An Implementation Technique for the Abstract Interpretation of Prolog

by

Annika Wærn

SICS R88004
Research Report
ISSN 0283-3638

An Implementation Technique for the Abstract Interpretation of Prolog

Annika Wærn
Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden

Abstract

An implementation technique for abstract interpretation is given which exploits stream communication. The implementation can alternatively be viewed as an ordinary interpretation collecting all solutions. The requirements for termination are discussed.

1 Introduction

Abstract interpretation /CoCo 77/ is a technique for global analysis of programs. For logic programs, one executes the program with an abstract description of a set of goals, producing an abstract description of the set of possible answer substitutions. Depending on what abstract descriptions are used, different information is yielded about the program.

As opposed to most other programming languages, the execution order for logic programs can be manipulated rather freely without changing the semantics. In abstract interpretation however, we are interested in yielding information about how information proceeds, rather than about execution independent properties. This makes it important to keep the same execution order as would be used in normal execution.

This paper discusses a general implementation technique for abstract interpretation of Prolog programs, assuming the normal execution order of Prolog. The abstract interpretation is performed top-down from a specified *calling pattern*, which indicates how the program is called by the user. The formal details concerning the correctness of the approach is beyond the scope of this paper. The approach can be viewed as an OLDT-resolution /TaSa 86/, for which abstract interpretation is discussed in /GaCo 87/. Appropriate discussions of correctness criteria for abstract interpretation of logic programs can be found in among other /JoSø 87/ and /Bru 87/.

2 Ensuring Termination

A Prolog interpreter can work in two different ways. One way is to build an and-or tree where substitutions only affect variables of the current clause or atomic goal. The other way is to keep a global substitution. Both these techniques can be used in abstract interpretation too, but the first makes it easier to construct terminating algorithms. Assume that the abstract interpretation represents an abstract substitution as a concrete substitution, except that variables take on values from some abstract domain rather than taking concrete terms as values. If the abstract value domain is finite, the first technique ensures that the complete set of abstract substitutions is finite, as each substitution is restricted to a finite number of clause variables. In the second approach substitutions can grow infinitely, simply by adding new variables. This property of the first approach makes it possible to construct execution schemes that are guaranteed to terminate given finite variable domains. The implementation technique discussed in this paper is such a scheme. In section 5 a version of the scheme is discussed, which terminates if the set of abstract substitutions is a noetherian lattice. (A noetherian lattice contains no infinitely increasing sequences a<b<c....)

3 The Framework Interpreter

We restrict ourselves to cut-free, static Prolog programs. To construct the abstract interpreter, we start out by constructing a Prolog (without cut) interpreter which keeps substitutions explicit, and collect for a goal the set of (global) answer substitutions.

In the interpreter given in figure 1, the following predicates are used, but not explicitly defined. This because the exact definition of them depend either directly on what abstract domain is intended, or else on the representation of clauses, goals and substitutions, for which the representation can vary somewhat depending on what abstract domain is intended.

get_clauses(Goal, Cllist):
: Get a list of clauses which are candidates for matching a goal. This predicate is the interface to the clause database. The predicate should fail if Goal is a built-in predicate, or if there are no candidate clauses for Goal (that is, Goal refers to an undefined predicate).

built_in(Goal, Subst, Substlist) :
: The predicate should be true only if Goal is a call to a built-in predicate. Substlist is a list of all possible resulting extensions of Subst from calling Goal.

failed(Goal, Subst) :
: Should succeed only if Goal is a call to an undefined predicate.

unify(Head, Goal, Subst, Subst') :
: Subst' is a smallest extension of Subst which unifies Head and Goal. Unify should fail if there is no such extension. The program can execute abstract or concrete interpretation depending on the definition of unify.

failed_unification(Head, Goal, Subst) :
: True only if there is no extension of Subst which unifies Head and Goal.

3.1 An Interpreter Keeping Substitutions Local to Clauses

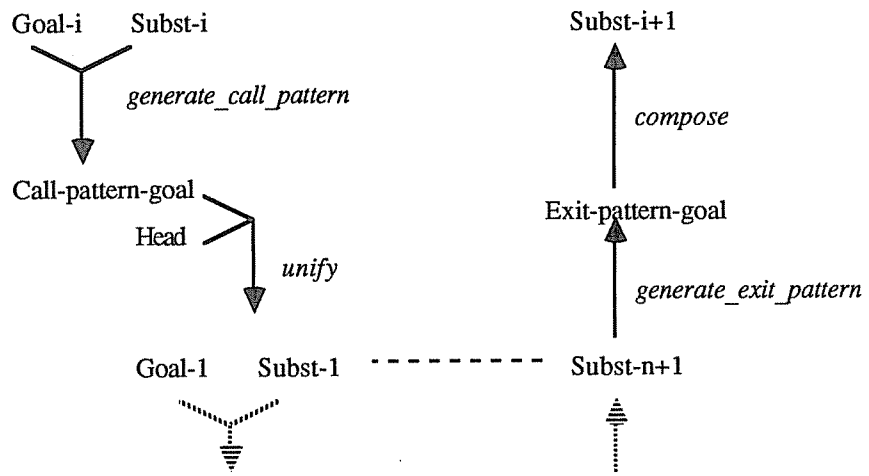
The next step is to obtain an interpreter which keeps each substitution local to a clause. This can be done by introducing a special goal format (a *call-exit pattern*) which is used to communicate information between a body goal and a clause. Although this is a single format, we will use the term *call patterns* as denoting call–exit patterns used when invoking a clause, and *exit patterns* as denoting call-exit patterns generated as the result of a clause execution. For concrete interpretation, a call or exit pattern is formed by simply applying the current substitution to the current goal. Since an abstract substitution usually is a relation on variables rather than a function from variables to values, a call-exit pattern is for abstract interpretation formed by restricting this relation to the variables of a goal. It is also possible to execute a generalization operation when forming a call or exit pattern. In general, the forming of a call or exit pattern can cause a loss of information.

When a goal is executed, it is first transformed into a call pattern (predicate *generate_call_pattern*). A head unification (*unify*) is executed between a clause head and a call pattern, and results in a substitution restricted to clause variables. When a clause is executed, its final substitution is used to generate an exit pattern (*generate_exit_pattern*), which transfers its information to the higher level clause (*compose*). It is worth noting that using this technique, the substitutions from a higher level clause and the next level clause never interact directly. The resulting program can be viewed alternatively as an ordinary interpreter, or as an abstract interpreter, depending on how the predicates *unify, get_clauses, compose, generate_call_pattern* and *generate_exit_pattern* are defined.

Fig. 1 A Prolog interpreter collecting substitutions
In the comments to the program 'set' denotes a list with no duplicate elements.

```
% Substlist is a set of all alternative extensions of Subst which are answer
% substitutions to Goal.

prove_goal(Goal, Subst, Substlist) :-
        get_clauses(Goal, Cllist),
        prove_goal_clauses(Cllist, Goal, Subst, Substlist).
prove_goal(Goal, Subst, Substlist) :-
        built_in(Goal, Subst, Substlist).
prove_goal(Goal, Subst, []) :-
        failed(Goal, Subst).


% Substlist is a set of all alternative extensions of Subst which are answer
% substitutions to Goal. The first argument to prove_goal_clauses is a list of
% alternative clauses for Goal.

prove_goal_clauses([],_,_,[]).
prove_goal_clauses([(Head :- Body)|RClauses], Goal, Subst, Substlist) :-
        unify(Head, Goal, Subst, Subst'),
        prove_conjunction_goals(Body, Subst', Substlist1),
        prove_goal_clauses(RClauses, Goal, Subst, Substlist2),
        union(Substlist1, Substlist2, Substlist).
prove_goal_clauses([(Head :- Body)|RClauses], Goal, Subst, Substlist) :-
        failed_unification(Head, Goal, Subst),
        prove_goal_clauses(RClauses, Goal, Subst, Substlist).


% Substlist is a set of all possible extensions of Subst, which are answer
% substitutions to the conjunction of the first argument list of goals.

prove_conjunction_goals([], Subst, [Subst]).
prove_conjunction_goals([Goal|RGoals], Subst, Substlist) :-
        prove_goal(Goal, Subst, Substlist'),
        prove_goals_alt_substs(Substlist', RGoals, Substlist).


% For each element in the first argument substitution list, find the
% corresponding set of answer substitutions to Goals. Substlist is the union of
% these sets.

prove_goals_alt_substs([],_,[]).
prove_goals_alt_substs([Subst|RSubsts], Goals, Substlist) :-
        prove_conjunction_goals(Goals, Subst, Substlist1),
        prove_goals_alt_substs(RSubsts, Goals, Substlist2),
        union(Substlist1, Substlist2, Substlist).

% The third argument is the set union of the first and second argument.
% Correct if all arguments are sets.

union([],Y,Y).
union([A|X], Y, [A|Z]) :-
        not(member(A, Y)),
        union(X, Y, Z).
union([A|X], Y, Z) :-
        member(A, Y),
        union(X, Y, Z).
```

Fig 2. Information flow between local substitutions



In figure 3, we use the term maplist intuitively for representing predicates that apply a predicate to all elements of a list. The first argument to maplist is the predicate to map, the second argument is the input list and the last the output list. All other arguments are duplicated for each call to the first argument.

Fig. 3 A Prolog interpreter collecting exit patterns, keeping substitutions local to clauses.

```
prove_goal(Call, Exitlist) :-
        get_clauses(Call, Cllist),
        prove_goal_clauses(Cllist, Call, Exitlist).
prove_goal(Call, Exitlist) :-
        built_in(Call, Exitlist).
prove_goal(Call, []) :-
        failed(Call).

prove_goal_clauses([],_,_,[]).
prove_goal_clauses([(Head :- Body)|RClauses], Call, Exitlist) :-
        unify(Head, Call, Subst),
        prove_conjunction_goals(Body, Subst, Substlist),
        maplist(generate_exit_pattern, Substlist, Head, Call, Exitlist1),
        prove_goal_clauses(RClauses, Call, Exitlist2),
        union(Exitlist1, Exitlist2, Exitlist).
prove_goal_clauses([(Head :- Body)|RClauses], Call, Exitlist) :-
        failed_unification(Head,Call),
        prove_goal_clauses(RClauses, Call, Exitlist).

prove_conjunction_goals([], Subst, (Subst.[])).
prove_conjunction_goals([Goal|RGoals], Subst, Substlist) :-
        generate_call_pattern(Goal, Subst, Call),
        prove_goal(Call, Exitlist),
        maplist(compose, Exitlist, Subst, Substlist'),
        prove_goals_alt_substs(Substlist', RGoals, Substlist).
```

```
prove_goals_alt_substs([],_,[]).
prove_goals_alt_substs([Subst|RSubsts], Goals, Substlist) :-
        prove_conjunction_goals(Goal, Subst, Substlist1),
        prove_goals_alt_substs(RSubsts, Goals, Substlist2),
        union(Substlist1, Substlist2, Substlist).

union(X, Y, Z) :- defined as in figure 1.
```

3.2 An Interpreter Including Loop Detection

The interpreter of figure 3 does not always terminate, even if the abstract domain used is finite, since it has no loop detection. Proper loop detection can easily be provided by a lookup table containing all call patterns which have occurred thus far. Then termination is ensured, as long as the number of possible call patterns is finite. (There exist also more complex forms of loop detection, which ensure termination if the set of call patterns is a lattice of finite height /Bru 87/.)

The question of what to do when a loop is detected is trickier, though. The trick is to recognize that the recurring goal should have the same set of exit solutions as the original one. We extend the lookup table to contain for each call its list of exit solutions. When a call recurs in an execution, we immediately instantiate its solution set to the list given in the lookup table. However, this list might not be completely instantiated yet. If we executed the program like this, the predicates operating on such lists would instantiate an uninstantiated tail to nil, with the possibility to backtrack later if the set of solutions is incomplete. This creates an enormous amount of completely unnecessary backtracking, since it is well-defined which predicates generate and consume these lists.

The solution is to convert all predicates operating on substitution or call-exit pattern lists into *stream predicates*. A call to a stream predicate is delayed until the first element of the input list is instantiated. (The input list does not need to be ground, since a recursive call to the stream predicate should be delayed in the same manner.) In the program of figure 2, the crucial predicates are *prove_goals_alt_substs*, for which the complete list of alternative substitutions might not be immediately provided, all predicates represented by *maplist*, and *union*. In program of figure 3, *map_stream* represents predicates of the same kind as map_list did in fig. 2, except these also exploit stream behavior. Furthermore *prove_goals_alt_substs* must exploit stream behavior. *Union* should delay until one of its input streams are instantiated, that is, it should exploit stream behavior for *two* lists. We add two predicates for handling lookup tables:

already_in((Call, Exitlist), Lookup) :
Succeed if *Call* already is in the lookup table, unifying *Exitlist* with its corresponding list of solutions. Fails if *Call* has not been previously encountered.

add((Call, Exitlist) Lookup) :
Succeeds if *Call* has not previously been encountered, adding the pair of *Call* and *Exitlist* to *Lookup*. Fails otherwise.

The lookup table in this program does not only prevent unnecessary loops, but also avoids unnecessary recomputation of goals that already are being computed in some other branch of the tree. Thus, the lookup table is used for recording what so far has been going on during the execution. Unfortunately, this use requires an non-logical variable test in the implementation of *already_in* and *add*. This would not be necessary if lookup was used only for loop detection, or if *prove_goals_alt_substs* could not suspend. The *add* predicate would in this case take three arguments, the tuple to add and the lookup table before and after adding the element.

Fig. 4 Including loop detection in the interpreter of Fig. 3

```
prove_goal(Call, Lookup, Exitstream) :-
        already_in((Call, Exitstream), Lookup).
prove_goal(Call, Lookup, Exitstream) :-
        get_clauses(Goal, Cllist),
        add((Call, Exitstream), Lookup),
        prove_goal_clauses(Cllist, Call, Lookup, Exitstream).
prove_goal(Call,_, Exitstream) :-
        built_in(Call, Exitstream).
prove_goal(Call,_,[]) :-
        failed(Call).

prove_goal_clauses([],_,_,_,[]).
prove_goal_clauses([(Head :- Body)|RClauses],Call,Lookup,Exitstream) :-
        unify(Head, Call, Subst),
        prove_conjunction_goals(Body, Subst, Lookup, Substream),
        mapstream(generate_exit_pattern,
                  Substream, Head, Call, Exitstream1),
        prove_goal_clauses(RClauses, Call, Lookup, Exitstream2),
        union(Exitstream1, Exitstream2, Exitstream).
prove_goal_clauses([(Head :- Body)|RClauses],Call,Lookup,Exitstream) :-
        failed_unification(Head, Call),
        prove_goal_clauses(RClauses, Call, Lookup, Exitstream).

prove_conjunction_goals([], Subst, _, (Subst . [])).
prove_conjunction_goals([Goal|RGoals], Subst, Lookup, Substream) :-
        generate_call_pattern(Goal, Subst, Call),
        prove_goal(Call, Lookup, Exitstream),
        mapstream(compose, Exitstream, Subst, Substream'),
        prove_goals_alt_substs(Substream', RGoals, Lookup, Substream).

prove_goals_alt_substs([],_,_,[]).
prove_goals_alt_substs([Subst|RSubsts], Goals, Lookup, Substream) :-
        prove_conjunction_goals(Goal, Subst, Lookup, Substream1),
        prove_goals_alt_substs(RSubsts, Goals, Lookup, Substream2),
        union(Substream1, Substream2, Substream).

union(X, Y, Z) :- example definition in figure 5.
```

When the program is used as an abstract interpreter, the argument 'Lookup' contains exactly the information we are looking for. For each call pattern formed during execution of the top-level goal 'Lookup' contains its corresponding set of exit patterns.

It is important to realize that the program does not 'terminate', in the sense that, at termination some goals might still be delayed on uninstantiated streams. This is not termination in the pure sense, the system just 'stops doing anything'. Procedurally it is a little bit difficult to accomplish pure termination. It can be done using a version of the shortcircuit technique which allows the detection of deadlock situations /Sh 85/. All uninstantiated streams can then be closed and pure termination can be achieved. It is unnecessary to do this however, since the fact that the system stops doing anything is a sufficient criterion to detect that all necessary elements have been added to the streams.

The program of figure 3 can be realized in a Prolog dialect which exploits the **freeze** predicate /Co 82/, /Ca 87/, where **freeze(X, B)** delays the execution of B until X is bound to a non–variable. It can also be implemented in a Prolog dialect with **wait** annotations, like MU-Prolog /Na 84/. (It is also fairly straightforward to define a Prolog + freeze interpreter in Prolog.) Due to the inherent determinism in the

program, it can also be implemented in a committed-choice language such as Concurrent Prolog /Sh 83/ or GHC /Ue 85/.

Fig. 5 The union operation on streams, defined using freeze.

```
union(X, Y, Z) :- union(X, Y, Z, []).

% Union needs an extra argument keeping record of what elements have
% been put on the output stream.

union(X, Y, Z, Memory) :-
        freeze_xor((X, unionhlp(X, Y, Z, Memory)),
                   (Y, unionhlp(Y, X, Z, Memory))).

unionhlp([], Y, Z, Memory) :-
        delete_already_generated(Y, Memory, Z).
unionhlp([A|X], Y, (A. Z), Memory) :-
        not(member(A, Memory)),
        union(X, Y, Z, (A . Memory)).
unionhlp([A|X], Y, Z, Memory) :-
        member(A, Memory),
        union(X, Y, Z, Memory).

% Delete_already_generated puts elements from the first arguments onto
% the output stream, only if they have not already been encountered.
% The first argument is assumed to be a set.

delete_already_generated([], _, []).
delete_already_generated([A|X], Memory, [A|Z]) :-
        not(member(A, Memory)),
        delete_already_generated(X, Memory, Z).
delete_already_generated([A|X], Memory, Z) :-
        member(A, Memory),
        delete_already_generated(X, Memory, Z).

% freeze_xor executes at most one of Goal1 and Goal2, depending on which one
% of Var1 and Var2 become instantiated first. If neither becomes
% instantiated, none of the goals are executed.

freeze_xor((Var1, Goal1), (Var2, Goal2)) :-
        freeze(Var1, xor(1, C, Goal1)),
        freeze(Var2, xor(2, C, Goal2)).

xor(C, C, Goal) :- !, Goal.
xor(_, _, _).
```

A complete interpreter, which differs only in minor details from the one described here is given in appendix A.

4 Inferring Information

When executing a unification, information is added about variable states and dependencies. This information does not only affect the variables participating in the unification, but also indirectly variables that have been coupled to these by unification. This kind of information can either always be immediately inferred and kept explicit, or kept implicit. In the latter case, it is necessary to do the inferences when a call or exit pattern is formed, since these are formed using only a subset of the variables of a clause. It is thus important that all information about these variables are gathered before forming the call or exit pattern.

For domains keeping track of variable dependencies the second, 'lazy' approach is more efficient, since variable dependencies change very rapidly. It presents an additional semantic problem though, since several abstract substitutions can represent the same set of concrete substitutions. (Call and exit patterns still behave nicely, since all information can be kept explicit in these.)

5 A Simple Domain Example

One application for abstract interpretation is groundness analysis. Given information about how a program is called, the abstract interpreter is used to infer information about what arguments are ground when the different predicates of the program are called. Of course, there can be several different call patterns for every predicate. The abstract interpreter described so far collects a set of call patterns which covers all possible runtime calls, and for every call pattern a corresponding set of exit patterns. (Section 6 describes a version which would generate a single exit pattern for each call pattern.)

Groundness analysis can be done using several different abstract domains of varying complexity. This section describes briefly a simple version which gives a rather coarse analysis. A very similar analysis is thoroughly described in /De 87/. Here, the aim is just to give a flavor of how the interpreter could be used.

In appendix B, a suboptimal but complete coding of the domain is contained, which is executable in connection with the interpreter given in appendix A.

We choose to represent an abstract substitution as a function from variables to values in the set **{free, ground, dont-know}**. It is possible to allow variables to take on the value 'empty' too, representing that unification surely failed, but instead we will construct an abstract unification algorithm that fails in such cases. A variable takes on the value 'dont-know' if it might be free, ground or instantiated. (Instantiated could also be a separate abstract value.) An abstract substitution is represented as a closed association list. Variables which are not explicitly given a value in an abstract substitution are assumed free.

This domain doesn't give any information about what variables are aliased to each other. Thus, we must assume that each time a predicate is called, the non-ground variables of the call might be aliased to each other, and that the execution of a goal might introduce variables between non-ground variables. This affects the unification algorithm, and the algorithm for incorporating information from an exit pattern into a clause substitution.

5.1 Generating Call and Exit Patterns

Call and Exit patterns are represented as tuples (Term, Subst) where Subst is an abstract substitution restricted to the variables in Term.

The predicate *generate_call_pattern(Goal, Subst, (CallGoal, CallSubst))* constructs CallGoal as a copy of Goal, only that the variables in Goal are renamed according to a specific traversal order. This makes it possible to compare call patterns by a simple equality. Variables are also marked to belong to a call pattern. This ensures that call variables will have different names from the clause variables they might unify against. CallSubst is then generated by restricting Subst to the variables of Goal, and simultaneosly renaming the variable keys in Subst according to the variable names use in CallGoal.

Figure 6 Call pattern generation

```
generate_call_pattern(Goal, Subst, (CallGoal, CallSubst)) :-
        rename(Goal, Rename_table, CallGoal),
        restrict(Subst, Rename_table, CallSubst).
```

The predicate *generate_exit_pattern(Subst, Head, Call, Exit)* generates an exit pattern with the same call goal as the original call but with a new substitution. This is done by comparing the current instantiation of the clause head to the original call, and instantiating such call variables which would unify against a head structure, or an instantiated variable.

5.2 Unification

Unification always takes place between a call pattern and a clause head whose variables initially are free, and produces a substitution on the clause variables. Since we don't keep track of aliases, we must assume that all variables might be dependent on each other. When a variable is unified against a non-variable, every other free variable changes its state to dont-know. For call variables this is obvious, since they might be dependent of each other when the unification is invoked. Clause variables might already have unified against call variables. We can do an interesting restriction though: if a variable has not previously been used at all, it cannot have unified against any other variable, and thus it cannot be affected by the instantiation of any other variable. Since we do not add a variable to a substitution unless it has participated in a unification, we can thus restrict the correction for possible aliases to variables occurring explicitly in the substitution.

Fig. 7 Unification between a call pattern and a clause head

```
unify((CallGoal, CallSubst), Head, Subst) :-
        unify(Head, [], Subst, CallGoal, CallSubst, _).

unify(Head, Subst, Subst2, Call, CSubst, CSubst2) :-
        (variable(Head); variable(Call)),!,
        replace_value(Head, Val1, Val, Subst, Subst1),
        replace_value(Call, Val2, Val, CSubst, CSubst1),
        combine_two_values(Val1, Val2, Val),
        correct_for_alias(Val, Subst1, Subst2),
        correct_for_alias(Val, CSubst1, CSubst2).
unify(Atom, Subst, Subst, Atom, CSubst, CSubst) :-
        atomic(Atom),!.
unify([Head | Tail], Subst, Subst2, [CallHead|CallTail], CSubst, CSubst2) :-
        unify(Head, Subst, Subst1, CallHead, CSubst, CSubst1),
        unify(Tail, Subst1, Subst2, CallTail, CSubst1, CSubst2).
unify(Struct, Subst, Subst1, CallStruct, CSubst, CSubst1) :-
        Struct =.. [F|Arg],
        CallStruct =.. [F|CallArg],
        unify(Arg, Subst, Subst1, CallArg, CSubst, CSubst1).

correct_for_alias(free, free, Subst, Subst).
correct_for_alias(free, Val, Subst, Subst1) :-
        Val \== free,
        correct_for_alias(Subst, Subst1).
correct_for_alias(dont-know, _, Subst, Subst1) :-
        correct_for_alias(Subst, Subst1).
correct_for_alias(ground, ground, Subst, Subst).

correct_for_alias([],[]).
correct_for_alias([(Var,free)|Subst],[(Var,dont-know)|Subst1]) :-
        correct_for_alias(Subst, Subst1).
correct_for_alias([(Var, Val)|Subst], [(Var, Val)|Subst1]) :-
        Val \== free,
        correct_for_alias(Subst, Subst1).
```

5.3 Propagating Information from Exit Patterns

When an exit pattern is returned from a body goal, the information yielded from the call should be carried over to the clause substitution. This is done by the predicate *compose(Exit, Goal, Subst, Subst1)*. The operation is simple, since the exit patterns contains variables in the exact same positions as the body goal for which it represents the exit. We must assume that all body variables might be dependent on each other before the call. Thus, if a variable changes its state from free to anything else during a call, all variables which were free before the call must change their state to dont-know. As for unification, we can restrict this correction to variables occurring in the represented substitution, as these are the only ones which might have participated in the head unification or a previous call, and thus the only ones which might have become aliased to anything.

Fig. 8 Propagating information from an Exit pattern to a clause Substitution

```
compose((ExitGoal, ExitSubst), Goal, Subst, Subst2) :-
        rename(Goal, Rename_table, ExitGoal),
        change_some(ExitSubst, Rename_table, Subst, Subst1, Some_change),
        correct(Some_change, Subst1, Subst2).

correct_for_alias(nochange, Subst, Subst) :- !.
correct_for_alias(change, Subst, Subst1) :- correct_for_alias(Subst, Subst1).
```

6 Using upper bound of abstract substitutions

The set of concrete answer substitutions to a goal is in this approach represented by a *set* of abstract substitutions. For some abstract interpretations, this set can be large even if finite, and it might be of interest to keep a single abstract substitution as output, which represents an upper bound of the set of output abstract substitutions. We could modify the scheme to accomplish this. Let all stream predicates do the following operation:

- When a new abstract substitution or call-exit pattern is formed, compute the *upper bound* of the new element and the element latest added to the stream. Add the upper bound element to the output stream only if it is different from the element latest added to the output stream.

Using this technique, a stream becomes an monotonely increasing sequence of approximations of an upper bound to the set of output abstract substitutions. The process will terminate if the lattices of abstract substitutions and call-exit patterns are noetherian. (Note that the loop detection technique used requires that the number of call-exit patterns is finite.) The approach is more efficient in two ways: the operation can be less complicated than stream union (depending on the complexity of the upper bound operation), and it will also keep streams shorter.

In general, using an upper bound operation results in loss of information. This is easily seen for abstract substitutions represented as concrete substitutions on abstract value domains.

Example 1

Assume that the abstract substitutions [ground/X, top/Y] and [top/X, ground/Y] are possible results for a certain goal. Then the union of these substitutions contains the information that either X or Y is ground. The upper bound of the two substitutions is [top/X, top/Y], from which we can conclude no such thing.

6 Current results and future work

Two interpreters have been developed, one the simple interpreter described in this paper, for which the code is enclosed in the appendix. The other is constructed somewhat differently from the interpreter described in section 3. In addition to the call and exit patterns the more complex interpreter also gathers information about the abstract substitutions occurring at *program points*. For each clause, a set of program points is defined; the first one being the time when head unification is finished and no body goal has been started, and the rest being the times when a body goal has finished executing, before starting the next body goal. The information is structured as a set of *trees* of abstract substitutions, such that one tree is associated to each pair of clause and matching call pattern.

For the construction of these trees, loop detection must be modified to detect when a recurring goal tries to match against an already used clause. This loop detection is still sufficient to ensure termination for finite domains, though less efficient.

The interpreter has been used in conjunction with two different domains, both representing an abstract substitution as an ordinary substitution where variables take on values from an abstract domain. The first domain is a simple groundness analysis; whereas, the second one also keeps track of variable relations. We are presently in the process of constructing and testing other domains.

An urgent topic of the project is to formally prove the correctness of the interpretation method. It should be proven that the interpreter implements an interpretation function which fulfils the correctness criteria for abstract interpretation /JoSø 87/. This interpretation function could either be one defined by other researchers in the field, or defined as a part of this project. The first would be preferable.

The interpreter was developed as a part of a project, whose goal is to develop an abstract interpretation which gives sufficient information for detection of independent and-parallelism /SjWæ 88/. We expect to use the general framework for testing different abstract domains on real programs, in order to estimate the actual information gain as related to the complexity of an abstract domain. For this sake, we are currently extending the abstract interpreter to deal better with built-ins.

7 Acknowledgements

I am indebted to Thomas Sjöland for many fruitful discussions and close cooperation in the complete project. I am also very much in debt to my supervisor Seif Haridi, without whom this research project would not exist. Many thanks also to Will Winsborough, Mats Carlsson and Manny Rayner who all read and commented on earlier versions of this paper.

8 References

/Bru 87/ **Bruynooghe M.,**
'A Framework for the Abstract Interpretation of Logic Programs'
Draft, Dept. of Computer Science,Katholieke Universiteit Leuven, Belgium (1987).

/Ca 87/ **Carlsson M.,**
'Freeze, Indexing and Other Implementation issues in the WAM'
in *Logic Programming*, Proceedings of the fourth Int. Conference, ed. Lassez, MIT Press,
ISBN 0-262-12125-5 (1987) pp. 40 - 58.

/Co 82/ **Colmeraur A.,**
' Prolog II, Manuel de référence et model theorique'
Groupe d'intelligence Artificielle, Universite Marseille II (1986).

/CoCo 77/ **Cousot P. & Cousot R.,**
'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints',
POPL 77 (1977).

/De 87/ **Debray S.K.,**
'Static Inference of Modes and Data Dependencies in Logic Programs',
TR - 87/24, Dept. of Computer Science,
University of Arizona (1987).

/GaCo 87/ **Gallagher J. & Codish M.,**
'Specialisation of Prolog and FCP Programs Using Abstract Interpretation'
ICT TC-2 Workshop on Partial Evaluation and Mixed Computation, Denmark (1987), (preliminary proceedings pp. 125 -134).

/JoSø 87/ **Jones N.D. & Søndergaard H.,**
'A Semantic-Based Framework for the Abstract Interpretation of Prolog'
in *Abstract Interpretation of Declarative Languages*,
ed. Abramsky & Hankin, Ellis Horwood (1987).

/Na 84/ **Naish L.,**
' Mu-prolog 3.1 Reference Manual'
Department of Computer science University of
Melbourne (1984).7

/Sh 83/ **Shapiro E. Y.,**
'A Subset of Concurrent Prolog and Its Interpeter' ICOT
Technical Report TR-003 (1983).

/Sh 85/ **Shapiro E.Y.,**
'Programming in Concurrent Prolog: Lecture Notes'
(1985).

/SjWæ 88/ **Sjöland T. & Wærn A.,**
'Compiling AND-Parallelism'
forthcoming as SICS report (1988).

/TaSa 86/ **Tamaki H. & Sato T.,**
'OLD Resolution with Tabulation',
in *proc. ICLP 86*, LNCS 225; Springer Verlag,
(1986).

/Ue 85/ **Ueda K.,**
'Guarded Horn Clauses'
ICOT Technical Report TR-103 (1985).

Appendix A

```
%************************************************************************
% Toplevel for abstract interpretation using substitutions local to clause.
% The abstract interpreter terminates under the following conditions:
%  -The value domain must be finite.
% - The set of possible call patterns must be finite.

%------------------------------------------------------------------------
% A call- and exit pattern should have the form (Term, Subst), where Subst is
% an abstract substitution affecting only the variables in Term.
% Use this toplevel together with an abstract interpretation defined by
% the following predicates:

% generate_call(Goal, Subst, Call)
%     should generate a call pattern from a body goal and its environment.

% generate_exit(Call, Head, Subst, Exit)
%     Takes a call pattern and a substitution and generates a analogous
%     exit pattern but with variables affected by the substitution.
%     This predicate is allowed to fail, for example if a substitution
%     describes an unreachable state.

% compose(G, Subst, Call, Exit, New_subst)
%     generates a new global substitution for a body goal, given:
%     G : The body goal
%     Subst: The old environment.
%     Call: The call pattern that produced the exit pattern Exit.

% unification(Head, Call, Subst)
%     The clause head Head is unified with the call pattern, generating Subst.
%     Unification is allowed to fail.

% builtin(Call, ExitStream)
%     Execution of built-in predicates. Should generate a set of
%     exit patterns which completely describe all possible results from the
%     call.

% preproc_goal(GoalDescr, Goal, Subst)
%     A preprocessor, which allows the user to use a common input format for
%     goals regardless of the abstract domain to be used. GoalDescr is an
%     ordinary prolog goal, except the two following constructs are allowed
%     to be used:
%     '@g'                  represents any ground term
%     '@i'(Arg1,..,Argn)    represents a non-ground structure. Arg1... Argn is
%                           a complete list of variables occurring in the
%                           structure, containing duplicates if a variable occurs
%                           several times in a structure.
%     Output from preproc_goal is a body goal + a substitution, in the domain-
%     dependent representation.

% preproc_clause(Pred,Args,Body,Tuple)
%     A preprocessor which takes an ordinary Prolog clause (split into Head +
%     Arguments + Body), and generates a clause represented in the domain-
%     dependent representation.
```

```
%*************************************************************************
% toplevel call

execute_intp(Goal, Lookup) :-
        preproc_goal(Goal, ProgramGoal, ProgramSubst),
        intp_conjunction_goals(ProgramGoal, ProgramSubst, Lookup, SubstStream).

%-------------------------------------------------------------------------
% Intp-goal:
% executes an atomic call, generating a list of exit patterns.

intp_goal(Call, Lookup, Exitstream) :-
        already_in((Call, Exitstream), Lookup).
intp_goal(Call, Lookup, Exitstream) :-
        builtin(Call, Exitstream),!.
intp_goal(Call, Lookup, Exitstream) :-
        add((Call, Exitstream), Lookup),
        get_clauses(Call, Cllist),
        intp_clauses(Cllist, Call, Lookup, Exitstream).

% intp_clauses:
% takes a call pattern and executes it against a list of possible clauses.

intp_clauses([],_,_,[]).
intp_clauses([(Head :- Body)| Rcl], Call, Lookup, Exitstream) :-
        unify(Head, Call, Subst),!,
        union_stream(Exitstream1, Exitstream2, Exitstream),
        intp_conjunction_goals(Body, Subst, Lookup, Subststream),
        generate_exit_stream(Subststream, Head, Call, Exitstream1),
        intp_clauses(Rcl, Call, Lookup, Exitstream2).
intp_clauses([Failed | Rcl], Call, Lookup, Exitstream) :-
        intp_clauses(Rcl, Call, Lookup, Exitstream).

% intp_conjunction_goals:
% executes a list of body goals and an environmental substitution,
% and generates a list of alternative result substitutions.

intp_conjunction_goals((Goal, RGoals), Subst, Lookup, Subststream) :-
        !,generate_call(Goal, Subst, Call),
        intp_goal(Call, Lookup, Exitstream),
        compose_stream(Exitstream, Call, Goal, Subst, Subststream1),
        intp_goals_alt_subst(Subststream1, RGoals, Lookup, Subststream).
intp_conjunction_goals(Goal, Subst, Lookup, Subststream) :-
        generate_call(Goal, Subst, Call),
        intp_goal(Call, Lookup, Exitstream),
        compose_stream(Exitstream, Call, Goal, Subst, Subststream).

% intp_goals_alt_subst:
% takes a list of body goals and a list of alternative substitutions, and
% executes the goals for each alternative substitution.

:- wait intp_goals_alt_subst/4.

intp_goals_alt_subst([],Goals, Lookup, Subststream).
intp_goals_alt_subst([Subst|Rsubst],Goals, Lookup, Subststream) :-
        union_stream(Subststream1, Subststream2, Subststream),
        intp_conjunction_goals(Goals, Subst, Lookup, Subststream1),
        intp_goals_alt_subst(Rsubst, Goals, Lookup, Subststream2).
```

```
% already_in:
% true if the call is already being explored.

already_in(A, Lookup) :- var(Lookup), !, fail.
already_in(A, [A | _]).
already_in(A, [X|Lookup]) :- \+ A=X, already_in(A, Lookup).

% add:
% Add a new entry to Lookup, only if it's not already in Lookup.

add(A, Lookup) :- var(Lookup), !, Lookup=[A|_].
add(A,[X|R]) :- \+ A = X, add(A,R).

%-------------------------------------------------------------------------------
% stream handling predicates

% union_stream:
% merge two streams into a stream without duplicate entries.

union_stream(X,Y,U) :-
        union_stream(X,Y,U,[]).

union_stream(X,Y,U,Mem) :-
        freeze_or((X,union_hlp(X,Y,U,Mem)),
                  (Y,union_hlp(Y,X,U,Mem))).

union_hlp([],Y,U,Mem) :-
        atss(Y,U,Mem).
union_hlp([A|X],Y,U,Mem) :-
        add_to_set(A,Mem,Mem1,U,U1),union_stream(X,Y,U1,Mem1).

% add_to_set_stream:
% U is a stream like Y, except duplicates are deleted.

add_to_set_stream(Y,U) :- freeze(Y,atss(Y,U,[])).

:- wait atss/3.
atss([],[],_).
atss([A|X],U,Mem) :-
        add_to_set(A,Mem,Mem1,U,U1),
        atss(X,U1,Mem1).

add_to_set(El, OldUnion, OldUnion, Out, Rest) :-
        member(El, OldUnion),
        Out = Rest.
add_to_set(El, OldUnion, [El | OldUnion], Out, Rest) :-
        \+ member(El, OldUnion),
        Out = [El|Rest].

% generate_exit_stream:
% apply generate_exit to each element in the stream Substlist, and generate
% monotonely increasing stream, where the last element always represents
% the union of the results produced sofar.

generate_exit_stream(Substream, Head, Call, Exitstream) :-
        gen_exit_stream(Substream, Head, Call, Exitstream, []).
```

```
:- wait gen_exit_stream/5.
gen_exit_stream([],_,_,[],_).
gen_exit_stream([Subst|Rsubst],Head,Call,Exitstream,Memory) :-
        generate_exit(Call, Head, Subst, Exit),
        add_to_set(Exit,Memory,Memory1,Exitstream, Exitstream1),
        gen_exit_stream(Rsubst,Head, Call, Exitstream1, Memory1).

% compose_stream:
% apply compose to each element in the stream GDlist, and generate a monotonely
% increasing stream of abstract substitutions, last element representing
% the union of the current results.

compose_stream(Exitstream, Call, Goal, Subst, Subststream) :-
        comp_stream(Exitstream, Call, Goal, Subst, Subststream, []).

:- wait comp_stream/6.
comp_stream([],_,_,_,[],_).
comp_stream([Exit|RExit], Call, Goal, Subst, Subststream, Mem) :-
        compose(Goal, Subst, Call, Exit, NSubst),!,
        add_to_set(NSubst,Mem,Mem1,Subststream,Subststream1),
        comp_stream(RExit, Call, Goal, Subst, Subststream1, Mem1).
comp_stream([Exit|RExit], Call, Goal, Subst, Subststream, Mem) :-
%       \+ compose(Goal, Subst, Call, Exit, _),
        comp_stream(RExit, Call, Goal, Subst, Subststream, Mem).

%-------------------------------------------------------------------------------
% Primitives for mutual exclusion between alternatives.
% List is a list of alternatives (Variable, Goal) such that Goal is executed
% if Variable is the first variable of the alternatives to become instantiated.
% If a pair is annotated as ndet(Variable, Goal) the program is allowed to
% backtrack to other alternatives if Goal fails.

freeze_cond(List) :-
        freeze_cond(List,1,_).

freeze_cond([],_,_).
freeze_cond([ndet(Var,Goal)|Rgoals],N,Cmt) :-
        !,freeze(Var,condhlp(N,Cmt,Goal)),
        N1 is N + 1,
        freeze_cond(Rgoals,N1,Cmt).
freeze_cond([(Var,Goal)|Rgoals],N,Cmt) :-
        freeze(Var,condhlp_det(N,Cmt,Goal)),
        N1 is N + 1,
        freeze_cond(Rgoals,N1,Cmt).

freeze_or(Case1,Case2) :-
        freeze_cond([Case1,Case2],1,_).

condhlp(Cmt,Cmt,G) :- G.
condhlp(_,_,_).

condhlp_det(Cmt,Cmt,G) :- !,G.
condhlp_det(_,_,_).

%-------------------------------------------------------------------------------
% Clause retrieval from database.

get_clauses((Term,Subst),Clauselist) :-
   Term =.. [F|Args],
   length(Args,N),
   clauses(F,N,Clauselist),!.
```

```
%*************************************************************************
% Framework for preprocessing clauses.

%-------------------------------------------------------------------------
% Read clauses from file, and ADD them to current program clauses, that is,
% COMPILE, not recompile.

preproc_file(Name) :-
   see(Name),
   read(First),
   preproc_file_hlp(First).

preproc_file_hlp(end_of_file) :- !,seen.
preproc_file_hlp(Clause) :-
   preproc_clause(Clause),
   read(Next),
   preproc_file_hlp(Next).


%-------------------------------------------------------------------------
% Preprocessor for clauses.
% Generates the predicate:
% clauses(Functor, Number_of_args, List_of_clauses).

:- dynamic clauses/3.

preproc_clause((H :- B)) :-
   !,H =.. [Pred|Args],
   length(Args,N),
   preproc_clause_hlp(Pred,N,Args,B).
preproc_clause(H) :-
   H =.. [Pred|Args],
   length(Args,N),
   preproc_clause_hlp(Pred,N,Args,true).

preproc_clause_hlp(Pred,N,Args,Body) :-
   clauses(Pred,N,List),!,
   preproc_clause(Pred,Args,Body,Tuple),
   retract(clauses(Pred,N,List)),
   append(List,[Tuple],Nlist),
   assert(clauses(Pred,N,Nlist)).
preproc_clause_hlp(Pred,N,Args,Body) :-
   preproc_clause(Pred,Args,Body,Tuple),
   assert(clauses(Pred,N,[Tuple])).

%-------------------------------------------------------------------------
% General useful primitives

onesol(G) :- G,!.

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

member(X, [X|_]).
member(X, [A|R]) :- X \== A, member(X, R).
```

Appendix B

```
%-----------------------------------------
% builtin(Call, Exit).

builtin((true, Subst), [(true,Subst)]).
builtin((A > B, Subst), [(A > B, Subst)]).
builtin((A < B, Subst), [(A < B, Subst)]).

%-----------------------------------------
% unify(Head, Call, Subst).


unify(Head, (CallGoal, CallSubst), Subst) :-
        unify(Head, [], Subst, CallGoal, CallSubst, _).

unify(Head, Subst, Subst2, Call, CSubst, CSubst2) :-
        (variable(Head); variable(Call)),!,
        replace_value(Head, Val1, Val, Subst, Subst1),
        replace_value(Call, Val2, Val, CSubst, CSubst1),
        combine_two_values(Val1, Val2, Val),
        correct_for_alias(Val1, Val, Subst1, Subst2),
        correct_for_alias(Val2, Val, CSubst1, CSubst2).
unify(Atom, Subst, Subst, Atom, CSubst, CSubst) :-
        atomic(Atom),!.
unify([Head | Tail], Subst, Subst2, [CallHead|CallTail], CSubst, CSubst2) :-
        unify(Head, Subst, Subst1, CallHead, CSubst, CSubst1),
        unify(Tail, Subst1, Subst2, CallTail, CSubst1, CSubst2).
unify(Struct, Subst, Subst1, CallStruct, CSubst, CSubst1) :-
        Struct =.. [F|Arg],
        CallStruct =.. [F|CallArg],
        unify(Arg, Subst, Subst1, CallArg, CSubst, CSubst1).

% correct_for_alias(OldVal, NewVal, Subst, Subst1):
% change all free variables in the substitution to dont_know if the
% OldVal-NewVal pair indicates that a variable might have changed its value.

correct_for_alias('@free', Val, Subst, Subst1) :-
        Val \== '@free',!,
        correct_for_alias(Subst, Subst1).
correct_for_alias('@dont-know', _, Subst, Subst1) :-
        !, correct_for_alias(Subst, Subst1).
correct_for_alias(_, _, Subst, Subst).


%-----------------------------------------
% generate_exit(Call, Head, Subst, Exit).

generate_exit((CallGoal,CallSubst), Head, Subst, (CallGoal,ExitSubst)) :-
        compare(Head, Subst, CallGoal, CallSubst, ExitSubst).

compare(Var, Subst, Struct, ExitSubst, ExitSubst1) :-
        variable(Var),!,
        find_value(Var, Subst, Val),
        set_value(Struct, Val, ExitSubst, ExitSubst1).
compare(Struct, Subst, Var, ExitSubst, ExitSubst1) :-
        variable(Var), !,
        find_value(Struct, Subst, Val),
        set_value(Var, Val, ExitSubst, ExitSubst1).
compare(Atom, Subst, Atom, ExitSubst, ExitSubst) :-
        atomic(Atom), !.
```

```
compare([Head|Tail], Subst, [CallHead|CallTail], ExitSubst, ExitSubst2) :-
        !, compare(Head, Subst, CallHead, ExitSubst, ExitSubst1),
        compare(Tail, Subst, CallTail, ExitSubst1, ExitSubst2).
compare(Struct, Subst, CallStruct, ExitSubst, ExitSubst1) :-
        Struct =.. [F|Arg],
        CallStruct =.. [F | CallArg],
        compare(Arg, Subst, CallArg, ExitSubst, ExitSubst1).


%-----------------------------------------
% generate_call(Goal, Subst, Call).


generate_call(Goal, Subst, (CallGoal, CallSubst)) :-
        rename(Goal, 1, _, Rename_table, CallGoal),
        restrict(Subst, Rename_table, CallSubst).

rename(Var,N,N1,Rename_table,CallVar) :-
        variable(Var),!,
        lookup_add(Var,Rename_table,CallVar),
        new_or_old(CallVar, N, N1).
rename(Atom, N, N, _, Atom) :-
        atomic(Atom),!.
rename([Head|Tail], N, N1, Rename_table, [CallHead|CallTail]) :-
        !, rename(Head, N, N2, Rename_table, CallHead),
        rename(Tail, N2, N1, Rename_table, CallTail).
rename(Struct, N, N1, Rename_table, CallStruct) :-
        Struct =.. [F|Arg],
        rename(Arg, N, N1, Rename_table, CallArg),
        CallStruct =.. [F|CallArg].

new_or_old('@var'('@call',N), N, N1) :- N1 is N + 1.
new_or_old('@var'('@call',X), N, N) :- X \== N.

restrict([], Rename, []).
restrict([(Var,Val)|Subst], Rename, [(CallVar,Val)|CallSubst]) :-
        lookup(Var,Rename,CallVar),!,
        restrict(Subst, Rename, CallSubst).
restrict([Pair|Subst], Rename, CallSubst) :-
        restrict(Subst, Rename, CallSubst).


%-----------------------------------------
% compose(Goal, Subst, Call, Exit, NSubst)

compose(Goal, Subst, _, (ExitGoal, ExitSubst), Subst2) :-
        rename(Goal, 1, _, Rename_table, ExitGoal),
        change_some(ExitSubst, Rename_table, Subst, Subst1, Var_changed),
        correct_for_alias(Var_changed, Subst1, Subst2).

change_some([], _, Subst, Subst, _).
change_some([(CallVar, NewVal)|ExitSubst], Rename, Subst, Subst2, Any_change) :-
        lookup_add(ClauseVar, Rename, CallVar),
        replace_subst(ClauseVar, OldVal, NewVal, Subst, Subst1),
        any_change(OldVal, NewVal, Any_change),
        change_some(ExitSubst, Rename, Subst1, Subst2, Any_change).

any_change('@free', '@free', _).
any_change('@free', X, change) :- X \== '@free'.
any_change('@ground', '@ground', _).
any_change('@dont-know', X, change).
```

```
% correct_for_alias(Change, Subst, Subst1)
% changes the value of each free variable to dont-know if the change argument
% indicates that some variable has changed.

correct_for_alias(nochange, Subst, Subst) :- !.
correct_for_alias(change, Subst, Subst1) :-
        correct_for_alias(Subst, Subst1).


%-----------------------------------------
% Primitives for substitutions

% Find_value:
% find the value (free, ground or dont-know) of a whole structure in a
% substitution.

find_value(Var, Subst, Val) :-
        variable(Var),!,
        mapsubst(Subst, Var, Val).
find_value(Atom, Subst, '@ground') :-
        atomic(Atom).
find_value([Head|Tail], Subst, Val) :-
        find_value(Head, Subst, Val1),
        find_value(Tail, Subst, Val2),
        combine_parts_value(Val1, Val2, Val).
find_value(Struct, Subst, Val) :-
        Struct =.. [_|Arg],
        find_value(Arg, Subst, Val).

% Set the value of each variable in a whole structure to Val (except if
% the value of a variable is already known to be ground).

set_value(Var, Val, Subst, Subst1) :-
        variable(Var), !,
        replace_subst(Var, OldVal, NewVal, Subst, Subst1),
        combine_two_values(OldVal, Val, NewVal).
set_value(Atom, _, Subst, Subst) :-
        atomic(Atom).
set_value([Head|Tail], Val, Subst, Subst2) :-
        Val \== '@free',
        set_value(Head, Val, Subst, Subst1),
        set_value(Tail, Val, Subst1, Subst2).
set_value(Struct, Val, Subst, Subst1) :-
        Struct =.. [_|Arg],
        set_value(Arg, Val, Subst, Subst1).

% Replace_value:
% replace the value of each variable in a structure with a new value, but
% check that ground variables remain ground.

replace_value(Var, OVal, NVal, Subst, Subst1) :-
        variable(Var),!,
        replace_subst(Var, OVal, NVal, Subst, Subst1).
replace_value(Atom, '@ground', _, Subst, Subst) :-
        atomic(Atom).
replace_value([Head|Tail], OVal, NVal, Subst, Subst2) :-
        !, replace_value(Head, OVal1, NVal1, Subst, Subst1),
        replace_value(Tail, OVal2, NVal2, Subst1, Subst2),
        combine_parts_value(OVal1, OVal2, OVal),
        combine_two_values(OVal1, NVal, NVal1),
        combine_two_values(OVal2, NVal, NVal2).
```

```
replace_value(Struct, OVal, NVal, Subst, Subst1) :-
        Struct =.. [F|Arg],
        replace_value(Arg, OVal, NVal, Subst, Subst1).

% mapsubst:
% Apply First argument to a variable, finding its value.

mapsubst([], _, '@free').
mapsubst([(Var,Val) | _], Var, Val).
mapsubst([(Other,_) | Subst], Var, Val) :-
        Other \== Var,
        mapsubst(Subst, Var, Val).

% replace_subst:
% replace the value of a variable with a new value. Note that this
% predicate does not check that the change is monotonic.

replace_subst(Var, '@free', NewVal, [], [(Var, NewVal)]).
replace_subst(Var, OldVal, NewVal, [(Var,OldVal)|Subst], [(Var,NewVal)|Subst]).
replace_subst(Var, OldVal, NewVal, [(Y,YVal) |Subst], [(Y,YVal)|Subst1]) :-
        Var \== Y,
        replace_subst(Var, OldVal, NewVal, Subst, Subst1).

% Generating the value of a structure from the value of its parts.

combine_parts_value('@free', '@free', '@dont-know').
combine_parts_value('@free', '@ground', '@dont-know').
combine_parts_value('@free', '@dont-know', '@dont-know').
combine_parts_value('@ground', '@free', '@dont-know').
combine_parts_value('@ground', '@ground', '@ground').
combine_parts_value('@ground', '@dont-know', '@dont-know').
combine_parts_value('@dont-know', '@free', '@dont-know').
combine_parts_value('@dont-know', '@ground', '@dont-know').
combine_parts_value('@dont-know', '@dont-know', '@dont-know').

% combine_two_values:
% Determine what is known about a variable from combining two sources.
% wait declarations inserted to avoid unnecessary backtracking, since this
% predicate is deterministic when the two first arguments are instantiated.

:- wait combine_two_values/3.
combine_two_values(X, Y,Z) :- combine_two_values_hlp(Y,X,Z).

:- wait combine_two_values_hlp/3.
combine_two_values_hlp(X, X, X).
combine_two_values_hlp('@free', '@ground', '@ground').
combine_two_values_hlp('@free', '@dont-know', '@dont-know').
combine_two_values_hlp('@ground', '@free', '@ground').
combine_two_values_hlp('@ground', '@dont-know', '@ground').
combine_two_values_hlp('@dont-know', '@free', '@dont-know').
combine_two_values_hlp('@dont-know', '@ground', '@ground').

variable('@var'('@call',_)).
variable('@var'('@body',_)).

% lookup:
% ensures that Var had a value in Tab, by checking that Val is a non-variable.

lookup(Var, Tab, Val) :-
        lookup_add(Var, Tab, Val),
        \+ var(Val).
```

```
% lookup_add:
% allows a variable to be added to Tab if it is not already in Tab.

lookup_add(Var, [(Var,Val)|_], Val).
lookup_add(Var, [(X,_)|Tab], Val) :-
        \+ X = Var,
        lookup_add(Var, Tab, Val).


% correct_for_alias:
% changes the value of every free variable explicitly represented to dont-know.

correct_for_alias([],[]).
correct_for_alias([(Var,Val)|Subst],[(Var,Val1)|Subst1]) :-
        combine_two_values('@dont-know', Val, Val1),
        correct_for_alias(Subst, Subst1).

%--------------------------------------
% preprocessing of clauses and goals.

preproc_clause(Pred, Args, Body, (Head :- Body)) :-
        preproc_vars(Args, 1, N),
        preproc_vars(Body, N, _),
        Head =.. [Pred|Args].

preproc_vars(Var, New, Next) :-
        preproc_new_or_old(Var, New, Next), !.
preproc_vars(Atom, New, New) :- atomic(Atom).
preproc_vars([Hd|Tl], New, Next) :-
        !,preproc_vars(Hd, New, New1),
        preproc_vars(Tl, New1, Next).
preproc_vars(Struct, New, Next) :-
        Struct =.. [F|Arg],
        preproc_vars(Arg, New, Next).

preproc_goal(PGoal, Goal, Subst) :-
        preproc_goal(PGoal, Goal, [], Subst, 1, _).

preproc_goal(Var, Var, Subst, Subst1, New, Next) :-
        preproc_new_or_old(Var, New, Next),!,
        replace_subst(Var, _, '@free', Subst, Subst1).
preproc_goal(Inst, Var, Subst, Subst1, New, Next) :-
        Inst =..['@i'|_],!,
        preproc_new_or_old(Var, New, Next),
        replace_subst(Var, _, '@dont-know', Subst, Subst1).
preproc_goal('@g', Var, Subst, Subst1, New, Next) :-
        !,preproc_new_or_old(Var, New, Next),
        replace_subst(Var, _, '@ground', Subst, Subst1).
preproc_goal(Atom, Atom, Subst, Subst, New, New) :-
        atomic(Atom),!.
preproc_goal([PHd|PTl], [Hd|Tl], Subst, Subst2, New, Next) :-
        !,preproc_goal(PHd, Hd, Subst, Subst1, New, New1),
        preproc_goal(PTl, Tl, Subst1, Subst2, New1, Next).
preproc_goal(PStruct, Struct, Subst, Subst1, New, Next) :-
        PStruct =.. [F|PArg],
        preproc_goal(PArg, Arg, Subst, Subst1, New, Next),
        Struct =.. [F|Arg].

preproc_new_or_old('@var'('@body',N), N, N1) :- N1 is N + 1.
preproc_new_or_old('@var'('@body',X), N, N) :- X \== N.
```