THE NU-PROLOG DEBUGGING ENVIRONMENT

Lee Naish
Philip W. Dart
Justin Zobel

Technical Report 88/31
Department of Computer Science
University of Melbourne

ABSTRACT

One of the advantages of logic programming is that it is easy to reason about what is computed by a program (the declarative semantics) independently of how it is computed (the procedural semantics). As well as being useful for writing, transforming and understanding programs, these two alternative viewpoints are useful for testing and debugging programs. By taking full advantage of declarative and procedural semantics, we anticipate that very powerful programming environments can be constructed.

In this paper we describe the NU-Prolog Debugging Environment (NUDE) which is currently under development. NUDE has three main components. First, several static program analysis tools are used for detecting program errors such as simple typing mistakes, 'type' errors, infinite loops and floundering. Second, dynamic tools are provided for diagnosing incorrect behaviour of a query at run time. There are tools for diagnosing wrong answers, missing answers, infinite loops and floundering. Third, there is a mechanism for automatically testing programs to detect queries with wrong or missing answers. The three components are integrated into the NU-Prolog system, supporting a rapid test-debug-recompile cycle.

1. INTRODUCTION

One of the advantages of logic programming is that it is easy to reason about what is computed independently of how it is computed. That is to say, programs have a simple ‘declarative’ semantics as well as a ‘procedural’ semantics. Programs can be understood and, to a lesser extent, written using either or both viewpoints. These two ways of viewing a logic program are complementary. Reasoning about (partial) correctness and completeness is best done at the declarative level whereas reasoning about termination and efficiency requires an procedural view. The declarative semantics is more high level and has a very well developed theory, so the benefits of using it are considerable.

The two views are also complementary when testing and debugging logic programs. The topic of this paper is a system called NUDE (NU-Prolog Debugging Environment), which exploits the declarative and procedural semantics more fully than previous systems and is designed to make use of and support the unique features of NU-Prolog[1] . It consists of several testing and debugging tools which have been integrated into one system, running under NU-Prolog. Testing programs and finding bugs which cause incorrectness or incompleteness can be done with several tools based on the declarative semantics. Flexibility of input-output modes is used widely, and interaction with the user concerning the intended behaviour of the program is simplified. The tools used for finding bugs which cause loops or abnormal termination are based on the procedural semantics.

2. SEMANTICS OF NU-PROLOG

The NU-Prolog manual[1] gives a full description of the syntax, declarative semantics and procedural semantics of NU-Prolog. The syntax is Edinburgh Prolog syntax with some additions based on ‘extended syntax’[2] . These include constructs for implication (for example, `Goal1 => Goal2`), if-then-else, aggregate predicates which examine all solutions to a goal (for example `max(Term, Goal, MaxTerm)`), soundly implemented negation and logical quantification (`all Vars Goal` and `some Vars Goal`). Logical quantification, restricts the scope of variables which occur syntactically in the term `Vars`, to the formula`Goal`. If the same variable name occurs in distinct scopes it refers to distinct variables.

The declarative semantics of NU-Prolog is given by interpreting constructs of the language as formulas of first order logic. A detailed treatment of this subject is given in [3] . Many built-in predicates of NU-Prolog (such as cut) are considered non-logical, and predicates which call them are not assigned any declarative semantics. There are advantages in avoiding non-logical code where possible, especially before a program has been fully debugged.

The default procedural semantics is the same as other Prolog systems: NU-Prolog normally selects the leftmost subgoal for evaluation first. However, NU-Prolog is able to ‘delay’ the execution subgoals until they are further instantiated, when they are ‘woken’. Many system predicates delay until they have a finite number of solutions and the implementation of negation guarantees soundness by delaying until global variables are ground. ‘when’ declarations may also be used to make calls to user-defined predicates delay until some arguments or parts of arguments are non-variable or ground. A goal has ‘floundered’ if it has terminated with some subgoals delayed but insufficiently instantiated to be woken.

Procedural semantics are defined for all the non-standard constructs, such as implication and quantifiers, as well the standard constructs. However, this procedural semantics can be overridden if a predicate is declared ‘pure’. In this case, the procedural semantics is not defined, allowing the NU-Prolog compiler to optimise the code in any way consistent with the declarative semantics. Pure declarations should not be used in predicates containing non-logical code.

By default, NU-Prolog predicates are ‘static’. That is, they can be only changed by redefining the whole predicate, (for example, with `consult`). Procedures can also be declared ‘dynamic’. Dynamic predicates can be changed by adding or deleting individual clauses (for example, with `assert`).

3. EXISTING DEBUGGING TOOLS

The most common debugging tools for Prolog are procedural debuggers based on the ‘four port’ model of Prolog execution[4] . In this model, the execution of procedures is characterised by being in one of four states: called, exited (succeeded), redone (backtracked into) or failed. Procedural debuggers print information when certain user-specified procedures enter one of the four port model states. In addition, the user can affect the execution in various ways, for example, initiating backtracking or restarting the execution from a particular call. Systems often allow all calls be traced but usually it is best to be more selective. Most debugging is done by only tracing calls to those procedures on which the user has placed ‘spy points’. Procedural debuggers have been extended in various ways. Probably the most advanced system is the Transparent Prolog Machine[5] . This system represents the Prolog search and/or tree graphically, displaying information about procedure invocations, matches with clause heads, bindings to variables, et cetera.

There have also been several methods, proposed and implemented, for detecting possible errors by static analysis at compile time. The simplest of these are the ‘style checkers’, which detect such things as variable names which don’t start with ‘_’ (underscore) but occur uniquely in a clause. These programs can find many simple typing mistakes and some more serious errors. The more complex static analysis methods use declarations supplied by the programmer and analyse such things as input-output modes of procedures and what ‘types’ procedures have6 7 [8] .

Another kind of Prolog debugger which is gaining popularity is the ‘declarative’ (or ‘algorithmic’) debugger[9, 10, 11, 12, 13] . These debuggers are based on the declarative semantics of Prolog and diagnose wrong and missing answers. They ask the programmer what the correct behaviour is for certain calls, for example, whether they should fail, or what the output should be. The cause of incorrect behaviour is traced to a single incorrect clause or procedure.

4. OVERVIEW OF NUDE

NUDE has three main components. The first consists of several static program analysis tools, the second of several dynamic tools to diagnose incorrect run-time behaviour, and the third is a mechanism for automatically testing a program. Prior to the implementation of NUDE, several of these tools existed independently, each with their own representation of programs. The first step in integrating the tools was to design a common way to access the program being debugged. An abstract data type was devised which gave the required functionality. Procedures were provided for: modifying the program, for example, consulting files, deleting a procedure and adding individual clauses; accessing parts of the program, such as details of clauses and declarations for given procedures; extracting program structure, such as what procedures are defined in what files and what procedures directly or indirectly call what other procedures; and miscellaneous tasks, for example printing clauses with the correct variable names, and listing procedures. Information which is relatively expensive to calculate, such as the call graph, is computed on demand then saved until the relevant parts of the program are changed. When available, the compiled code for procedures is loaded as well as the source. This is useful for manual testing and several debugging tools which run some code in the normal way.

The static program analysis tools NUDE provides detect simple errors, such as misspelt variable and procedure names and missing arguments, as well as more subtle coding errors, for example, errors which result in programs always failing. These tools are also provided as a separate non-interactive style checker called *nit*[1] . *nit* can be used in the same way as other style checkers but, because of the extra analysis, more subtle errors can be discovered. Within NUDE, the static checks can be run interactively on parts of the program where it is known there is a bug, rather than being applied to the whole program or a whole file. The static checks can be invoked in the following ways:

```
?- nit.
?- nit Goal.
?- nit(Pred, Arity).
```

The first method performs the static checks on the whole program. The second method performs static checks on all procedures directly or indirectly called by the goal, and the goal itself. The third method checks a single procedure.

The second component of NUDE consists of several dynamic tools for diagnosing incorrect behaviour of the program at run time. This debugging is symptom directed: a goal is executed, and if it returns an incorrect answer, misses a correct answer, (apparently) doesn't terminate or flounders, one of four tools is used to diagnose the bug. They are invoked as follows:

```
?- wrong Goal.
?- miss Goal.
?- loop Goal.
?- flounder Goal.
```

In each case, relevant static checks are optionally performed on the procedures called by the goal. If this does not uncover the bug, the goal can be interpreted and the execution analysed. In the case of incorrect and missing answers, a declarative debugger is invoked and the user is asked questions related to what answers should be returned by various parts of the computation. A database containing this information is built up. This can be saved in and restored from files, so it is not necessary to build it up from scratch in each debugging session. There are also facilities to correct mistakes in this database, which can occur if the user answers a question incorrectly. The dynamic tools generally isolate the bug to a single clause or procedure. The user can then edit the file containing the procedure definition. The modified parts of the file are then recompiled and reloaded into NUDE so debugging can continue with the new definitions.

The third component of NUDE is a mechanism for automatically testing the program. It is invoked as follows:

```
?- test Goal.
?- test(Pred, Arity).
```

The second form is equivalent to the first form, where the goal is a single call to the procedure with all arguments variables. Wrong and missing answers to a goal can be detected automatically by running instances of the goal which are in the database containing the desired behaviour of goals. If any instance incorrectly succeeds or fails, the appropriate dynamic tool is called to diagnose the error. Further wrong answers can be detected by running the goal and asking the user if each answer is correct. If the computation eventually fails, missing answers can also be detected in this way. When testing using these mechanisms leads to a bug being isolated and the program modified, the same goal can be retested very simply. Because the database grows continually, the ability to detect and isolate bugs improves with each iteration.

The three components are integrated into the NU-Prolog system. The various tools interface with each other and parts of the NU-Prolog system such as spy points.

5. STATIC CHECKS

A variety of checks can be applied to programs in the absence of any goal. Some of these checks analyse the program by treating the body of each clause as a goal, while others look for inconsistencies within each clause. Briefly, the static checks applied in NU-Prolog are: type checking, loop detection, floundering analysis (all discussed in detail later), checking for inconsistencies other than type errors, and a stratifiability test.

Some of these static checks relate to programming style. For example, the existence of two procedures or functors with the same name and different arity (number of arguments) generates a warning; this is particularly useful if both arities are large. Similarly, the programmer is warned if two variables in the same clause have near-identical names; in the current implementation, 'near-identical' means that the names are identical if case and any underscores are ignored. NUDE also checks the convention that a variable name begins with an underscore if and only if the variable only occurs once in a clause. Since NU-Prolog includes quantifiers to define scope, the same variable name in the same clause can refer to different variables; if this occurs, a warning is printed. All of these errors could occur in a 'correct' program, but could well arise from typographical mistakes, and should be pointed out to the user. Non-observance of these conventions can detract from the clarity of a program.

Other checks relate to program correctness. The programmer is warned if a procedure is declared but not used, or is used but not declared. Also, if a procedure is modified but not declared 'dynamic', or contains non-logical features but is declared 'pure', the error is reported; the semantics of procedures containing such errors is not defined. Similarly, if a clause contains a variable which is used within a negation but never used in subgoals which could instantiate that variable to a ground term, execution using that clause will always flounder, and so the user is warned. *nit* can also detect when a program is not stratifiable, that is, the program contains a recursive call inside `not` or some other negative construct. Unstratifiable programs are often incomplete.

5.1. Infinite loops

In conventional Prolog systems, where sub-goals are always executed from left to right, the termination of a call to a procedure generally depends on which arguments are instantiated. Therefore, mode analysis is needed for useful termination analysis. Consider the following example:

```
        % append two lists together to get a third
append([], A, A).
append(H.A, B, H.C) :- append(A, B, C).

        % append three lists together
append3(A, B, C, ABC) :-
        append(B, C, BC),
        append(A, BC, ABC).
```

If `append3` is always called with its first two arguments instantiated to lists, `append` will always be called with its first instantiated to a list. In this mode, `append` has a finite number of solutions and will not cause an infinite loop. However, if `append3` is called with only its last two arguments instantiated, `append` will be called with its first and last arguments uninstantiated. In this mode, `append` has an infinite number of solutions and can cause an infinite loop (for example, if all solutions to `append3` are sought).

In NU-Prolog, this loop can be avoided by adding control information so that the two calls to `append` are not executed from left to right. Calls to `append` which have an infinite number of solutions can be delayed until they are further instantiated.[14] gives an algorithm which analyses potential infinite loops and generates control information to prevent them (for example, the control generated for `append` delays it until the first or last argument are instantiated). There is an implementation of a variation of this algorithm for NU-Prolog called *nac*.

The idea of the loop analysis code of NUDE is to compare the control information in the program with the control information required to prevent infinite loops. The current implementation is a simplification of this, more ideal approach. Any recursive call in the program which could never delay is treated as a potential cause of an infinite loop. In the example above without control information, the user would be warned that the second clause of `append` may cause an infinite loop. With the control information added, the recursive call to `append` could delay (since the first and last arguments are variables), and the potential for an infinite loop is eliminated.

If programmer does not follow the NU-Prolog philosophy and add control information to their programs, the static loop checker will generate many spurious warnings. However, a very good way of detecting or avoiding infinite loops in such programs is to run *nac*, which automatically adds control information where possible and reports potential loops which cannot be avoided by adding extra control.

There is one other special case which the static loop checker detects, which is useful even when control information is not used. Clauses which contain a call to `repeat` but no call to cut cause a warning message. If `repeat` is used then cut must be used somewhere to avoid a loop, and it is safest and most clear if the cut is in the same clause.

5.2. Types

When programmers define a procedure, they intend that the procedure be applicable only to particular types. For example, although the definition of `append`, as given above, can succeed for non-lists, it is intended that `append` only succeed for lists. A procedure could include logic to restrict it to the intended types, but such a definition would be inefficient to execute, and type information would be distributed throughout a program. If type declarations rather than extra logic was used, the type information is made explicit, and the type declaration should be considered as part of the specification of the program. This is summed up in the equation Specification = Program + Types[15] . For example, if `append` has a type declaration stating that all of its arguments should be lists, then for the following definition of `append`

```
append([], A, a).                    % 'a' should be 'A'
append(H.A, B, H.C) :- append(A, B, C).
```

it can be demonstrated that the first clause cannot succeed in the specification. Similarly, the use of `append` in the clause

```
last(E, L) :- append(_, E.nil, L). % 'nil' should be '[]'
```

is inconsistent with the type declaration (that is, it should never succeed). A programmer is unlikely to deliberately include useless clauses in a program, and it is therefore probable that something different was intended. Thus, although both of the faulty clauses above can succeed in the program, they are flagged as containing type errors because they cannot succeed in the "specification".

Two distinct type systems based on these principles have been implemented. Programs with optional type declarations on procedures are examined for clauses (or queries) which, according to the specification, should not take part in a successful execution. One, *regular* types, is based on a precise system of polymorphic types defined in a regular language. The other, *unrestricted* types, is based on types defined by arbitrary Prolog procedures, which have greater expressive power than regular languages, but in which type checking is incomplete.

5.2.1. Regular types

A system in which types are defined by a regular language has been implemented under NU-Prolog[8, 16] . In this system, types can be defined as follows:

```
?- type $list(X) -> { [], X.$list(X) }.
?- type $list -> { $list($_) }.
?- type $odd -> { $_.[], $_.$_.$odd }.
?- type $even -> { [], $_.$_.$even }.
```

There are also several predefined types, such as `$_` for 'don't care' and `$integer` for integers. In the definitions above, functors starting with '`$`' are type symbols. In type definitions, type symbols may have parameters, as in the definition of `$list(X)`; these parameters must be given a value when the type definition is used in a type declaration. Type declarations are of the form

```
?- type append($list, $list, $list).
?- type sumlist($list($integer), $integer).
?- type isOdd($odd).
?- type isEven($even).
```

In this example, arguments to `append` must be lists, the first argument of `sumlist` must be a list of integers and the second an integer, and the arguments of `isOdd` and `isEven` must be odd and even length lists respectively.

One of the advantages of using a regular language to define types is that algorithms can be developed for unification and comparison of such types. This type checker uses type unification to infer a type for a clause based on the terms in the clause and the declared type of the procedures called in the clause. This inferred type is compared to the declared type of the procedure containing the clause. If the inferred type is empty, or is disjoint from the declared type, the clause cannot be used in a successful execution without conflicting with a type declaration, and probably contains an error. The intersection of the inferred and declared types is used as the declared type for that procedure when clauses calling the procedure are considered. For example, the clause

```
query :- odd(X), even(X).
```

has a type error, because the type unification of the types `$odd` and `$even` is empty.

This regular type system was developed as part of a separate system[16] in which declared types are automatically incorporated into the program, so that the new program embodies the original specification. Extra type checking code is added only where it is needed, so that the new program is not significantly less efficient than the original program. The extra tests in the code could be used to detect wrong answers at run time, or the need for extra tests could be reported to the user immediately, to indicate the possibility of type errors.

5.2.2. Unrestricted Types

The definition a type in[15] is more general than in other schemes. Types of procedures are specified by arbitrary Prolog goals rather giving the name of a type (which is defined in a restricted way) for each argument. This added flexibility compared to other type schemes means that more errors can potentially be detected. However, for any given implementation of this scheme, not all type errors will be detected; the problem of detecting clauses which cannot succeed is not decidable.

As a simple example of the extra power the scheme gives, the type information for `append` can specify that each argument is a list *and* the sum of the lengths of the first two arguments equals the length of the last argument:

```
?- append(A,B,C) type
     length(A,AL), length(B,BL), length(C,CL),
     plus(AL,BL,CL).
append([], A, A).
append(H.A, C, H.B) :-        % C and B swapped
     append(A, B, C).

query(X) :- append([a,b], [b,c], X).  % returns X = [a, c]
```

Unlike other systems, our implementation is able to detect that `query` should never succeed. From the initial call to `append`, the system deduces that the length of `X` should be four. In the recursive call to `append`, although all the arguments are lists, their lengths are not compatible with the length of `X` being four. Any solution to `query` must therefore violate the type constraints somewhere in the execution, so an error is reported. The system does not report an error for the recursive clause of `append` because there are instances of it which succeeds correctly. Even when a clause can never succeed correctly, the system may not report an error because the implementation is necessarily incomplete.

The current implementation of the scheme is very simple, consisting of a meta-interpreter for programs and type declarations. The interpreter is run for each clause to be checked. If it fails, the clause is printed with a warning that it should never succeed. Clauses with explicit calls to `fail` are ignored, to avoid spurious messages. Clauses containing disjunctions are treated like multiple clauses with each disjunct being checked independently. The goal which is interpreted consists of the body of the clause plus the type information for the head of the clause. Whenever the interpreter calls a procedure, it also calls the corresponding type goal (if one exists).

The only requirement on the meta-interpreter is that it is sound with respect to finite failure. This makes it easy to deal with even the non-logical aspects Prolog, such as `assert`, cut, `var`, `read`, unsound negation and so on. Everything which is difficult to deal with is ignored; it is assumed that the call succeeds. Several system predicates which normally cause problems for type systems, such as `=..` and `arg`, can be called however, because they behave logically (delaying when insufficiently instantiated, rather than failing). To avoid infinite loops, the normal Prolog search space is not searched completely. This is very important since the bodies of clauses are not instantiated before they are run. The current implementation has a limit on recursion. Any recursive call beyond a certain depth is assumed to succeed.

5.3. Groundness analysis

As stated earlier, a goal flounders if it terminates with delayed subgoals which are insufficiently instantiated to be woken. Analysis of the propagation of variable bindings in a program can detect likely causes of floundering during execution. For example, consider a definition of `member` which avoids repeated solutions.

```
member(H, H._).
member(E, H.L) :- not ( E = H ), member(E, L).

query :- member(heart-Val, [club-10, heart-4, spade-9]).
```

Since sound negation is only be evaluated when its argument is ground, `query` will flounder. This problem can be identified by groundness analysis; this analysis can be extended to encompass extended syntax and evaluable predicates. For example, consider the program

```
increment(N, Nplus1) :- N is Nplus1 - 1.

query :- increment(5, X).
```

In this example, `query` will flounder, because `is` requires that the right-hand argument be ground. Analysis can demonstrate that `increment` can only propagate bindings from its second argument to its first. If the definition of `increment` is changed to

```
increment(N, Nplus1) :- plus(N, 1, Nplus1).
```

in which the call to `plus` will ground the first or third argument, but not both, then analysis can demonstrate that `increment` will propagate bindings in either direction.

Groundness analysis is particularly suited to deductive database applications. NU-Prolog provides a sophisticated first order logic query language[17] ; in complicated queries, it is not always apparent whether a variable will be ground by evaluation of the query. Groundness analysis may be applied to queries before they are passed to the deductive database system for evaluation.

6. DYNAMIC CHECKS

The dynamic checks of NUDE analyse a program with respect to a particular goal. The user is first asked if static checks should be performed. The choice of static checks depends on what dynamic check is being called. The simple checks, such as those concerning variable and functor names, are always chosen, but the loop checker is only run with `loop` and the type checker is run with `miss` and `wrong`, et cetera. These checks are performed on the goal and all predicates indirectly called by it. After the static checks are performed, the goal is generally executed in some way and this execution is monitored.

6.1. Wrong answers

Declarative or algorithmic debugging for Prolog was first presented in[9] . This was extended to allow coroutining and arbitrary first order logic formulas in the bodies of clauses in[11] . The theory behind declarative debugging is that the user has an *intended model* of the program (that is, whether certain calls should succeed or fail) but the completion of the program is not true in that model (that is, some procedure definitions are inconsistent with the intended model). If a wrong answer is returned, normally there is a clause which is false in the intended model; that is, for some instance of the clause, the body correctly succeeds but a call to the atom in the head should fail. The debugging algorithm finds such a clause instance.

The algorithm can be summarised as follows. The user gives as input to the algorithm an atomic goal which should fail, but in fact succeeds without further instantiating any variables. A matching clause instance with a body which succeeds is found; this clause instance is found by running the body. If any literal in the clause body should fail then the bug is be sought recursively. Otherwise, an incorrect clause instance has been found. If a negated subgoal succeeds incorrectly then the missing answer algorithm is invoked to find the bug (described briefly in the next section). It straightforward to extend this algorithm to work for all goals which return an incorrect answer, even if they bind variables in the process or have correct as well as incorrect answers.

To find out if a goal should succeed, the system queries the user. The response is stored in a database, so the user is not asked the same question repeatedly. The user can also indicate that the procedure in question has been completely debugged. The system will then just run the code and assume the answer is always correct. A similar but complementary idea is to have two definitions of a procedure; one assumed to be correct (the specification, which may be very inefficient) and another which may need to be debugged[12] . Rather than asking the user questions, the specification code is run. We plan to add this facility to NUDE.

For a brief example of how the wrong answers diagnosis works, consider the buggy `append` procedure from section 5.2.2. Parts typed by the user are in **bold**.

```
?- wrong query(X).
Do static analysis? n
Do dynamic wrong answer analysis?       % default=yes always
query([a, c])   Valid? n
append([a, b], [b, c], [a, c])     Valid? n
append([b], [c], [b, c])   Valid?

Incorrect Clause Instance:
append([a, b], [b, c], [a, c]) :-
        append([b], [c], [b, c]).

Matching Clauses:
append([H|A], C, [H|B]) :-
        append(A, B, C).

Do you want to edit append/3 ?
```

6.2. Missing answers

The method used to diagnose missing answers is similar in spirit to that for wrong answers. The bug normally returned is an ‘uncovered atom’; that is, an atomic goal which should succeed, but the bodies of all clauses it matches with correctly fail. It indicates that an extra clause is needed in the procedure or one of the clauses needs to be made weaker. Diagnosing missing answers is actually a more difficult problem than diagnosing wrong answers and the algorithm is more complex. One of the consequences for users is that the questions asked by the system are not always yes/no questions. Sometimes the system asks if a goal is satisfiable (true for some values of the variables in it), then asks for particular instances which are valid. The user must provide answers to goals rather than just verify whether answers are correct.

Details of the algorithm used are given in[13] . It is based on the algorithms of Shapiro and Lloyd, but has several enhancements which reduce the number of questions asked. It is particularly important to reduce the number of questions where the user must provide valid instances of goals. One technique used to find valid instances of a goal is to call the goal (rather than ask the user). The user is then asked a yes/no question to make sure the answer is valid. If the answer is not valid, `wrong` is called. Missing answer bugs in top level goals are often caused by wrong answers somewhere deeper in the computation. Since wrong answers are easier to diagnose, this feature can avoid many difficult questions. `wrong` is also called when a negated goal misses an answer.

6.3. Infinite loops

Infinite loops are dealt with at run time by a meta-interpreter. The interpreter maintains a list of the ancestor calls, the clauses they were matched with and the current recursion depth. If the recursion depth reaches a multiple of some value (the default is 30) then the interpreter succeeds. A message is printed indicating the depth and the number of times each clause has been used in the branch of the proof tree. If the user is not convinced the loop has been found, the interpreter can continue the computation. If a call matches with any ancestor call then the interpreter succeeds and a message is printed in a similar manner.

Many infinite loops only occur after a substantial amount of backtracking, for example, after all solutions to a goal have been found. This is due the to practice of putting simple clauses such as facts (base cases) before recursive clauses. The interpreter dynamically changes the clause selection rule to avoid the extra backtracking and find the infinite loop as quickly as possible. Rather than always trying the first clause first, the interpreter tries the most recently used clause in the current branch of the tree first. Consider the following example:

```
      % standard O(N) reverse
reverse(L, R) :- reverse1(L, [], R).

      % first arg should be instantiated
      % or second clause can cause a loop
reverse1([], R, R).
reverse1(H.L, R0, R) :- reverse1(L, H.R0, R).

      % palindrome: logic correct
      % but reverse causes loops
palindrome(LR) :- reverse(L, R), append(L, R, LR).
palindrome(L_R) :- reverse(L, R), append(L, _.R, L_R).

?- palindrome("able was I ere I saw elba").
```

With the standard order for executing calls and selecting clauses, the first call to `reverse` repeatedly generates longer and longer lists which cause `append` to (eventually) fail and backtrack. It takes O(N^2) time to reach a recursion depth of N. With the modified clause selection rule, the first call to `reverse1` matches with the fact and succeeds, causing `append` to fail immediately. On backtracking, the second (recursive) clause of `reverse1` is selected. When the recursive call to `reverse1` is made, it is matched with the recursive clause first. This clause is chosen first for each subsequent call, so a recursion depth of N is reached after N+1 calls. Note that no call matches with any ancestor call due to the way the second argument of `reverse1` is constructed.

When the recursion gets deep enough, the interpreter will succeed and inform the user that the first clause of `palindrome` and `reverse` have been used once and all other calls used the second clause of `reverse1`. The bug can be fixed most simply by adding control information to `reverse1` so that calls to it delay until the first argument is instantiated. It is also desirable to change the order of subgoals in `palindrome`.

6.4. Floundering

In NU-Prolog, calls to both system and user-defined procedures can delay when they are not sufficiently instantiated; this can lead to floundering as discussed in Section 2. When a goal flounders, the NU-Prolog system has neither proved nor refuted the goal. With sensible control information, NU-Prolog programs will often flounder when the goal has an infinite number of solutions. However, bugs in the program, both in the logic and in the control, can cause floundering in quite reasonable goals.

When a goal flounders in NU-Prolog, the system simply prints a message to that effect. No indication of which calls were delayed and never woken is given. This can make tracking down the problem quite difficult. In MU-Prolog, when a goal flounders, each delayed call is printed, much simplifying diagnosis. NUDE provides a similar facility with the `flounder` procedure. It is implemented using a meta-interpreter which understands NU-Prolog's control facilities. Because the control is specified at the procedure level, rather than the call level, the meta-interpreter must be correspondingly more complex. The meta-interpreter delays whenever the call being interpreted would delay, and is woken whenever the call would have been woken. It also has an extra argument which is instantiated when the computation terminates. This wakes up any remaining delayed calls to the meta-interpreter, which then print the calls instead of interpreting them. A future implementation will further simplify debugging by also printing the procedure from which the call was made.

6.5. Spy points

When a NU-Prolog procedure which has a spy point is executed, the call (and other information) is printed and the user has a large number of options (continue the computation, fail, turn off debugging partially or completely etc). One of the options, '|', causes a user-defined procedure `spyHook` to be executed with the current call, port and other information. This facility is used as an interface between the four port debugger and the dynamic debugging tools of NUDE. At an exit (succeed) port, the '|' option is equivalent to calling `wrong` with the current call. First the user is asked if static analysis is desired, then the declarative debugger is used. At a fail port, the '|' option is equivalent to calling `miss`, and at a call port it is equivalent to calling `loop`.

This facility is very useful when the top levels of a program do various non-logical operations, such as initializing tables, reading data and writing results, but call logical code to do major parts of the computation. Calls to the logical procedures can be quite complex, so rather that typing in such calls, a spy point can be added and the non-logical top level code can be called. When control enters the logical code, the '|' option can be used to change to the declarative debugger, for example. The technique is also useful for large programs if the user has a good idea as to which part of the program contains the bug. This can significantly reduce the number of questions put to the user.

7. AUTOMATIC TESTING

The testing mechanism has three phases. The first runs static checks on everything called by the goal. The second checks that the goal behaves consistently with all information in the database. The database contains atoms which are known to be satisfiable (they should succeed, possibly binding some variables), unsatisfiable (they should fail) and valid (they should succeed without binding any variables). Atomic goals are simply matched with the atoms in the database then run. If any goal instance fails incorrectly then `miss` is called and if any goal instance succeeds incorrectly then `wrong` is called. Only valid instances of non-atomic goals are tested. They are constructed by finding valid instances of the literals in the goal. It is possible to do the same for unsatisfiable instances also.

Matching goals with valid and unsatisfiable atoms in the database is done by unification, but satisfiable atoms must be subsumed by the goal. This is implemented by applying `numbervars` to all the satisfiable atoms in the database, which binds all variables in the atoms to terms which are unlikely to occur elsewhere. `numbervars` is also applied when testing goal instances which are known to be valid, to make sure no variables in the goal get bound when it is run. The use of `numbervars` is not ideal, since it can bind variables to terms which occur in the program. Debugging programs which use `numbervars` can be difficult because of this.

Testing with each of the three kinds of atoms is optional. There are two reasons for this. Sometimes the user already knows that the program is consistent with the database for that goal (for example, that phase of testing may have already been completed successfully). Also, running some of the goal instances can cause infinite loops. Rather than preventing further testing of (partial) correctness, this can be ignored, at least temporarily. The infinite loop may disappear when the partial correctness bugs are fixed, or the loop may be because the program was run in a mode which will not be needed.

The third phase of testing interprets the goal. The user is asked whether each solution returned is valid. If the solution is valid the system continues searching for more solutions, otherwise `wrong` is called. If the goal eventually fails, `miss` is called to check if any solutions were missed. If the goal has an infinite number of solutions the system will keep generating solutions until the user is convinced that the procedure is correct or suspicious that some answers are being missed (these can then be checked separately). If Prolog's (unfair, depth first) search strategy was used, an infinite search space could result in some answers which are derivable from the program never being returned. This is clearly undesirable, so a fair search strategy is used, which guarantees that every answer will be returned eventually.

The most obvious fair search is breadth first. Unfortunately, this is very expensive to implement in Prolog. Whenever a call is matched with a clause, the call and the entire continuation must be duplicated to maintain information about variable bindings on different branches of the tree. This takes a great deal of space and time and does not take advantage of the Prolog language well. A much superior approach is to use a bounded depth first search with iterative deepening. For each successive depth, Prolog's backtracking mechanism can be used to search very quickly and with little space overhead. Detecting when the tree has been searched completely requires the use of side effects. Whenever the search is stopped by the depth bound, a flag is set. If the flag is not set at all during the search at one depth, then the tree has been searched completely and the depth does not have to be increased further. Another desirable enhancement is to avoid repeatedly returning answers found near the root of the tree. This is achieved checking the depth bound of the previous iteration when each solution is found. Answers found above the previous bound have been returned already, so they are ignored.

The `append` program of Section 5.2.2 is used again, to illustrate the third phase of testing.

```
?- test(append, 3).
Do static analysis? n
Test known valid instances? n
Test known satisfiable atoms? n
Test known unsatisfiable atoms? n
Generate solutions to goal?
Solution found: append([], A1, A1).
append([], A1, A1) Valid?
Solution found: append([A1], B1, [A1|B1]).
append([A1], B1, [A1|B1])  Valid?
Solution found: append([A1, B1], [B1|C1], [A1|C1]).
append([A1, B1], [B1|C1], [A1|C1]) Valid? n
```

When the wrong answer is found, debugging continues as in the previous example. After the procedure has been edited and the new version has been loaded the user is asked if they want `append` to be tested again. Because of the information now stored in the database, the system can quickly check whether the bug has been corrected and whether the new code still behaves correctly for the other cases.

8. FURTHER WORK

There is a great deal of work still to be done on NUDE. First, several of the tools discussed still need more work on implementation and/or integration with NUDE. All the tools work for simple code, but some have limitations (such as not being able to handle cut) and others run as separate programs but are not yet incorporated into NUDE. The two type systems should be integrated, hopefully keeping the best features of each. Several enhancements mentioned in this paper should be implemented. More consideration needs to be given to the debugging of very large programs and of parts of programs which contain non-logical code. Some features of the current system, such as the '|' option on spy points and the ability to declare procedures correct, provide some support already. Another area for improvement is the treatment of code which is transformed in some way before execution, such as definite clause grammar rules and Parallel NU-Prolog[18] . Currently NUDE only knows about the transformed code, whereas debugging may be better done at the original source code level.

There is no doubt that NUDE can be improved by adding more tools along the same lines as the existing tools, and by using better algorithms for some of the tools. For example, the work on most specific programs[19] includes algorithms which can be used for detecting failing clauses and for automatic testing. However, there is probably more to be gained by putting more high-level knowledge about bugs into the system. NUDE incorporates relatively low level concepts of what bugs are, such as incorrect clause instances. There are no idea that a clause can be wrong because the programmer forgot to put in a test, for example. Some systems do incorporate knowledge about various common kinds of bugs, which can lead to bugs being found more quickly and reported to the user in a nicer way[20] . It is also important to investigate the different misunderstandings which lead to bugs[21] . For example, a bug may be caused because the programmer did not understand the behaviour of a particular system predicate. Currently, NUDE will just trace the bug to the clause or procedure in which the system predicate is called. Ideally it should get to the crux of the problem and display the manual entry for the predicate.

NUDE is currently aimed only at detecting and locating bugs, but there is obvious merit in adding tools to help fix bugs. The techniques of[9] and[12] can be used to fix bugs resulting in wrong and missing answers. Some bugs which result in looping or floundering are also mistakes in the logic, not just the control, and they may also be amenable to similar treatment. A fair search procedure, such as the one used for testing, can find mistakes in the logic even if the program would not terminate with Prolog's search strategy. The algorithm used in *nac* is also useful for fixing infinite loop bugs and floundering bugs which are caused by user-defined control. Dataflow analysis such as that used for groundness analysis can also be useful.

9. CONCLUSION

The dual declarative and procedural semantics of logic programming permit the building of very powerful programming environments. We have described the NUDE system, which combines the standard Prolog debugging tools, static program analysis, runtime bug diagnosis, and automatic testing in a natural way. It allows programmers to use many interacting tools for detecting and locating bugs. There is potential for further improvement by incorporating knowledge about common kinds of bugs and the addition of tools to help correct errors.

ACKNOWLEDGEMENTS

Giles Lean contributed to many discussions and helped implement the system. Jeffrey Schultz implemented most of the NU-Prolog system, including the four port debugger. James Thom provided much helpful criticism during the development of *nit*. This research was partly supported by the Australian Commonwealth Department of Science and ARGS (now ARC).

References

1. James Thom, Justin Zobel, "NU-Prolog reference manual, version 1.3," Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia (1988).
2. J.W. Lloyd, R.W. Topor, "Making Prolog more expressive," *Journal of Logic Programming* **1**(3) pp. 225-240 (October 1984).
3. John W. Lloyd, *Foundations of logic programming,* Springer-Verlag, New York (1984).
4. Lawrence Byrd, "Understanding the control flow of Prolog programs," *Proceedings of the 1980 Logic Programming Workshop*, pp. 127-138 (July 1980).
5. Marc Eisenstadt, Mike Brayshaw, "The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming," Technical Report No. 21a, The Open University, Human Cognition Research Laboratory, Milton Keynes, United Kingdom (October 1987). To appear, Journal of Logic Programming
6. Maurice Bruynooghe, "Adding redundancy to obtain more reliable and more readable Prolog programs," *Proceedings of the First International Logic Programming Conference*, pp. 129-133 (September 1982).

7. Alan Mycroft, Richard A. O'Keefe, "A polymorphic type system for Prolog," *Artificial Intelligence* **23** pp. 295-307 (1984).

8. J. Zobel, "Derivation of polymorphic types for Prolog programs," *Proceedings of the Fourth International Conference on Logic Programming*, pp. 816-838 (May 1987).

9. Ehud Y. Shapiro, *Algorithmic program debugging,* MIT Press, Cambridge, Massachusetts (1983).

10. Luis Moniz Pereira, "Rational debugging in logic programming," *Proceedings of the Third International Conference on Logic Programming*, pp. 203-210 (July 1986). published as Lecture Notes in Computer Science 225 by Springer-Verlag

11. J.W. Lloyd, "Declarative error diagnosis," *New Generation Computing* **5**(2) pp. 133-154 (1987).

12. Nachum Dershowitz, Yuh-Jeng Lee, "Deductive debugging," *Proceedings of the Fourth IEEE Symposium on Logic Programming*, pp. 298-306 (August 1987).

13. Lee Naish, "Declarative diagnosis of missing answers," Technical Report 88/9, Department of Computer Science, University of Melbourne, Melbourne, Australia (May 1988).

14. Lee Naish, "Automating control of logic programs," *Journal of Logic Programming* **2**(3) pp. 167-183 (October 1985).

15. Lee Naish, "Specification = program + types," *Proceedings of 7th Conference on Foundations of Software Technology and Theoretical Computer Science*, (December, 1987).

16. Philip W. Dart, Justin Zobel, "Transforming typed logic programs into well-typed logic programs," Technical Report 88/11, Department of Computer Science, University of Melbourne, Melbourne, Australia (May 1988).

17. Kotagiri Ramamohanarao, John Shepherd, Isaac Balbin, Graeme Port, Lee Naish, James Thom, Justin Zobel, Philip Dart, "The NU-Prolog deductive database system," *IEEE Data Engineering* **10**(4) pp. 10-19 (December 1987).

18. Lee Naish, "Parallelizing NU-Prolog," *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pp. 1546-1564 (August 1988).

19. K. Marriott, L. Naish, J.L. Lassez, "Most specific logic programs," *Proceedings of the Fifth International Conference/Symposium on Logic Programming*, pp. 909-923 (August 1988).

20. Marc Eisenstadt, "Retrospective zooming: a knowledge based tracking and debugging methodology for logic programming," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 717-719 (September 1985).

21. Paul Brna, Alan Bundy, Helen Pain, Liam Lynch, "Programming tools for Prolog environments," Research Report, Department of Artificial Intelligence, University of Ediburgh (1987).