The Knowledge Engineering Review, Vol 7:2, 1992, 157–171

Trends in applying abstract interpretation

ANDREW BOWLES

Department of Artificial Intelligence, University of Edinburgh, Edinburgh, UK

Abstract

Abstract interpretation is a principled approach to inferring properties of a program's execution by simulating that execution using an interpreter which computes over some abstraction of the program's usual, concrete domain, and which collects the information of interest during the execution. Abstract interpretation has been used as the basis of research in logic and functional programming, particularly in applications concerned with compiler optimizations. However, abstract interpretation has the potential to be used in other applications, such as debugging or verification of programs. In this paper we review the use of abstract interpretation to both compiler optimizations and to other applications, attempting to give a flavour of the kind of information it is possible to infer and some of the issues involved

1 Introduction

There are many situations where it is desirable to know some property of the states of execution a program may reach that is not immediately obvious from the program text. An example of this is information concerning the form of arguments that a procedure may be passed. This information is useful during program debugging—this information ought to match the programmers expectations. An instance of this, specific to Prolog, is knowing that a predicate is always called with its first argument ground, in which case a compiler can save serveral instructions in the code it produces (Mellish, 1987). An obvious way to work out these properties would be to actually run the program and make a note of the states that it gets into. While this is plausible when it is possible to perform the analysis 'by hand', there are some problems in using this approach to construct automatic program analysers:

- To be able to make a strong statement such as 'this argument is always ground' the inputs to the program would need to be carefully chosen. Generally, the set of example inputs would need to be infinite to provide a guarantee of such statements.
- Programs do not always terminate.

A solution to both of these problems is to step back from the details of how a program is executing and consider only the execution in terms of the properties of interest; that is, consider only an abstraction of a program's execution. This is *abstract interpretation*. In doing this abstraction the 'size' of the domain over which the interpreter is executing can be very much reduced, avoiding the above problems. If the usual interpreter executes over the *concrete* domains of Prolog terms, then an abstract interpreter executes over this reduced *abstract* domain of properties of Prolog terms.

The foundation of abstract interpretation are given in the work Cousot and Cousot (1977, 1979). These papers formalize the relationship between the concrete and abstract domains, and formulate correctness conditions for guaranteeing the correctness of the information inferred and the termination of the abstract interpretation. These foundations are applicable to a standard imperative language, but have been reformulated for functional languages (Mycroft, 1981) and for Prolog (Mellish, 1987; Jones & Søndergaard, 1987; Bruynooghe, 1988).

Abstract interpretation has recently been used in a wide variety of applications in logic and functional programming*. In this paper we review some of this work to demonstrate the kind of information inferable using abstract interpretation, and the range of applications that this has been applied to. Until recently, abstract interpretation has been applied mainly to developing optimizing compilers. Whilst this is relevant to those developing knowledge based systems—since performance is always an issue—we here attempt also to highlight the potential that abstract interpretation based analyses have in other areas, particularly in debugging knowledge bases.

The paper is structured as follows. The following section presents a simple example of an abstract interpretation which highlights a number of issues. In the third section we describe some example applications in the field of logic programming compiling. This will give a flavour of the kind of information it is possible to infer using abstract interpretation, as well as the range of applications which have been attempted. In the third section we describe two pieces of work oriented towards debugging programs. Also included is a technical appendix containing a simple formalization of abstract interpretation.

2 Example Prolog abstract interpretation

In this section we present a simple example of an abstract interpretation on Prolog. We demonstrate a possible approach to inferring the 'argument groundness' information mentioned above. Note, however, that this an instance of a general class which are concerned with determining argument properties of procedures—a similar problem in Lisp would be determining whether a function is always called with arguments which are atoms rather than lists.

The problem is to determine which arguments to a predicate, if any, are at run time always ground. Since we are interested only in the groundness of terms we can ignore the particular constants used to make up those terms, and simply record the groundness of each variable in the substitution built by the abstract interpreter. That is we could base an abstract domain on the groundness property of terms. To illustrate this, consider the execution of the following program:

$p(f(A), B)$:- $q(A), C = g(A), r(C), D = g(B), s(D)$.
?- $p(f(a), X)$.

After the successful head match, the variable A is bound to the atom $a/0$. Being an atom, it is ground. This means that the call to the predicate $q/1$ is completely ground, as is the call to $r/1$ because the variable C is constructed out of ground components. The call to $s/1$, on the other hand, involves the variable B which is still uninstantiated, and so this call is not ground.

With an abstract interpreter we can execute this program with regard only to this groundness information. During execution, instead of constructing the usual Prolog terms, two special tokens are used; $\boldsymbol{g}$ representing all ground terms and $\boldsymbol{a}$ representing all Prolog terms. Thus, if in the usual interpreter a variable is bound to any ground term, it would be bound to a $\boldsymbol{g}$ in this abstract interpreter; if the term is not (known to be) ground the variable would be bound to $\boldsymbol{a}$. We call the set of these tokens the *abstract domain*.

The initial query to the above program is written as $p(\boldsymbol{g},\boldsymbol{a})$ since $f(a)$ is a ground term and X is a free variable. The abstract interpreter will contain an abstract version of unification which corresponds to the usual unification. Here the abstract unification attempts to match the call $p(g,a)$ with the head $p(f(A),B)$. Abtract unification is defined as follows:

- Unifying any term with $\boldsymbol{g}$ binds all the variables in the term to $\boldsymbol{g}$.
- Unifying any term with a binds all the variables in the term to $\boldsymbol{a}$ unless already bound to $\boldsymbol{g}$.
- Unifying any variable to a term binds the variable $\boldsymbol{g}$ if the term is ground, otherwise to $\boldsymbol{a}$.
- Constants are unified as in usual unification.

*For a number of papers on the abstract interpretation of declarative languages, see Abramsky and Hankin (1987).

The unification of $p(\boldsymbol{g},\boldsymbol{a})$ and $p(f(A),B)$ results in the substitution $(A/\boldsymbol{g}\ B/\boldsymbol{a})$. The two local variables in the clause C and D are initially free and so are bound to $\boldsymbol{a}$. This gives the initial substitution of the body as $(A/\boldsymbol{g}\ B/\boldsymbol{a}\ C/\boldsymbol{a}\ D/\boldsymbol{a})$.

The first subgoal in the clause calling the predicate $q/1$ is completely ground, as is the call to the predicate $r/1$ since the variable C is unified to a term $g(A)$, where A *is* $\boldsymbol{g}$. The variable D, on the other hand, is unified with $g(B)$, where nothing is known about the groundness of B, hence nothing can be determined about the groundness of D. Hence the groundness of the call to the predicate $s/1$ cannot be determined.

The execution of the abstract interpreter has correctly inferred that, if the orginal call is of the form $p(\boldsymbol{g},\boldsymbol{a})$, all the calls to the predicates $q/1$ and $r/1$ are ground, but the groundness of the call to $s/1$ cannot be determined. Note, that this does *not* imply that the call certainly involves free variables, even though it does when executed on the usual Prolog interpreter. That is, we can only make statements about the program that reflect the particular abstraction used in the abstract interpreter. Here we have chosen an abstraction where one can only say whether a variable is ground or anything.

To illustrate this point further, consider adding an extra clause to the above program:

$p(f(A,B)$:- $q(A), C = g(A), r(C), D = g(B), s(D)$.
$p(h(A), B)$:- $q(B)$.
?- $p(f(a), X)$.

Note that this addition has no effect on the usual Prolog execution because the head matching unification on the new clause will fail. However, it will succeed on the abstract interpreter—because information about which constants make up terms has been lost—leading to a nonground call to the predicate $q/1$. For this program, the abstract interpreter could no longer infer the groundness of all the calls to $q/1$, only the calls in the first clause. Whether this information is still useful depends on the application. There are three points to be made here:

- An abstraction involves a loss of information, and so implies a reduction in the precision of the information an abstract interpreter can infer.
- Despite this, the information is always correct; saying that a variable is bound to $\boldsymbol{a}$ is always correct.
- Similarly, if a variable is bound to a $\boldsymbol{g}$ then we can be sure that the variable is always ground.

This abstract domain is not sufficiently expressive to be able to represent variables bound together, or *aliased*. After unification $X = Y$, where both are initially bound to $\boldsymbol{a}$, they are still both bound to $\boldsymbol{a}$. If subsequently X is bound to $\boldsymbol{g}$ then it is not possible to also bind Y to $\boldsymbol{g}$ and so we cannot infer the groundness of Y. It is still correct to say that Y is bound to $\boldsymbol{a}$ because $\boldsymbol{a}$ represents all Prolog terms, not just the nonground ones. This is another source of imprecision in this particular abstract domain.

Abstract interpretation allows us to solve the two problems we stated at the beginning of this paper as follows:

- Prolog execution is approximated by considering only abstract properties of the calls. Thus a single abstract call can represent a large set of normal calls which would otherwise have to be treated individually.
- Because the set of abstract calls is smaller than that of normal calls, and often is also finite, it is possible to use conventional techniques for handling non-terminating programs (Dietrich, 1987).

3 Example applications

We now take a closer look at some of the applications of abstract interpretation.

3.1 Strictness analysis

Strictness analysis was the first application of abstract interpretation in a declarative language (Mycroft, 1981). Strictness analysis infers information useful for optimizing lazy functional

programs by identifying the arguments of functions which may be passed simply by value, rather than by constructing a closure for the argument. This both avoids the need of building and handling closures, and also opens up the possiblity for increased parallel evaluation. A strict argument is one which is certain to be eventually evaluated—the argument can then be evaluated when constructing the arguments of the function without altering its semantics.

For example, consider the function

$$f(c,x,y) = \textbf{if } c \textbf{ then } x \textbf{ else } y$$

Clearly, the argument c always needs to be evaluated, and so it may be passed strictly. The arguments x and y, on the other hand, may not be evaluated, and so they must be passed lazily.

The information required, therefore, is simply whether each argument for a function is certain to be evaluated. Thus a suitable abstraction is a domain of two values for each argument in a function call, one indicating that the argument is certain to be evaluated, and the other indicating that may not be evaluated.

The abstract interpretation provides a semantics for the primitives in the functional language. In the above case, the if construct must assign a 'lazy' value to both the arguments x and y. On the other hand, the laziness of the argument c depends on the exact form of the argument.

3.2 Reference counting

Implementations of declarative languages make use of sharing of data-structures so that values need not be recomputed, and so space is not consumed with copies of these values. So that the space may be reclaimed, data-structure representations can contain a count of the number of references of abstract interpretation (Hudak, 1986) attempt at compile time to infer values for these reference counts—in particular, whether a data-structure has at most one reference to it. In this case a number of optimizations are possible. For example, if a change has to be made to a data-structure referenced only once it may be made in place, avoiding the need for copying. The optimization allows the functional implementation of quicksort to have linear space complexity.

One difficulty is that reference counts are purely an implementational notion. The usual functional programs contain no notion of references to data-structures (or indeed, any notion of memory management at all.) Thus the first step is to provide a semantics for functional programs which deals explicitly with a store and with references to the data-structures in that store.

Given this semantics, however, a suitable abstract domain is quite simple. Instead of keeping an exact count of the referemces to each data-structure, this number is abstracted to a domain of three tokens: 0 for when there are no references to a data-structure (that is, it is garbage), 1 for when there is exactly one reference, and 'many' for when there are more than one reference.

Thus if an extra reference is made to a data-structure with an abstracted reference count of 1 then this abstracted reference count becomes 'many'. This happens when a data-structure is passed by reference.

If a reference is removed from a data-structure with an abstracted reference count of 1 then new count is 0. This happens when a function whose argument was passed by reference terminates. If, however, a reference is removed from a data-structure with an abstracted reference count of 'many' then this stays the same. Thus if more than one reference is made to a data-structure then the abstraction contains little useful information.

In place updates are possible whenever a data-structure with an abstracted reference count of 1 is copied, and then that reference is lost and changes are subsequently made to the copy. In this situation the copying is unnecessary, and the changes may be made directly to the original.

3.3 Mode inference

A mode of a goal is the call time instantiation pattern of its arguments; that is whether they are free or instantiated*. Abstract interpreters for mode inference attempt to characterize all the possible

*The groundness example we used above can be seen as a cut down mode inference.

calls to each predicate in the program in terms of the instantiation state of their arguments. Abstract domains for mode inference map variables onto a small set of tokens; Debray & Warren, (1988) uses tokens for certainly ground, certainly free, anything and no information. In an attempt to gain precision, Mellish (1987) uses a slightly more complex domain which includes a token for instantiated terms which have free arguments.

Both of these abstract domains lose precision through not being able to represent aliasing between variables; that is variables which have been bound together (see also section 3.5). This leads to a considerable loss of information in some circumstances. For example, take the program

```
p(X,Y) :- q(X,Y), r(X), s(Y).
q(X,X).
r(a).
s(-).
```

The variables X and Y in $p/1$ are aliased by the call to $q/2$. When the call to $r/1$ grounds X it also grounds Y, so the call to $s/1$ is also ground. Using an abstract interpreter based on the above abstract domains, any unification which could result in aliasing between variables must bind these variables to the anything token; this is the only way to ensure that the analysis will produce correct information. If the variables were left bound to the variable token the groundness of X would not be propagated to Y and the call to $s/1$ would appear to involve a free variable. It is possible to add aliasing information to the abstract domain to avoid these problems; this is the approach used by Bruynooghe *et al*, (1987). However, the complexity of this domain is considerably greater. This illustrates the trade-off between designing an abstract domain which infers the information needed and designing an abstract domain which can be efficiently implemented.

3.4 Call/Exit Type Inference

Even though Prolog is a single-type language, it is natural to group certain sets of data-structures together into types. For example, lists form a natural type parameterized on the type of the elements contained. A 'call type' is some specification of the data-structures that can possibly appear in a subgoal's arguments at call time (Mishra, 1984). Since such arguments are not necessarily ground, a call type for a predicate subsumes the mode of that predicate. Analogously, the exit type of a predicate is some specification of the data-structures that can possibly appear in a subgoal's arguments at exit time.

This notion of type is in contrast to the conventional computer science notion of typing as exemplified in the functional language ML (Harper *et al.*, 1986). In terms of Prolog, this corresponds to a type of a predicate specifying some superset of the atoms which are true in the model of the program (Mycroft & O'Keefe, 1984; Zobel, 1987). Consider the standard Prolog predicate *append/3*:

```
append([],L,L).
append([H|L1],L2, [H|L3]) :-
    append(L1, L2, L3).
```

Append/3 is used for joining or breaking up lists, i.e., the type of each argument is a list. The call type of *append/3* in some program might be

append(***list*(*integer*)**, ***list*(*integer*)**, ***var*)**.

and the success type

append(***list*(*integer*)**, ***list*(*integer*)**, ***list*(*integer*)**).

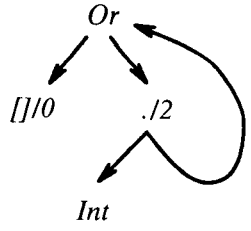
Here ***list*(*integer*)** is the name for the type of lists with integer elements. However, the model for *append/3* includes atoms such as *append*([],*a*,*a*) which follow from the first clause. This atom is not in the call or success type specification, but should appear in the 'conventional' typing for *append/3*.

Type inference for both notions of typing can be obtained by abstract interpretation. However, here we describe only call/success typing.

The two most significant pieces of recent work are the type inferencing schemes of Bansal & Sterling (1990) and Bruynooghe & Janssens (1988). They are both fairly similar in that an abstract substitution is a mapping from variables to type trees. These trees are the standard tree representation of terms supplemented with

- nodes for representing a choice;
- backarcs for representing recursive data-structures.

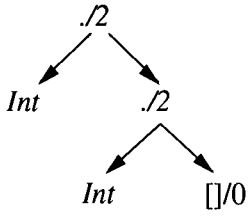
For example, the 'list of integer' type above might appear as



where '*Or*' represents the choice node, and '*Int*' represents all integers.

The main difference between these two domains is the way in which aliasing between values is represented. Bruynooghe does not distinguish between variables in trees—that is, all variables are represented by a special token—and sharing is represented by two separate components specifying certain and possible aliasing between variable nodes. Bansal, on the other hand, can distinguish between variables in trees, and so there is no need for the extra components. This has corresponding gains in the simplicity of the algorithms, and also their proofs, used to implement unification in his interpreter.

The abstract domain is infinite, but with backarcs and 'or' nodes an infinite set of increasing data-structures may be represented in one tree. The primitives in the abstract interpreter must be designed so that backarcs are introduced if a potentially unbounded data-structure appears to be being built. The primitives described by Bruynooghe look out for repeated constants on the branches of the tree being built. That is, if the tree appears as



then the repeated constant *./2* would be detected, and the tree would be replaced by the 'list of integer' tree above. Bansal does not deal with termination, assuming that the programs being analysed always terminate.

3.5 Variable aliasing

Considerable work has been done in trying to infer when subgoals in a clause execution are independent. Two subgoals are independent if they do not have any variables in common. A special case of this is when a subgoal is ground. If they are independent they may be executed

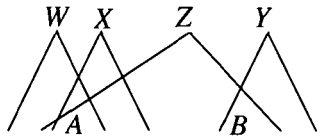


Figure 1 Example substitution where W, X, Y and Z are the local variables and A and B are the only non-local variables

in parallel with no communication between the separate process. For example, take the program

```
contained(empty,-).
contained(node(LT,X,RT),L) :-
    member(X,L). contained(RT,L).
?- contained(T,L).
```

This determines whether all values appearing in the first argument, a binary tree, are members of the second argument, a list. The recursive calls in the second clause may execute in parallel if L is ground and LT and RT are independent. Because variables may be bound together, or aliased, during the program execution this cannot be inferred simply by checking the program text. With abstract interpretation it is possible to infer at each point in the program some idea of which variables are certainly not aliased together and which are certainly ground. This information may be used to minimize the amount of independence checking needed at run time.

The most elegant abstract domain currently formulated for this application is that of Jacob and Langen (1989). Their approach is to characterize each variable in the substitution with the set of local variables—that is, the ones local to the currently executing clause—which contain it. The abstract substitution is the set of these sets. For example, the concrete substitution (W/A X/A Y/B $Z/f(A,B)$), where W, X, Y and Z are the local variables (see Figure 1) is represented as $\{\{W,X,Z\},\{Y,Z\}\}$. The substitution contains the set $\{W,X,Z\}$ because the variable A is contained in W, X and Z, and it contains $\{Y,Z\}$ because the variable B is contained in Y and Z. If there were a third non-local variable C contained only in Z the abstract substitution would be $\{\{W,X,Z\},\{Y,Z\},\{Z\}\}$.

This captures and propagates groundness and independence information with great precision. In the above, X and Y are independent because they do not appear in the same set, grounding either X or W grounds the other, and grounding Z grounds W, X, and Y. This can be inferred from the set representation.

3.6 Detecting shared data-structures

Another way of improving code generated by a Prolog compiler is to ensure that it is capable of reusing space that can be shown at compile time to be redundant; this is known as 'compile time garbage collection'. Data-structures may then be reused during the computation, avoiding the cost of creating new space and effectively giving Prolog programs the efficiency of destructive assignment. The application is the Prolog equivalent of the reference counting application for functional languages (see section 3.2).

A data-structure which is bound to a variable may be reused if that variable is not used later on in the clause or is needed on backtracking, provided that the data-structure does not share with any other data-structures which are used later on. Sharing is very much an implementational notion, treating the pointers that the Prolog system builds as a graph; two variables share if their graphs contain a common subgraph. A related notion is containment; a variable contains another if its graph completely contains the other variable's graph. Thus the information required is a partitioning of the local variables into sets between which there is no sharing at points just before each subgoal in a clause. This can be seen as a data-structure equivalent of variable aliasing.

Bruynooghe *et al.* 1987 have sketched an abstract domain where an abstract substitution consists of two sets of pairs of variables*. One of these sets represents the pairs of variables X, Y which possibly share, each of which are written as $A\ AL\ Y$. The other represents the pairs of variables of which the first one is certainly contained by the other, which is written as $X\ PART\ Y$. Note that the AL relation is reflexive and symmetric, but the $PART$ relation is only reflexive. Also, if $X\ PART\ Y$ then $X\ AL\ Y$. We write an abstract substitution as a set of these pairs, but the extra pairs implied by these reflexive and symmetric properties are not given explicitly.

For example, consider executing the following query with an initial substitution where all variables are known not to share, that is {}

$$?\text{-}\ X = f(A,A),\ X = f(B,C).$$

After execution of the first unification, the substitution is

$$\{A\ PART\ X\}$$

since A's graph is certainly contained in the graph of X. After the second unification the substitution is

$$\{X\ AL\ B,\ X\ AL\ C,\ A\ AL\ B,\ A\ AL\ C,\ B\ AL\ C,\ A\ PART\ X,\ B\ PART\ X\}$$

that is, all the variables possibly share with each other. Note that because the actual constants used to make up a variable's graph are not represented in the abstract substitution, it is impossible for abstract unification to determine where a variable appears in a graph. This partly leads to the poor quality information derived above.

Another cause of this is that sharing inside a variable's value cannot be represented because the same variable cannot appear on both sides of a pair. This means that a data-structure which is a graph cannot be distinguished from one that is a tree. This forces a worst case assumption about sharing in unification which leads to a considerable loss of precision. For example, take the call

$$?\text{-}\ x = [H|T]$$

If X is known to be a tree, then H and T do not share afterwards. If X is a graph, H and T will in general share.

This representation problem is solved by Mulkers *et al.* (1990) by merging a type inferencing component with the possible sharing set. Here the set is composed of pairs of nodes in the type trees rather than pairs of variables. This abstract domain is essentially a merge of the type inferencing domain of Bruynooghe and the simple data-structure sharing domain above.

3.7 Program specialization

Program specialization involves constructing a new program which can follow only certain computational paths of the original one, but not others. The work of Gallagher & Bruynooghe (1990 a) has developed techniques for specializing both Prolog and Flat Concurrent Prolog programs. Abstract interpretation plays a part by its ability to characterize all possible computational paths which a program may follow. This characterization is of course approximate, but any parts of the original program which contribute nothing to this characterization may be safely removed in the new program.

An elegant example of a program specialization scheme is given by Gallagher & Bruynooghe (1990 b). They describe an abstract interpretation scheme which characterizes the most general call made to each predicate in a program. With this information it is possible to remove redundant

*It contains somewhat more information which is specific to their application, but which we ignore at present.

constructor functions. For example, in the standard flatten predicate using a constructor function *-/2* to join together the two different list arguments

```
flatten(L, FL) :- flatten(L, FL-[]).

flatten([H|T], L-L0) :-
    flatten(H, L-L1),
    flatten(T, L1-L0).
flatten([],L-L),
flatten(X, [X|L]-L) :-
    X\ = [], X\ = [_|_].
```

The most general call to *flatten/2* is *flatten(A,B-C)*; that is, the second argument of every call to *flatten/2* contains the *-/2* constructor function. Knowing this it is possible to specialize *flatten/2* by replacing all occurrences of *-/2* with a pair of separate arguments resulting in considerably faster code

```
flatten(L, FL) :- flatten(L, FL, []).

flatten([H|T], L, L0) :-
    flatten(H, L, L1),
    flatten(T, L1, L0).
flatten([], L, L),
flatten(X, [X|L], L) :-
    X\ = [], X\ = [_|_].
```

4 More applications

Historically, compiler optimizing was the first area of application for abstract interpretation. In the previous section we listed a variety of abstract interpretation research in this area. In this section we describe two more recent applications more concerned with debugging programs. The techniques used are exactly the same as above. This work illustrates the potential for using abstract interpretation in other stages in the software lifecycle.

4.1 Non-termination analysis

The ability to detect non-terminating programs is a major requirement of any program debugging or verifying system. The approachs taken to detect potential non-terminations may be put in two categories:

1. *Dynamic*: here infinite loops are detected at run time by comparing successive recursive calls (van Gelder, 1987). Although this can be made quite efficient, it still imposes a considerable overhead on execution. This checking is necessary for every execution of the program.
2. *Static*: here potentially looping programs are detecting by checking the text of the code (Wong & Shyamashundar, 1990). Although these impose no run time overhead, they are not in general complete—that is, they do not detect all possible non-terminating programs.

The abstract interpretation approach to non-termination is static, and so imposes no run time overhead, but promises to be more complete than other static approaches since abstract interpretation is capable of modeling run time behaviour.

Here we summarize the work of Verschaetse & de Schreye (1991). A proof of termination usually consists of determining a well-founded ordering on the successive calls in a recursion which strictly decreases with each call. Less formally, we want some measure of the 'size' of each call and some proof that this size decreases. Consider the following program

```
quicksort(L, S) :- qsort(L, S,[]).

qsort([H|T], S, S0) :-
    partition(H, T, Ls, Gs),
    qsort(Ls, S, [H|S1]),
    qsort(Gs, S1, S0).
qsort([], S, S).

partition(_, [], [], []).
partition(X, [H|T], [H|Ls], Gs) :-
    X @ ≥ H,
    partition(X, T, Ls, Gs).
partition(X, [H|T], Ls, [H|Gs] ;-
    X @ < H,
    partition(X, T, Ls, Gs).
```

To prove that this program terminates it is necessary to demonstrate that both recursive calls to *qsort/3* are 'smaller' than the original. This involves reasoning about the relationship between the size of the arguments of *partition/4*. To determine these relationships the program may be transformed into an equivalent abstract form which compute over the sizes of the data-structures. This is an application of abstract interpretation. The abstraction is to regard a program in terms of some constraint between the 'size' of its arguments.

A measure of the size of arguments which is appropriate here is the length of the lists. The abstract form of the *partition/4* predicate is

```
patition(0, 0, 0, 0).
partition(X, T + 1, Ls + 1, Gs) :-
    partition(X, T, Ls, Gs).
partition(X, T + 1, Ls, Gs + 1) :-
    partition(X, T, Ls, Gs).
```

From this it is possible to determine that the size of both output arguments is certainly smaller than the input, and this information is sufficient to prove the termination of *quicksort/2*.

This is quite straightforward because we have a convenient measure of the size of the arguments. In general, it is only possible to arrive at this measure after performing some type analysis on the program. Such an analysis is of course another application of abstract interpretation.

4.2 Student program debugging

We have developed an approach to debugging students programs (Bowles, 1991) which made use of a notion of programming technique—common code patterns systematically used by Prolog programmers (Brna *et al.*, 1990). For example, the standard quick reverse list predicate illustrates a class of techniques called *accumulator pairs*

```
qrev([], R, R).
qrev([H|T], R0, R) :-
    qrev(T, [H|R0], R).
?-qrev([a,b,c], [], X).
```

Another common technique is that of difference structures. Difference lists are a subclass of this technique where the type of structure being 'differenced' is a list—a difference list is used in the *qsort/3* predicate in the previous section.

The approach is based on that of Looi (1988). Here a student's program is compared with a number correct solutions to the problem each using a different algorithm. These solutions are stored in a canonical form, which allows us to ignore details such as the exact names used in the

program, and which specifies expected mode and type information and the form of recursive and base cases. The system matches the student's program against each correct solution, and determines some score as to how good the match is. The algorithm of the solution which has the best score is treated as the algorithm the student intended to use. The correct solution can be used as a source of code for suggesting bug fixes if necessary.

We enhance this approach by incorporating explicitly a notion of programming techniques into the canonical representation for correct solutions. Thus a correct solution program is described in terms of the techniques used in the program. For example, the representation for the quick reverse predicate given above specifies that there should be an accumulator pair in two of its arguments. The use of techniques allows the implementation of many algorithms to be described much more accurately.

For this approach to work, however, we need to be able to detect occurrences of techniques in the student's code. We define techniques using patterns of data-flow between the variables in clauses. Abstract interpretation is used to analyse the data-flow of the student's code. This information can then be used to determine the techniques being used. The abstract domains used are similar to the domains used for the compile-time garbage collection application (see section 3.6)

5 Summary and conclusions

Abstract interpretation is a principled approach to inferring properties of a program's execution by simulating that execution using an interpreter which computes over some abstraction of the program's usual, concrete, domain and which collects the information of interest during the execution. An abstraction is used so that

- the properties of interest are more explicit;
- a whole class of queries may be specified by one abstract predicate call;
- and to ensure that, unlike the concrete interpreter, the abstract interpreter terminates for all possible programs and queries.

The cost of abstracting is precision; an abstract interpreter cannot be expected to exhibit the exact behaviour of the normal interpreter, only to approximate it. Note that the properties inferred can be correct without being precise. For example, for some program it might be possible to guarantee that some property holds, whereas for other programs it may only be possible to infer that this property *may* hold. An extreme case would be a *don't know* result which is completely correct but totally imprecise. The possibility of inferring imprecise information should not be remarkable, since many properties of program states will not be decidable.

Abstract interpretation has recently been used in a wide variety of applications in logic and functional programming. The work we have reviewed includes

- Strictness analysis (Mycroft, 1981).
- Reference counting (Hudak, 1986).
- Mode inference (Codish *et al.*, 1988; Mellish, 1987; Debray & Warren, 1988; Kanamori; *et al.*, 1989).
- Type inference (Bruynooghe & Janssens, 1988; Mulkers *et al.*, 1990).
- Occur check reduction (Søndergaard, 1988).
- Shared variables (Jacobs & Langen, 1989; Muthukumor & Hermenegildo, 1989).
- Shared data-structures (Bruynooghe *et al.*, 1987; Mulkers *et al.*, 1990)
- Program specialization (Gallagher *et al.*, 1988; Gallagher & Bruynooghe, 1990 a,b).
- Program termination (Verschaetse & de Schreye, 1991).
- Student program debugging (Bowles, 1991).

We have attempted to illustrate the broad range of applications that abstract interpretation has been applied to. Despite the bulk of these being oriented towards compiler optimizations, abstract

interpretation has the potential for being applied in other areas. We have highlighted work in debugging Prolog programs, but the techniques used are transferable to other applications and to other languages.

References

Abramsky, S and Hankin, C, eds., 1987, *Abstract Interpretation of Declarative Languages* Ellis Horwood.

Bansal, A and Sterling, L, 1990, "An abstract interpretation scheme for logic programs based on type expressions," *New Generat. Comput.* **7** 273–324.

Bowles, A, 1991, *Detecting Prolog Programming Techniques Using Abstract Interpretation* PhD thesis, University of Edinburgh.

Brna, P, Bundy, A, Dodd, T, Eisenstadt, M, Looi, CK, Pain, H, Robertson, D, Smith, B and van Someren, M, 1990, "Prolog programming techniques," *Instructional Sci.* **19**(4/5),

Bruynooghe, M, 1988, "*A practical framework for the abstract interpretation of logic programs.* (To appear in *Journal of Logic Programming*)

Bruynooghe, M and Janssens, G, 1988, "An instance of abstract interpretation integrating type and mode inferencing," In: *Fifth Symposium and Conference on Logic Programming* pp 669–683, Seattle, WA.

Bruynooghe, M, Janssens, G, Callebaut, A and Demoen, B, 1987, "Abstract interpretation: towards the global optimization of Prolog programs," In: *Logic Programming Symposium*, pp 192–204, San Francisco, CA.

Codish, M, Gallagher, J and Shapiro, E, 1988, "Using safe approximations of fixed points for analysis of logic programs," In: H Abramson and M Rogers, editors, *Proceedings of the First Workshop on Meta-Programming in Logic-Programming*, pp 233–261, Bristol, England.

Cousot, P and Cousot, R, 1977, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," In: *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages,* pp 238–252.

Cousot, P and Cousot, R, 1979, "Systematic design of program analysis frameworks," In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pp 269–282.

Debray, S and Warren, DS, 1988, "Automatic mode inference for logic programs," *Journal of Logic Programming* **5**(3) 207–230.

Dietrich, SW, 1987, "Extension tables: memo relations in logic programming," In: *Logic Programming Symposium* , pp 264–272, San Francisco, CA.

Gallagher, J and Bruynooghe, M, 1990 a, "The derivation of an algorithm for program specialization," In: DHD Warren and P Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pp 732–746, Jerusalem, Israel.

Gallagher, J and Bruynooghe, M, 1990, "Some low-level source transformations for logic programs," In: M Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic*, pp 229–244, Leuven, Belgium.

Gallagher, J, Codish, M and Shapiro, E, 1988, "Specialization of Prolog and FCP programs using abstract interpretations," *New Generat Computing*, **6**(2/3) 159–186.

Harper, R, MacQueen, D and Milner, R, 1986, "*Standard ML*, Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh.

Hudak, P, 1986, "A semantic model of reference counting and its abstraction," In: *ACM Conference on Lisp and Functional Programming*, pp 351–363, Cambridge, MA.

Jacobs, D and Langen, A, 1989, "Accurate and efficient approximation of variable aliasing in logic programs," In: E Lusk and R Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pp 154–165, Cleveland, OH.

Jones, ND and Søndergaard, 1987, "A semantics-based framework for the abstract interpretation of Prolog," In: S Abramsky and C Hankin, editors, *Abstract Interpretation of Declarative Languages*, pp 123–142, Ellis Horwood.

Kanamori, T, Kawamura, T and Maeji, M, 1989, "*Logic program analysis by abstract hybrid interpretation*, TR 485, ICOT.

Looi, CK, 1988, "*Automatic Program Analysis in a Prolog Intelligent Technical System* PhD thesis, University of Edinburgh.

Mellish, C, 1985, "Some global optimizations for a Prolog compiler," *Journal of Logic Programming*, **2**(1) 43–66.

Mellish, C, 1987, "Abstract interpretation of Prolog programs," In: S, Abramsky and C, Hankin, editors, *Abstract Interpretation of Declarative Languages*, pp 181–198, Ellis Horwood.

Mishra, P, 1984, "Towards a theory of types in Prolog," In: *Logic Programming Symposium*, pp 289–298, Atlantic City, NJ.

Mulkers, A, Winsborough, W and Bruynooghe, M, 1990, "Analysis of shared structures for compile-time garbage collection in logic programs," In: DHD Warren and P Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming* pp 747–762, Jerusalem, Israel.

Muthukumar, K and Hermenegildo, M, 1989, "Determination of variable dependence information through abstract interpretation," In: E Lusk and R Overbeek, editors, *Proceedings of the North American Conference on Logic Programming* pp 166–185, Cleveland, OH.

Mycroft, A, 1981, *Abstract Interpretation and Optimizing Transformations* PhD thesis, University of Edinburgh.

Mycroft, A and O'Keefe, R, 1984, "A polymorphic type system for Prolog," *Artificial Intelligence*, **23** 296–307.

Søndergaard, 1986, "An application of abstract interpretation of logic programs: Occur check reduction," In: B Robinet and R Wilhelm, editors, *ESOP 86 European Symposium on Programming*, pp 327–338, Saarbrucken, Germany.

Tamaki, H and Sato, T, 1986, "Old resolution with tabulation," In: E, Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming* pp 84–98, London, England.

van Gelder, A, 1987, "Efficient loop detection in Prolog using the tortoise-and-hare techniques," *Journal of Logic Programming* **4** 23–31.

Verschaetse, K and de Schreye, D, 1991, "Deriving termination proofs for logic programs, using abstract procedures," In: K Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pp 301–315, Paris, France.

Wang, B and Shyamasundar, R, 1990, "Towards a characterization of termination in logic programs," In: *Proceedings of the International workshop PLILP'90*, Volume 456 of *Lecture Notes in Computer Science*, Springer-Verlag, 204–221.

Zobel, J, 1987, "Derivation of polymorphic types for Prolog programs," In: J-L Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming* pp 817–838, Melbourne, Australia.

Appendix A: Formalization of Prolog abstract interpretation

Here we present a simple formalization of Prolog abstract interpretation which highlights the correctness and termination issues involved. Although specific to Prolog, it can be readily adapted to other languages.

We require a concrete semantic function for Prolog which will be abstracted to form an abstract semantic function. Such a concrete semantic function could have the form $Prolog_P : C \rightarrow C$ where P is a program and each member of C is a set of variable substitutions*. This roughly corresponds to a conventional concrete semantic function for Prolog; *Prolog* is a mapping from a set of substitutions (one corresponding to each of the queries which will be run on the program†) to a set of substitutions (one for each successful resolution of those queries).

The purpose of abstract interpretation, however, is to infer some property of the execution of a program; the conventional variable substitution output of the program is not usually of interest. Rather, it is the details of the execution of the program that is of interest. Accordingly, the concrete interpreter *Prolog* should provide a description of the states reached during the execution of P. Depending on the application, there is plenty of scope for deciding the exact nature of an execution state. For example, with a semantic function based on SLD resolution an execution state would be the **and/or** tree constructed so far, or, for a particular Prolog implementation, the image of the executing Prolog system. Any interesting property of the execution of a program, such as the groundness of the variables or the height of the recursion stack, should be inferable from the execution state. A semantic function *Prolog* of this kind is called a *static* or *collecting* semantics (Cousot & Cousot, 1977; Mycroft, 1981), since it characterizes a program by collecting information during its execution.

We regard C as a domain of sets of execution *states*. To extend the SLD example further, the input to *Prolog* would be a set of **and/or** trees which are empty apart from the root (the entry predicate of the program) annotated with a substitution.

*We usually drop the P argument.

†Without loss of generality we assume the program has a single entry point.

C inherits the set inclusion ordering of powersets; we call this ordering The least element is the empty set of execution states and the greatest element is the set of all possible execution states.

The *concrete* domain C is abstracted to give the *abstract* domain A. This will be a homomorphic, but smaller, image of C. Since we are interested in inferring properties of execution states, an element in A will be a description of some execution states in terms of those properties. Intuitively, such an element—which we call an abstract execution state—represents the set of all concrete execution states which have those properties. A also has an ordering, which is called $\subseteq_A$. An abstract execution state should be greater in $\subseteq_A$ than another if it represents a larger set of concrete execution states.

The homomorphism between the two domains is formalized by defining two mapping functions between them: abstraction $abs{:}C \rightarrow A$ and concretization $conc{:}A \rightarrow C$. These functions should satisfy the following:

$$\begin{array}{lll} \forall c,d \in C, \quad \forall a,b \in A & & \\ \qquad c \subseteq_C d & \Rightarrow & abs(c) \subseteq_A abs(d) \\ \qquad c \subseteq_A b & \Rightarrow & conc(a) \subseteq_C conc(b) \\ \qquad\qquad c & \subseteq_C & conc(abs(c)) \\ \qquad\qquad a & =_A & abs(conc(a)) \end{array}$$

The first two conditions state that *abs* and *conc* are monotonic.

To complete the abstract interpreter an abstract semantic function *AbsProlog* is needed. This will compute a result corresponding to the result of *Prolog* in the abstract domain $AbsProlog\ p{:}A \rightarrow A$. Figure 2 illustrates the relationship between C and A.

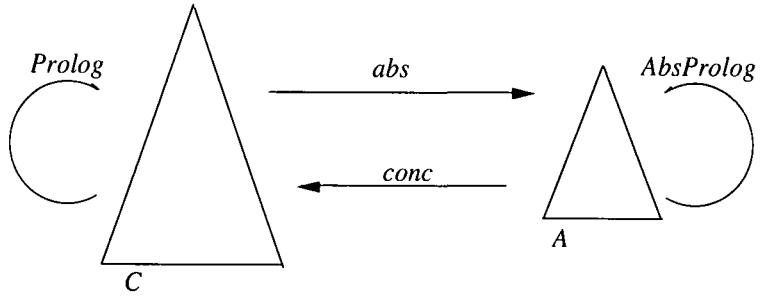


Figure 2 Relationship between C and A

There are two important requirements that an abstract interpretation must meet to form the basis of automatic program analysers: correctness and termination.

A.1 Correctness

The information an abstract interpreter infers should always be correct*, though it may not always be precise. Correctness can be guaranteed by placing some conditions on the abstract interpretation.

Informally, the correctness condition states that if the abstract interpreter infers some description of the possible executions of a program and a set of queries then all executions of the concrete interpreter with that program and set of queries should be included in that description, i.e.,

$$\forall c \in C, \quad Prolog(c) \subseteq_C conc(AbsProlog(abs(c)))$$

It is necessary to demonstrate that this holds for all programs and queries.

Note that if the abstract interpreter always returns some top element in $\subseteq_A$ which represents all possible execution states then the above condition will always hold. However, no useful properties

*Correctness is sometimes called soundness or safety.

can be inferred from this top element; that is, the result is completely correct but totally imprecise. One of the aims in designing as abstract domain is to ensure that it can satisfy the above condition and produce an output which is as small (with respect to $\subseteq_A$), and therefore as precise, as possible.

A.2 Termination

To be automatic, a program analyser based on abstract interpretation must guarantee that any execution will terminate. Semantics for recursive programs usually involve some iterative, often bottom-up, fixpoint characterization of the function over some partially ordered set. For automatic abstract interpretation it is necessary to place some restrictions on the abstract domain A so that this characterization can be finitely computed.

An obvious restriction is to make A finite; any fixpoint computation over a finite set is guaranteed to terminate. This is the solution most researchers have adopted but it is, in general, too strong a restriction.

Computations over infinite sets can be made to terminate provided that the set does not contain an indefinitely increasing chain (that is a sequence $a \subseteq_A b \subseteq_A c \ldots$ where the elements $a,b,c \ldots$ are distinct). An iterative bottom-up computation involves stepping along some chain. Removing all infinite chains will ensure that these computations terminate. This is the termination condition used by Bruynooghe (1988) in his Prolog abstract interpretation framework.

There is an alternative way of viewing this restriction. Infinite chains may be permitted provided that in practice no iteration tries to go up them. That is, the subset of A used in a particular analysis should conform to this restriction, even if A does not. Consider an abstract domain characterizing lists as L_n which represents all lists with length n or less. Thus the infinite chain $L_0 \subseteq_A L_1 \subseteq_A L_2 \ldots$ occurs in A . An iteration along this chain may be terminated by adding a token L, which represents all lists, into the abstract domain, provided the iteration can be made to 'jump' straight to this general description of lists. This approach gains some flexibility in deciding at which point this generalization should occur*.

*This generalization mechanism is used in OLDT resolution to ensure completeness of Prolog (Tamaki & Sato, 1986).