Practical Declarative Debugging of Mercury Programs

Ian Douglas MacLarty

A thesis submitted to the University of Melbourne in accordance with the requirements of the degree of Masters by Research in the Faculty of Engineering and the Department of Computer Science.

July 2005

Abstract

Debugging is the most unpredictable and potentially expensive phase of the software development life-cycle. Declarative debuggers ask the user questions about the correctness of subcomputations in their program. Based on the user's answers, subcomputations that cannot be the cause of the buggy behaviour are eliminated. Eventually one subcomputation is left which must be the cause of the buggy behaviour. Declarative debuggers thus keep track of which parts of the computation are still suspect, relieving the user of the burden of having to do so. They also direct the bug search, something that many users (especially novices) would find difficult to do manually. Even expert users often find it hard to explore large search spaces systematically, a limitation that does not apply to software systems. Declarative debuggers thus have the potential to make the debugging process easier and much more predictable.

Despite these expected benefits, declarative debugging is not yet widely used in practice to find real bugs. There are three main reasons for this:

1. Most previous declarative debuggers only support a subset of the features of their target language that is not sufficient to express real programs.

2. Previous declarative debuggers do not scale well when applied to problems with large search spaces.

3. Previous declarative debuggers do not do enough to make the questions easier for the user to answer.

The declarative nature of Mercury makes it relatively easy to implement a declarative debugger that can handle the full language. The version of the Mercury declarative debugger that was the starting point for this thesis already handled almost all of Mercury. By extending it to handle exceptions we made it handle the full language

One problem posed by large search spaces is that they cannot be stored in memory all at once. This requires only portions of the search space to be stored in memory at any one time, materializing missing pieces when they are needed by reexecuting the program. We present the first algorithm for controlling this rematerialization process that is practical in the presence of multiple search strategies, minimising reexecutions while keeping memory consumption within acceptable limits.

Another problem with large search spaces is that previous search strategies either asked far too many questions, demanded too much in the way of CPU and/or memory resources

or were too inflexible to coexist with other search strategies. For example, the divide-and-query search strategy is query-optimal in the worst case, however previous implementations of it often required more memory than is typically available. We have implemented heuristics which enable divide-and-query to be used on partially materialized search spaces, making it practical.

We also address the third problem, namely the problem of reducing the complexity of the debugger's queries. The new declarative debugger allows users to specify exactly which part of an atom is wrong. The subterm dependency tracking strategy exploits this extra information to jump directly to the part of the program that computed the wrong subterm. In many cases, only a few such jumps are required to arrive at the bug. Subterm dependency tracking can converge on the bug even more quickly than divide-and-query, and it tends to yield questions and question sequences that are easier for users to answer.

We also support a variety of other methods of making questions easier to answer. By trusting some predicates the user can automate answers to all questions about those predicates (implementing this capability, especially in the presence of higher order code, is trickier than it seems). We also support a novel technique that allows custom visualisations of terms to be easily created. If a call fails a precondition then neither 'yes' or 'no' is an appropriate answer to a question from the debugger about the validity of an answer computed for that call. Our debugger therefore allows users to answer 'inadmissible' to such questions. If all else fails, users can also skip hard questions.

We give evidence that the new declarative debugger can be used on complex, real world programs by presenting several case studies of real bugs found in real programs with the aid of the debugger.

Declaration

I certify that

1. other than as specified in the preface, this thesis comprises my original work;
2. due acknowledgment has been made in the text to all other material used;
3. the thesis consists of approximately 35 000 words, exclusive of tables, maps, bibliographies and footnotes.

Signature:

Ian MacLarty

Acknowledgments

I would like to sincerely thank my supervisor, Zoltan Somogyi, for all his help and encouragement.

I would also like to thank Mark Brown for carrying me to the ambulance, Julien Fischer for his help on many sep*e*rate occasions and Ralph Becket for eh, well, just thanks.

The entire Mercury group also deserve special thanks for all the bugs they have introduced into the Mercury compiler. Without these I wouldn't have had much of a thesis.

Thanks also to Lee Naish and Bernie Pope for many stimulating discussions.

Contents

List of Figures

List of Tables

Preface

This thesis comprises eight chapters of which the first and last are an introduction and conclusion.

Chapter 2 gives the background necessary for the rest of the thesis. Sections 2.1, 2.2 and 2.1.3 give overviews of the Mercury language, the Mercury procedural debugger and exceptions in Mercury. Parts of section 2.3, specifically sections 2.3.2 and 2.3.3, are based on a design by Mark Brown and Zoltan Somogyi [4]. In those sections, the idea of excluding `exit` nodes which produce no output and `negs` nodes is mine. The analysis algorithm in section 2.3.1 is my work. Section 2.3.4 is based on previous work done by Mark Brown, though the handling of `try` and `try_all` is my work. In section 2.4, the idea of being able to resume a declarative debugging session from the procedural debugger is mine.

Chapter 3 describes the features of the debugger which make it easier to use. The implementations of inadmissibility, trusted predicates and term visualisation are mine. The idea of inadmissibility is based on [33]. I/O tabling was implemented by Zoltan Somogyi [46], before I started my thesis.

Chapter 4 describes the search strategies implemented for the Mercury declarative debugger. The top-down search is from the previous version of the debugger which was implemented by Mark Brown and Zoltan Somogyi. Sections 4.2 and 4.3 are an expanded version of [27]. The ideas presented in section 4.2 are all mine. In section 4.3, Zoltan Somogyi implemented the original algorithm for determining the origin of a subterm within a call before I started my thesis. I extended the implementation to track subterms between calls including higher order calls.

Chapter 5 presents a new method for controlling the memory consumption and execution time of the debugger. That chapter is all my original work.

Chapter 6 gives examples of the debugger being used on real programs. Case studies 1 and 3 are based on my own experiences, while case study 2 is based on a bug found by Julien Fischer.

Chapter 7 explores future research opportunities.

Chapter 1

Introduction

Almost all software contains bugs. These are defects in the software which cause it to behave in a manner other than intended. Failures caused by bugs can be benign (such as a character printing in the wrong colour) or they can be catastrophic and cost hundreds of millions of dollars (for an example see [2]).

There are numerous ways to reduce the number of bugs in a software system. Applying sound software engineering practices, such as code reviews and thorough testing certainly helps reduce the number of bugs in a system as do applying good design principles such as data abstraction. Formal verification can be used to prove mathematically that a program will do what its specification says. However there is no guarantee that the specification isn't buggy or that the proof is error free. Indeed, a formal specification may not even be available. Ultimately all software is specified and written by *human* beings, not *perfect* beings.

Locating a bug is often the most frustrating and unpredictable task in the software development cycle. To save time and money, programmers need to locate defects in their programs once a *symptom* is observed as quickly as possible (a symptom is some form of incorrect behaviour of the program). For small programs, simply reading through the code with a critical eye can often reveal the bug. For large pieces of software, developed by teams of programmers, this approach becomes impractical.

One method often used is to augment the code with extra statements which log the values of variables at points of interest (so called "printf" debugging). This approach can be very time consuming as the program must be recompiled and reexecuted each time the programmer wishes to observe a different aspect of the state of the program. The extra statements also make the program less readable and may actually change the behaviour of

the program, masking the effects of the defect being sought. The introduced statements could also be buggy themselves. There is also the added risk that the programmer forgets to remove logging statements before releasing the software.

Numerous tools have been developed that try to address this problem. These tools can aid the programmer in their debugging task, without them having to change the source code. Such tools are generally referred to as *debuggers*.

Tracing debuggers allow the programmer to view the state of the program at any point during its execution and quickly jump to a point of interest, or skip parts of the execution which are not of interest. Some debuggers also allow the programmer to run the program backwards, which is useful for tracking the cause of a symptom (for example [9, 23]).

Even with the aid of such tools, debugging can be an extremely difficult endeavour. This is because the user of the debugger must direct the bug search manually — the user looks for the bug by examining the state of the program at various points in its execution, and the debugger merely facilitates this activity. The user can only do this effectively if they have some idea of where the bug might be. Recent debuggers can tell the user why a program behaved in a certain manner (for example [54]), or can track where the value of a variable came from (for example [50]). However, because the user never communicates to the debugger which subcomputations are wrong and which are right, the debugger can only show the user *what* the program is doing, and not *where* the bug is.

Another class of debuggers, called *declarative* or *algorithmic* debuggers take a different approach. Instead of the user directing the bug search by telling the debugger what parts of the execution they would like to see next, declarative debuggers try to automate most of the reasoning behind the bug search.

The scientific method of finding a bug is to formulate an hypothesis, test the hypothesis, and then formulate a new hypothesis based on the validity of the previous hypothesis. Each hypothesis reduces the set of possible causes of the bug symptom. Eventually there is only one explanation left, so it must be the cause of the bug symptom.

Declarative debuggers automate the scientific method. The hypothesis formed is about the correctness of a subcomputation in the buggy program. The hypothesis is tested by consulting an oracle, typically the user. If the oracle asserts that the subcomputation is correct, then the bug must be in a subcomputation in the program *outside* the subcomputation the hypothesis was about, so the next hypothesis is about the correctness of a subcomputation outside the subcomputation the previous hypothesis was about. If the oracle asserts that the subcomputation is incorrect, then the bug must be in the subcomputation

the hypothesis was about or one of its (direct or indirect) descendant subcomputations, so the next hypothesis is about one of these. Eventually only one subcomputation is left. This must be the bug.

The ability of a declarative debuggers to remember which subcomputations are correct and which are incorrect and use this information to isolate the bug is what distinguishes them from other types of debugging tools.

Since the scientific method is guaranteed to find the bug if the hypothesis test is reliable, declarative debuggers have the potential to make the bug location task more predictable and therefore cheaper in terms of time and money.

The original declarative debugging method was proposed by Shapiro [45]. Shapiro's debugger is written for the pure subset of Prolog, but its ideas are applicable to general programming. Ferrand [15] studied its application to definite logic programs, while Lloyd [24] presented a declarative debugger for general logic programs with negation.

Most of this early work tended to be only of theoretical interest. This may in part be due to the fact that most early declarative debuggers were for Prolog (examples include [15, 24, 40, 45]). and they had no or very limited support for the impure features of Prolog, such as the cut operator and side effects. Consequently they could not be used to debug realistic programs.

Algorithmic debugging has also been applied to imperative languages (most notably [16]), although current implementations only work on toy examples.

Recently progress has been made in developing declarative debuggers that can be used on realistic Haskell programs (for example [35, 43, 49]). This progress is in part due to Haskell's clean declarative semantics, although non-strict evaluation proved a challenge and most of that research is concerned with overcoming this hurdle.

Another reason some previous declarative debuggers fall short when applied to realistic problems, is the lack of complex programs written in the programming languages targeted by these debuggers. (For example virtually no large Prolog programs are written in the pure subset of Prolog.) Most of the test cases presented are manufactured and are only useful for testing or demonstrating the debugger.

Debugging is an extremely complex problem. The insight which can be gained from a purely theoretical analysis is very limited. The most valuable lessons are learnt by applying declarative debugging to complex, real-world programs. That is what this research aims to do.

Mercury is a purely declarative logic and functional programming language intended

to support the creation of large, reliable programs.

Mercury is an attractive target for an investigation into practical declarative debugging because its clean declarative semantics mean it is amenable to declarative debugging. In Mercury, the operational semantics are also much more closely coupled with the declarative semantics. This makes it much easier to implement a declarative debugger for Mercury, than it would be for a non-strict language, such as Haskell.

An essential part of any practical investigation must involve trying out the debugger on real bugs in complex programs. We have a large Mercury program on which to test the declarative debugger, namely the Mercury compiler itself. This is a Mercury program of approximately 300 000 lines of code. It is over ten years old and under continual development. This means we have a good supply of real bugs on which to test the declarative debugger, which is important for a practical study. Using the declarative debugger on the Mercury compiler has led to most of the innovations in this thesis.

The effectiveness of declarative debuggers, like other debugging tools, is measured by how long it takes to find a bug with their help. One obvious objective when designing the declarative debugger is therefore to minimize the number of questions asked of the user. The fewer questions the user needs to answer, the sooner the bug is found. Reducing the number of questions has been the focus of much past research (for example [13, 16, 19, 28, 39, 40, 45]). However most of the techniques in the literature result in poor performance when applied to large search spaces because they require a representation of the search space to be resident in memory. This is infeasible if the search space is large.

The time required to find the bug is the product of the number of questions asked and the average time required to answer each question. Making the questions asked by the declarative debugger easier to answer is therefore as, if not more, important than reducing the total number of questions. The focus of this research is on implementing techniques to reduce both the number of and the complexity of the debugger's questions.

The remainder of this thesis is divided up as follows.

In chapter 2 we give the background which will be needed for the rest of the thesis. We introduce the Mercury programming language, followed by an overview of the Mercury procedural debugger. We then describe the design of the Mercury declarative debugger.

In chapter 3 we describe some of the usability features of the declarative debugger which make it practical to use on realistic programs. Most real programs perform I/O, so our debugger includes support for debugging code which performs I/O. In some cases the user may not always know what a part of their program is supposed to do, we therefore

also allow them to answer 'don't know' to questions from the debugger. The debugger can also be told to trust certain predicates. This results in fewer questions, since the declarative debugger will not ask questions about trusted predicates. The debugger also includes a novel approach to creating custom visualisations of data terms appearing in questions. The user may also assert that certain questions are 'inadmissible' which means the subcomputation the question is about is neither correct nor incorrect, but instead should never have occurred in the first place.

The most important part of the declarative debugger is the *search strategy*. The search strategy is used to decide what question to next ask the user. Ideally, the questions asked should be simple and there should be as few of them as possible so that the total time to find the bug is minimised.

The previous implementations of the most promising query minimisation strategies in the literature, namely divide-and-query (first proposed by Shapiro [45]) and subterm dependency tracking (first proposed by Pereira [40]), cannot easily be applied to long running programs which generate large search spaces. In chapter 4 we present an approximation to Shapiro's divide-and-query algorithm that is efficient, even if the search space contains hundreds of millions of subcomputations. This approximation is accurate enough to yield good results. Subterm dependency tracking allows the user to tell the declarative debugger exactly which part of an incorrectly computed value is incorrect. The debugger can use this information to reduce dramatically the number and complexity of the questions it asks.

In chapter 5 we present a new approach to controlling the resources consumed by the debugger.

In chapter 6 we give several examples of real bugs in the Mercury compiler as well as in the declarative debugger itself, that were found using the declarative debugger. The examples also suggest new areas for future research. These are explored in chapter 7.

All the ideas presented in this thesis are implemented in the latest version of the Mercury declarative debugger, which is distributed with the Mercury system and is available for download from http://www.cs.mu.oz.au/mercury/. The Mercury User's Guide is available from the same site. It documents all the features of the declarative debugger and how to use them.

Chapter 2

Background

2.1 Mercury

Mercury [48] has its roots in logic programming. Its syntax looks like the syntax of Prolog. However, programming in Mercury feels different from programming in Prolog. One reason is that unlike Prolog, Mercury is purely declarative. Another is that Mercury's design objective is to support teams of programmers building large, reliable software systems, and thus the Mercury compiler insists on knowing a lot more information about the program. This includes information about types, modes and determinisms.

2.1.1 Types, modes and determinisms

Mercury has a Hindley-Milner type system very similar to Haskell's. A type restricts the set of function symbols a value of the type can have. Each function symbol has an arity which determines how many arguments the function symbol can have. The type declaration also gives the types of these arguments. There are four builtin types: '`int`' for integers, '`float`' for double precision floating point numbers, '`string`' for character strings and '`char`' for single characters. Polymorphic types are defined using type variables.

Here is an example type declaration.

```
:- type account
    --->    savings(float)
    ;       cheque(float, float).
```

Values of the type `account` can have the function symbol `savings` with one floating point argument (the argument represents the interest rate), or they can have the function

symbol `cheque` with two floating point arguments (the interest rate for a positive balance and the interest rate for a negative balance). Examples of valid terms of this type are `savings(1.03)` and `cheque(1.02, 1.16)`.

The arguments of a predicate are declared to be of a certain type with a declaration such as the following.

```
:- pred append(list(T), list(T), list(T)).
```

Here the `append` predicate is declared to have three list arguments. The lists can contain elements of any type, but all three lists must contain elements of the same type.

Functions can be declared in much the same way as predicates. For example

```
:- func fib(int) = int.
```

A mode classifies each argument of a predicate or function to be either input or output. If input, the argument passed by the caller must be a ground term; if output, the argument passed by the caller must be a free variable, which the predicate or function will instantiate to a ground term. It is possible for a predicate or function to have more than one mode; the usual example is `append`, which has two principal modes: `append(in,in,out)` and `append(out,out,in)`. We call each mode of a predicate or function a *procedure*. The Mercury compiler generates different code for different procedures, even if they represent different modes of the same predicate or function; in fact, different procedures are handled as separate entities by most parts of the Mercury debugger and by all parts of the compiler after mode checking. The mode checking pass of the compiler is responsible for reordering conjuncts in conjunctions as necessary to ensure that for each variable, the goal that generates the value of the variable comes before all goals that use the value of the variable.

Each mode of a predicate or function has a determinism associated with it. This limits the number of solutions that procedure may have. Procedures with determinism *det* succeed exactly once; procedures with determinism *semidet* succeed at most once; procedures with determinism *multi* succeed at least once; and procedures with determinism *nondet* may succeed zero or more times.

If mode and determinism declarations are omitted from a function declaration, then the function is assumed to have one mode where all the arguments are input, the return value is output and the determinism is det. Unless otherwise specified, all points in this thesis pertaining to predicates in Mercury also pertain to functions, since in Mercury a function is just a special kind of predicate.

If a predicate has only one mode, then type, mode and determinism declarations can be combined.

```
:- pred sum(list(int)::in, int::out) is det.
```

In our experience, few predicates are designed to have more than one solution; most have exactly one. For example, in the Mercury compiler, roughly 85% of procedures are det, 14% are semidet, and only 1% multi or nondet. There are a few reasons why this is the case. One reason is the fact that nondet or multi code cannot perform I/O operations, because it is impossible to backtrack over an I/O operation once it has been performed (a printed document cannot be unprinted). Another is to do with the fact that control information must be reflected in the declarative reading of the problem, instead of with extra-logical features such as cuts. This often means search algorithms must be encoded using if-then-else goals and recursion, instead of with backtracking and cuts.

Mercury has a module system that supports information hiding. Each module has an interface section in which all exported types and predicates are declared. Each module also has an implementation section where all the exported predicates are defined, along with the internal predicates and types for the module. The types and predicates declared in the implementation section can only be accessed from the module in which they are declared.

Programmers must declare the types, modes and determinisms of predicates and functions exported from their defining modules, and common practice is to declare them for internal predicates and functions as well, though these could be inferred. The compiler verifies these declarations. This process catches most simple errors in the program, leaving only the relatively complex ones to be found by the debugger.

To illustrate consider the following predicate.

```
:- pred apply_interest(account::in, float::in, float::out) is det.
apply_interest(savings(I), Amount0, Amount0 * I).
apply_interest(cheque(IPos, INeg), Amount0, Amount) :-
    ( if Amount > 0 then
        Amount = IPos * Amount0
    ;
        Amount = INeg * Amount0
    ).
```

(Note that unlike in Prolog, in Mercury evaluable functions such as `*` can appear anywhere, not just on the right hand side of the `is` operator.)

The `apply_interest` predicate performs an interest calculation based on a given type of account.

The mode and determinism declarations say that for any account type and amount there is exactly one way to apply interest to the amount.

Now suppose the program is later modified and a new function symbol is added to the `account` type.

```
:- type account
    --->    savings(float)
    ;       cheque(float, float)
    ;       credit(float, float).
```

Suppose further that the programmer forgets to update the `apply_interest` predicate definition to hand credit accounts.

In other logic languages such as Prolog and Gödel [18] which do not have strong mode and determinism systems, this would become a bug in the resulting executable and the error would only be detected at runtime if at all. If the predicate were implemented in an imperative language as a case statement, then the error would also go undetected by most compilers.

The Mercury compiler, however, spots this error, since the determinism of the predicate no longer matches its declared determinism (because the predicate will fail if it is passed a `credit` account type).

Other types of errors are also impossible to express in Mercury. One cannot, for example, dereference an invalid pointer, since Mercury has no concept of pointers. The mode system also ensures that variables cannot be used before they are initialised.

2.1.2 Input and Output

The Mercury mode system also supports uniqueness through the use of the modes `di` which stands for *destructive input* and `uo` which stands for *unique output*. If a value is passed as destructive input to a procedure, then there may be no other references to the value live at the time of the call, since the call will clobber that value. Calls that accept destructive input cannot be backtracked over or fail, since it is impossible to reconstruct the clobbered value. The compiler checks these conditions. A unique output argument is guaranteed to be unique, so it can be passed as destructive input to other calls.

Unique modes are particularly useful for performing I/O. The Mercury standard library

provides an 'io' type which abstractly represents the state of the world at a particular point during program execution. All predicates that perform I/O must accept a pair of arguments of the type 'io'. One of the arguments must be destructive input and the other must be unique output. This ensures that I/O operations are never backtracked over.

Every valid Mercury program contains a `main` predicate that accepts a `di`, `uo` pair of arguments of type `io`. The `main` predicate is the first predicate executed in a program. This, together with the fact the I/O arguments are uniquely moded, ensures that there is only ever one reference to the current I/O state at any point in the execution of a program.

I/O states enable I/O to be handled in a purely declarative way in Mercury. They therefore do the same job as monads in Haskell, and linear types in Clean.

Because threading a pair of I/O states through predicates can be laborious, Mercury allows some syntactic sugar called *state variable notation*. This allows predicates such as the following

```
:- pred write_html_tag(string::in, io::di, io::uo) is det.
write_html_tag(ElementName, IO0, IO) :-
    write_string("<", IO0, IO1),
    write_string(ElementName, IO1, IO2),
    write_string(">", IO2, IO).
```

to be written as

```
:- pred write_html_tag(string::in, io::di, io::uo) is det.
write_html_tag(ElementName, !IO) :-
    write_string("<", !IO),
    write_string(ElementName, !IO),
    write_string(">", !IO).
```

The '`!IO`' above are called *state variables* and represent an input/output pair. If a state variable appears in a fact, then the variables in the pair are simply unified.

2.1.3 Exceptions

Mercury allows exceptions[1] to be thrown and caught during the execution of a program. This means that beside succeeding or failing, a procedure call may also throw an exception.

[1]It should be noted that the exceptions we refer to here have nothing to do with the exceptions proposed by Kowalski [22]. His exceptions are logic programming rules of the form "$\neg p \leftarrow q$", and are unrelated to the exceptions we discuss here. The exceptions discussed in this section are similar to the exceptions in some imperative programming languages, such as Java.

This is an essential feature when developing large software systems as it allows runtime assertion checking and graceful recovery from unexpected behaviour.

The value of the exception can be any ground Mercury term. The special purpose predicate `throw` is used to throw exceptions. Declaratively `throw` has no interpretation. Operationally, calling `throw` causes its argument to be thrown as an exception. For example:

```
divide(N, D, Q) :-
    ( if D = 0 then
        throw("division by zero")
    else
        Q = N / D
    ).
```

If a thrown exception is not caught, then the program aborts. For procedures which may succeed at most once, a thrown exception can be caught by passing a call to the procedure to the builtin predicate `try`. For example:

```
maybe_divide(D, N, MaybeQuotient) :-
    try(divide(D, N), Result),
    ( if Result = succeeded(Q) then
        MaybeQuotient = yes(Q)
    else
        MaybeQuotient = no
    ).
```

The first argument to `try` is the closure which may throw an exception. The procedure should have one output argument. In the example we use currying to convert `divide` into a procedure which returns one output argument. After the procedure has been executed the second argument is unified with one of three functors:

1. `succeeded(R)`, if the call succeeds, where `R` is bound to the output of the procedure,
2. `failed`, if the call fails or
3. `exception(E)`, if the call or one of its descendant calls throws an exception, where `E` is the exception which was thrown.

To catch exceptions thrown by procedures that may succeed more than once, another predicate is provided. `try_all(Goal, MaybeException, Solutions)` tries to finds all

solutions to `Goal`. If no exception is thrown by `Goal` in doing this, then `MaybeException` is bound to `no` and `Solutions` is bound to the list of all solutions for `Goal`. If `Goal` does throw an exception, then `MaybeException` is bound to `yes(E)`, where `E` is the thrown exception, while `Solutions` is bound to the list of all solutions found before the exception was thrown.

Versions of `try` and `try_all` that accept a pair of I/O states are also provided. These allow exceptions thrown by predicates which do I/O to be caught.

Whenever a procedure call throws an exception, an `excp` event is generated instead of the `exit` or `fail` event which would have been generated had the procedure succeeded or failed respectively. Thrown exceptions propagate up the call stack until a `try` or `try_all` stack frame is found. A new `excp` event is generated for each stack frame as it is popped off the stack while searching for the nearest `try` or `try_all`. Every generated `excp` event therefore has a matching `call` event earlier in the trace.

2.1.4 Higher order terms

Mercury allows higher order terms to be passed as arguments to procedure calls. The higher order term appears as a variable in the procedure it is passed to and can be called by placing arguments in parenthesis after the higher order variable.

For example a '`map`' predicate which applies a higher order term to the elements of a list to produce a new list could be defined as follows.

```
map(_, [], []).
map(P, [H0 | T0], [H | T]) :-
    P(H0, H),
    map(P, T0, T).
```

Higher order terms can be constructed by currying or by in-place anonymous procedure definitions.

2.1.5 Extra terminology

In Mercury a disjunction where each disjunct unifies the same ground variable with a different function symbol of the same type is called a *switch*. Switches can be implemented more efficiently than normal disjunctions since at most one of the disjuncts can possibly be true, so at most one disjunct in a switch is evaluated.

The body of a predicate is simply the disjunction of all the clause bodies with overlapping variables replaced by unique variables. All arguments in the head are transformed to unique variables, and the appropriate unifications are added to the body.

Because of this, we will refer to the body or head of a *predicate*, instead of the body or head of a *clause* as is usually done in logic programming literature.

The Mercury compiler first compiles a Mercury program down to an imperative program (usually C, though Java and IL backends have also been partially developed). An executable (or bytecode in the case of Java or IL) is then generated from the imperative program.

2.2 The Mercury procedural debugger

When a program is compiled with debugging enabled, the generated imperative code is instrumented with callbacks to the runtime system. These callbacks are placed at points in the program that might be of interest, such as the entry and exit points of procedures. If the program is run normally then these callbacks are null operations, however if the program is run under the debugger then the callbacks stop the execution of the program and give control to the debugger.

Once the debugger has control it interacts with the user. The user can then step through the execution of the program in much the same way he or she would step through the execution of a C program using gdb. The procedural debugger supports most features one would expect from a modern tracing debugger such as conditional breakpoints and the ability to view the values of variables on the call stack. The user may also rewind the state of the program to the state it was in at a particular call, even in the presence of I/O. This is achieved using the '`retry`' command and I/O tabling [46].

The internals of the procedural debugger are described in more detail in [47].

Events can be classified into two categories, *interface* events and *internal* events. Interface events describe the interaction between one invocation of a procedure and its caller, while internal events describe the flow of control inside the call. The are five types of interface events.

`call` A call event occurs just after a procedure has been called, and control has just reached the start of the body of the procedure.

`exit` An exit event occurs when a procedure call has succeeded, and control is about to

return to its caller.

`redo` A redo event occurs when all computations to the right of a procedure call have failed, and control is about to return to this call to try to find alternative solutions.

`fail` A fail event occurs when a procedure call has run out of alternatives, and control is about to return to the rightmost computation to its left that has remaining alternatives which could lead to success.

`excp` An exception event occurs when control leaves a procedure call because that call or one of its descendants has thrown an exception.

`call`, `exit`, `redo` and `fail` events correspond to the four ports in Byrd's box model [5]. `excp` events are useful for debugging code which throws exceptions. Whenever execution enters a call a `call` or `redo` event occurs. Whenever execution leaves a call an `exit`, `fail` or `excp` event occurs.

There are eight kinds of internal events. Their purpose is to record the outcomes of decisions about the flow of control, and to mark the boundaries of (possibly) negated contexts. The second kind were added specifically to support the declarative debugger.

`cond` A cond event occurs when execution reaches the start of the condition of an if-then-else.

`then` A then event occurs when the condition of an if-then-else succeeds, and execution reaches the start of the then part.

`else` An else event occurs when the condition of an if-then-else fails, and execution reaches the start of the else part.

`nege` A negation enter event occurs when execution reaches the start of a negated goal.

`negf` A negation failure event occurs when a negated goal succeeds, which means that the negation failed.

`negs` A negation success event occurs when a negated goal fails, which means that the negation succeeded.

`disj` A disj event occurs when execution reaches the start of a disjunct in a disjunction.

`swtc` A switch event occurs when execution reaches the start of one arm of a switch.

At each event, the debugger has access to several kinds of information about the event. The event number uniquely identifies the event, and the call number uniquely identifies a specific invocation of a procedure. The event depth gives the number of ancestors linking the call to the initial invocation of `main`. The debugger of course knows the identity of the procedure within which the event occurs (the name of the predicate or function, its arity, its mode number, etc), and the list of the variables that are live at the time of the event, including their names, types and storage locations.

At each internal event, the debugger also has access to the *goal path*. This gives the identity of the subgoal associated with the event. For most kinds of internal events, the goal path identifies the goal that execution is about to enter when the event occurs, the exceptions being `negf` and `negs` events, for which it identifies the goal that execution has just left when the event occurs.

2.3 The Mercury declarative debugger

The Mercury declarative debugger is divided into three main components.

1. The *analyser* which searches a tree for bugs. The tree it searches is a representation of the execution of the program, and is an instance of the scheme proposed by Naish [32]. In this scheme the tree and the algorithm used to search the tree are separate components. The analysis algorithm has an abstract view of the tree. The tree need only have the property that if an erroneous node has only correct children, then it represents a bug in the program. This decoupling of search algorithm and search tree allows greater flexibility in our implementation. We will refer to the tree searched by the analyser as the *evaluation dependency tree* or EDT. This name is borrowed from the lazy functional declarative debugging community, and was first coined by Nilsson and Sparud [37]. Although its meaning is slightly different in our context (mainly because Mercury is not a lazy functional language) we have adopted the term because it aptly describes the trees we generate. We are able to do both missing and wrong answer diagnosis using the EDT. In addition we are also able to diagnose the cause of unexpected exceptions.

2. The *oracle* which the analyser queries to ascertain which nodes are erroneous and which nodes are correct. Normally the user will act as the oracle. However, the answers given by the user will be remembered and used if the oracle is asked the

same question twice. Users may also declare certain predicates or entire modules to be trusted. The oracle is able to answer questions about trusted predicates without consulting the user.

3. The *backend* which is responsible for generating the EDT. The nodes in the EDT are generated by replacing the callbacks which normally invoke the procedural debugger with callbacks which construct the nodes in the EDT.

2.3.1 The analysis algorithm

Following the ideas proposed by Naish [32], the analysis algorithm searches an abstract tree for a node which is buggy. To do this it only needs to know how to get the children of any node in the tree and whether a particular node is erroneous or not.

Naish [32] gives the following definition of a buggy node: A node is *buggy* if it is erroneous and has no erroneous children. The definition of erroneous depends on the actual tree which is being debugged. We will give a definition of erroneous nodes for our trees in section 2.3.2.

The algorithm proposed by Naish to find buggy nodes in a tree is shown (in Mercury syntax[2]:) in figure 2.1.

```
debug(Root, Bug) :-
    erroneous(Root),
    ( if
        child(Root, Child),
        debug(Child, Bug1)
    then
        Bug = Bug1
    else
        Bug = Root
    ).
```

Figure 2.1: Naish's top-down debugger.

Note that `(if` C `then` T `else` E`)` is declaratively equivalent to $(C \wedge T) \vee ((\neg\exists C) \wedge E)$ in Mercury, so the call to `child` can be backtracked into to produce more solutions.

[2]We show the analysis algorithms here using Mercury syntax, because in this form they are easier to reason about formally than they would be in a pseudo-imperative form. Later on, algorithms will be presented in an imperative pseudo-code so as to be more accessible to a general audience.

A *top-most buggy node* is a buggy node all of whose ancestors are erroneous. Naish [32] proves that his version of `debug` will find all top-most buggy nodes and only all top-most buggy nodes.

We use a modified version of Naish's algorithm (figure 2.2).

```
debug(Root, Bug) :-
    erroneous(Root),
    ( if pick_descendant(Root, Descendant) then
        ( if erroneous(Descendant) then
            debug(Descendant, Bug)
        else
            tree_minus(Root, Descendant, NewTree),
            debug(NewTree, Bug)
        )
    else
        Bug = Root
    ).
```

Figure 2.2: Our debugger.

`pick_descendant(Node, Descendant)` deterministically chooses an arbitrary descendant of `Node` (excluding `Node` itself). If `Node` has no descendants then `pick_descendant` fails. `tree_minus(Tree1, Tree2, Result)` is true iff `Result` is the tree `Tree1` with `Tree2` removed.

Our algorithm differs from Naish's in two respects:

1. Through customization of the implementation of `pick_descendant`, arbitrary search strategies can be used, so we are not limited to just top-down search.

2. Our algorithm is not complete (although as we will show it is sound). This means it is not guaranteed to find all buggy nodes, nor even all topmost buggy nodes. However it is guaranteed to find at least one buggy node if the root of the tree is erroneous. This behaviour is perfectly acceptable, since in general programmers only search for and fix one bug at a time.

Our algorithm is very similar to Pereira's Select&Query algorithm [39] in the way we separate the search strategy from the diagnosis algorithm.

We show soundness by induction over sets of trees of different sizes.

Proposition 1. *For all trees t,* `debug(`*t*`, Bug)` $\Rightarrow$ `Bug` *is buggy in t.*

Proof. Let T_n be the set of all trees of size less than or equal to n.

Base case: suppose $t \in T_1$ and `debug(`t`, Bug)` is true. Then `pick_descendant(`t`, Descendant)` is false for all values of `Descendant` because t has no descendants. Thus the `else` branch of the outer if-then-else must succeed which implies that `Bug` is buggy in t by the definition of buggy.

Now suppose

$$\forall t \in T_k(\texttt{debug}(t, \texttt{Bug}) \Rightarrow \texttt{Bug} \text{ is buggy in } t) \tag{2.1}$$

for some $k \geq 1$.

Suppose that $t \in T_{k+1} \setminus T_k$ and `debug(`t`, Bug)` is true. t has descendants, so `pick_descendant(`t`, Descendant)` will be true for exactly one value of `Descendant` (since `pick_descendant` returns only one descendant). If `Descendant` is erroneous then (2.1) tells us that `Bug` is buggy in the subtree rooted at `Descendant` since `Descendant` $\in T_k$. This implies that `Bug` is buggy in t, since the tree rooted at `Descendant` is a subtree of t. If `Descendant` is not erroneous then `Bug` is buggy in `NewTree`, since `NewTree` $\in T_k$. To show that `Bug` is also buggy in t we must consider whether `Descendant` is a child of `Bug` or not.

- If `Descendant` is not a child of `Bug`, then all the children of `Bug` in `NewTree` must also be children of `Bug` in t and since `Bug` is buggy in `NewTree` it must also be buggy in t.
- If `Descendant` is a child of `Bug`, then `Bug` is buggy in t, because `Descendant` is not erroneous.

□

To prove that the debugger will find at least one buggy node if the root is erroneous we again use induction over T_k.

Proposition 2. *If t is erroneous then there exists some* `Bug` *such that* `debug(`t`, Bug)`.

Proof. Base case: suppose $t \in T_1$ and t is erroneous, then `debug(`t`, t)` succeeds.

Now suppose

$$\forall t \in T_k(\texttt{erroneous}(t) \Rightarrow \exists \texttt{ Bug such that } \texttt{debug}(t, \texttt{Bug})) \tag{2.2}$$

for some $k \geq 1$.

Let $t \in T_{k+1} \setminus T_k$ and suppose that t is erroneous.

t has descendants, so there exists exactly one `Descendant` such that `pick_descendant(`t`, Descendant)` is true. If `Descendant` is erroneous then because `Descendent` $\in T_k$ there exists a `Bug` such that `debug(Descendant, Bug)` is true and so too then is `debug(`t`, Bug)`. If `Descendant` is not erroneous, then the root of `NewTree` will be erroneous (since t is erroneous and t and `NewTree` share the same root) and thus there is a `Bug` such that `debug(NewTree, Bug)` is true because `NewTree` $\in T_k$, which means `debug(`t`, Bug)` is true. □

Note that the soundness of the algorithm is independent of the search strategy used. This gives us great flexibility to try out different search strategies without risking the soundness of our debugger.

It should also be noted that `debug` is merely an abstraction of the actual Mercury program that implements the analyser. For example in `debug` above, the result of calls to `erroneous` depend only on the node passed to it. In actual fact the result also depends on the state of the oracle as well as the I/O state (since the I/O state must be consulted if the user is to act as the oracle). Since in Mercury even I/O is handled declaratively, the I/O state must be threaded through calls to the `erroneous` predicate. The proof of soundness presented above serves only to show that our implementation is based on a sound design. It by no means proves that our implementation is bug free. In fact bugs found in the declarative debugger have proved useful case studies. (We can use the declarative debugger on itself as long as we are careful not to trigger the bug we are trying to find.)

At any point during the execution of the analysis algorithm, we call nodes which have not yet been eliminated from the tree *suspects*. We call the set of suspect nodes the *suspect set*.

2.3.2 The evaluation dependency tree

In this section we describe the EDT in more detail. We describe what each node in the tree represents as well as what constitutes an erroneous node. We also define what the children of any particular node are.

EDT nodes

During the execution of a program a call can succeed and produce a solution by binding its output arguments. A call can also fail, possibly after producing a number of solutions.

Every time a call succeeds there will be a corresponding `exit` event in the execution trace. The existence of each `exit` event makes an assertion about the semantics of the program. The assertion made is that the solution generated by the call is in the intended interpretation of the program.

Every time a call fails the program executes a `fail` event, which implicitly makes the assertion that the (possibly empty) set of all previous solutions generated by the call is a superset of all the expected solutions. There may be solutions which are not expected, however, the presence of `exit` events for unexpected solutions does not affect the truth of the assertion associated with the `fail` event. The assertion associated with a `fail` event will be false if and only if there is a solution *missing* from the solution set.

We will consider calls which abort, resulting in an `excp` event, later on in this chapter.

For now, each node in the EDT corresponds to the assertion made by an `exit` or `fail` event in the program. We consider a node *erroneous* if the assertion it makes is inconsistent with the intended semantics of the program. We call the nodes associated with `exit` events *wrong answer* nodes and the nodes associated with `fail` events *missing answer* nodes.

We define the nodes in the EDT in terms of the events generated by the procedural debugger because this allows us to implement the declarative debugger on top of the existing infrastructure of the procedural debugger. This also allows us to more closely integrate the procedural and declarative debuggers, allowing the user to switch between the two at will.

In order to determine if a node is erroneous or not, the analyser will ask the user questions about the assertion made by the associated event.

For wrong answer nodes the question will be of the form

```
sort([1, 4, 2, 3], [1, 2, 3, 4])
Valid?
```

where the displayed atom is the solution generated at the `exit` event.

Missing answer questions are of the form

```
Call member(_, [1, 2, 3])
Solutions:
        member(1, [1, 2, 3])
        member(2, [1, 2, 3])
        member(3, [1, 2, 3])
Complete?
```

or for calls which fail without producing any solutions:

```
Call member(_, [])
Unsatisfiable?
```

The oracle answers 'yes' or 'no' to each question, with an answer of 'no' indicating that the node is erroneous.

The alternative way to ask about failed calls is to require the oracle to give valid instances of the call which failed (as done by Shapiro [45] and Lloyd [24]). If there is no matching clause instance with a valid body, then there is a bug in the completion of the program.

Naish [31] points out that these questions can be much harder to answer since they require the user (if the user is acting as the oracle) to think of valid instances. It is much easier to examine a set of instances and decide if that set is complete or not, than to have to give the set from scratch. It is also very difficult to give correct instances by hand if the terms involved are large. Pereira's debuggers [39, 40] adopt the same approach as we do and display all generated solutions to the user.

It now remains for us to define what exactly the children of any particular node are.

EDT children

The children of any node in the EDT are the `exit` and `fail` events generated by child calls which could have affected the result of the parent call. Thus if a node is erroneous and all its children are correct, then there must be a bug in the definition of the predicate the erroneous node relates to, since the correct results of the children are combined together in the erroneous predicate's definition to form an erroneous result.

In the absence of negations and if-then-elses, the children of an `exit` node will be the last `exit` events generated by all non-backtracked-over child calls before the parent `exit`. (A child call is backtracked-over if a failure after the call causes execution to return to a goal to the left of the call, or to an alternate disjunct if the call is in a disjunction.) Only the solutions generated by these events are used in the calculation of the answer given at the parent `exit`. The tree generated under an `exit` node corresponds to the *proof tree* [25] for the atom at the `exit` event.

In the absence of negations and if-then-elses, the children of a `fail` node will be *all* the `exit`s resulting from child calls which produced variable bindings, as well as *all* the `fail` nodes resulting from child calls which failed. This is because if a child call succeeded, producing an `exit` event, then it may have succeeded with a wrong answer, the correct version of which might have caused the parent call to succeed instead of fail. Child calls

which succeed, but do not produce any variable bindings, cannot be the cause of the parent call failing, so they are excluded. Such calls could cause the parent call to fail earlier, but they cannot be the cause of any missing answers in the parent. On the other hand if a child call fails, producing a `fail` event, then it may have missed a solution which in turn could have caused the parent call to succeed instead of fail.

To illustrate, consider the following predicate.

```
common_element(List1, List2, Common) :-
    member(Common, List1),
    member(Common, List2).
```

Suppose `common_element` is called with `List1` bound to `[1, 2, 3]` and `List2` bound to `[4, 2, 5]` then the sequence of events generated inside `common_element` would be as follows (event numbers start at 3 because of events generated before the call to `common_element`).

Event#	Call#	Port	Atom
3	2	`call`	`common_element([1, 2, 3], [4, 2, 5], _)`
4	3	`call`	`member(_, [1, 2, 3])`
5	3	`exit`	`member(1, [1, 2, 3])`
6	4	`call`	`member(1, [4, 2, 5])`
7	4	`fail`	`member(1, [4, 2, 5])`
8	3	`redo`	`member(_, [1, 2, 3])`
9	3	`exit`	`member(2, [1, 2, 3])`
10	5	`call`	`member(2, [4, 2, 5])`
11	5	`exit`	`member(2, [4, 2, 5])`
12	2	`exit`	`common_element([1, 2, 3], [4, 2, 5], 2)`

Since in this case `common_element` succeeds, only the last `exit` events of the calls to `member` are included as children of the `exit` node for the call to `common_element`, since only they contribute to the found solution. This results in the EDT shown in figure 2.3.

Suppose a call to the right of the call to `common_element` then failed, causing a `redo` into `common_element`. This would add the following events to the trace.

Event#	Call#	Port	Atom
15	2	`redo`	`common_element([1, 2, 3], [4, 2, 5], _)`
16	3	`redo`	`member(_, [1, 2, 3])`
17	3	`exit`	`member(3, [1, 2, 3])`
18	7	`call`	`member(3, [4, 2, 5])`

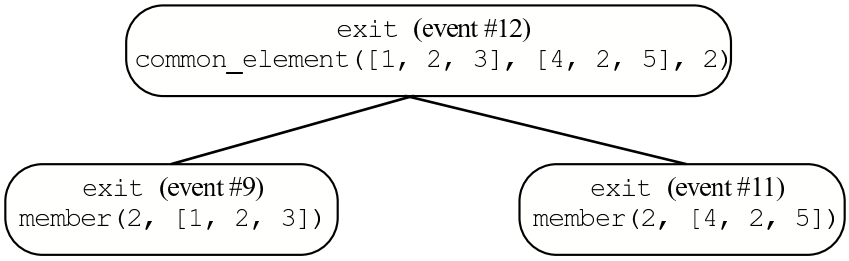


Figure 2.3: The EDT for a succeeded call to `common_element`

```
19      7      fail    member(3, [4, 2, 5])
20      3      redo    member(_, [1, 2, 3])
21      3      fail    member(_, [1, 2, 3])
22      2      fail    common_element([1, 2, 3], [4, 2, 5], _)
```

The children of the `fail` event for `common_element` would be *all* the `exit` and `fail` events generated by the calls to `member`, except event number 11, since it succeeds, but doesn't produce any output. The EDT for the `fail` at event number 18 is shown in figure 2.4.

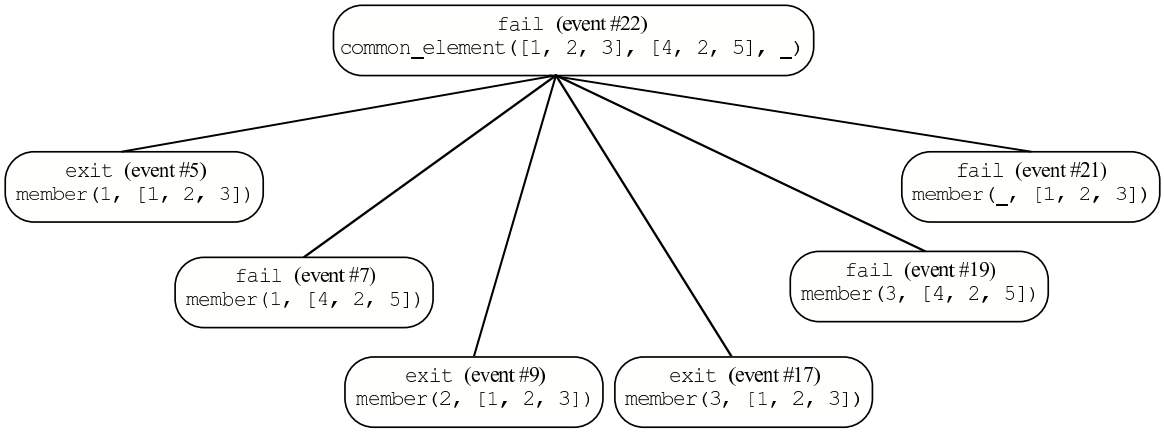


Figure 2.4: The EDT for a failed call to `common_element`.

Handling negation

Negations and if-then-elses complicate matters. This is because the criterion that governs which events are of interest inside a negated goal can be different from the criterion that

```
:- pred danger(animal::in, danger::out) is nondet.
danger(Animal, Danger) :-
    ( if
        ( fangs(Animal) ; poisonous(Animal) )
    then
        Danger = high
    else
        (
            hungry(Animal),
            fangs(Animal),
            Danger = high
        ;
            not hungry(Animal),
            Danger = low
        ;
            fangs(Animal),
            Danger = medium
        )
    ).

:- pred fangs(animal::in) is semidet.
fangs(snake).
fangs(lion).

:- pred hungry(animal::in) is semidet.
hungry(lion).

:- pred poisonous(animal::in) is semidet.
poisonous(snake).

:- pred afraid(danger::in) is semidet.
afraid(high).
afraid(medium).
```

Figure 2.5: Example program fragment with an if-then-else, disjunction and negation.

decides what events are of interest outside the negated goal. An if-then-else can have the same effect as a negation if its condition fails. Recall that, in Mercury, `(if` C `then` T `else` E`)` is declaratively equivalent to $(C \wedge T) \vee ((\neg\exists C) \wedge E)$, although operationally the condition is evaluated only once.

To illustrate consider the program fragment in figure 2.5.

Suppose the goal `( danger(caribou, Danger), afraid(Danger) )` appears in the body of the `main` predicate. The call to `danger` in this goal will result in the sequence of events depicted in figure 2.6.

Notice that the set of events for a given procedure call is not necessarily in a contiguous block, but may be interspersed with events from other calls, from outside as well as inside the call-tree of the call concerned. For example, the `exit` and `fail` events generated by the call to `danger` (call 2) are interspersed with the events of the call to `afraid` (call 7) as well as e.g. the calls to `fangs` (calls 3 and 8).

Event #	Call #	Port		Atom
3	2	call		danger(caribou, _)
4	2	A cond		
5	2	A B disj		
6	3	A B call		fangs(caribou)
7	3	A B **fail**		fangs(caribou)
8	2	A B disj		
9	4	A B call		poisonous(caribou)
10	4	A B **fail**		poisonous(caribou)
11	2	A **else**		
12	2	A disj		
13	5	A call		hungry(caribou)
14	5	A **fail**		hungry(caribou)
15	2	A disj		
16	2	A nege		
17	6	A C call		hungry(caribou)
18	6	A C **fail**		hungry(caribou)
19	2	A **negs**		
20	2	**exit**		danger(caribou, low)
21	7	call		afraid(low)
22	7	**fail**		afraid(low)
23	2	redo		danger(caribou, _)
24	2	D disj		
25	8	D call		fangs(caribou)
26	8	D **fail**		fangs(caribou)
27	2	**fail**		danger(caribou, _)

Figure 2.6: Execution trace of a call to `danger`.

Suppose we are interested in finding the children of the `exit` node at event number 20. This `exit` corresponds to the atom `danger(caribou, low)`. We need to find the `exit` and

`fail` events, generated by child calls of the call to `danger`, whose corresponding assertions are sufficient to prove that `danger(caribou, low)` is true in the semantics of the program.

Clearly we could make *all* the child `exit` and `fail` events (events 7, 10, 14 and 18) children in the EDT, since they are sufficient to prove `danger(caribou, low)` is true in the semantics of the program. However the `fail` at event number 14 is irrelevant, since the truth of the disjunct in which that `fail` event occurs is irrelevant to the truth of the body (because another disjunct in the disjunction is true). However, *all* the `fail` events in the if-then-else condition (events 7 and 10) are relevant, since if any of these was an `exit` instead of a `fail` (i.e. if `fangs(caribou)` or `poisonous(caribou)` were true) then the condition of the if-then-else would have succeeded, which would have changed the outcome of the `exit` at event number 20.

In practice we are able to eliminate certain nodes which are not necessary to prove the result of the parent, but not all such nodes all of the time. The child nodes we create will always, however, be *sufficient* to prove the assertion of the parent node in the semantics of the program.

In order to describe exactly how the children of a particular node in the EDT are derived in the presence of negations and if-then-elses it is helpful to first consider an intermediate tree structure, which we will call the *raw EDT*.

Raw EDT nodes

`exit` and `fail` events are not the only types of events that have assertions about the semantics of the program associated with them. The events which signal that execution is leaving a negated goal also have assertions about the negated goal associated with them. These include `negs` and `negf` events which indicate that a negation has succeeded or failed respectively (i.e. the negated goal failed or succeeded respectively), as well as `else` events which indicate that the condition of an if-then-else has failed.

The presence of a `negs` event asserts that the negated goal, with all substitutions computed at the time of the corresponding `nege` event applied, is unsatisfiable in the semantics of the program. The presence of a `negf` event asserts that the negated goal, with all substitutions computed at the time of the corresponding `nege` event applied, is satisfiable in the semantics of the program. The presence of a `else` event asserts that the condition of the if-then-else, with all substitutions computed at the time of the corresponding `cond` event applied, is unsatisfiable in the semantics of the program.

`negs` and `else` events are like missing answer nodes, since they indicate that the goal

inside the negation or if-then-else condition has failed, while `negf` events are like wrong answer nodes, since they indicate that the goal inside the negation has succeeded.

We call events of type `exit`, `fail`, `negs`, `negf` or `else` *assertion* events. In figure 2.6 the assertion events are in a bold font.

Each node in the raw EDT corresponds to one of these assertion events.

Raw EDT children

In order to define how the children of node in the raw EDT are obtained, we first need some additional concepts.

We define the events *inside* the call identified by call c to consist of the events generated by call c and the events generated by the calls that are the direct and indirect descendants of call c. This concept is useful because a bug exhibited by call c cannot be caused by any event outside call c, so when diagnosing such a bug, the debugger can restrict its attention to the events inside c. In the example given in figure 2.6, the events inside call 2 (the only call to `danger`) consist of the events generated by calls 2 – 6 and 8, while the events inside call 3, the first call to `fangs`, consist only of the events generated by call 3.

When looking for the raw EDT children of an `exit` or `fail` event generated by call c, not all the events inside call c are needed at once. The ones which are of immediate concern are the *body events* of c, which consist of

- the internal events of call c, and
- the interface events of the children of call c.

These events represent the execution of (all or part of) the bodies of the predicate called by call c. Note that although the interface events of c are "inside" c, they are not counted as "body events".

In a set of body events, the internal events relating to negated goals come in pairs: `nege`–`negs`, `nege`–`negf`, and `cond`–`else`. (There are also `cond` events which are matched not by an `else` event but by one or more `then` events. These `cond` events are not related to negated goals, because the condition of the if-then-else has not failed.) These pairs divide the body events of calls into segments. We will say that a subsequence of the body events of a call is a *negated context* if the subsequence includes all the body events of the call chronologically between such a pair of *delimiter* or *anchor* events, but not the anchor events themselves.

We define a *context* to be either a negated context or a segment of body events for some call c which is delimited by the `call` event of call c and either an `exit` event or a `fail` event of call c. Contexts can have other contexts nested inside them, where the nesting site is a pair of events anchoring a negated context (the anchor events are considered part of the outer context only). This nesting is somewhat simpler than the nesting of one procedure call inside another: while execution can leave a call and then later backtrack into it, execution can never backtrack into a negated context, because, in Mercury, negated goals cannot bind non-local variables, and therefore cannot succeed more than once.

In our example in figure 2.6, we can identify four contexts: A, B, C and $A \cup D$.

We define a *stratum* to be the set $S \setminus \bigcup_i c_i$, where S is the set of events from some context, and the c_i are the sets of events from all the contexts nested in S. A stratum clusters together events for goals which have been negated by the same set of constructs in the procedure. This set may of course be the empty set. All strata are anchored at the end by an event of type `exit`, `fail`, `else`, `negs` or `negf`. Conversely, events of these types always have a matching `call`, `cond` or `nege` event respectively earlier in the same body, so each `exit`, `fail`, `else`, `negs` or `negf` event anchors the end of a (unique, possibly empty) stratum. We can therefore identify a stratum by the event which anchors it at the end.

The body events of a call are important because they tell us *how* the results from its children were put together to derive the call's result. However, due to the presence of negation, we must treat different body events differently depending on their context. The reason why it is useful to consider just the events in a single stratum is that the events of a single stratum *are* treated uniformly.

The two events anchoring a stratum define a goal. If the stratum is delimited by a `call`–`exit` or a `call`–`fail` pair, the goal is the body of the relevant procedure. If the stratum is delimited by a `cond`–`else` pair, the goal is the condition of the if-then-else identified by their goal paths inside the relevant procedure. If the stratum is delimited by a `nege`–`negs` or a `nege`–`negf` pair, the goal is the goal inside the negation identified by their goal paths inside the relevant procedure.

The assertions made by all the assertion events inside a stratum are sufficient to explain the assertion made by the event anchoring the end of the stratum, though they are not always necessary.

In our example in figure 2.6 we can identify four strata:

1. B (anchored by event number 11),
2. C (anchored by event number 19),

3. $A \setminus (B \cup C)$ (anchored by event number 20) and

4. $[A \setminus (B \cup C)] \cup D$ (anchored by event number 27).

When diagnosing a missing answer to the goal defined by a stratum, the declarative debugger is interested in all the events in the stratum, regardless of whether they led to a solution or not. However, when diagnosing a wrong answer, we are interested only in events that describe how that particular answer was computed; alternative disjuncts or exits that did not lead to the wrong answer can be ignored. The events of interest are those which represent how forward execution generated the solution—any events which represent backtracking, or which had been backtracked over at the point the solution was generated, are not relevant to the solution. We call a sequence of stratum events like this a *contour*. Most event types can appear in a contour; however, `redo`, `fail` and `negf` events indicate that backtracking has happened or is about to happen, so these events are never part of a contour.

The assertion events lying on a contour are sufficient to explain the existence of the `exit` or `negf` event at the end of the contour.

We therefore define the children of an `exit` or `negf` node in the raw EDT to be the assertion events lying on the contour leading up to that `exit` or `negf` event. We define the children of a `fail`, `negs` or `else` node to be all the assertion events, except `exit` events which produce no variable bindings and `negs` events, on the stratum anchored at the end by the parent `fail`, `negs` or `else` event.

We do not include `exit` events which produce no variable bindings or `negs` events (which *cannot* produce any variable bindings), since these cannot be responsible for the failure of the goal associated with the stratum.

Applying this definition to our example in figure 2.6 yields the raw EDT depicted in figure 2.7 for the `exit` node at event number 20, and the raw EDT depicted in figure 2.7 for the `fail` node at event number 27. Note that the `negs` at event number 19 is not a child of the `fail` node at event number 27. This is because the `negs` event cannot produce any bindings, and so even if it was a `negf` event instead (i.e. the negated goal had succeeded instead of failing), this would not have changed the fact that the parent call failed.

Obtaining the EDT from the raw EDT is simply a matter of removing the `negs`, `negf` and `else` nodes and adding any children of these nodes to the parent node. Applying this procedure to the raw EDTs in figures 2.7 and 2.8 yields the EDTs in figures 2.9 and 2.10 respectively.

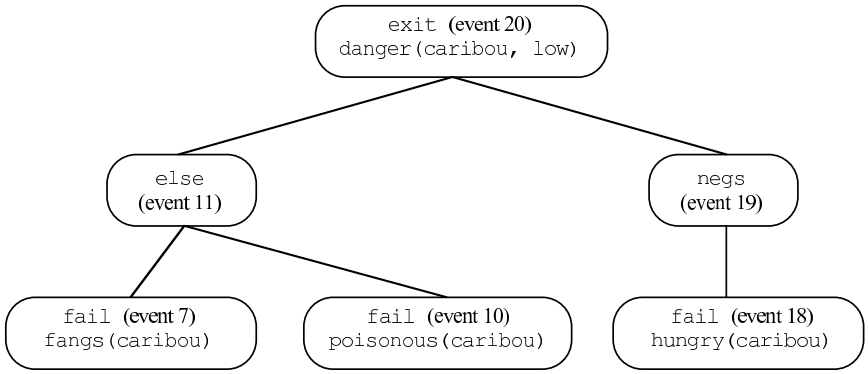


Figure 2.7: The raw EDT for the `exit` node at event number 20.

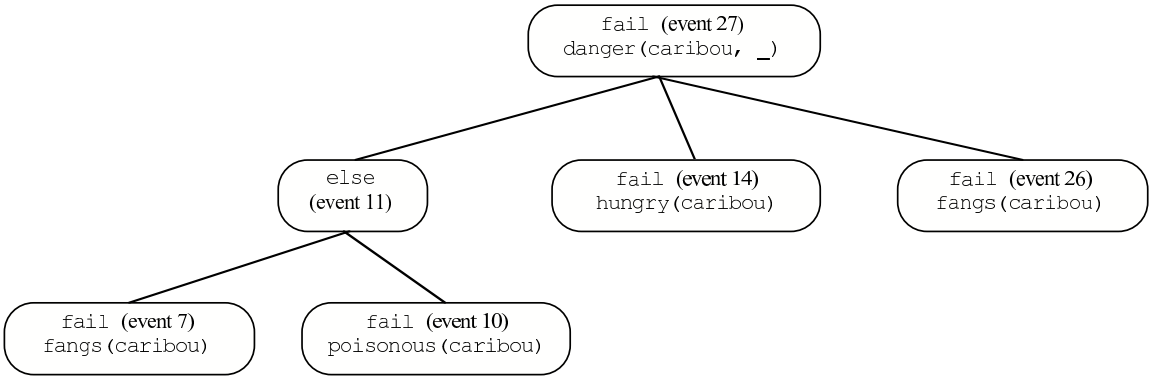


Figure 2.8: The raw EDT for the `fail` node at event number 27.

Why do we search for bugs in the EDT, instead of the raw EDT? The advantage of using the raw EDT to search for bugs is that we gain finer grained bug location — we can localise the bug to a specific negation or if-then-else condition, whereas with the EDT the bug can only be localised to the body of a predicate. The price we pay for this increased accuracy, however, is a bigger search space (since in general the raw EDT will have more nodes than the EDT), and harder questions for the oracle to answer. The questions will be harder for the user to answer because in general the user will not assign an intended interpretation to a negated goal (which may consist of many atomic goals). The user has an intended interpretation for each predicate, but not for each negated goal. To answer questions about negated goals the user will also need to know the context of the negated goal, since an instance of a goal may be correct in the body of one predicate and erroneous in the body of another. We therefore sacrifice the ability to localise a bug to a specific negation or if-then-else condition, for fewer, simpler questions.

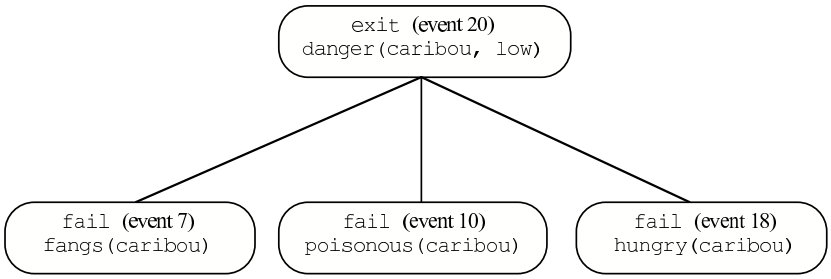


Figure 2.9: The EDT for the `exit` node at event number 20.

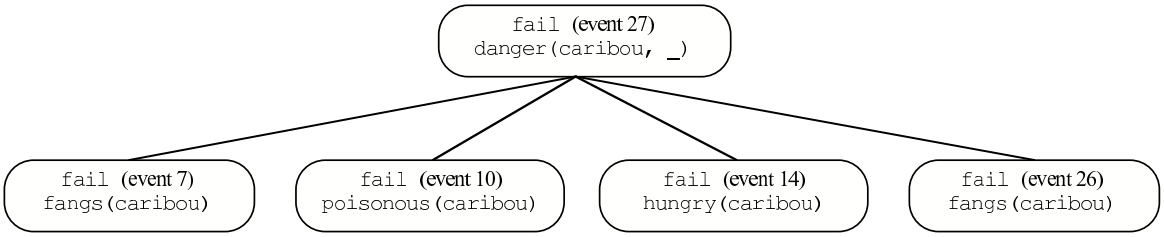


Figure 2.10: The EDT for the `fail` node at event number 27.

A special case

There is one notable exception to the rules for determining the children of a node in the raw EDT which we have described above. The exception is the `solutions` builtin predicate.

`solutions(Goal, Solutions)` is true iff `Solutions` is a list of all the solutions to the procedure `Goal`. For example suppose a predicate `p` is defined as follows.

```
p(a).
p(b).
p(c).
```

A call to `solutions(p, Solutions)` would bind `Solutions` to the list `[a, b, c]`. The resulting execution trace would be as follows.

Event#	Call #	Port	Atom
3	2	`call`	`solutions(p, _)`
4	3	`call`	`p(_)`
5	3	`disj`	
6	3	`exit`	`p(a)`
7	3	`redo`	`p(_)`
8	3	`disj`	

```
9       3       exit        p(b)
10      3       redo        p(_)
11      3       disj
12      3       exit        p(c)
13      3       redo        p(_)
14      3       fail        p(_)
15      2       exit        solutions(p, [a, b, c])
```

Now suppose we wished to find the children of the `exit` node for the call to `solutions` (event number 15). Normally to find the children of an `exit` node we would look at the contour leading up to that `exit`, however in this case we want to include *all* `exit` and `fail` events for `p` in the children of the `exit` node for `solutions`. This is because if any of the assertions made by any of `p`'s `exit`s is wrong then the list produced by `solutions` would contain an incorrect element, and if the assertion made by the `fail` event is wrong then the list returned by `solutions` would contain too few elements.

Thus we make a special case of calls to `solutions` (and related builtin predicates) and collect *all* assertion events on the stratum anchored by an `exit` event of a call to `solutions`.

Making a special case of these builtin predicates is okay, since they are not reproducible using pure Mercury code. It *is* however possible to create similar predicates using Mercury's impurity system. Currently we do not support declarative debugging of impure code. Section 7.5 explores the possibility of supporting impurity in the future.

2.3.3 The annotated trace

We now concern ourselves with how we can efficiently construct the raw EDT.

We construct the raw EDT on demand, based on a representation of the execution we call the *annotated trace* [4]. The annotated trace is like a chronologically ordered linked list of trace events, with two main differences. The first is that we only record events down to a given depth; if the analysis algorithm needs to know about EDT nodes and hence events below this depth, then it will use the machinery of the procedural debugger's '`retry`' command [46, 47] to repeat that part of the execution, this time recording events to a deeper level. Nodes below the depth limit are said to be in an *implicit* subtree. These nodes are not represented (or materialized) in memory. Execution traces can consist of hundreds of millions of events, so doing this allows us to trade-off the space required to store

the annotated trace against the time required to construct it. This space-time trade-off is explored in detail in chapter 5.

The second difference is that we maintain extra links between nodes in the annotated trace. These links make it easy and efficient to step through strata and contours in the annotated trace. This allows us to efficiently find the children of any node in the raw EDT and thus the EDT.

The nodes of the annotated trace

Each node in the annotated trace corresponds to an event generated in the program. Each node is represented by a structure whose fields point to other nodes in the annotated trace or which give information about the node. Each event type has different fields.

- All nodes have a `preceding` field which points to the node of the immediately preceding event; this field is `NULL` for the first event in the trace.
- The nodes of internal events have a `goal_path` field that holds the goal path of the event. This allows us to identify the goal associated with an internal event and is particularly useful for subterm dependency tracking, which is described in more detail in section 4.3.
- Nodes for `disj` events have a `first_disj` field, which contains the event number for the `disj` event generated when execution enters the first disjunct of that disjunction. In the node for the `disj` event for the first disjunct, this field is thus self-referential.
- The nodes of `else`, `negs` and `negf` events have a `context_start` field that identifies the start of the negated context they represent the end of. For `else` events, it points to the node of the corresponding `cond` event; for `negs` and `negf` events, it points to the node of the corresponding `nege` event.
- The nodes of `exit` and `fail` events have a `call` field that points to the node of the corresponding `call` event.
- The nodes of `exit` and `fail` events also contain a `redo` field. If the call has succeeded before, this field points to the node of the `redo` event that has asked for another solution to be generated; if the call has not produced any previous solutions, this field will be `NULL`.

```
step_in_stratum(this) returns next is

switch on the type of this node
    case call:  raise an exception
    case exit:  If this.redo = NULL
                        next := this.call.preceding
                Else
                        next := this.redo.preceding
    case redo:  next := this.exit.preceding
    case fail:  If this.redo = NULL
                        next := this.call.preceding
                Else
                        next := this.redo.preceding
    case cond:  next := this.preceding
    case then:  next := this.preceding
    case else:  next := this.context_start.preceding
    case nege:  raise an exception
    case negs:  next := this.context_start.preceding
    case negf:  next := this.context_start.preceding
    case swtc:  next := this.preceding
    case disj:  next := this.preceding
```

Figure 2.11: Algorithm for stepping to the previous event in a stratum.

- At each `call` node we also store the identity of the procedure being called as well as references to the values of the input arguments of the call.
- At `exit` nodes we store references to the output arguments of the atom at the `exit` event. This, together with the input arguments and procedure information stored at the `call` node allow us to display questions to the user.
- At `redo` nodes we store a reference to the `exit` node corresponding to the last generated solution of the call which is being redone.

These links are enough to be able to efficiently step through any stratum or contour in the execution trace.

Figure 2.11 shows the algorithm for stepping one node to the left in the current stratum. If we encounter an `exit` or `fail` event then the next event in the stratum will be the last `redo` for that call, or if there is are no `redo`s then it is the corresponding `call` event. The step_in_stratum function skips over `call` events, `cond` events for failed if-then-else

```
step_left_in_contour(this) returns next is

switch on the type of this node
      case call:  raise an exception
      case exit:  next := this.call.preceding
      case redo:  raise an exception
      case fail:  raise an exception
      case cond:  next := this.preceding
      case then:  next := this.preceding
      case else:  next := this.context_start.preceding
      case nege:  raise an exception
      case negs:  next := this.context_start.preceding
      case negf:  raise an exception
      case swtc:  next := this.preceding
      case disj:  next := this.first_disj.preceding
```

Figure 2.12: Algorithm for stepping to the left in the current contour.

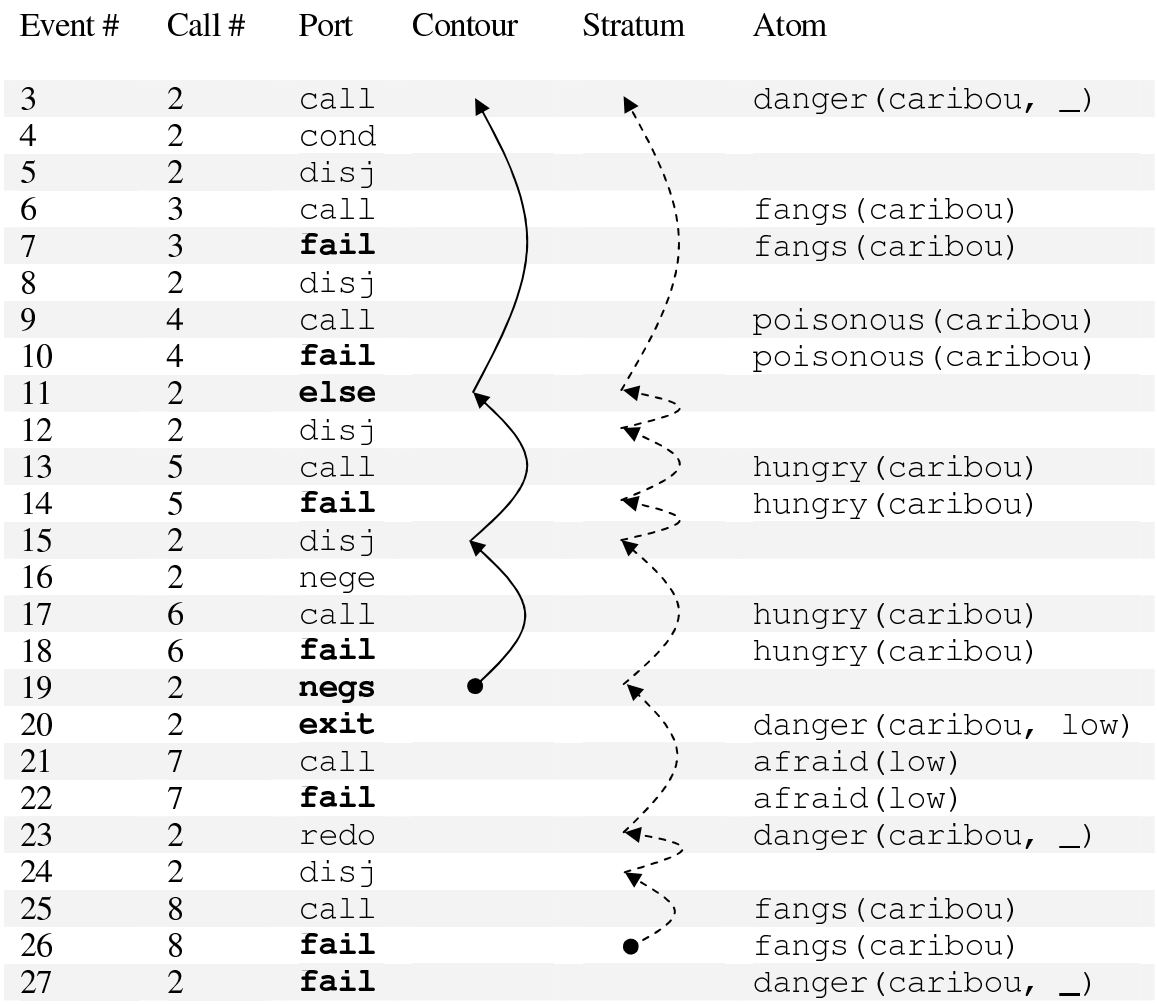
conditions and `nege` events in the stratum. These events don't represent nodes in the EDT.

This function can be used to find the children of a `fail`, `negf` or `else` node in the raw EDT by repeatedly applying it to the node preceding the `fail`, `negf` or `else` node until either a `call`, `nege` or `cond` node is found. Along the way all the assertion events in the stratum, excluding `exit` events which don't produce any output and `negs` events, are gathered as children. Note that we abort in the cases where *this* is a `call` or `nege` event, since step_in_stratum should never be called with such an event (since we always step over them). `cond` events, on the other hand, are handled. This is because the condition may have succeeded which means the condition is not in a negated context. If the condition failed and we are stepping through the stratum anchored at the corresponding `else`, then we will stop as soon as step_in_stratum returns a `cond` event.

The "Stratum" column in figure 2.13 illustrates the nodes traversed through repeated applications of the step_in_stratum function, starting with event number 26. The events traversed are all from the stratum anchored by event number 27. Although some of the events in this stratum are skipped (such as event number 13), all *assertion* events in the stratum are traversed.

Figure 2.12 shows that algorithm for stepping one node to the left in the current contour. This function can be used to find the children of an `exit` or `negs` node in the raw EDT. To

find the children of an `exit` or `negs` node we simply repeatedly apply step_left_in_contour to the event preceding the `exit` or `negs` event until we encounter a `call` or `nege` event, gathering all assertion events as we go. Note that we don't stop at `cond` events as we did with step_in_stratum, since returned `cond` events can only represent succeeded if-then-else conditions (this is because if the condition had failed, then we would be looking along the stratum, not the contour). step_left_in_contour should never be called with a `call`, `redo`, `fail`, `nege` or `negf` event since all such events will be jumped-over by the step_left_in_contour function.



Event #	Call #	Port	Atom
3	2	call	danger(caribou, _)
4	2	cond	
5	2	disj	
6	3	call	fangs(caribou)
7	3	**fail**	fangs(caribou)
8	2	disj	
9	4	call	poisonous(caribou)
10	4	**fail**	poisonous(caribou)
11	2	**else**	
12	2	disj	
13	5	call	hungry(caribou)
14	5	**fail**	hungry(caribou)
15	2	disj	
16	2	nege	
17	6	call	hungry(caribou)
18	6	**fail**	hungry(caribou)
19	2	**negs**	
20	2	**exit**	danger(caribou, low)
21	7	call	afraid(low)
22	7	**fail**	afraid(low)
23	2	redo	danger(caribou, _)
24	2	disj	
25	8	call	fangs(caribou)
26	8	**fail**	fangs(caribou)
27	2	**fail**	danger(caribou, _)

Figure 2.13: An example contour and stratum.

The "Contour" column in figure 2.13 shows the nodes traversed through repeated applications of the step_left_in_contour function, starting with event number 19. The events

traversed are all from the contour anchored by event number 20. Notice how alternate disjunctions, for example the disjunct at event number 12, are skipped, since they are not relevant to the answer computed at event number 20.

Constructing the annotated trace

Nodes are added to the annotated trace in chronological order. This means that whenever a new node is added to the annotated trace, the nodes which the fields of the new node must point to are already in the trace. More importantly, with the exception of the `preceding` field, all the fields will point to nodes in the same stratum as the node being added.

To populate the `previous` field we simply keep a record of the previous node which was added to the trace. If the new node is a `call`, `cond`, `then`, `swtc` or `nege` node then there are no other fields to populate.

To populate the fields of the other nodes we first need to find a previously added node which is on the same stratum. We can then apply the step_in_stratum function to traverse the stratum.

Note that if we are inside an if-then-else condition, then it doesn't matter whether the condition failed or not, as all the nodes required to populate the fields of nodes added inside the condition will themselves be inside the condition, so for the purposes of populating the fields of each node, we can consider `cond` nodes to mark the start of a nested negated context, even though we don't necessarily know if the condition has failed or succeeded yet.

For `exit`, `fail`, `negs`, `negf` and `else` nodes there are two cases to consider.

1. The previous node is in the same stratum as the node being added. This will only happen if the previous node is a `call`, `redo`, `nege` or `cond` node. If the previous node is a `call`, `nege` or `cond` then it can be used to populate the field of the current node. If the previous node is a `redo` then we can look up the previous `exit` for the call by looking up the appropriate field in the `redo` node. From the `exit` we can then look up the `call` node.

2. The previous node is in a stratum one level down from the node being added. This will happen if the previous node is anything other than a `call`, `redo`, `nege` or `cond`. In this case we must first traverse the nested stratum until we find a `call`, `redo`, `nege` or `cond` from the stratum the current node is in. This can be done by repeatedly

applying step_in_stratum starting with the preceding node, until a `call`, `redo`, `nege` or `cond` node is returned. See below for an optimisation.

For `redo` nodes we must find the `exit` node which corresponds to the previous solution generated by the call. The node preceding a `redo` will either be in the same stratum or in a stratum one level up. The preceding node cannot be in a nested stratum one level down, because `redo` events do not mark the ends of strata. The only time the previous event will be in a stratum one level up is when the preceding node is another `redo`. In this case we can follow the `exit` field of the preceding `redo`. The node preceding this `exit` must be on the same stratum as the `redo` being added. (The stratum anchored by this `exit` cannot be empty, because then there would be no call to redo.) Otherwise the `redo` being added, and its preceding node, must both be on the same stratum. Once we have a node on the same stratum we can traverse the stratum looking for the `exit` node corresponding to the last generated solution.

In practice we use a modified version of step_in_stratum to traverse the stratum when populating the fields of a newly added node. The modified version looks for the first event in the stratum which is also on a contour. We can then use step_left_in_contour instead which will skip irrelevant nodes. This optimisation is described in [4].

New portions of the annotated trace are constructed piecemeal, as and when they are needed by the analyser. We explore in detail the problem of deciding how much of the annotated trace to materialize at once in chapter 5.

2.3.4 Handling exceptions

It is very important that we be able to debug code which throws exceptions, since many bugs exhibit themselves by causing some assertion in the program to be violated, which then causes an exception to be thrown. This is especially true for Mercury where often the programmer is required, in order to satisfy a determinism declaration, to throw an exception in the branch of a disjunction or if-then-else which they know cannot be executed, but where the compiler is unable to infer that the branch is unreachable. If the branch is reached then the programmer made a mistake somewhere, and the value of the thrown exception helps narrow down where that mistake might be.

Exception EDT nodes

The existence of an `excp` event implicitly makes an assertion about the operational semantics of the program, though nothing about the declarative semantics is said. This assertion says that the associated procedure call is allowed to throw the thrown exception, given the values of its input arguments. We include `excp` events as nodes in the EDT.

Questions about `excp` nodes are of the form:

```
Call divide(1, 0, _)
Throws "division by zero"
Expected?
```

The user is expected to answer 'no', indicating that the node is erroneous, if the named exception should not have been thrown by the procedure call. This tells the declarative debugger that either (a) a different exception should have been thrown, or (b) no exception should have been thrown. In either case the bug lies in the definition of the predicate the question is about or in one of its descendants. The bug is either a missing `try` or `try_all`, an incorrectly calculated exception value, or some other buggy descendant which caused the call to `throw` to be executed when it should not have been.

It may also be the case that the user *expects* exceptions to be thrown in certain situations. For example the user may expect the input to `divide` above to sometimes be zero and they may have code which handles any `"division by zero"` exceptions. In this case the user would be expected to answer 'yes' to the question about the aborted call to `divide`. This asserts that the associated EDT node is correct in the intended (operational) semantics.

Exception node children

The children of an `excp` node in the EDT are the `exit`, `fail` and `excp` events generated by child calls which could have caused the exception to be thrown.

Backtracked over goals could not have been responsible for the thrown exception. This is because a backtracked over goal couldn't have thrown the exception itself (since then it wouldn't have been backtracked over), nor could any of the bindings it computed have been used in the computation which lead to the exception being thrown (since the bindings are lost on backtracking).

If the `excp` event does not occur in a negated context, then we use the same criteria for deciding the children of the `excp` event as we do for deciding the children of an `exit`

event, i.e. we include all assertion events (including child `excp` events) along the contour leading up to the `excp` event.

If the `excp` event is in a negated context then there will be at least one `cond` or `nege` event between the `excp` event and the corresponding `call` event. This `cond` or `nege` event will not have a corresponding `else`, `negf` or `negs` event, since the exception is thrown before that event can be generated. In this case we look for child nodes on the contour between the `excp` event and the `cond` or `nege` node marking the start of the negated context the exception is thrown in. If the `excp` event is inside further, nested, negated contexts, then we continue to look at the contours between the `cond` or `nege` events marking the beginning of nesting negated contexts. Finally we look at the contour between the `call` event and the `cond` or `nege` event marking the start of the outermost negated context the exception is thrown in. All these contours together explain how the exception came to be thrown in the body of the procedure. The step_through_contours function in figure 2.14 can be called repeatedly, starting with the event to the left of the `excp` event, and stopping when a `call` event is returned, to traverse the contour(s) leading up to the `excp` event.

step_through_contours(*this*) returns *next* is

If *this* is a `nege` or `cond` event
    next := *this*.preceding
Else
    next := step_left_in_contour(*this*)

Figure 2.14: Algorithm for stepping through the contours leading up to an exception which was possibly thrown in a negated context.

All assertion events on the contour(s) leading up to an `excp` node are counted as children of the `excp` node in the raw EDT. The raw EDT is then converted to the EDT in the usual manner.

For example consider a call to the following predicate

```
p :- q, not (r, throw(error)).
```

and suppose it generates the following execution trace.

Event#	Call#	Port	Atom
1	2	`call`	`p`

```
2       3       call        q
3       3       exit        q
4       2       nege
5       4       call        r
6       4       exit        r
7       5       call        throw(error)
8       5       excp        throw(error)
9       2       excp        p
```

The children of the node corresponding to event number 9 are events 8, 6 and 3. The `exit` events 6 and 3 are in different contexts, and therefore on different contours, however if either of the associated calls had behaved differently (for example by failing instead of succeeding), then the exception might not have been thrown.

The only exception to this rule is when we want to find the children of the `exit` node of a call to `try_all`. The reasons are the same as the reasons we treat calls to the builtin predicate `solutions` differently and are explained in section 2.3.2. The problem is solved in the same way: by collecting *all* the assertion events in the stratum anchored at the `exit` event generated by the call to `try_all`.

2.4 The relationship between the procedural and declarative debuggers

The Mercury declarative debugger is invoked from within the procedural debugger. Typically a user will have some idea of the procedure which is behaving erroneously. They would start the procedural debugger, set a breakpoint on the procedure and continue execution until the procedure is called. Once they have established that the procedure is indeed erroneous, they can then start the declarative debugger at an `exit` or `fail` event which exhibits the erroneous behaviour. The declarative debugger then reexecutes the erroneous call, generating the annotated trace as it does so.

The user is also permitted to start the declarative debugger at an `exit` or `fail` event which does not exhibit erroneous behaviour. Answering 'yes' to the question posed by the declarative debugger about this node will then cause the EDT to be expanded upwards. To do this an ancestor call of the call where the declarative debugger was started is reexecuted (exactly how far up this ancestor is can be controlled by the user). This allows the user to do a kind of bottom up search from a correct node in the EDT. As we will see later this is

particularly useful for debugging code which throws exceptions (see section 6.2).

The user may switch from the declarative debugger to the procedural debugger at any time. They can return to the procedural debugger either at the event corresponding to the current question, or at the event where the declarative debugger was started. If the user returns to the procedural debugger from the declarative debugger, they can resume their declarative debugging session (from the same question where they left off) at any time. This is useful if the user wishes to use the facilities of the procedural debugger in order to answer a declarative debugger question. For example the user might wish to trace execution through the body of a call before answering a declarative debugger question about the call.

Chapter 3

Usability features

In this chapter we will examine some of the features of the debugger that make it easier to use.

3.1 Inadmissibility

Often, for reasons of efficiency, the intended interpretation of a predicate will only be defined in the programmer's mind for a subset of the ground instances of the predicate.

For example consider a predicate `merge(List1, List2, List3)` which merges two sorted lists `List1` and `List2` to form the new sorted list `List3`.

Usually we have an intended interpretation of this predicate for all atoms where the first two arguments are sorted lists. However for atoms where the first two arguments are unsorted we might not have a clear intended interpretation. This is because we would not check that the two lists are sorted in the body of `merge`, because that would be too inefficient. We instead rely on any callers of `merge` to always pass it two sorted lists.

Suppose, during the course of a debugging session a question such as the following is asked:

```
merge([3, 2, 1], [4, 5, 6], [3, 2, 1, 4, 5, 6])
Valid?
```

To answer the question we must know if the given atom is in the intended interpretation or not. Since we have no predefined intended interpretation for this atom (we did not anticipate such an atom while writing the program) we cannot know how to answer the question. We might be tempted to answer 'no', but this would direct the bug search down

the subtree rooted at this atom, which is not what we want, since the bug is not there — it is outside the subtree rooted at this atom. Giving an answer of 'yes', however will direct the bug search in the right direction. An answer of 'yes' asserts that the atom is valid in our intended interpretation. Now what if, instead of succeeding, the call failed and the following question was asked:

```
Call merge([3, 2, 1], [4, 5, 6], _)
Unsatisfiable?
```

Giving an answer of 'yes' would still direct the bug search in the right direction, but would mean that all ground instances of the atom are not in our intended interpretation. This contradicts our previous intended interpretation, in which the atom was valid. This situation is unsatisfactory, because it means we have to adapt what we say our intended interpretation is to the behaviour of the program. The program should be based on the intended interpretation and not the other way around!

The solution proposed by Naish [33] is to allow a third answer of '*inadmissible*' in addition to 'yes' and 'no'. Answering 'inadmissible' to a question allows the oracle to direct the bug search in the right direction, without making any commitment to the intended interpretation of the predicate. Answering 'inadmissible' is saying 'I don't have an intended interpretation for this atom, because the associated call shouldn't have occurred in the first place'.

This gives rise to a new type of bug: one where an erroneous node has one or more inadmissible children and all the other children are correct. Naish calls such bugs *i-bugs* [33] (bugs where an erroneous node has only correct children are called *e-bugs*).

Admissibility is closely related to Hoare style pre-conditions adapted for logic programs: if a call is inadmissible then one or more of its inputs violates a pre-condition of the call. Pedreschi and Ruggieri [38] define a *specification* as a pair of Herbrand interpretations $\langle Pre, Post \rangle$. Pre specifies the admissible atoms, and $Post$ specifies which admissible atoms are true and which are false. An atom may be in Pre, but not in $Post$ if it is admissible, but should fail. Atoms not in Pre should not occur at all during the execution of the program. By answering 'inadmissible' the user asserts that the atom being asked about is not in Pre. Often calls that throw exceptions are inadmissible because the inputs violate some precondition of the procedure.

Pereira also introduces inadmissible answers in his rational debugger [40], although he does not elaborate in any great detail on their usefulness.

In our case inadmissibility exposes opportunities to reduce the search space further. Instead of just eliminating the subtree rooted at an inadmissible node, as we do with correct nodes, we can eliminate all the nodes that do not contribute to the computation of the inputs to the inadmissible call.

As a first approximation we can eliminate all nodes resulting from calls that were executed after the inadmissible call. These nodes could not have been responsible for the inputs to the inadmissible call. Pereira makes a similar observation about inadmissible calls [39]. He uses this to reduce the search space in his Select&Query debugger.

A refinement would be to use slicing [51] to determine which calls prior to the inadmissible call could have participated in the computation of the inputs to the inadmissible call and eliminate all nodes not in this slice. The slice would have to include all nodes that could have influenced the values of the input arguments, not just the nodes through which the inputs were passed. Such a slice is called a relevant slice. An efficient method for determining such a slice is presented in [17].

These refinements warrant a new definition of i-bug. Instead of a node being i-buggy if it is erroneous with at least one inadmissible child and *all* other children are correct, we instead say a node is i-buggy if it is erroneous, has at least one inadmissible child, and all child nodes *that could have influenced the inputs to the inadmissible call* are correct. We do not care about nodes that could not have affected the inputs to the inadmissible call. Erroneous siblings that were not responsible for the inputs to the inadmissible call represent separate bugs.

We have not yet implemented these refinements in the Mercury declarative debugger , but we do have most of the slicing machinery in place already, so this would not be too hard to do (see section 4.3). At the moment the same nodes are eliminated from the search space when the oracle answers "inadmissible" as when the oracle answers "yes" to a question about a node, however we distinguish between e-bugs and i-bugs when reporting a diagnosis to the user.

3.2 Trusted predicates

Sometimes the user is positive that a predicate in their program is correct, either because they have proved it correct with respect to some specification (or the predicate is so simple that it *is* its specification) or because the predicate has been rigorously tested in the past. (Of course neither of these guarantee that the predicate will always behave as expected

since there may be a bug in the specification, or there may have been scenarios that were missed by previous tests. However the user may feel that the chances of the predicate being buggy are negligible.) In these cases the user is not interested in being asked questions about such predicates, since they are unlikely to be the cause of any bugs in the program.

The user may request that certain predicates or entire modules should be *trusted* by the debugger. The debugger will assume that trusted predicates cannot be buggy and will not ask the user any questions about them.

However, the debugger does not assume that calls to trusted predicates are *correct*. Instead it will simply *ignore* any nodes in the EDT that result from calls to trusted predicates. We therefore include a fourth possible oracle answer in addition to 'correct', 'erroneous' and 'inadmissible': An answer of '*ignore*' from the oracle means the node is not buggy, though its descendants may possibly be buggy.

A node that is ignored can be considered erroneous if it has an erroneous child and correct if all its children are correct or inadmissible.

There are two reasons for treating trusted nodes this way. First, one of the arguments in a call to a trusted predicate may be a higher order term. This higher order term will most likely be called by the trusted predicate or one of its descendants. The higher order call may behave erroneously. Since the higher order call will be a descendant of the trusted node in the EDT, if we assume the trusted node is correct, then we will eliminate the erroneous higher order call from the suspect set. This means that we will never be able to diagnose a bug in the subtree rooted at the higher order call's `exit`, `fail` or `excp` node(s).

Second, it may be the case that a change is made to a predicate, say p, that is called by another predicate, say q. Now suppose the user is in the habit of trusting both p and q since they have always performed correctly in the past. The user now decides not to trust p anymore since a change has recently been made to it. The user chooses to continue to trust q though, since no change has been made to q's code. Now if the debugger were to assume that all trusted nodes were correct, then it would have to assume that any calls to q should be correct. However, if the change in p has introduced a bug, then the debugger will never be able to find this bug if it is in a call to p which is a descendant of the call to q.

The user never gives an answer of 'ignore', instead the user indicates which predicates (or entire modules) should be trusted. The oracle then gives an 'ignore' response to questions about trusted predicates.

The user may assert which predicates or modules are trusted before starting a declara-

tive debugging session, or the user may decide to trust predicates if and when the declarative debugger asks a question about the predicate.

By default all predicates in the Mercury standard library are trusted. This avoids many questions about obviously valid atoms, such as:

```
+(2, 2) = 4
Valid?
```

(`+` is a function in the Mercury standard library). It is also very helpful to trust predicates that take higher order arguments such as `map` which applies a given predicate to a list of values to generate another list of values, since this avoids many irrelevant questions about the internal workings of `map`, that are known to be correct.

Drabent et al [13] present a generalisation of trusted predicates where the user can specify arbitrary assertions that the oracle can then use to answer debugger queries. This technique has the potential to greatly reduce the number of questions. However, this introduces the possibility that the assertions contain bugs themselves (since the assertions can be arbitrary logic programs). The risk of this increases with the age of the software, since the assertions must be kept up to date whenever the program is changed.

3.3 'Don't know' answers

In order to locate bugs the oracle needs to be able to answer 'correct', 'erroneous', 'inadmissible' or 'ignore' to at least some of the questions. However, in some cases, the user might feel that the question asked is particularly difficult to answer, and may prefer to take their chances with a different part of the search space first. We also provide a fifth answer of '*skip*' for use in such cases.

The key difference between the four other answers and a 'skip' answer is that 'skip' provides no knowledge of the intended interpretation. This means that a 'skip' answer will not be used by the oracle to infer the answers to other questions. In particular, if the same question comes up later in the search, the previous answer of 'skip' will not necessarily be reused.

An answer of this kind will only be used by the search algorithm to try to find a better question to ask. When a question is skipped, the search algorithm will keep a record of this and try to avoid asking that question again.

Our 'skip' answer is similar to the 'maybe' answer proposed by Westman and Fritzon [53] with one important difference: we do not allow a skipped node to guide the search

in any way. In Westman and Fritzon's debugger a 'maybe' node is initially assumed to be erroneous. It is then later re-asked if no erroneous descendants are found. We believe this to be the wrong approach for a debugger supporting multiple search strategies as ours does, since this excludes parts of the search space prematurely.

It is only when all suspects in the suspect set have been skipped that skipped questions will be asked again. The user may of course answer 'skip' again, but at some stage the user has to face the inevitable and provide an answer which will advance the oracle's knowledge of the intended interpretation and will therefore advance the search for the bug. We indicate to the user that a question has been previously skipped, so that they know the above situation has occured.

Internally, skipped nodes are treated similarly to ignored nodes, until all the nodes left in the suspect set are skipped. At this point the debugger will re-ask the user about the skipped suspects in chronological order of when they were first skipped.

Being able to skip questions can be a big help when debugging since it means the user can ignore very difficult questions. Often simpler questions will later present themselves, the answers to which are sufficient to find the bug. See chapter 6 for a real example.

3.4 Visualising terms

Much work has been done on making the execution tree easier for the user to visualize [7, 26, 29, 53, 56].

Many of these visual debuggers also support the visualisation of data structures in the program being debugged. Westman and Fritzson [53], for example, have specialized interfaces to display the contents of arrays in an easily browsable way. Zimmermann and Zeller [56] have implemented a tool for the visualisation of C data structures. Their system depicts pointer references between structures as arcs between nodes. This makes it much easier to spot cycles than it would be by examining the actual pointer values. Lönnberg et al [26] have implemented a similar visualisation tool. Their Matrix Visual Tester allows visual manipulation of the data structures of a running Java program.

The Mercury debugger (both the declarative and procedural variants) support text-based interactive browsing of terms. The user can browse the term as they would a file system, with each functor representing a directory. The navigation commands are analogous to the commands used in a Unix shell. For example '`cd`' can be used to navigate into a particular argument of the current functor and '`pwd`' can be used to display the current

path. Paths are sequences of functor argument numbers (or field names if the argument is named) which identify subterms, much like the paths in file systems which identify subdirectories. For example the path '`3/1`' identifies the first argument of the third argument of the current term and '`/2/4`' identifies the fourth argument of the second argument of the root term. There is also a command for navigating very deep recursive data structures, such as longs lists; the '`cdr`' command repeatedly descends into a specified argument a specified number of times. For example '`cdr 5 2`' is equivalent to '`cd 2/2/2/2/2`'. An example session is shown in figure 3.1. We also allow SICStus style syntax for navigating terms (i.e. `^2^1` instead of 'cd 2/1') Most commands can also be abbreviated by their first letter, for example a user could enter 'p' instead of 'print'. In our examples the full commands will be shown for clarity.

```
typecheck_goal(
  unify(30, functor(cons(...), ...), ...)  - goal_info(erroneous, ...),
  unify(30, functor(cons(...), ...), ...)  - goal_info(erroneous, ...),
  typecheck_info(
    module(module_sub(qualified(...), ...), predicate_table(map([...]), ...),
      ...), call(predicate - //2), 0, ...),
  typecheck_info(
    module(module_sub(qualified(...), ...), predicate_table(map([...]), ...),
      ...), call(predicate - //2), 0, ...), ...)
Valid?  browse
browser> cd 1
browser> print
unify(30, functor(cons(unqualified(...), ...), no, ...),
  (free -> free) - (free -> free), ...)  -
goal_info(erroneous, unreachable, ...)
browser> ^2^1
browser> print
erroneous
browser> cd ..
browser> pwd
/1/2
```

Figure 3.1: Example interactive term browser session.

All the visualisation and browsing tools mentioned so far can make the job of debugging easier, since the programmer is able to explore the state of the program. However, all these tools still suffer from a common problem: they don't abstract the data structures

being represented. The programmer is forced to consider the low level representation of every data structure in the program. This is in contrast to how most modern software is developed, where high-level abstract data types are used by the programmer without them having any knowledge or interest in their internal workings. Some of the tools (for example [26, 56]) allow uninteresting data to be hidden, but this doesn't necessarily give the user a view of the data structure corresponding to their mental model.

As an example consider a 'map' abstract data type that maps arbitrary keys to arbitrary values. There are many ways to implement such a data type; 234-trees, hash tables or red-black trees could all be used to implement efficient maps. Most of the time the programmer is not interested in the underlying data structure, they are only interested in looking up, inserting and/or deleting key/value pairs from or into the map. The programmer will be no more interested in the underlying data structures during the debugging phase than they would be interested in these data structures in the development phase.

Suppose a programmer is trying to debug a program which uses 234 trees to implement maps. Perhaps the programmer would like to know whether a particular value is stored in the map, but the programmer doesn't know much about the implementation of 234 trees. The programmer would therefore prefer not to have to navigate the nodes of the 234 tree in order to find the value of interest. What the programmer would like is an abstract visualisation of the 234 tree as a table of key/value pairs.

Since 234 trees are a commonly used data structure it might be a good idea to build specialised visualisations for them into the debugger. Indeed the Mercury debugger's text-based term browser does display 234 trees as lists of key/value pairs, because they are ubiquitous in the Mercury compiler. However, there are many other examples of data structures that are less common, or indeed are specific to only one application (for example the EDT data structure used by our declarative debugger). It can be convenient to have an abstract view of the data structures in a buggy program, especially if the bug is unlikely to be in the code which implements the interface to the abstract data type.

Myers [30] makes the observation that it is often difficult to implement custom visualisations of data structures, since substantial extra programming effort is usually required. The Mercury declarative debugger supports a new mechanism whereby the programmer can relatively easily create customised visualisations of different types of data structures in their programs.

Our technique relies on the fact that Mercury terms all have a tree structure. Each node in the tree corresponds to a functor in the term and the children of each node are the

arguments of the functor. Leaf nodes are zero arity functors. This tree structure makes it straight forward to convert any Mercury term into an XML document. This is precisely what we allow the programmer to do. Each functor is converted to an XML element with information about the functor (such as its type and field name) as attributes.

The advantage of using XML is that there are many existing tools and techniques for transforming and visualising XML documents. To create a custom visualisation of a certain type of data structure, a suitable stylesheet needs to be created which will transform the XML version of the data structure into the appropriate format.

As an example we have implemented a stylesheet to convert 234 trees into an HTML table of key/value pairs (a screen of the stylesheet in action is shown in figure 3.2). The paths displayed can be used to access each key or value in the 234 tree from the text-based interactive term browser without having to know anything about the structure of 234 trees. Displaying the table in a web browser is particularly convenient, because it means we can use the browser's text search facilities to look for arbitrary strings of characters in the tree. The key or value containing the text can then be further browsed using the text-based interactive term browser, or another custom stylesheet.

We have also created an XSLT stylesheet for transforming arbitrary Mercury terms into XUL tree widgets. (XUL is a graphical user interface language renderable by Mozilla based browsers [10].) A XUL visualisation of an internal Mercury compiler data structure is shown in figure 3.3. Because of the greater flexibility offered by the GUI we are able to show extra information, such as field names and term paths, which would be more difficult to display in the text-based browser. The user also has more control of how much of the structure is shown as well as which parts are expanded or collapsed. These benefits come with no extra development cost, since they are features of the XUL browser.

The main benefits of this scheme are the relative ease with which new visualisations can be put together and the increasing range of software able to understand, manipulate and display XML.

This approach also has drawbacks though. One major drawback is the size of the generated XML documents. Some large data structures in the Mercury compiler can result in XML documents of over a hundred megabytes in size! This is partly due to the verbosity of XML and partly due to the fact that we export extra information like field names and full type names that are not always used. One way to get around this is to limit the depth of generated XML and place appropriate marker elements where the XML document has been pruned. This would indicate to the user that the displayed document

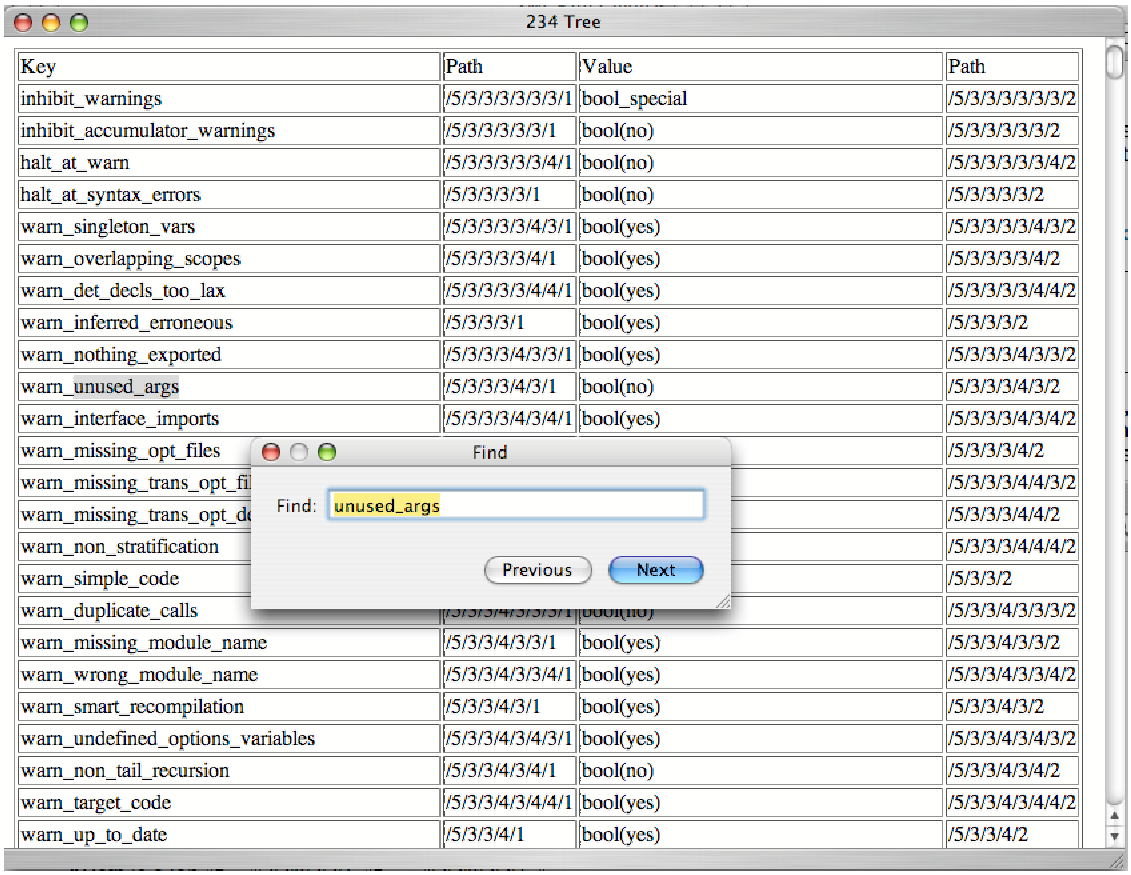


Figure 3.2: A 234 tree rendered as an HTML table of key/value pairs.

is not the complete picture.

Our current integration of this scheme with the Mercury debugger is also not as flexible as it could be. The user must manually change the stylesheet (and possibly the browser application) each time they want to use a different visualisation. Ideally the user should be able to associate stylesheets with different types and the debugger should know which stylesheet to choose automatically. This is future work.

3.5 Handling I/O

In its original form, declarative debugging only works on purely declarative code — i.e. code that has no side effects.

Declarative debugger implementations for languages with side effects have either had to take special measures to model the side effects declaratively, or have ignored the issue

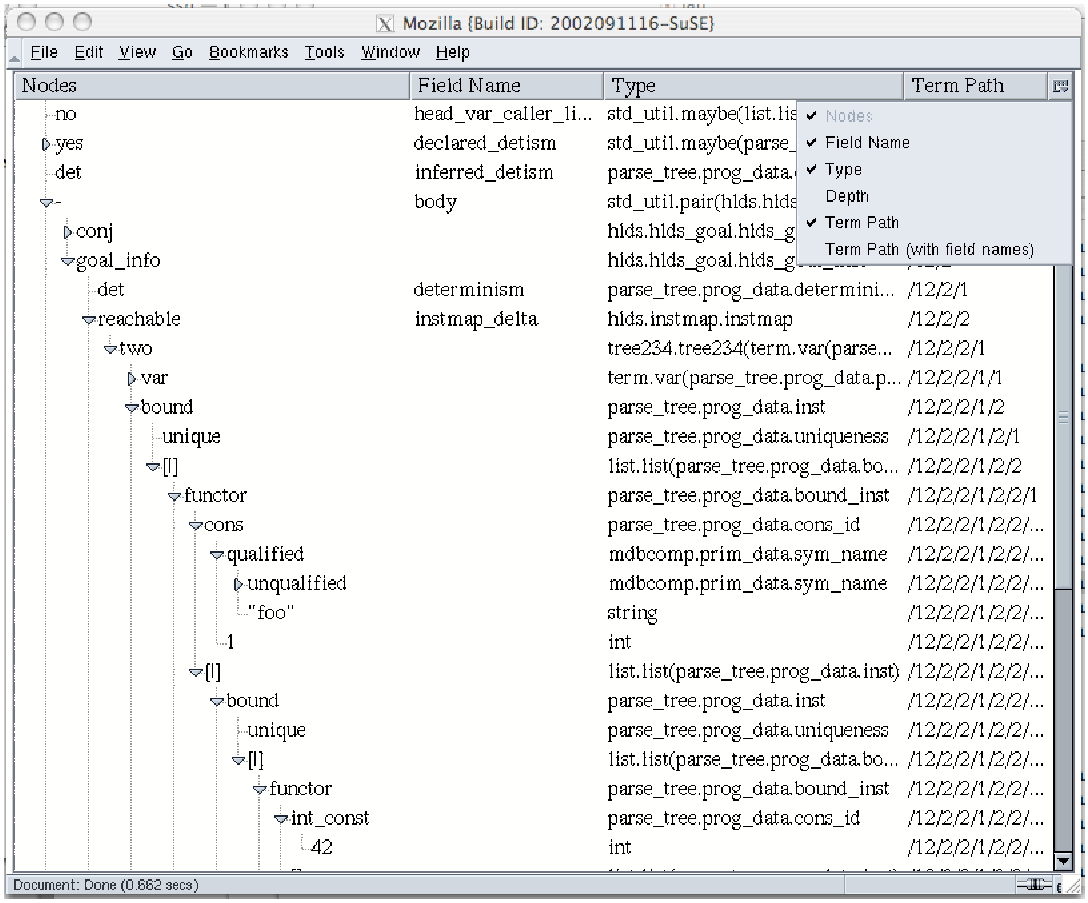


Figure 3.3: The generic graphical term browser showing one of the data structures used by the Mercury compiler.

altogether.

For example Fritszon et al [16] have implemented an algorithmic debugger that works on imperative programs. They transform the program into an equivalent program where each procedure does not have any side effects.

Pereira and Calejo [6, 41] propose a method to debug Prolog programs that produce output via side-effects. All the output side-effects produced during a predicate call (even those which are backtracked over) are encapsulated in what they call a *segment*. The segments for each call are presented to the user along with the question about the call. The user is then expected to assert that a call behaves erroneously if either its segment is wrong (i.e. it produces incorrect output) or its output arguments are wrong.

In Mercury I/O is not done via side effects, so we do not need to consider the possibility that I/O operations were backtracked over, since this can never happen.

Despite this there remain problems with declarative debugging code which does I/O. The first is how to let the user know what I/O actions a particular call performs. The second problem stems from the need to debug large programs. When constructing the annotated trace described in section 2.3.3 we only construct pieces of it as they are needed. Constructing the entire trace would be infeasible for long running programs (see chapter 5 for more information on how this problem is resolved). If an earlier portion of the trace needs to be constructed then we rewind execution to a point just before the portion of the trace which needs to be materialized and reexecute that part of the program again, this time collecting events into the annotated trace. If this portion of the program performs I/O, then the I/O will be performed again. This is a problem, because it means the program may not necessarily behave in the same way the second time around. The program could even crash, if for example it tried to close a file which was previously closed during the first execution.

Somogyi [46] has implemented a solution to both these problems by making I/O idempotent. Each primitive I/O operation performed by the program is given an *I/O action number*. The input and output argument values and the name of the procedure of each primitive I/O call are recorded in a table and indexed by the assigned I/O action number. If execution is rewound, then the I/O operations are not reexecuted, but instead the output argument values of the primitive call are looked up in the table and passed to the calling procedure as if the I/O operation had been executed. By recording the current I/O action number at each `call` and `exit` event we can also display to the user all the primitive I/O calls which occurred during the procedure call. The correctness of the call depends on the output of the call given its inputs as well as the correctness of all the I/O actions performed by the call. The I/O state arguments can be thought of as two lists of I/O actions. The input I/O state argument is a list of all the I/O actions performed before the call and the output I/O state argument is the list of I/O actions performed by the call appended to the input I/O state. We can then ask the user questions such as

```
read_and_write([...], [..., read("a"), write("a")])
Valid?
```

where '...' represents the I/O actions executed before the call. We are able to simply display '...' to the user, since knowledge of I/O operations before the call are not necessary to determine the correctness of the call as long as we assume the primitive I/O operations themselves are correct. The only way a call can know what the effects of previous I/O actions were is for the call to either execute a *new* I/O action that observes those effects

or for the call to explicitly accept arguments that describe those effects. This means that any dependence on I/O actions executed outside the call are reflected in new I/O actions executed during the call, or the input arguments of the call.

In our implementation we do not show the I/O actions for the call in the I/O state argument positions as shown above, but instead list them separately as this is generally easier to read.

To illustrate, consider the program fragment in figure 3.4 that writes out a list of strings as an HTML table.

```
:- pred print_html_table(list(string)::in, io::di, io::uo) is det.
print_html_table(Items, !IO) :-
    write_string("<table>\n", !IO),
    print_rows(Items, !IO),
    write_string("</table>\n", !IO).

:- pred print_rows(list(string)::in, io::di, io::uo) is det.
print_rows([], !IO).
print_rows([H | T], !IO) :-
    write_string("<tr><td>", !IO),
    write_string(H, !IO),
    write_string("</td></tr>\n", !IO),
    print_rows(T, !IO).
```

Figure 3.4: A buggy program fragment that performs I/O.

Executing the goal
`print_html_table(["<unknown>", "Ted", "Sue"], IO0, IO)` would result in the following output.

```
<table>
<tr><td><unknown></td></tr>
<tr><td>Ted</td></tr>
<tr><td>Sue</td></tr>
</table>
```

This output is buggy since the '`<`' and '`>`' characters in the string `"<unknown>"` should have been replaced with '`<`' and '`>`' respectively.

Invoking the declarative debugger on this goal results in the session in figure 3.5.

Because a program may perform many millions of I/O actions during its execution, we allow the user to control what events are I/O tabled from within the procedural debugger.

```
print_html_table(["<unknown>", "Ted",
  "Sue"], _, _)
11 tabled IO actions:
  write_string("<table>\n")
  write_string("<tr><td>")
  write_string("<unknown>")
  write_string("</td></tr>\n")
  write_string("<tr><td>")
  write_string("Ted")
  write_string("</td></tr>\n")
  write_string("<tr><td>")
  write_string("Sue")
  write_string("</td></tr>\n")
  write_string("</table>\n")
Valid?  no
print_rows(["<unknown>", "Ted",
  "Sue"], _, _)
9 tabled IO actions:
  write_string("<tr><td>")
  write_string("<unknown>")
  write_string("</td></tr>\n")
  write_string("<tr><td>")
  write_string("Ted")
  write_string("</td></tr>\n")
  write_string("<tr><td>")
  write_string("Sue")
  write_string("</td></tr>\n")
Valid?  no
print_rows(["Ted", "Sue"], _, _)
6 tabled IO actions:
  write_string("<tr><td>")
  write_string("Ted")
  write_string("</td></tr>\n")
  write_string("<tr><td>")
  write_string("Sue")
  write_string("</td></tr>\n")
Valid?  yes
```

```
Found incorrect contour:
print_rows(["<unknown>", "Ted",
  "Sue"], _, _)
9 tabled IO actions:
  write_string("<tr><td>")
  write_string("<unknown>")
  write_string("</td></tr>\n")
  write_string("<tr><td>")
  write_string("Ted")
  write_string("</td></tr>\n")
  write_string("<tr><td>")
  write_string("Sue")
  write_string("</td></tr>\n")
Is this a bug?
```

Figure 3.5: Finding a bug in the presence of I/O.

The tabled I/O actions must be in a contiguous range and *all* I/O actions occurring inside a call on which the user wishes to use the declarative debugger, must be I/O tabled.

Chapter 4

Search Strategies

The search strategy employed by a declarative debugger decides the sequence of questions the debugger asks in its search for the bug. The search strategy is therefore extremely important for effective debugging, especially if the search space is large. The search strategy should not only aim to reduce the number of questions, but should also present questions that are easier for the user to answer.

In this chapter we explore the search strategies implemented for the Mercury declarative debugger. We allow the user to choose which search strategy the debugger uses and to change the strategy on the fly.

4.1 Top-down search

The top-down search strategy first asks the oracle about the root node in the EDT and then proceeds to ask about the children. If a child is erroneous, then the search progresses down the subtree rooted at the erroneous child, otherwise a sibling is asked about. Naish's version of top-down search [32] is given in figure 2.1. Lloyd [24] and Ferrand [15] also present similar top-down debuggers. Top-down search is the simplest (usable) search strategy to implement.

Top-down search has the advantage that it presents questions to the user (assuming the user is acting as the oracle) in a logical order. The next question will always be either about a child or a sibling of the call the current question is about. This means it is simple to keep track of where the debugger is by examining the source code.

The major disadvantage of top-down search is that it may require the user to answer an extremely large number of questions if the search space is large. This is usually the case

for realistic programs.

4.1.1 Variations on top-down search

Top down zooming is explored by Maeji and Kanamori [28]. In this version of top-down search, recursive child calls are preferred. This results in successive questions about the same predicate. This makes answering the questions easier, since the meaning of the predicate is "cached" in the user's mind.

Binks [1] proposes a top-down search, called *heaviest first*, that selects the child with the most total number of descendant nodes. This has a similar effect to divide-and-query for balanced trees with high branching factors. If the tree is not balanced then the search tends to behave like top-down zooming, since recursive calls tend to have the most descendant nodes.

Another variation that we intend to include in the Mercury declarative debugger is *exception zooming*. This would prefer to ask about children that throw an exception.

Our current implementation of top-down search performs a standard top-down, left-to-right traversal of the EDT. This type of top-down analyser has been studied extensively. We have included a top-down search strategy for completeness and because it is a very useful strategy when the search space is small. We can also use it as a baseline for comparing the effectiveness of other search strategies.

It should be noted that because each mode of a predicate in Mercury is compiled to a different procedure, where the goals in the body of each procedure are reordered to satisfy the mode constraints, the order in which child nodes are asked about will vary for different procedures, even if the procedures represent different modes of the *same* predicate. This behaviour is not noticeable, since most predicates tend to be declared with only one mode and the order of execution of goals is usually the same as the order that they appear in the source code.

4.2 Divide-and-query

4.2.1 Overview

For long running programs, the sheer number of nodes in the EDT makes it often impractical to use a search strategy based on top down search. For such situations, we need a search algorithm that can eliminate large numbers of nodes from the search space in one

step. The classic search algorithm designed for this task is Shapiro's divide-and-query algorithm [45]. This algorithm chooses a node in the suspect set that divides the suspect set into two parts, each of equal weight (or as close to equal weight as possible) according to some weighting metric. Each time the oracle supplies an answer, the weight of the suspect set should be reduced by almost a factor of two. Given an EDT with an initial weight w, this allows the bug to be found with $O(\log w)$ questions being asked of the oracle in most cases.

```
middleweight(LastErroneousNode, StepSize) returns Middle is

    CurNode := LastErroneousNode
    TargetWeight := weight(CurNode) / 2
    Repeat
        If CurNode is the root of an implicit subtree
            materialize the next StepSize levels
            of the subtree rooted at CurNode
        PrevNode := CurNode
        CurNode := the heaviest child of PrevNode
    Until weight(CurNode) < TargetWeight
    If PrevNode is closer to TargetWeight than CurNode
        Middle := PrevNode
    Else
        Middle := CurNode
```

Figure 4.1: Algorithm for finding the middle weight node in an EDT.

Our version of the divide-and-query algorithm is shown in figure 4.1. The greedy search works because at each step *CurNode* is guaranteed to be at least as close to half the weight of *LastErroneousNode* as any of its siblings. It is more similar to Hirunkitti's version of divide-and-query [19], which looks for the closest node to the middle, than Shapiro's original version [45], which isn't always guaranteed to find the closest node to the middle.

Because we are able to approximate the weight of a node without having its entire subtree in memory, we are able to selectively materialize the heaviest subtrees: if an implicit root is not *CurNode* at any point in the algorithm in figure 4.1, its subtree won't be materialized. This means that the algorithm will materialize only small portions of the EDT while searching for the middle weight node. The *StepSize* parameter allows us to control the tradeoff here: higher values require more memory to store more nodes of the

annotated trace and the EDT, but require fewer reexecutions of parts of the program. This trade-off is explored in detail in chapter 5.

This is a more general version of the algorithm proposed by Shapiro [45] since we allow an arbitrary weighting to be used for each subtree. This allows us to experiment with different weighting heuristics besides the number of nodes (which is the weighting that Shapiro uses). Because we include succeeded, failed and aborted calls in the EDT we can use divide-and-query to search for both wrong and missing answers as well as unhandled or incorrect exceptions.

4.2.2 Calculating the weight of a subtree

In this section we will discuss the reasons why it is difficult to accurately calculate the weight of a subtree in practice using the traditional weighting metric. We will then explore some alternative weighting metrics that are easier to calculate, but still good enough to yield effective results in most cases.

The traditional weighting metric

The most obvious weighting of a subtree in the EDT is the number of nodes in that subtree. This metric directly reflects the number of questions represented by the tree. Shapiro [45] has shown that using this metric, divide-and-query is query optimal in the worst case.

This weighting is easy to compute for subtrees we have in memory – we simply traverse the subtree and count the nodes.

However, in general, we do not have the entire EDT available in memory, because this would require too much space for long running programs, so we need a way to calculate the weights of implicit subtrees too. Calculating the weight of an implicit subtree is, however, not simple. To calculate the weight of an implicit subtree we might, while executing the part of the program represented by the implicit subtree, try to count the events that would be EDT nodes. We could then store this weight with the root node of the implicit subtree and use it when calculating the weight of any ancestor subtrees in the EDT.

This turns out to be harder than it may at first seem. There may be calls in the implicit subtree that produce multiple solutions. All the `exit`, `fail` and `excp` nodes for such calls will be included in the EDT only if the call fails. When we are executing the program in the implicit subtree, we will need to remember how many solutions have been produced for each multi or nondet call, as well as the weights of the subtrees rooted at these solutions, in

case the call ultimately fails and we need to include all previous solutions in the EDT. This becomes quite difficult to do without an explicit version of the entire subtree in memory. At the least we will require memory proportional to the number of multi or nondet calls, since for each such call we will need to remember the cumulative weights of all the subtrees rooted at previous solutions.

For example consider the predicates `p` and `q`.

```
p(X) :- q(X), X > 1.
```

Suppose we wish to calculate the weight for the node corresponding to the result of a call to `p` without having a copy of the EDT in memory. As we progress through forward execution of the call to `p`, suppose `q` exits with the result 0. Potentially this is a child of the call to `p` in the EDT but we will only know if it is when we know whether `q` fails inside the call to `p` or not. The potential EDT, excluding all nodes corresponding to calls to '`>`', is shown in figure 4.2. The dashed arcs indicate that we don't yet know if the depicted children will be children in the final EDT or not. Suppose the subtree under the first `exit` event for `q` has weight X. Now the generated solution of `q(0)` makes the body of `p` false, so we retry `q` and get the new answer 1 (figure 4.3). Suppose the weight of the subtree under this `exit` node is Y.

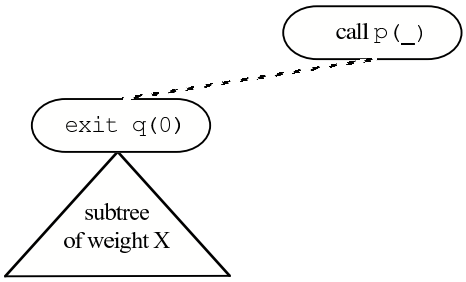


Figure 4.2: The potential EDT after `q` has produced one solution.

Figures 4.4 and 4.5 show two possible next scenarios. In the scenario in figure 4.4, the next retry of `q` yields the solution 2, which causes the body of `p` to be true and `p` to exit. In this case only the last `exit` is included in the EDT, so the weight of the subtree rooted at the call to `p` is $Z + 1$.

In the scenario in figure 4.5, `q` produces no more solutions, causing the call to `p` to fail. In this case we include all `q`'s previous `exit`s as well as the `fail` node, so the weight of the (failed) call to `p` is $X + Y + Z + 1$.

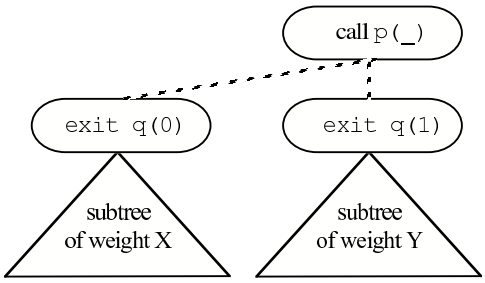


Figure 4.3: The potential EDT after `q` has produced two solutions.

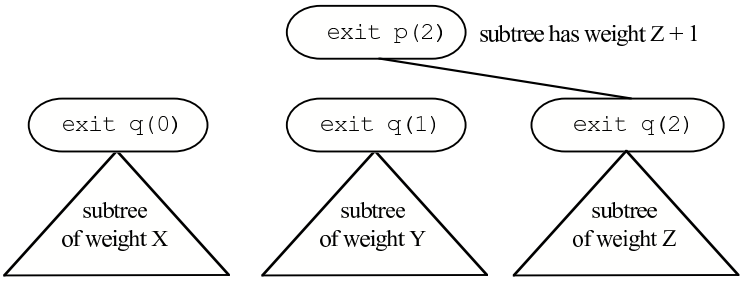


Figure 4.4: Scenario 1, `q` produces another solution.

In this example we need to remember the weight $X + Y$ in case we need to use it in the calculation of the weight of the call to `p`. We have to do the same for all multi or nondet nodes in the EDT if we wish to accurately calculate the weights of their ancestors. If the entire EDT is available in memory, this is easy, since then we know immediately whether a call will fail or not. If parts of the EDT are not available, we need an alternate data structure that gives us this information. Such a data structure would duplicate the parts of the missing EDT that involve procedures that can succeed more than once, but would have to have a more complicated structure than the EDT itself. The design and implementation of such a data structure seems a very high price to pay, and we would strongly prefer not to pay it. Instead, we have looked at alternate metrics for the weight of a subtree which are easier to calculate in the absence of an explicit subtree.

A practical approximation to the traditional weighting metric

For det and semidet code we can calculate the number of EDT nodes by simply counting the number of descendant `exit`, `fail` and `excp` events (or equivalently the number of `call` events, since for det and semidet code each `call` will have exactly one matching `exit`,

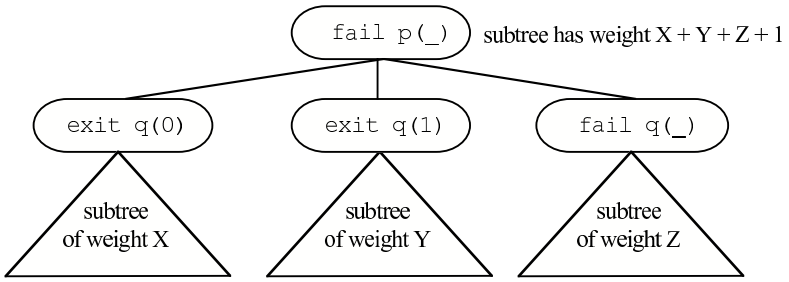


Figure 4.5: Scenario 2, `q` fails.

`fail` or `excp` event).

When constructing the EDT we execute the corresponding part of the program, adding nodes to the EDT that are above a certain depth. Calls below this cut-off depth still need to be executed, even though they aren't added to the EDT (calls below the cut-off depth must be executed so that subsequent exit and fail events above the cut-off depth can be added to the EDT). When executing the program below the cut-off depth we can count the number of `exit` and `fail` events and store this with the node at the cut-off depth in the EDT (i.e. the root of the implicit subtree).

For implicit subtrees where multi or nondet code is executed we can *approximate* the weight by counting the number of descendant `exit`, `fail` and `excp` events between the event which is the root of the implicit subtree and the corresponding `call` event.

The weight of any subtree can then be approximated by adding the sum of all the materialized EDT nodes in the subtree, plus the approximated weights of any descendant implicit subtrees.

For subtrees with only det or semidet code, this is a completely accurate calculation of the number of nodes in the EDT, since for det and semidet code each `exit`, `fail` and `excp` event will appear as a node in the tree.

For subtrees which also contain nondet or multi code, the approximation is just that, and is not guaranteed to be accurate.

In the presence of calls which could succeed more than once, however, we may violate a property of the weights of each subtree that we wish to preserve. Namely that the weight of the subtree should be a constant plus the total weights of all the children. We need this property to hold to ensure that if we subtract the weight of a subtree from all its ancestors (because the oracle asserts that the subtree is correct), then we don't make the weight of any ancestors negative.

To overcome this problem we can add the (possibly estimated) weights of any backtracked over solutions to all the ancestors of a node when we are building a new portion of the EDT and discover that the previously backtracked over solutions should be included in the EDT.

A biased weighting metric

Instead of counting the number of `exit`, `fail` and `excp` events only, we might count *all* the descendant events (both interface and internal) in a subtree and use this as a weighting metric.

For det and semidet code this is trivial to calculate: we simply take the difference between the event number of the root of the subtree and the event number of the corresponding call node plus one — we needn't traverse any of the subtree even if it is in memory. We *could* do the same for the `exit` and `fail` count metric described in the previous section by keeping a count of the number of `exit` and `fail` events at each materialized node in the EDT, however this would require an extra word per node as well as some extra calculation at each event when building the EDT. We have the event numbers available at each node already, for switching between the procedural and declarative debuggers and for building unmaterialized portions of the EDT.

For calls that produce multiple solutions, we can approximate the number of descendant events by adding the number of events between previous `redo`s and `exit`s. This is an over approximation, since not all the events generated for previous solutions will contribute to the generation of later solutions, i.e. some of the events may be inside backtracked-over descendant calls.

For example, suppose a call generates the following sequence of interface events.

Event#	Call#	Port
⋮		
4	3	`call`
⋮		
7	3	`exit`
⋮		
12	3	`redo`
⋮		
17	3	`exit`
⋮		

```
23      3       redo
⋮
45      3       fail
⋮
```

Our estimate of the weight of the subtree rooted at the final `fail` event will be (45 - 23 + 1) + (17 - 12 + 1) + (7 - 4 + 1) = 33. Our estimate of the weight of the subtree rooted at the second `exit` event would be (17 - 12 + 1) + (7 - 4 + 1) = 10. The weight of the first `exit` would be 7 - 4 + 1 = 4.

Using this over approximation, can cause the weights to become inconsistent. For example suppose that the event number of the parent `call` was 3, the call failed without producing any solutions and the event number of the parent `fail` was 46. Then the approximated weight of the parent `fail` node in the EDT would be 46 - 3 + 1 = 44, however the sum of the weights of the children would be at least 33 + 10 + 4 = 47.

To avoid this situation where the weight of a subtree is less than the sum of the weights of the child subtrees, we need to add any double counted events to ancestor subtrees. We can do this on the fly if and when we encounter such a situation. This situation would only arise in the presence of multi or nondet code. As we mentioned such code is quite rare in practice.

An interesting property of this weighting metric is that it is biased towards nodes whose calls generate more internal events. Calls which generate more internal events are generally to predicates with more complicated bodies (i.e. bodies with more disjuncts, switches, if-then-elses, etc). It seems likely that predicates with more complicated bodies would be more likely to contain bugs, so this bias would seem justified.

To illustrate the effect of the biased weighting metric, consider the program fragment in figure 4.6. The body of the predicate `animal` is more complex than the body of the predicate `make_list`. Let's assume that `animal` has a bug in it (which seems pretty likely given its definition). Two debugging session are shown in figures 4.7 and 4.8. Both use divide-and-query to debug the call `make_animal_list(6, 3, 3, _)`. Figure 4.7 shows the results of using the traditional weighting metric of the number of nodes (which, since there is no non-deterministic code, is simply the number of descendant calls). Figure 4.8 shows the same session, but using the biased weighting metric. Notice that with the biased weighting metric there is one less question about `make_list`. This is because the subtree rooted at the call to `animal` is assigned a greater weight than it would be with the

traditional weighting metric. The three if-then-elses and numerous disjunctions generate more internal events in the call to `animal` which gives it a greater weight.

```
:- pred animal(int::in, int::in, int::in, animal::out) is det.
animal(Legs, Eyes, Arms, Animal) :-
    ( if ( Legs = 8 ; Arms = 8 ; Eyes = 8 ) then
        Animal = spider
    else if Eyes = 1 then
        Animal = cyclops
    else if ( Arms = 2 ; Eyes = 2 ; Legs = 2 ) then
        Animal = human
    else
        Animal = ant
    ).

:- pred make_animal_list(int::in, int::in, int::in, int::in,
    list(animal)::out) is det.
make_animal_list(N, Legs, Eyes, Arms, List) :-
    animal(Legs, Eyes, Arms, Animal),
    make_list(N, Animal, List).

:- pred make_list(int::in, T::in, list(T)::out) is det.
make_list(N, X, L) :-
    ( if N = 0 then
        L = []
    else
        make_list(N - 1, X, L1),
        L = [X | L1]
    ).
```

Figure 4.6: Example program fragment used to compare the biased weighting metric with the usual weighting metric.

Admittedly this is a very contrived example, but it does serve to show the effect of the biased weighting metric. With longer running programs which produce more events the effect will be less obvious. The point is that it is unlikely to have a detrimental effect on the debugging session. Its efficiency in terms of execution speed make it a worthwhile alternative to Shapiro's original divide-and-query.

```
make_animal_list(6, 3, 3, 3, [ant, ant, ant, ant, ant, ant])
Valid?  no
make_list(3, ant, [ant, ant, ant])
Valid?  yes
make_list(5, ant, [ant, ant, ant, ant, ant])
Valid?  yes
animal(3, 3, 3, ant)
Valid?  no
Found incorrect contour:
animal(3, 3, 3, ant)
```

Figure 4.7: Debugging session using the traditional weighting metric.

```
make_animal_list(6, 3, 3, 3, [ant, ant, ant, ant, ant, ant])
Valid?  no
make_list(4, ant, [ant, ant, ant, ant])
Valid?  yes
animal(3, 3, 3, ant)
Valid?  no
Found incorrect contour:
animal(3, 3, 3, ant)
```

Figure 4.8: Debugging session using the biased weighting metric.

The effect of oracle answers on the weight calculation

'Yes' or 'inadmissible' oracle answers have the obvious effect of causing the weight of the subtree rooted at the correct or inadmissible node to be subtracted from the weights of its ancestors. Since all the ancestors must already be materialized (otherwise the search would not have been able to progress to the node about which the question was asked), we can store the new weight at each ancestor.

The effect of an 'ignore' answer is a little more subtle. We do not wish to remove the weight of the subtree rooted at the ignored node, since there may be buggy nodes in this subtree. Instead we remove the difference between the weight of the subtree rooted at the ignored node, and the sum of the weights of the subtrees rooted at all the children of the ignored node. (In the case of our biased weighting metric this will be the number of internal events plus interface events with the same call sequence number as the ignored

node).

4.2.3 Related work

Shapiro's original method of rerunning the erroneous part of the program with a modified interpreter each time the middle node needs to be found is impractical for long running programs. Ironically, long running programs which produce large search spaces are precisely the programs for which divide-and-query is most useful.

Plaisted [42] proposed a more efficient version of Shapiro's divide-and-query algorithm for Prolog. His technique involves saving the input and output values of calls at carefully chosen points in the call tree. The oracle is queried about only the saved calls until a much smaller subtree containing no saved calls is left. The process is then repeated on the remaining subtree. This greatly reduces the time spent reexecuting the program. The nodes are chosen in such a way as to approximate Shapiro's divide and query algorithm. This approach has two main drawbacks. First, the tree has to first be transformed into a new tree that has a constant branching factor of two. To achieve this the meanings of the questions at each node must be modified. This results in questions of the form "If procedure P was called with such and such inputs, then should it be possible to reach a state after Q returns in which the variables accessible to P have such and such values?" for each child call Q of call P. Such questions are more complex than ours and would seem to require a knowledge of the operational semantics of the program, something we would prefer to avoid. Second, because only selected nodes are materialized, the set of search strategies that can be applied are severely limited. For example we could not perform the usual top-down search or do subterm dependency tracking. In our practical experience this loss of flexibility would be intolerable.

4.3 Subterm dependency tracking

Most previous declarative debuggers have asked users to say, for each atom, simply whether the atom is valid, erroneous or inadmissible. By accepting only these three answers, they fail to gather information that could improve the search significantly. This information is the precise difference in the user's head between the correct behavior of the predicate concerned and the actual behavior.

When users say that a particular atom is erroneous, it is because they know, at least implicitly, what the set of correct solutions is for the call, and they see that the output

arguments of the actual atom computed by the program differ from output arguments in all the correct solutions. Frequently, the actual output is *almost* right: most parts of most output arguments are correct, and only a small number of parts in just one or two output arguments are wrong. However, unless the debugger allows users to specify exactly *which* parts of which output arguments are wrong, the search inside the computation represented by the atom will not be able to focus on the part of the computation that computed the wrong part of the erroneous atom.

Similarly, when users say that a particular call is inadmissible, it is because they know that some part(s) of some input argument of the call fail a precondition of the call. The debugger can focus on the part of the computation that generated that wrong subterm only if the user can tell the debugger *which* part of which input argument violates the precondition.

The above observation was first made by Pereira with his rational debugger [40]. Weiser [52] provides empirical evidence which shows that the origins of data values are valuable in debugging.

The Mercury declarative debugger includes a mechanism that allows users to mark function symbols in arguments when browsing an atom that the declarative debugger is asking about. If they mark a subterm of an output argument, they say that the atom is erroneous. If they mark a subterm of an input argument, they say that the atom is inadmissible. In both cases, the system will use the information about the identity of the wrong subterm to guide the search for the bug. Specifically, the system will start asking questions about the atoms that generated the marked subterm, since it is very likely that either these atoms have bugs inside their call tree, or they were given incorrect information themselves. (The third possibility, which in our experience is less likely, is that this computation is correct and was given correct inputs, but its output was supposed to be processed further before being passed on, and this post-processing is missing.)

Pointing out an incorrect subterm generally doesn't require much extra effort on the part of the user (if the user is acting as the oracle), because they would have had to have known which subterm(s) were incorrect in order to answer the question accurately in the first place. We allow the user to mark an incorrect subterm from within the interactive term browser, which is the tool the user would typically use to explore large terms anyway.

Focusing the search onto a wrong subterm can be a huge win. If the atom is large, and only a small part of it is incorrect, then not exploring the parts of the computation that generated the correct parts of the atom will avoid a large number of questions that don't

have anything to do with the bug; the larger the atom, the more unnecessary questions can be avoided. We are acutely aware of this point, because we use the Mercury declarative debugger to debug the Mercury compiler, many of whose predicates pass around multi-megabyte data structures as arguments.

Sometimes it can be hard to know which subterm to mark. Consider a predicate that expects its input to be a sorted list. If it is given the list `[1, 3, 2]`, is the wrong subterm the subterm `2` or the subterm `3`? Or is it the cons node of the sublist `[3, 2]`? All of these subterms could be replaced with different subterms which would make the list sorted. Which one should the user mark? The answer is that it depends on which subterm the user is more curious about. Marking the `3`, for example would direct the user to the code that first generated the `3`, which could be in a very different place to the code that generated the cons node. If the user suspects that the `3` shouldn't appear in the list at all, then the `3` should be marked. If the elements in the list are correct, but the order is incorrect, then it is more likely that the code which created the incorrect cons node is wrong and the user should mark this term.

4.3.1 A short example

The following example illustrates how the subterm dependency tracking feature of the debugger is used.

```
:- pred get_payment(loan::in, int::in, payment::out) is det.
get_payment(Loan, PaymentNo, Payment) :-
    get_payment_amount(Loan, PaymentNo, Amount),
    get_payment_date(Loan, PaymentNo, Date),
    Payment = payment(Date, Amount).
```

Suppose that `get_payment` produces an incorrect result and the declarative debugger asks the following.

```
get_payment(loan(...), 10, payment(date(9, 10, 1977), 10.000000000000)).
Valid?
```

If we know that this is the right payment amount for the given loan, but the date is incorrect, we can mark the `date(...)` subterm and the debugger will then ask us about `get_payment_date`.

```
get_payment(loan(...), 10, payment(date(9, 10, 1977), 10.000000000000)).
```

```
Valid?  browse
browser> cd 3/1
browser> print
date(9, 10, 1977)
browser> mark
get_payment_date(loan(...), 10, date(9, 10, 1977)).
Valid?
```

In this example irrelevant questions about `get_payment_amount` are avoided. `get_payment_date` was only one level down in the tree, however it could have been at a much deeper level. In general the further away the node where the subterm is bound, the greater the number of questions that can potentially be skipped.

4.3.2 The subterm tracking algorithm

Our method of tracking a marked subterm to its ultimate source can be best described in two steps: the algorithm for tracking subterms within a single procedure call, and the algorithm for tracking subterms across calls.

Consider an erroneous atom in which one subterm of an output argument is marked as wrong. (We will consider inadmissible calls later.) The first task in tracking the marked subterm is to find out what goal in the body of the procedure generated that subterm.

The Mercury mode system's knowledge of where each variable is bound makes this task significantly easier than it would be in most other languages. If the program is compiled with the right options, the compiler will include a representation of the bodies of all procedures in the executable. This representation includes, for each goal, the list of variables bound by that goal.

For example, consider the following predicate.

```
:- pred rational_add(rational::in, rational::in, rational::out) is det.
rational_add(A, B, C) :-
    A = r(An, Ad),
    B = r(Bn, Bd),
    lcm(Ad, Bd, Cd),
    CA = Cd // Ad,
    CB = Cd // Bd,
    Ap = An * CA,
    Bp = Bn * BA,
    Cn = Ap + Bp,
    C = r(Cn, Cd).
```

The mode information recorded for `rational_add` can tell the declarative debugger immediately that the producer of the `Cd` part of the output argument is the call to `lcm` (the least common multiple predicate), and that the producer of the `Cn` part of the output argument is the call to the builtin function `+`.

This works for all predicates whose body is a simple conjunction. However, most predicates have more complex bodies that include if-then-elses and/or disjunctions.

```
:- pred search(bintree(K, V)::in, K::in, V::out) is semidet.
search(Tree, K, V) :-
    Tree = tree(K0, V0, Left, Right),
    compare(Result, K0, K),
    ( if Result = (=) then
        V = V0
    else if Result = (<) then
        search(Right, K, V)
    else
        search(Left, K, V)
    ).
```

In this case, the mode system knows that `V` is produced by the unification `V = V0` or by one of the two recursive calls, exactly one of which is executed in the process of computing a solution, but it can't know which one was executed in any specific case. However, the debugger can, since it has access to the execution history of the call. If during the relevant call the first condition failed and the second succeeded (i.e. if `Result = (<)`), the debugger will know it, because it will have seen an `else` event for the outer if-then-else and a `then` event for the inner if-then-else (it knows which goal the `else` and `then` events correspond to because of the goal paths stored at the `else` and `then` nodes in the annotated trace). It can thus reconstruct the sequence or conjunction of goals executed to compute the solution. This sequence is a kind of dynamic slice [21, 51]:

```
search(Tree, K, V) :-
    Tree = tree(K0, V0, Left, Right),
    compare(Result, K0, K),
    Result = (<),
    search(Right, K, V).
```

While procedure bodies may contain conjunctions, disjunctions, negations and if-then-elses nested arbitrarily, with the atomic goals being unifications and calls, the slice we compute is always a conjunction in which all conjuncts are either unifications, calls or

negated goals and negated goals cannot bind any variables visible from the outside (this restriction being necessary for the safety of negation as failure). You can boil a procedure body down to such a slice by discarding those arms of if-then-elses and disjunctions which did not contribute to the solution being considered.

The slice we search through in the body of the call corresponds exactly with the goals lying on the contour leading up to the `exit` event for the solution (see section 2.3.2). We can use the step_left_in_contour function given in figure 2.12 to step through the contour and gather up the goals responsible for generating the output of the call.

The marked subterm is identified by the argument number and position within that argument. The position is a *subterm path*, which is a sequence of argument numbers. A subterm path can be used to uniquely identify a subterm in a term. The first number in the sequence represents the position of the subterm in the top level functor of some term. Successive argument numbers give the functor argument number in which the subterm appears for terms nested in the top level term. For example the subterm path of the second $f(a)$ in the term $h(f(a), g(h(b, c, f(a))), b)$ is [2, 1, 3]. This is exactly the path which would be used to navigate to the term using the interactive term browser (described in section 3.4), except we would write "cd 2/1/3" or "^2^1^3" instead of "[2, 1, 3]" in the interactive term browser.

The algorithm for tracking subterm dependencies within procedure bodies is implemented as the origin function, shown in figure 4.9. This algorithm relies on the fact that the compiler converts the bodies of predicates to what we call superhomogeneous form. In this form, multiple clauses are converted to disjunctions in the body of the completion of the predicate, all predicate heads and calls have distinct variables as arguments, all unifications are explicit, and each unification contains at most one function symbol. The mode system classifies all unifications into four categories:

- unifications of the form $X = f(Y_1, ..., Y_n)$ in which the Y_i are input and the X is output. We write these *construction* unifications as $X \leftarrowtail f(Y_1, ..., Y_n)$.

- unifications of the form $X = f(Y_1, ..., Y_n)$ in which the X is input and the Y_i are output. We write these *deconstruction* unifications as $X \rightarrowtail f(Y_1, ..., Y_n)$. (Actually, it is possible for some of the Y_i to also be input, but that is of no relevance here.)

- unifications of the form $X = Y$ in which one variable (say X) is input and the other is output. We write these *assignment* unifications as $X := Y$.

origin(*Conj*, *Var*, *SubtermPath*) returns *Origin* is

Find the goal G that produces *Var* in *Conj*
If G is a construction $X \leftarrowtail f(Y_1, ..., Y_n)$ then
    Var must be X
    If *SubtermPath* = [] then
        Origin := unify(G)
    Else
        SubtermPath must be [*First* | *Rest*]
        First must be in 1..n
        Origin := origin(*Conj*, Y_{First}, *Rest*)
Else if G is a deconstruction $X \rightarrowtail f(Y_1, ..., Y_n)$ then
    Var must be one of the Y_is, say Y_k
    Origin := origin(*Conj*, X, [k | *SubtermPath*])
Else if G is an assignment unification $X := Y$ then
    Var must be X
    Origin := origin(*Conj*, Y, *SubtermPath*)
Else if G is a call $p(A_1, ..., A_n)$ then
    Var must be one of the A_is, say A_k
    Origin := call(G, k, *SubtermPath*)
Else
    Var must be an input argument
    Let *ArgNum* be the number of that input argument
    Origin := head(*ArgNum*, *SubtermPath*)

Figure 4.9: The origin function.

- unifications of the form $X = Y$ in which both variables are input and have atomic types. We write these *test* unifications as $X == Y$.

All other unifications are either (a) transformed into calls to compiler-generated unification predicates or (b) disallowed, which results in either that goal being reordered relative to other conjuncts or in an error message. For example, a unification of two non-atomic ground terms is transformed into a call, while a unification of two free variables is delayed until one variable is bound, if that is possible.

Conj contains only calls, unifications and negated goals. Among unifications, only construction, deconstruction and assignment unifications can bind variables; test unifications cannot. Negated goals also cannot bind variables. The cases handled by the origin function are therefore all the cases.

The origin function returns one of three possible functors:

- unify(U), indicating the subterm is bound by the unification U appearing in *Conj*.
- call(C, k, P), indicating that the call C, appearing in *Conj*, binds the subterm in its kth argument. The path of the subterm in this argument is P.
- head(k, P), indicating that the subterm is passed as input to the call which has *Conj* in its body. The subterm appears in the k argument and at path P in this argument.

Consider the `rational_add` example from before, and suppose we want to find the origin of the computed numerator. Since the numerator is the first argument of `r`, the declarative debugger calls `origin(Body, C, [1])`, where `Body` is the body of the clause. The goal that produces `C` is `C = r(Cn, Cd)`. Since this is a construction unification and the path isn't empty which means the unification doesn't create the subterm, we call `origin(Body, Cn, [])`, which finds that the origin is the call to the builtin addition function.

If we want to find the origin of the computed denominator, the declarative debugger calls `origin(Body, C, [2])`. This time, the processing of `C = r(Cn, Cd)` leads to the recursive call `origin(Body, Cd, [])`. This tells us that the origin is the third argument of the call to the `lcm` predicate.

This algorithm can be adapted trivially to handle the dependency tracking needs of inadmissible calls. There are only two differences.

- The conjunction we give it as the second argument is from the body of the caller of the marked atom, not the body of the predicate involved in the marked atom itself.
- We need to use the modified version of the step_left_in_contour function presented in section 2.3.4 (figure 2.14) to collect all the atomic goals in the conjunction we pass to the origin function. This is because an input subterm can be passed into a negated goal, though a negated goal cannot produce any output. We need to track the origin of the input subterm through these negated goals.

Consider the `all_pairs_are_in_table` predicate below. This predicate checks whether `Struct` contains a `Pairs` element such that all the key/value pairs in `Pairs` in are present in `Table`.

```
:- pred all_pairs_are_in_table(struct(K, V)::in, map(K, V)::in) is semidet.
all_pairs_are_in_table(Struct, Table) :-
```

```
    extract_pairs(Struct, Pairs),
    not (
        list_member(Key - Value, Pairs),
        not map_search(Table, Key, Value)
    ).
```

The part of the procedure body after the call to `extract_pairs` tests whether `list_member(Key - Value, Pairs)` implies `map_search(Table, Key, Value)`. The Mercury compiler applies the equivalence $A \Rightarrow B \equiv \neg(A \wedge \neg B)$ yielding the code above.

If a call to `map_search` is inadmissible and one of its input arguments is marked, then in the call to the origin function, the conjunction leading up to that call and the corresponding head, will be

```
all_pairs_are_in_table(Struct, Table) :-
    extract_pairs(Struct, Pairs),
    list_member(Key - Value, Pairs),
    ⋮
```

If the origin function returns a unification, we have found the true origin of the subterm we are tracking. If it returns a reference to an argument in the clause head, then the true origin is in a sibling call to the left. We can take another step towards that true origin by marking the indicated subterm of the indicated argument (which must be input) and invoking the origin function on the conjunction leading up to that call, stepping one level up in the EDT.

If a call to origin returns a reference to a call, then the true origin is somewhere probably in the subtree below the call, and we can take another step towards that true origin by marking the indicated output argument of the call and invoking the origin function on the conjunction leading up to the `exit` event that computed that atom, stepping one level down in the EDT.

Even if a call to the origin function returns a reference to a call, it is possible that the true origin is not somewhere in the subtree below the call, because it is possible that the marked output argument of the call was simply copied from an input argument. In such cases, our dependency tracking algorithm will first follow the subterm down the EDT and then back up again, to get back to the body of its caller. However, this time it will be searching for the origin of a different variable in that scope, and the producer of that

track(*Node*, *ArgNum*, *Path*, *StepSize*) returns *BindingNode* is

If *ArgNum* is an output argument of the atom at *Node*
    (this can only happen if *Node* is an `exit` node)
  Let *CurNode* := *Node*
  If *CurNode* is an implicit root
    materialize the next *StepSize* levels of the subtree rooted at *CurNode*
  Let *Conj* be the conjunction of goals along the contour leading up to *CurNode*
  Let *Var* be the *ArgNum*'th argument of the head of *Node*'s predicate
Else
  ArgNum must be an input argument of the atom at *Node*
    (so *Node* could be an `exit`, `fail` or `excp`)
  Let *CurNode* be *Node*'s parent in the EDT
  Let *Conj* be the conjunction of goals in the body of *CurNode*'s predicate
    along the contour(s) leading up to the call for *Node*
  Let *Var* be the *ArgNum*'th argument of the call atom
    for *Node* in the body of *CurNode*'s predicate

Let *Origin* := origin(*Conj*, *Var*, *Path*)
If *Origin* is call(*SubAtom*, *SubArgNum*, *SubPath*)
  There must be exactly one `exit` node corrsponding to *SubAtom*
    on the contour(s) used to construct *Conj*
  Let *SubNode* be this `exit` node
  SubArgNum indicates one of *SubAtom*'s output arguments
  BindingNode := track(*SubNode*, *SubArgNum*, *SubPath*, *StepSize*)
Else if *Origin* is head(*HeadArgNum*, *HeadPath*)
  BindingNode := track(*CurNode*, *HeadArgNum*, *HeadPath*, *StepSize*)
Else
  Origin must be unify(*G*)
  BindingNode := *CurNode*

Figure 4.10: The track function.

variable will be to the left of the call the algorithm dived into and out of. This guarantees that the algorithm makes progress.

The dependency tracking algorithm is shown in figure 4.10. It returns the node in the EDT in which the subterm appearing in the *ArgNum*'th argument of *Node* at path *Path* is bound by a construction unification.

In general, the dependency tracking algorithm may make many steps both up and down

in the EDT in its search for the unification that creates the subterm being tracked. The depicted track function stops only when it finds the construction unification which binds the subterm it is tracking. In practice there are two other conditions which cause the track function to stop tracking a subterm (these extra conditions are not shown in figure 4.10 merely for simplicity). First, it cannot step into predicates whose bodies it doesn't have access to. This can happen either if the module containing that predicate wasn't compiled with the option that tells the compiler to include predicate body representations in the executable, or if the predicate is a Mercury builtin predicate or is defined not in Mercury but in a foreign language. (The Mercury foreign language interface allows Mercury predicates to be defined in other languages such as C.) Second, the subterm dependency tracking algorithm will not take a step that would take it above the current suspect set, since there is no point in asking questions about nodes there. If necessary, it will take steps that take it *below* the suspect set, into the non-suspect regions rooted at correct or inadmissible EDT nodes. If tracking the subterm leads out of such non-suspect regions, we carry on as usual. If the subterm being tracked was created in such a non-suspect region, we return the last suspect which we tracked the subterm through before leaving the suspect set.

4.3.3 Tracking a subterm through higher order calls

To understand the problem with tracking a subterm through a higher order call, let us first consider an example. Suppose we have the predicates `p`, `add`, `map` and `one` as defined below.

```
:- pred p(list(int)::out) is det.
p(L) :-
    one(One),
    map(add(one), [1, 2, 3, 4, 5], L).

:- pred add(int::in, int::in, int::out) is det.
add(A, B, A + B).

:- pred map(pred(T, T), list(T), list(T)).
:- mode map(pred(in, out) is det, in, out) is det.
map(_, [], []).
map(P, [H0 | T0], [H | T]) :-
    P(H0, H),
    map(P, T0, T).
```

```
:- pred one(int::out) is det.
one(10).
```

`map` transforms the list in its second argument to the list in its third argument by applying the higher order term in its first argument to each element. `add` simply adds its first two arguments to produce its third argument. `one` is supposed to unify its argument with the number one, but is buggy. The predicate `p` curries `add` to form a new predicate which unifies its second argument with its first argument plus one. It then passes this new predicate to `map` to produce the list `[11, 12, 13, 14, 15]`.

Suppose we wish to track the first argument of the atom `add(10, 5, 15)`, appearing as a descendant of a call to `p`, to its source using the track function (figure 4.10). Since the first argument is input we construct the slice leading up to the (higher order) call to `add(10, 5, 15)` in `map`, which turns out to be the empty slice, since there are no goals before the higher order call. So in the track function we compute *Conj* to be $\emptyset$. However we cannot find a value for *Var*. The call atom for the `exit` node produced by `add(10, 5, 15)` is `P(H0, H)`. Following the algorithm blindly would mean we set *Var* to `H0` which would be wrong.

The solution is to treat the higher order call `P(H0, H)` as the deconstruction unification `P` $\rightarrowtail$ `add(A)` (where the variable `A` contains the 10 we are tracking). We then proceed to track the subterm at path [1] in the variable `P`. (i.e. we treat the higher order term `P` as if it were a regular first order term).

Eventually repeated application of the origin function will lead to the goal `map(add(One), [1, 2, 3, 4, 5], L)`. In superhomogeneous form this would be broken up into several atomic goals, one of which would be the unification `P = add(One)`, where the first argument passed to map is constructed by currying the `add` predicate. For the purposes of subterm tracking we can treat this as the construction unification `P` $\rightarrowtail$ `add(One)`, which means we now track the subterm in the variable `One` with its path in that variable being []. This leads us to the actual origin of the first argument of the atom `add(10, 5, 15)`, which is the call to `one`.

Applying the declarative debugger to the above example gives the following output.

```
add(10, 5, 15)
Valid?  browse 1
browser> mark
one(10)
```

```
Valid?
```

4.3.4 Using incorrect subterm information

Once the oracle has asserted that a particular subterm in a node in the EDT makes the node inadmissible or erroneous, we can call the track function as we described above and locate the node in which the subterm was bound. We define the *dependency chain* as the sequence of EDT nodes corresponding to the atoms returned by successive calls to the origin function made by the track function. The first node in the dependency chain will be the node that the oracle asserted was erroneous or inadmissible because of an incorrect subterm. The last node will correspond to the node where the subterm was initially constructed.

Our current implementation asks the oracle about the correctness of the node in which the incorrect subterm was bound, provided the node wasn't previously eliminated from the suspect area If tracking was stopped because of missing information or because the origin of the subterm lies outside the suspect set, then we ask about the last node on the dependency chain that is in the suspect set. We also tell the user the location in the source file of the construction unification that bound the subterm. This behaviour is easy for the user to understand since it is predictable and gives the user some control over the bug search — they can direct the bug search to the call responsible for binding a particular subterm appearing in an atom.

If the oracle then asserts that the node in which the incorrect subterm was bound is erroneous, then the search will continue down the new erroneous subtree, using the same search strategy as before the incorrect subterm was pointed out by the oracle.

If the binding node is not erroneous then it seems likely that the bug lies somewhere on or near the dependency chain. Thus a search strategy that favours nodes on the dependency chain if the user asserts that the binding node is correct would seem to be a good idea.

As a first attempt to implement such a search strategy we currently perform a binary search on the path between the binding node and the last erroneous node if the oracle asserts that the binding node is correct. This effectively does a binary search on a modified version of the dependency chain if the marked subterm is output. The modified dependency chain has all the nodes of the original chain, except nodes where the subterm was passed in and out of without any modification. This strategy has not proved particularly effective in practice and other strategies are suggested in chapter 7.

4.3.5 Related work

The idea of focusing the search on a marked subterm is not new. It was first proposed two decades ago by Pereira [40], who named it *rational debugging*. Pereira's implementation worked by modifying the usual Prolog unification algorithm to keep track of where each part of the output was bound, effectively recording the entire history of the program execution.

This idea has been applied several times since then. For example, Sparud and Runciman [49] used redex trailing to implement subterm dependency tracking for Haskell. Redex trails record backward data dependencies between computed values in the program and so can be used to trace an incorrect result to its source. Recently redex trails have been augmented with additional information which allow for more flexible views of the trail [8, 50]. Specifically the augmented trail can be used to do normal top-down style algorithmic debugging as well as tracking the origins of values. Some experimental debuggers for Java [9, 23] can also find out where a particular variable was last assigned, their equivalent of subterm tracking.

Unfortunately, this approach has significant overhead, in both space and time. The space overhead in particular is a killer; a program running for a minute or two can generate enough data to overflow memory or even disk capacity (since most disks are always close to full). The difference from the standard execution algorithm is itself a problem: the code for recording execution history is a large body of code that must be maintained, and due to differences in data formats, most of these systems do not allow history recording to be switched on for only part of a program or program run.

By contrast, the static availability of mode information in Mercury allows our subterm dependency tracking algorithm to work *without* a complete history of execution, which makes the system much more practical. We don't need any changes to the runtime system; the only two things we needed to add to our existing system to support dependency tracking were compiler support for recording procedure bodies in the executable and the algorithms in the debugger to interpret them. In a sense, we made virtue out of necessity. While Pereira could modify the general purpose unifier to keep track of dependencies, we could not, since the Mercury runtime doesn't *have* a general purpose unifier. The only unifications allowed in Mercury programs are one-way *matches*[1], and all matches are compiled into sequences of primitive operations such as constructions and deconstructions. Recording history at runtime would therefore have demanded huge changes in the code

[1]This is changing as we add constraint solving capabilities to Mercury, but we don't support the debugging of constraint code just yet.

generator. As it is, the only cost our algorithm imposes when not being used is the cost of storing representations of procedure bodies, which leads to larger executables but not to slowdowns (except possibly through cache effects). As an indicator of this space cost, consider the Mercury compiler. Normally, its executable is 8.5 Mb in size; enabling debugging increases that to 47 Mb. Adding representations of procedure bodies, which is the information needed only by the dependency tracking algorithm, increases it further to 57 Mb, a relative increase of 22%.

The dependency chain we compute is a form of inter-procedural dynamic slice [20, 21, 51], with the slicing criterion being the contribution to the value of the marked subterm. Unlike many applications of slicing, we don't need to construct an executable extract of the program; we just construct a sequence of EDT nodes.

Schoenig and Ducassé [44] propose a slicing technique for Prolog programs that yields executable slices based on a static slicing criteria. That work has different goals to ours. Schoenig and Ducassé wish to produce an executable slice which has the same (partial) declarative and operational semantics (with respect to the slicing criteria) as the original program. On the other hand we are only interested in computing a dynamic, non-executable, backslice for one particular value (or subvalue) of a variable. The combination of the single-assignment nature of Mercury and the static availability of mode information makes the algorithm for constructing our kind of slice particularly simple and cheap to execute. Another difference is that Schoenig and Ducassé's slicing criteria is limited to the argument of a literal in the body of a clause, whereas we compute a dynamic backslice based on an arbitrarily deeply nested subterm. This makes our method more suited to algorithmic debugging. This is because the outer functors of arguments in a literal are normally not computed very far away from the literal, where as a subterm in an argument of a literal could be created some distance from the literal. This means we can generally make bigger jumps in the search tree if we allow the slicing criteria to be the subterm of an argument, instead of just the argument. This also generally results in simpler questions because the subterms will be smaller than their enclosing terms.

Fritzson et al have developed a debugger for an imperative language that combines slicing with algorithmic debugging [16]. Their Generalized Algorithmic Debugger has separate components for computing the slice and for doing algorithmic debugging on the execution tree. If the user asserts that the value of a variable is wrong, the backslice with respect to that variable is computed and all nodes in the execution tree not on that slice are eliminated. Algorithmic debugging then proceeds as normal on the reduced tree. They

therefore make use of the computed slice in a different way to us. They use the slice to further prune the debug tree, something which we do not currently do, because we want to retain maximum flexibility. We cannot simply eliminate nodes that are not on the dependency chain, since, although the subterm does not pass through these nodes, they may still influence the value of the subterm (for example by failing instead of succeeding, causing a different branch of an if-then-else to execute). The slice computed in the Generalized Algorithmic Debugger is more general and so nodes not on the slice can be eliminated. Another difference is that we zoom in immediately on the source of the incorrect value which in most cases results in much simpler questions and homes in on the bug quicker. Fritzson et al also do not seem to be able to determine the origin of a *sub*structure of a larger structure as we are able to do.

Chapter 5

Resource considerations

In this section we will consider some of the technical problems with using the declarative debugger on large, long-running programs.

The main problem is that, for such programs (such as the Mercury compiler, which can generate hundreds of millions of events) it is infeasible to store the entire annotated trace in memory at once.

The memory cost of collecting a large number of events in the annotated trace is difficult to predict. Each annotated trace node can consume between three and ten words. Each `call` and `exit` event will also maintain references to the argument values at the time of the `call` or `exit` as well as flags indicating whether each argument was ground at the time of the `call` or `exit` and whether it is a user visible argument (the Mercury compiler may add extra, non-user-visible arguments to procedures; for more details see [12]). This all adds five extra words per argument for each `call` and `exit` event. It also prevents the garbage collector from recycling the memory occupied by the argument values. So in storing `call` and `exit` events in the annotated trace we introduce a kind of memory leak, the effects of which will vary depending on the sizes of the values passed around by predicates in the program, how frequently new values are created, and how frequently previously created values would become garbage under normal conditions.

To avoid storing the entire annotated trace in memory at once we can build pieces of it as they are needed. The downside to this is that parts of the program need to be reexecuted each time we want to build a new piece. We have a space-time trade-off: if we store larger portions of the annotated trace in memory then the program will need to be reexecuted less often, but the memory consumption will be higher. On the other hand if we store smaller pieces of the annotated trace in memory then the memory cost will be

lower, but the program will need to be reexecuted more often.

We would like to be able to parameterize this space-time trade-off in an easily predictable way. This would allow predictable adjustment of the space-time trade-off according to available memory and CPU power.

5.1 An overview of the events gathering mechanism

To generate an annotated trace for a call the call must be reexecuted and the resulting events collected. Not all the events need be collected though. We may collect a subset of the events generated by the call and ask the declarative debugger to try to find a bug in one of these. If the declarative debugger needs to explore events not collected the first time around, then the missing events can always be added by reexecuting the appropriate call.

On each reexecution of a call we require the set of events gathered during that run to form a tree (the EDT) which the declarative debugger can use to search for bugs. It is no good to collect, for example, every third event, since in general these events cannot be combined to form a coherent EDT. For each node in the EDT derived from a generated portion of the annotated trace we require that either all the children of the node are present in the annotated trace, or none of them are present. If none of them are present then we mark the node as an *implicit root*. An implicit root is the root of a subtree in the EDT whose nodes have not been materialized in the annotated trace.

If the declarative debugger needs to search the nodes in an implicit subtree, the call corresponding to the `exit`, `fail` or `excp` event at the implicit root must be reexecuted. To do this we use the machinery of the procedural debugger's '`retry`' command [47] and I/O tabling [46] to rewind the state of the program to a point just before the `call` event corresponding to the `exit`, `fail` or `excp` event at the root of the implicit subtree. We then proceed to reexecute the program from that point, gathering events into the annotated trace, until we arrive at the `exit`, `fail` or `excp` event at the implicit root.

We decide on a depth limit before reexecuting the call. Events below this depth limit are not included in the annotated trace, while those at or above the depth limit are. The general algorithm we use to decide which events are to be added to the annotated trace on a particular run is depicted in figure 5.1 (this algorithm is taken from [4]).

Here *depth_limit* controls the depth of each generated portion of the annotated trace (or

```
build_annotated_trace(call_number, end_event, depth_limit) returns trace is

trace := NULL
inside := false
Rewind execution to a call before or equal to call_number
For each executed event e loop
        If e is a call or redo event for call call_number
                inside := true
        If inside
                If depth(e) ≤ depth_limit
                        trace := create_annotated_node(e, trace)
                If e is an exit, fail or excp and depth(e) = depth_limit
                        trace := mark_as_implicit_root(e, trace)
        If e is an exit, fail or excp event for call call_number
                inside := false
Until the event number of e is end_event
```

Figure 5.1: Algorithm for building the annotated trace to a predefined depth limit.

rather, the depth of the EDT represented by the generated portion of the annotated trace). The depth function returns the depth of an event relative to the root of the portion of the EDT currently being materialized. Initially *end_event* will be the event where the '`dd`' command was issued in the procedural debugger. On subsequent invocations *end_event* will be the event at the root of an implicit subtree we wish to materialize. *call_number* is the call sequence number of the call corresponding to the event at the root of the implicit subtree. The declarative debugger keeps a reference to every list of annotated trace nodes generated by calls to build_annotated_trace in a structure which allows it to view the separate traces as one complete EDT.

The algorithm is not actually implemented as an explicit loop as depicted in figure 5.1, but instead the target code generated by the Mercury compiler is instrumented with callbacks to a function that implements the body of the loop and the variables are global variables in the target imperative language. When the event with event number *end_event* is executed the runtime system calls the analyser, which searches the generated trace for bugs (and is implemented entirely in Mercury). For our purposes it is simplest to express the algorithm as the loop depicted in figure 5.1.

If the analyser requests the implicit subtree of a node *after end_event* be built, then we continue execution from the current event (which will be the value of *end_event* from

the previous call to build_annotated_trace).

5.2 Limiting the depth by a predefined constant

The simplest way to control the space-time trade-off is to vary the value of *depth_limit* in figure 5.1. For less memory consumption, but more reexecutions we can set *depth_limit* to a low value. If we have more memory available then we can set *depth_limit* to a higher value which would result in fewer reexecutions.

The problem with this solution is that it is impossible to give a value for *depth_limit* that will work sensibly in all cases.

If we make the depth limit small, and the branching factor of the program is low, then we may end up collecting too few events on each run, resulting in lots of reexecutions of the program.

If our debugger only implements top down search and this is the only search strategy we are interested in ever implementing, then setting the depth limit to a low number will work fine most of the time. This is because the program will only need to be reexecuted after a nonzero number of questions have been asked of the user (the number of questions depends on the value of *depth_limit* and the user's answers to the questions). This means the user will normally spend most of their time answering debugger questions instead of waiting for the program to be reexecuted. However, with search strategies such as divide-and-query and subterm dependency tracking, which need to explore large portions of the EDT before asking the user a question, a small depth limit can be disastrous. For example suppose *depth_limit* is fixed at 5 and the program generates a stick-like tree of depth 100 000. For divide-and-query to ask about the node at depth 50 000 (which would divide the tree into two portions of equal weight) the program must be reexecuted 10 000 times!

If, on the other hand, we use a large predefined depth limit, then if the branching factor of the program is large, too many nodes will be collected. For example suppose we fix *depth_limit* at 10 000. In our example above this would mean the program would only have to be reexecuted 5 times for divide-and-query to arrive at the middle node. However using a depth of 10 000 on a program with a higher branching factor, like the Mercury compiler, causes all available memory to be used up and the debugger to come to a thrashing halt before it ever gets the chance to ask the user any questions.

Instead of limiting the depth of each tree by a predefined constant, it would be preferable

for *depth_limit* to be a function of the shape of the implicit tree we wish to materialize.

Besides the heap space used by the program under normal conditions, there are two additional memory costs when the annotated trace is being built.

1. Each node in the annotated trace consumes a fixed amount of memory, though the amount depends on the node type.

2. Because each `call` and `exit` node in the annotated trace keeps references to the arguments of the call or exit atom, the values of such arguments cannot be garbage collected. This doesn't add directly to the memory consumption, but prevents previously allocated space from being freed when it otherwise would be. The effects of this can be extremely difficult, if not impossible, to predict. We do, however, know that the amount of memory held in this way will increase with the number of nodes which are added to the annotated trace.

The amount of memory held by or allocated for annotated trace nodes increases with the number of nodes which are added to the annotated trace. Thus if we are able to control the number of nodes added to the annotated trace during a single reexecution, we would stand a better chance of controlling the amount of memory consumed and thus gain more control over the space-time trade-off. The more linear the relationship between the number of annotated trace nodes and the amount of memory consumed, the more control we can have over the space-time trade-off. As we will see it turns out that in practice we can achieve a satisfactory level of control of the space-time trade-off by controlling more accurately how many nodes are added to the annotated trace with each reexecution.

5.3 Estimating the ideal depth

Given the root of an implicit tree, we need to find a value for *depth_limit* which will cause no more than a specified number of nodes, *node_limit*, to be added to the annotated trace in one run. We'll call the maximum such depth limit the *ideal depth* for the given tree. The algorithm for building a portion of the annotated trace then becomes the one depicted in figure 5.2.

Notice how the algorithm is now parameterized by *node_limit*, the desired number of nodes to be generated and not by *depth_limit*. *depth_limit* is instead calculated using the function get_ideal_depth. Notice too that we have introduced a new variable d_{max}, which

```
build_annotated_trace(call_number, end_event, node_limit) returns trace is

trace := NULL
inside := false
d_max := 0
Rewind execution to a call before or equal to call_number
depth_limit := get_ideal_depth(end_event, node_limit)
For each executed event e loop
        If e is a call or redo event for call call_number
                inside := true
        If inside
                If depth(e) ≤ depth_limit
                        trace := create_annotated_node(e, trace)
                Let depth_in_implicit_subtree := depth(e) - depth_limit
                If d_max < depth_in_implicit_subtree
                        d_max := depth_in_implicit_subtree
                If e is an exit, fail or excp and depth(e) = depth_limit
                        trace := mark_as_implicit_root(e, d_max, trace)
                        d_max := 0
        If e is an exit, fail or excp event for call call_number
                inside := false
Until the event number of e is end_event
```

Figure 5.2: Algorithm for building the annotated trace and recording the maximum depth of each implicit subtree.

records the maximum depth of each implicit subtree. The purpose of d_{max} will be made clear shortly.

We now look at some methods to efficiently estimate the ideal depth of a tree for a particular value of *node_limit*.

We can think of the annotated trace as a weighted tree where each `call` event represents a node in the tree. Each node in this tree has a weight which is the number of events which have the same call sequence number as the node. This includes the internal events generated directly by the call as well as the `call` event and its corresponding `redo` and `exit` event(s) and/or `fail` or `excp` event. We will call such a tree a *weighted call tree*.

Note that this tree does not in general correspond to the EDT. In the EDT each node represents an `exit`, `fail` or `excp` event, whereas in the weighted call tree each node represents a `call` event. Calls which succeed more than once will be represented by multiple

nodes in the EDT, but only one node in the weighted call tree.

The weighted call tree has the property that the sum of the weights of all the nodes equals the number of annotated trace nodes represented in the tree. This property makes it useful for modeling the approximate memory consumed by a materialized portion of the annotated trace.

For a weighted call tree with a constant branching factor b, the same weight for each node w, and maximum depth d_{max}, the number of events represented by the tree N, is given by the following formula.

$$N = w \sum_{i=0}^{d_{max}-1} b^i \tag{5.1}$$

To calculate the average weight we simply divide the number of events in the tree by the number of calls. We can easily find out the number of events and the number of calls in the tree by looking at the last allocated event number and call sequence number at the time of the `call` event and then at the time of the `exit`, `fail` or `excp` event at the root of the implicit subtree.

Calculating the average branching factor requires some additional information. But first let us define precisely what we mean by the average branching factor. We will define the *average branching factor* of a tree as the branching factor of another tree with the same maximum depth and number of nodes as the original tree, but with a constant branching factor. Using this definition we can find the average branching factor by solving for b in equation 5.1.

We have w (the average weight) and N (the total number of events in the implicit tree). The missing piece of the puzzle is d_{max}, the maximum depth of the implicit subtree. Now every time we build a new portion of the annotated trace, the events inside implicit subtrees must still be executed, even though they are not added to the annotated trace. This means we have an opportunity to collect and summarize information about implicit subtrees whenever we build a new portion of the annotated trace. This information can be stored at implicit root nodes in the materialized portion of the annotated trace, ready to be accessed when the implicit subtree needs to be materialized. One piece of information that we can gather with minimal impact on execution time is the maximum depth of each implicit subtree, as we do in figure 5.2.

We are now able to solve for b, the average branching factor. This is quite straight forward to do since $f(b) = w \sum_{i=0}^{d_{max}-1} b^i - N$ is a strictly increasing function for positive

b, so numerical methods such as Newton's method work well to find b where $f(b) = 0$.

Once we have b and w we can then find a value for *depth_limit* which causes approximately *node_limit* events to be added to the annotated trace. To do this we need to find the root of the strictly increasing function $g(d) = w \sum_{i=0}^{d-1} b^i - n$. Again this is easy to do using numerical methods. d can then be used as the return value of get_ideal_depth.

When the first tree is being materialized we do not yet have access to the maximum depth of the tree, so in this single case get_ideal_depth returns a conservatively small constant. This only happens once, so at worst results in one extra reexecution of the program.

Tables 5.2 – 5.6 show the results of some experiments run using this implementation of the get_ideal_depth function.

"fib" is a naive implementation of the Fibonacci function (which produces an exponential number of events relative to the maximum depth). "stick" is a simple recursive predicate which produces a stick-like tree. "smallbig" and "bigsmall" produce trees similar in shape to the ones depicted in figure 5.3. We also test this technique with a realistic Mercury program: the Mercury compiler itself. The compiler is invoked on a small, 51 line source file. (Once we have improved our method, we will also show an example with a bigger source file.) Table 5.1 gives the total number of events in the search space for each benchmark.

The results were obtained by simulating a declarative debugging session where the bug is at a leaf node in the tree (this is the worst case since it requires the most reexecutions of the program). The search strategy used in all cases was divide-and-query and in all cases the search was started from the top level `exit` node for the `main` predicate. 'No' was answered to all questions, causing the search to end at a leaf node. For interest the number of questions asked using divide-and-query is also given in table 5.1. In the tables *node_limit* is the number of nodes that we want to be collected for each reexecution. We do not include the initial reexecution or the final reexecution in the shown measurements, since the initial reexecution uses a small constant depth limit (since nothing is known about the tree at this time) and the final reexecution would in general collect fewer nodes that the other reexecutions. *total_created* is the total actual number of annotated trace nodes which were produced during the complete debugging session (except for the first and last reexecutions). *exec_count* is the number of reexecutions which were required to locate the bug (again minus the first and last reexecutions). Dividing *total_created* by *exec_count* gives the actual average number of nodes collected per reexecution, which we would like

Benchmark	Total events	D&Q questions
fib	24 232 831	30
stick	1 000 007	18
smallbig	15 076 715	29
bigsmall	18 404 873	21
Mercury compiler	18 135 308	12

Table 5.1: Total events in the search space for each benchmark.

to be as close to *node_limit* as possible. The total CPU time (T_{CPU}) and total wall clock time (T_{WC}) of the complete debugging session is also shown. Both these times are given in seconds. Times were averaged over ten runs. Next the total resident set size (RSS) is displayed in megabytes along with the total virtual size (VSZ) of the process (including swap space used by the process). Both these measurements were captured at the time of bug location which would be when they are at their maximum.

All tests were run on a 2.4 GHz Intel Pentium IV with 512MB of RAM under SuSe Linux 8.1.

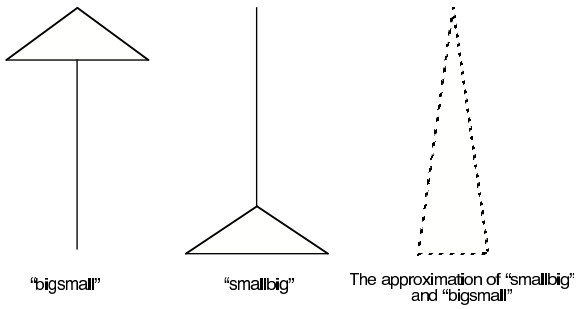


Figure 5.3: The shape of the trees produced by "smallbig" and "bigsmall".

Judging from the results in tables 5.2 – 5.6 it seems our first attempt to approximate the ideal depth has not been very successful.

For "stick" and "fib" the node limit and the average number of nodes gathered per reexecution seem at least proportional. Notice the long running times for smaller values of *node_limit* in the "stick" example (table 5.3). This is because of the high number of reexecutions of the program required. This is not as much of an issue for the "fib" example since the tree produced is much shorter. Instead in the "fib" example (table 5.2) reexecutions take longer for larger values of *node_limit* because of the overhead of constructing the nodes. (In the "stick" example this overhead occurs for all values of

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
50	7	3 516	502	25	27	16	26
100	5	7 324	1 464	25	27	17	26
200	4	7 818	1 954	25	28	17	26
500	3	20 020	6 673	25	27	18	26
1 000	2	64 710	32 355	25	27	22	30
2 000	2	113 364	56 682	26	28	27	41
5 000	2	174 420	87 210	26	28	32	41
10 000	2	541 946	270 973	28	30	66	74
20 000	1	2 059 706	2 059 706	38	41	211	223
50 000	1	6 454 422	6 454 422	59	109	468	647
100 000	1	6 454 422	6 454 422	59	110	465	647

Table 5.2: "fib" using the average branching factor.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
1 000	258	1 003 552	3 889	509	524	236	242
2 000	131	1 000 454	7 637	273	281	228	234
5 000	55	998 440	18 153	130	137	225	234
10 000	29	993 546	34 260	84	87	211	217
20 000	16	992 704	62 044	61	64	220	226
50 000	8	999 732	124 966	48	50	227	234
100 000	4	921 266	230 316	40	43	219	226
200 000	3	999 812	333 270	39	41	227	233
500 000	1	742 844	742 844	28	39	194	200

Table 5.3: "stick" using the average branching factor.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
2 000	33	173 664	5 262	311	313	44	58
5 000	21	232 152	11 054	208	209	48	58
10 000	17	390 994	22 999	170	171	68	82
20 000	13	1 952 040	150 156	148	149	211	224
50 000	10	1 146 974	114 697	118	119	147	157
100 000	9	313 478	34 830	92	93	55	66
200 000	7	624 342	89 191	85	85	95	107
500 000	Out of Memory						
1 000 000	Out of Memory						

Table 5.4: "smallbig" using the average branching factor.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
50	534	4 233 398	7 927	124	2 067	434	689
100	Out of Memory						

Table 5.5: "bigsmall" using the average branching factor.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
100	37	895 982	24 215	50	52	203	228
200	35	2 584 138	73 832	66	69	400	428
500	11	4 591 288	417 389	70	168	314	671
1 000	8	6 532 290	816 536	85	469	330	888
2 000	Out of Memory						

Table 5.6: The Mercury compiler using the average branching factor.

node_limit because all nodes are always constructed.)

For the "fib" example the average branching factor of the tree is calculated to be 1.55. This is too low. A value closer to 2 would give better values for *depth_limit* when collecting nodes near the top of the tree (i.e. the part of the tree where there are fewer leaf nodes).

We might expect the estimate to be more accurate for the "stick" example, since the approximation tree should also be a stick of the same depth as the actual tree. This is, however, not the case, since the "stick" example does not in fact produce a perfect stick. This is because along with each recursive call there are also calls to some builtin predicates such as `+`. (These calls are used to determine when to stop recursion.) This results in a tree which is a good approximation to a perfect stick when the tree is very deep, but which starts to look less like a stick as the maximum depth decreases. This is reflected in the calculated average branching factors which are close to one for the initial reexecution, but which increase for later reexecutions. For example with *node_limit* set to 100 000, the calculated average branching factor is 1.000023 for the first reexecution, but is 1.000297 for the last reexecution.

In the "smallbig" example (table 5.4) there doesn't appear to be any correspondence at all between *node_limit* and the average number of nodes constructed per reexecution. The runs with *node_limit* set to 20 000 and 50 000 seem particularly out of place. The individual reexecutions for these runs are shown in table 5.7 together with the reexecution where *node_limit* is set to 100 000. It seems the second last reexecution is responsible for the large memory usage in both cases. Even though the value of *depth_limit* is small

Re-execution #	Nodes created	*depth_limit*
node_limit = 20 000		
2	23 914	2 390
3	19 944	1 993
4	16 104	1 609
5	12 554	1 254
6	9 414	940
7	6 744	673
8	4 614	460
9	3 014	300
10	1 854	184
11	1 094	108
12	614	60
13	402	31
14	1 851 774	16
node_limit = 50 000		
2	33 724	3 371
3	24 514	2 450
4	16 884	1 687
5	10 954	1 094
6	6 674	666
7	3 804	379
8	2 024	201
9	1 024	101
10	484	47
11	1 046 888	19
node_limit = 100 000		
2	41564	4155
3	26544	2653
4	15744	1573
5	8644	863
6	4384	437
7	2054	204
8	894	88
9	16638	36
10	197012	18

Table 5.7: "smallbig" individual re-executions using the average branching factor.

Re-execution #	Nodes created	*depth_limit*
node_limit = 50		
2	4 122 802	17
3	3248	19
4	204	19
5	204	19
6	204	19
7	204	19
8	204	19
9	204	19
10	204	19
⋮	⋮	⋮

Table 5.8: "bigsmall" individual re-executions using the average branching factor.

for the second last reexecution in both cases, the number of nodes collected is huge. This is because the tree suddenly gets exponentially large at its base (as shown in figure 5.3). This means the number of nodes collected will increase dramatically even for small values of *depth_limit* near the bottom of the tree. The approximation therefore only has to be a little bit too optimistic for the number of nodes constructed to explode. This is what has happened for the runs with node limits of 20 000 and 50 000. Observe that the number of nodes collected for reexecutions before this explosion gradually get smaller with each reexecution. This is because the approximation trees for implicit subtrees rooted lower down are fatter and shorter than the approximation trees for implicit subtrees rooted higher up. So the calculated average branching factor will be higher for later reexecutions which causes the *depth_limit* to be lower.

The run with a node limit of 100 000 is also shown in table 5.7. Although the node limit is more than the two previous runs, fewer nodes are constructed. This is because we get lucky and only just touch the "big" portion of the "smallbig" tree in reexecution # 9. Because the next reexecution then starts inside the "big" portion of the tree we are able to make a much better guess at the ideal depth.

The "bigsmall" example runs out of memory with the node limit set to only a hundred (table 5.5). This is because the approximation tree (as shown in figure 5.3) is way too deep and narrow, resulting in all the nodes in the "big" part of the tree being materialized in the first few reexecutions. This can clearly be seen in table 5.8 which shows the individual reexecutions where *node_limit* is set to fifty.

The Mercury compiler session also performs poorly using the average branching factor heuristic (table 5.6).

The reason our heuristic performs poorly is because the approximation tree doesn't have the same shape as the actual tree, even though it has the same maximum depth and number of nodes. For example in the "fib" case, there will be only a few events at the maximum depth, whereas the approximation tree will be completely balanced, with many events at the maximum depth. Even though "smallbig" and "bigsmall" have very different shapes, they are approximated by the same tree (figure 5.3).

5.4 A better approximation

Instead of approximating an implicit tree by a tree with the same maximum depth, perhaps we could use the *average* depth of events in the implicit tree instead. This way the approximation tree would be shorter and fatter if more events were at the top of the tree and taller and thinner if more events were at the bottom of the tree. The "smallbig" and "bigsmall" examples would then be approximated by the trees in figure 5.4.

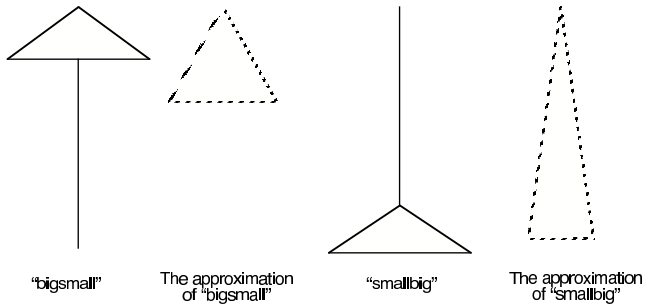


Figure 5.4: Approximating "smallbig" and "bigsmall" using the average depth of events in the tree.

The average depth of the tree can be calculated in a manner similar to the way we calculate the maximum depth, except that while we are in an implicit subtree we sum the depths of all the events and then when we exit the implicit subtree we divide this by the total number of events in the implicit subtree. The sum of the depths of all the events in an implicit subtree can cause overflow if we use 32 bits to represent the sum. We must be careful to avoid this problem (for example by using a 64 bit representation).

We now solve for β in

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
50	16	1 472	92	28	30	16	24
100	11	2 492	226	26	28	16	24
200	7	3 820	545	25	27	16	26
500	4	6 096	1 524	25	27	16	26
1 000	4	7 106	1 776	25	27	16	26
2 000	4	7 106	1 776	25	27	16	26
5 000	3	10 204	3 401	25	27	17	26
10 000	2	16 360	8 180	25	27	18	26
20 000	2	40 922	20 461	25	27	20	30
50 000	2	89 386	44 693	25	27	25	35
100 000	1	65 524	65 524	25	27	29	41
200 000	1	131 060	131 060	25	27	33	41
500 000	1	131 060	131 060	25	27	33	41
1 000 000	1	262 132	262 132	26	28	43	57
2 000 000	1	262 132	262 132	26	28	43	57

Table 5.9: "fib" using the biased branching factor.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
1 000	269	1 002 886	3 728	381	395	236	242
2 000	141	1 000 004	7 092	213	221	236	242
5 000	63	996 372	15 815	110	116	235	242
10 000	36	992 774	27 577	74	79	227	234
20 000	21	980 714	46 700	56	59	211	217
50 000	11	954 644	86 785	45	48	220	226
100 000	7	923 838	131 976	39	44	211	217
200 000	4	826 606	206 651	31	41	194	199
500 000	2	684 698	342 349	27	38	194	200

Table 5.10: "stick" using the biased branching factor.

$$N = w \sum_{i=0}^{d_{ave}-1} \beta^i \tag{5.2}$$

where d_{ave} is the average depth of the nodes in the tree. Because we are using the average depth and not the maximum depth, β is not an average branching factor according to our definition. Instead we will call β the *biased branching factor*.

As can be seen from the results in tables 5.9 – 5.13 this approach does perform better,

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
2 000	35	112 702	3 220	322	324	42	49
5 000	24	111 006	4 625	222	223	42	49
10 000	18	112 522	6 251	171	172	42	49
20 000	15	173 882	11 592	147	148	50	58
50 000	12	195 772	16 314	119	119	50	58
100 000	7	2 159 786	308 540	96	97	244	257
200 000	7	232 038	33 148	72	72	50	66
500 000	Out of Memory						
1 000 000	7	116 452	16 636	81	81	55	66

Table 5.11: "smallbig" using the biased branching factor.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
50	551	706 318	1 281	79	96	133	141
100	279	498 044	1 785	51	59	100	107
200	145	230 714	1 591	35	39	58	66
500	65	283 128	4 355	51	54	75	83
1 000	37	703 712	19 019	47	49	141	149
2 000	Out of Memory						

Table 5.12: "bigsmall" using the biased branching factor.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
100	54	601 527	11 139	47	50	155	186
200	36	1 061 339	29 481	49	51	217	244
500	18	2 441 239	135 624	63	64	387	411
1 000	13	3 264 682	251 129	65	72	307	520
2 000	12	3 795 007	316 250	64	124	375	562
5 000	8	4 340 017	542 502	66	131	174	654
10 000	6	4 757 083	792 847	68	158	400	704
20 000	6	5 196 205	866 034	74	353	423	754
50 000	5	5 761 007	1 152 201	79	459	298	787
100 000	4	6 223 895	1 555 973	82	396	425	871
200 000	Out of Memory						

Table 5.13: The Mercury compiler using the biased branching factor.

Re-execution #	Nodes created	*depth_limit*
node_limit = 100 000		
2	36 644	3 663
3	22 994	2 298
4	23 654	2 364
5	9 034	902
6	5 474	546
7	2 254	224
8	2 059 732	19

Table 5.14: "smallbig" individual re-executions using the biased branching factor.

although the estimates are still far from perfect.

"smallbig" (table 5.11) still suffers from the same problem that it did when using the average branching factor, except the problem now occurs when *node_limit* is set to 100 000. Table 5.14 shows the individual reexecutions when the node limit is 100 000.

The situation is slightly improved for the Mercury compiler debugging session using the biased branching factor (table 5.13), since we now only run out of memory when *node_limit* is set to 200 000 (although we are still generating way too many nodes compared to the node limit).

Clearly approximating an implicit subtree as a tree with a constant branching factor is not a general solution, since realistic programs do not behave this way. Typically real programs call all sorts of different predicates, some of which are simple recursive predicates which produce stick-like trees and some of which have long conjunctions in their bodies which produce wide trees with large branching factors. Non-deterministic code can also lead to a high branching factor if there is lots of backtracking.

5.5 Calculating the ideal depth

Instead of playing guessing games, ideally we'd like to *calculate* an accurate upper bound for *depth_limit* which would result in no more than *node_limit* nodes being added to the annotated trace.

Any method of calculating the ideal depth limit will need to take into account the shape of the tree. One way to do this is to build a histogram of the number of events at each depth in the tree. The histogram could then be consulted to work out at what depth the total number of events exceeds *node_limit*.

Now an obvious problem with such a method is that it requires an array whose length is as big as the deepest event in the call tree. If the program has deep recursion, then we might consume more memory building the histogram than actually generating the annotated trace, which instead of solving the space problem makes it worse.

Fortunately a simple observation saves us. In actual fact we don't need the number of events at *every* depth. If we wish to build an annotated trace containing *node_limit* events, then it suffices to record the number of events at each depth down to a depth of $\lfloor node\_limit/2 \rfloor$. This is because the minimum number of events we can have a each depth is two (a `call` event and its corresponding `exit`, `fail` or `excp` event). We can reuse the same array to calculate the ideal depth for all the implicit subtrees encountered during a particular run. Reserving $\lfloor node\_limit/2 \rfloor$ words of memory for this purpose is not a problem, since we have already committed to adding at most *node_limit* annotated trace nodes anyway.

The version of build_annotated_trace adapted to calculate the ideal depth for each implicit subtree is depicted in figure 5.5.

Experimental results for our toy examples are given in tables 5.15, 5.16, 5.17 and 5.18. The results for the Mercury compiler are given in table 5.19. We now also test the Mercury compiler on a bigger source file (table 5.20). The module compiled performs type checking in the Mercury compiler and is one of the biggest modules in the Mercury compiler. It consists of 6 418 lines of source code (including blank lines and comments). The total number of events generated is 204 640 323. Using divide-and-query to find a (pseudo) bug in a leaf node required 22 questions.

The average actual nodes gathered per reexecution in most cases is quite far below *node_limit*. This is because, with each reexecution, we build the tree to the maximum depth which will produce *at most node_limit* nodes. In most cases less than *node_limit* nodes are produced.

The results show that the extra calculation required to compute the ideal depth limit (instead of estimating it) is well worth it. Table 5.21 nicely demonstrates the flexibility of this approach. Here two runs of the "bigsmall" example are shown. Notice how very small values for *depth_limit* are calculated for the top of the tree where the branching factor is high, while large values for *depth_limit* are calculated for the stick-like section of the tree.

When the Mercury compiler is invoked on a larger source file the total time to find the bug in a leaf node is about ten minutes when *node_limit* is set to 500 000 (table 5.20). This is due to the high number of events (over 200 million). Ten minutes is not too bad

```
build_annotated_trace(call_number, end_event, node_limit) returns trace is

trace := NULL
inside := false
Initialise histogram to size ⌊node_limit/2⌋
Rewind execution to a call before or equal to call_number
depth_limit := get_ideal_depth(end_event, node_limit)
For each executed event e loop
        If e is a call or redo event for call call_number
                inside := true
        If inside
                If depth(e) ≤ depth_limit
                        trace := create_annotated_node(e, trace)
                Let depth_in_implicit_subtree := depth(e) - depth_limit
                If 0 < depth_in_implicit_subtree ≤ ⌊node_limit/2⌋
                        Add 1 to histogram[depth_in_implicit_subtree]
                If e is an exit, fail or excp and depth(e) = depth_limit
                        ideal_depth := calculate_ideal_depth(histogram, node_limit)
                        trace := mark_as_implicit_root(e, ideal_depth, trace)
                        Reset histogram
        If e is an exit, fail or excp event for call call_number
                inside := false
Until the event number of e is end_event
```

Figure 5.5: Algorithm for building the annotated trace and calculating the ideal depth of each implicit subtree.

considering there are seven reexecutions. In a realistic debugging session the user would likely spend a lot more time answering debugger questions. For higher values of *node_limit* memory space starts to become an issue.

It now becomes feasible to use a parameter independent of the program being debugged to control the space-time trade-off. A value for *node_limit* in the range of 50 000 to 200 000 allows us to feasibly debug the simple "stick" example as well as the Mercury compiler.

5.6 Related work

For non-strict functional languages such as Haskell there is the additional technical problem of dealing with non-strict evaluation when building the EDT. Non-strict evaluation is tricky,

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
50	26	520	20	39	41	16	24
100	13	676	52	32	34	16	24
200	8	928	116	29	31	16	24
500	5	2 486	497	27	29	16	24
1 000	4	2 000	500	27	29	16	24
2 000	4	4 746	1 186	26	29	16	27
5 000	3	11 828	3 942	26	28	17	27
10 000	2	16 360	8 180	26	28	18	27
20 000	2	32 730	16 365	26	28	19	30
50 000	2	80 596	40 298	26	28	24	35
100 000	1	65 524	65 524	27	29	27	41
200 000	1	131 060	131 060	27	29	31	41
500 000	1	262 132	262 132	28	30	42	58
1 000 000	1	524 262	524 262	29	32	66	74
2 000 000	1	1 046 852	1 046 852	34	36	117	132

Table 5.15: "fib" using the calculated ideal depth.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
1 000	1 010	1 003 940	994	641	685	228	234
2 000	502	1 000 988	1 994	332	354	203	209
5 000	200	998 800	4 994	151	161	203	209
10 000	100	999 400	9 994	94	101	195	201
20 000	50	999 700	19 994	63	67	195	201
50 000	20	999 880	49 994	47	49	203	209
100 000	10	999 940	99 994	40	42	203	209
200 000	5	999 970	199 994	37	38	203	208
500 000	2	999 988	499 994	35	37	244	250

Table 5.16: "stick" using the calculated ideal depth.

because the order of evaluation is usually very different to the order nodes should appear in the EDT. Much research has and is being done to overcome this problem [34–36, 43]. Mercury doesn't have this problem, because the sequence of calls corresponds closely with their position in the EDT.

Nilsson and Fritzson [35, 36] also propose constructing the program trace piece by piece. They introduce the concept of the *query distance* to a node. This is the number of questions required to get to the node using a top-down, left-to-right search. Figure 5.6 (taken from

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
1 000	105	102 400	975	949	953	42	50
2 000	54	104 942	1 943	486	488	42	50
5 000	23	111 904	4 865	204	205	42	50
10 000	12	116 396	9 699	112	113	42	50
20 000	7	132 746	18 963	67	68	42	50
50 000	4	180 600	45 150	40	41	45	58
100 000	2	165 524	82 762	31	32	50	58
200 000	1	165 520	165 520	22	22	50	58
500 000	1	362 128	362 128	23	24	77	91
1 000 000	1	624 258	624 258	26	26	117	125
2 000 000	1	1 146 848	1 146 848	30	30	166	174

Table 5.17: "smallbig" using the calculated ideal depth.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
50	2 526	110 596	43	156	226	42	50
100	1 124	105 230	93	81	112	42	50
200	534	103 092	193	50	65	42	50
500	210	102 436	487	33	39	42	50
1 000	105	102 646	977	28	31	42	50
2 000	54	105 112	1 946	25	27	42	50
5 000	23	113 772	4 946	23	24	42	50
10 000	12	118 350	9 862	23	24	42	50
20 000	7	136 982	19 568	22	23	45	58
50 000	4	181 874	45 468	23	23	58	66
100 000	2	173 708	86 854	22	23	50	58
200 000	1	147 442	147 442	22	24	44	58
500 000	1	294 928	294 928	23	25	60	74
1 000 000	1	590 692	590 692	25	27	92	107
2 000 000	1	1 182 706	1 182 706	29	32	161	174

Table 5.18: "bigsmall" using the calculated ideal depth.

[35]) illustrates the query distance of some nodes in an EDT.

Nilsson optimistically adds nodes to the EDT and then uses the query distance to decide which nodes should be discarded if memory usage becomes too high. Nodes with higher query distances are the first to be discarded.

This, as Nilsson demonstrates in [35] works well for top-down search, since most of the

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
100	206	14 422	70	156	164	110	136
200	151	21 226	140	121	127	102	128
500	111	32 058	288	97	101	102	128
1 000	83	67 193	809	80	83	102	128
2 000	62	102 199	1 648	77	80	105	136
5 000	42	190 746	4 541	73	75	110	136
10 000	30	279 637	9 321	71	72	117	145
20 000	19	366 148	19 270	60	61	128	153
50 000	10	490 280	49 028	57	58	138	162
100 000	6	588 080	98 013	57	58	159	187
200 000	4	750 617	187 654	53	54	191	220
500 000	2	983 303	491 651	47	48	211	237
1 000 000	1	997 904	997 904	52	54	251	279
2 000 000	1	1 939 334	1 939 334	62	63	358	387

Table 5.19: The Mercury compiler using the calculated ideal depth.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_{CPU}	T_{WC}	RSS	VSZ
1 000	50	39 272	785	1 238	1 242	202	228
5 000	31	105 326	3 397	1 010	1 015	206	236
10 000	24	202 919	8 454	942	945	217	244
50 000	15	651 764	43 450	734	736	248	278
100 000	12	1 079 520	89 960	688	690	315	344
500 000	7	3 259 056	465 579	509	594	467	561
1 000 000	6	5 872 026	978 671	500	1 480	463	877
5 000 000	Out of Memory						

Table 5.20: The Mercury compiler with a bigger source file.

time the next question will be in a materialized portion of the EDT. However for other search strategies, such as divide-and-query or subterm dependency tracking, the query distance as described above ceases to become a useful heuristic to decide which nodes to throw away.

Modifying the notion of the query distance to work for different search strategies is not a viable option either. It is unclear how this modified query distance heuristic could be measured efficiently for the divide-and-query search strategy and it is impossible to measure for the subterm dependency tracking strategy, since it is impossible to know which subterms a user will wish to track beforehand. Also the user should be able to

Re-execution #	Nodes created	*depth_limit*
node_limit = 20 000		
2	18 418	10
3	18 590	10
4	19 998	1 934
5	19 994	1 999
6	19 994	1 999
7	19 994	1 999
8	19 994	1 999
node_limit = 50 000		
2	36 850	11
3	45 034	12
4	49 996	4 988
5	49 994	4 999

Table 5.21: "bigsmall" individual re-executions using the calculated ideal depth.

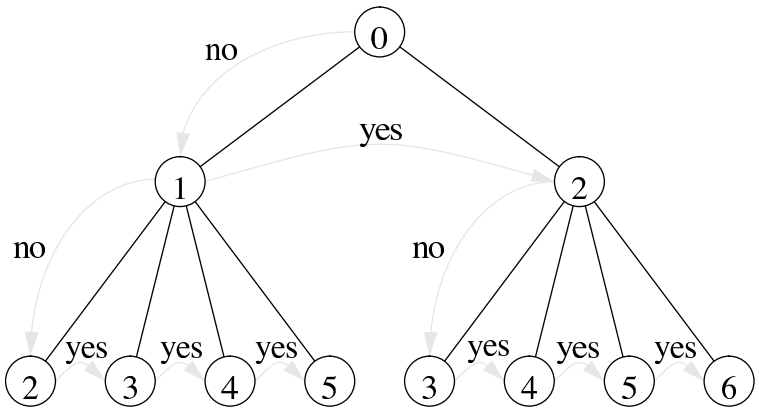


Figure 5.6: Query distance of nodes from the root node

switch search strategies mid-session. Nilsson's approach cannot be adapted to handle this situation without a substantial redesign.

Also, as Nilsson points out in [35] there is a penalty to be paid for first creating a node in the EDT and then disposing of that node later when resources become tight (mostly due to the extra work garbage collection must do). This is tolerable if a new portion of the EDT only needs to be built after having asked the user a few questions, however if several EDT portions need to be materialized between successive questions (as can be the case with divide-and-query and subterm dependency tracking) then this penalty becomes unacceptable.

Using our method we are able to know *ahead of time* how much of the EDT can be viably generated in a single reexecution, so no nodes are created only to be destroyed later.

Certainly for some long running programs it will become necessary to dispose of nodes which have been eliminated from the bug search. We have not (yet) implemented this feature in the Mercury declarative debugger, though conceivably that would not be hard to do. Nodes should only be deleted as a last resort, though, since even nodes which have been eliminated from the suspect area may be revisited if the user decides to revise a failed bug search, because they answered some questions incorrectly the first time round.

Naish and Barbour [34] also propose piecemeal generation of the trace using a predefined constant depth limit. The limitations of this approach have already been pointed out in section 5.2.

Another idea which has been around for a while is to not collect events for trusted predicates (both Sparud and Nilsson do this in their debuggers [35, 49]). We implement trusting in the Mercury declarative debugger too (see section 3.2), however, we do not throw trusted nodes away, for the simple reason that if we omitted trusted predicates from the annotated trace then we could not track subterms through them (see section 4.3). It is often the case that we wish to track a subterm through calls to library predicates, which are normally trusted. For example if we wish to track the origin of a term after it has been inserted and then extracted from a map, we cannot do so if the calls to `map.insert` and `map.search` (and all their descendant calls) have been omitted from the trace.

Another reason we keep trusted calls is that the user's trust in a predicate may be misplaced (see section 6.1 for a real example of this). In this case the user may wish to untrust a predicate. We do not then want to have to reexecute the program to collect the previously trusted nodes.

Chapter 6

Case studies

In this chapter we will examine four debugging sessions where real bugs were found in non-trivial programs using the declarative debugger.

Two of the bugs were in the Mercury compiler itself. This is a Mercury program of over 300 000 lines of code. The two remaining bugs occurred in the declarative debugger, which is also a Mercury program of approximately 10 000 lines of code.

6.1 Case study 1: A bug in the Mercury compiler

The first bug is one we found in the Mercury compiler. Its symptom is a thrown exception when a test program is compiled with the `--optimize-saved-vars-const` option.

We won't go into the meaning of the `--optimize-saved-vars-const` option here, save to say that it is an optimization which is inapplicable to the test case in question and that the compiler aborts when compiling the test case with this option.

Since the optimization shouldn't affect the output of the compiler in this case, our first course of action is to observe the call to the predicate which implements this optimization in the Mercury compiler. This predicate is named `saved_vars_const`. We fire up the procedural debugger, turn on I/O tabling (since we may have to retry across I/O actions — see section 3.5), put a break-point on the predicate `saved_vars_const` and continue until we hit the first call to `saved_vars_const`. We then use the `finish` command to continue to the corresponding `exit` event.

```
1:  1 1 CALL pred top_level.main/2-0 (det) top_level.m:48
```

```
mdb> table_io start
I/O tabling started.
mdb> break saved_vars_proc
  0:  + stop interface pred ll_backend.saved_vars.saved_vars_proc/8-0 (det)
mdb> continue
  2567051:  745876 25 CALL pred ll_backend.saved_vars.saved_vars_proc/8-0 (det)
    saved_vars.m:63 (mercury_compile.m:2694)
mdb> finish
  2570947:  745876 25 EXIT pred ll_backend.saved_vars.saved_vars_proc/8-0 (det)
    saved_vars.m:63 (mercury_compile.m:2694)
```

The three numbers at each event indicate the event number, call sequence number and call depth respectively. The first point to note here is the relative ease with which we can get to the interesting part of the program using the procedural debugger. Concepts such as break-points are not as intuitively integrated into the declarative debugger and nor do they need to be, because the procedural debugger does an excellent job of navigating the program execution. This reenforces the idea that the procedural and declarative debuggers play a complementary role.

The head of the `saved_vars_proc` predicate looks like the following.

```
saved_vars_proc(PredId, ProcId, ProcInfo0, ProcInfo, !ModuleInfo, !IO) :-
```

The arguments of interest here are `ProcInfo0` and `ProcInfo` which are representations of a procedure body, annotated with extra information, such as determinism information, before and after the optimization is applied. `ProcInfo0` and `ProcInfo` should be the same since the optimization shouldn't affect any of the procedures in the input program. Now we cannot easily compare the values of `ProcInfo0` and `ProcInfo` by eye, since they each require a few hundred lines to display. To find out if they are equal we use the procedural debugger's `save_to_file` command to first save their textual representations to two separate files and then perform a diff operation on the two files. [1] This reveals a difference between the information about one of the goals in `ProcInfo0` and `ProcInfo`. Specifically the difference is in a structure which describes the state of instantiation of the variables in the goal. For `ProcInfo0` it is:

⋮

[1] We have just added a feature to the debugger which allows a diff operation to be done between two values recorded at any point in the execution from within the debugger. This tools, however wasn't available at the time we were looking for the bug in question.

```
goal_info(det,
  reachable(
    map([
      2 ->
        bound(unique,
          [functor(
            cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
            [bound(unique, [functor(int_const(42), [])])])])])),
⋮
```

while in `ProcInfo` the value of the structure is:

```
⋮
goal_info(det,
  reachable(
    map([
      2 ->
        bound(unique,
          [functor(
            cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
            [free])])])),
⋮
```

The difference is in the last line which gives the instantedness of the argument of the functor `"foo"` which variable number 2 is bound to. In `ProcInfo0` it is bound to 42, while in `ProcInfo` it is free. This is erroneous behaviour, since the instantedness of variable 2 should remain unchanged. We start up the declarative debugger which asks us about the `saved_vars_proc` atom we have just been exploring.

```
mdb> dd
saved_vars_proc(0, 0,
  proc_info(context("unused_args_test2.m", 4),
    varset(var_supply(4), map(
      [1 -> "HeadVar__1", 2 -> "Y", 4 -> "Z"]), map([])),
    map([
      2 ->
        functor(atom("."),
        [functor(atom("unused_args_test2"), [], context("", 0)),
        ⋮
        1875 further lines, representing the (partial!) values of
        all the arguments of saved_vars_proc.
```

```
        ⋮
Valid?
```

The question takes up many screen pages, because we have set the print depth to quite a high value. This illustrates a major problem with answering declarative debugging questions, namely that they can simply be too big to even read through in a reasonable amount of time! Luckily in this case we were able to determine an answer to the question by using external tools (namely the Unix diff command).

6.1.1 Using subterm dependency tracking

Since we know that the value of the fourth argument of the call to `ProcInfo` is incorrect, we navigate to the incorrect variable-to-state-of-instantiation map and mark it, which produces the following result.

```
browser> browse
browser> cd 4/12/2/2
browser> print
reachable(
  map([
    2 ->
      bound(unique,
        [functor(
          cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [free])])]))
browser> mark
instmap_delta_restrict(
  reachable(
    map([
      2 -> bound(unique,
        [functor(cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [free])]),
      4 -> bound(unique, [functor(int_const(42), [])])])),
  [2],
  reachable(
    map([
      2 -> bound(unique,
        [functor(cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [free])])])))
Valid?
```

It should be noted that the process of navigating to the incorrect subterm was somewhat more involved than the simple '`browser>` *`cd 4/12/2/2`*' shown here. We had to navigate to the right term by checking at each stage where we were (much like a Unix user would navigate an unfamiliar file system). The generic graphical term browser also helped here (in fact the screen shot in figure 3.3 is taken from this debugging session). We omit these details here for simplicity.

Since the entire variable-to-instantiation map is incorrect the user marked the `reachable` functor enclosing it. We will investigate the effect of marking subterms nested inside the `reachable` functor later on in this section.

The next question is considerably simpler than the previous one. Instead of over 1800 lines we are now presented with only 13!

`instmap_delta_restrict` is supposed to eliminate variables from its first argument which are not in the list in its second argument and return the result in its third argument. It appears to be doing its job correctly: eliminating variable number 4, which doesn't appear in the list in the second argument (the second argument contains only the variable number 2). However the first argument of `instmap_delta_restrict` is wrong. It still says the argument of functor "`foo`" is free when it should be bound to 42. Marking the first argument leads to the following.

```
Valid?  browse 1
browser> mark
instmap_delta_apply_instmap_delta(
  reachable(
    map([
      2 -> bound(unique,
        [functor(cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [free])])])),
  reachable(
    map([
      2 -> bound(unique,
        [functor(cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [bound(unique, [functor(int_const(42), [])])])]),
      4 -> bound(unique, [functor(int_const(42), [])])])),
  large_overlay,
  reachable(
    map([
      2 -> bound(unique,
```

```
        [functor(cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [free])]),
      4 -> bound(unique, [functor(int_const(42), [])])])))
Valid?  info
Context of current question :  instmap.m:588 (mode_util.m:1075)
Search mode                 :  binary search on path
Number of suspect events    :  3,865
The current question was chosen because the marked subterm was bound by
the unification inside the predicate
hlds.instmap.instmap_delta_apply_instmap_delta/4 (instmap.m:619).  The
path to the subterm in the atom is 4.
```

The 'info' command gives information about the state of the search. If a subterm was marked it also gives the location of the unification that bound the subterm.

`instmap_delta_apply_instmap_delta` is supposed to produce a union of the variable maps in its first and second arguments and return the result in its fourth argument. If a variable appears in both arguments then the second argument should take priority. This is not happening, since variable number 2 appears in both the first and second arguments, but in the fourth argument the instiantedness from argument one is used. The third argument gives some operational information about which algorithm should be used to perform the union. Clearly the answer to the question must be "no". Giving this answer results in the following diagnosis from the debugger.

```
Found incorrect contour:
instmap_delta_apply_instmap_delta(
  reachable(
    map([
      2 -> bound(unique,
        [functor(cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [free])])])),
  reachable(
    map([
      2 -> bound(unique,
        [functor(cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [bound(unique, [functor(int_const(42), [])])])]),
      4 -> bound(unique, [functor(int_const(42), [])])])),
  large_overlay,
  reachable(
    map([
      2 -> bound(unique,
```

```
        [functor(cons(qualified(unqualified("unused_args_test2"), "foo"), 1),
          [free])]),
      4 -> bound(unique, [functor(int_const(42), [])])])))
Is this a bug?
```

The reason a bug is diagnosed is because all the descendants of the call to `instmap_delta_apply_instmap_delta` are in the Mercury standard library and therefore trusted. Examining the source code, there is only one child, which is a call to the library predicate `overlay_large_map`. Examining the code for `overlay_large_map` reveals a very obvious copy-and-paste bug in its definition. Fixing this then causes the test case to compile without any problems.

In this case our trust in the Mercury standard library was misplaced, since one of its (recently changed) predicates turned out to be buggy. However this did not turn out to be a problem for us, since the bug was a direct child of the call to `instmap_delta_apply_instmap_delta`.

What would have happened if the user had marked a subterm inside `reachable`, instead of `reachable` itself? If the user had marked the map one level down, the sequence of question would have been exactly the same. However, if the user had marked the `functor` or `free` functors, then the next question would have been about a direct child of the call to `saved_vars_proc`. The reason for this is that the subterm dependency tracking algorithm traces the subterm back to an input to `saved_vars_proc` (specifically the `ProcInfo0` argument). Because this node is erroneous, tracking stops, and the debugger asks about the last node the subterm was tracked through.

In this session the call to `saved_vars_proc` is erroneous, however in other cases the node where the subterm is marked may not be erroneous, even if the subterm is output. The user may wish, in these cases, to see the origin of the subterm, regardless of whether the origin lies outside the subtree rooted at the node where the subterm was marked. The best solution here might be to not assume the node where the subterm was marked is erroneous or inadmissible, and to allow the user to manually browse the dependency chain.

The user should be able sort and filter the nodes in the dependency chain based on different criteria. For example the user may wish to see only calls to certain predicates or to all predicates in a certain module, through which the marked subterm was passed. It would also be useful to sort the dependency chain by how deep the marked subterm is nested in its argument at each node. This way the user can examine nodes where the subterm is near the top of the argument it appears in first. Applying this ordering to the

current example, the node for `instmap_delta_apply_instmap_delta` would be one of the first nodes viewed by the user, since the subterm is near the top of its argument in this node. The node outside the call to `saved_vars_proc` where the subterm is bound, would also be one of the first nodes viewed by the user. Since the questions about these nodes would be small, the user would easily be able to spot the erroneous node. Implementing these enhancements is future work.

The original session illustrates how subterm dependency tracking can reduce the number of questions. The resulting questions also tend to be simpler to answer.

It is worth noting that the user found this bug *without* having any familiarity with the code. He was able to answer the questions based only on their names, comments in the code and the fact that he knew `saved_vars_proc` should not have modified its arguments. In the real world it is very often the case that software systems must be maintained and debugged by programmers who did not originally write the software. Any debugging tool capable of skipping parts of the execution irrelevant to the cause of a bug is therefore very useful.

6.1.2 Using divide-and-query

As a comparison, let us now recreate the debugging session using divide-and-query, this time without making use of the subterm dependency tracking feature.

We issue the command '`dd -s divide_and_query`' to start the declarative debugger in divide-and-query mode. The first question asked is the same as before, since the debugger always asks about the node where the `dd` command is first issued, regardless of the search strategy. This time, instead of marking the incorrect subterm we simply answer 'no' to the question.

This results in a question about the predicate `implicitly_quantify_conj_2`. The question requires 184 lines to display in total. This is at least one order of magnitude smaller than the previous question, but is also one order of magnitude *bigger* than the question we got from marking the incorrect subterm. To someone unfamiliar with this code (as the user was) the question is almost impossible to answer — especially given the fact that this particular predicate has no informative comments in the source code, being only a helper predicate of another algorithm.

One solution is to skip the question and hope an easier one comes along, but for the purposes of comparison let us suppose that the user knows the answer to the question (*we do* know the answer to the question, since we know where the bug is, from the previous

session).

It turns out that the answer to the question is 'yes', because the atom is valid. Progressing with the session in this manner, results in five further questions before the same bug is found (making seven questions in total). Each of the questions, except for the last one, required over a hundred lines to display.

Although finding a bug in a search space of 3 896 events with seven questions is not too bad, subterm dependency tracking is the clear winner in this case since it found the bug in fewer, simpler questions.

6.2 Case study 2: A bug in the termination analyser

This was a bug which was found in one of the termination analysers in the Mercury compiler (there are two).

Again the symptom is a thrown exception when the termination properties of a certain predicate are being checked by the compiler.

In this case there was no prior suspicion of any particular predicate in the compiler, so we couldn't simply put a break-point on a procedure to see if it was behaving badly. However, the procedural debugger has a feature which allows the program to be run until an exception is thrown. So this was the logical first course of action.

```
  1:  1 1 CALL pred top_level.main/2-0 (det) top_level.m:48
mdb> table_io start
I/O tabling started.
mdb> exception
  3702795:  1159300 39 EXCP pred exception.throw_impl/1-0 (erroneous)
    exception.m:700 (exception.m:361)
```

`throw_impl` is a builtin predicate in the 'exception' standard library module which is used to implement the `throw` predicate (exceptions in Mercury are discussed in more detail in section 2.3.4). This means that `throw_impl` is trusted by default. There is no problem, however, with invoking the declarative debugger on a trusted predicate. If this is done, it will simply search for the first descendant of the call which is not trusted. If there are no untrusted descendants it will look in the subtrees rooted at ancestor nodes for the first untrusted call it finds and ask about that. In this case `throw_impl` has no descendants, so

the search looks in its ancestors. The following is what is output when the user issues a 'dd' command at the excp event for throw_impl.

```
mdb> dd
Call entailed(varset(var_supply(19), empty, empty),
  [eq([], r/2)], lte([], r(1, 1)))
Throws "entailed/3:  inconsistent constraint set."
Expected?
```

The declarative debugger has searched up the EDT to find the first excp node for an untrusted predicate, which turned out to be for a call to entailed. This is useful behaviour since it represents the call where the exception was thrown at the user's level. Because all the internal predicates of throw as well as throw itself are trusted by default, the declarative debugger does not bother asking the user about their correctness.

The user then answers 'yes' to the question, since the constraint set passed to entailed is indeed inconsistent. The user should have answered 'inadmissible', since entailed should never have been called with the given constraint set. However, this doesn't make much difference here, since in the current implementation of the debugger, 'yes' answers have the same pruning effects as 'inadmissible' answers.

Note that the question was quite small, since the search is essentially started at a leaf node of the EDT. This suggests that this type of bottom up search can be useful when dealing with code which throws exceptions, since one can start at the throw (or in this case throw_impl) node and work up to find the cause of the thrown exception.

The search up the tree continues with the user answering 'yes' to the next three questions. The questions start to get bigger further up the tree. Eventually the user is unsure how to answer the debugger's question (because the question is now quite complicated), so the user responds with 'skip' which tells the declarative debugger to ask another question (see section 3.3). The next question is also too complicated, so the user skips it too.

The search continues in this way with the user skipping questions which seem too complicated and answering only questions they feel confident about. This results in a total of five skipped questions, five 'no' answers, twelve 'yes' answers and three 'inadmissible' answers, before finding a buggy node.

Although this seems like a fair number of questions, each question took no more than a few minutes to answer, since questions requiring more time were simply skipped.

This example highlights the importance of allowing the user to answer 'don't know' to

questions and shows that the debugger can still locate the bug even if not all the questions are answered. In a sense the 'skip' answer gives the user some basic control over the bug search, since they can redirect the debugger simply by skipping nodes.

The found buggy node turned out not to be the whole bug, because the actual bug was distributed over more than one predicate. The found node did however highlight an inconsistency between the user's intended interpretation and the model of the program which quickly lead the user to the real bug.

The total session took approximately one hour.

6.3 Case study 3: Debugging the debugger

Earlier versions of the declarative debugger had a defect where, if the declarative debugger was started at a trusted node, it would abort. The problem was being caused by two separate bugs. We successfully used our implementation of divide-and-query to find both bugs (we can use the declarative debugger on itself as long as we don't use a feature that triggers the bug we are trying to find).

The search space for the first bug consisted of 893 events. The debugger asked eleven questions before finding the bug. The search space for the second bug consisted of 166 events and the debugger asked eight questions before finding the bug. The questions were all relatively simple to answer, because they never grew bigger than a few dozen lines.

We can see here the logarithmic relationship between the number of events in the search space and the number of questions it took to find the bug.

As a comparison the sessions were reproduced post-mortem, using only the basic top-down search, with surprising results. Using top-down search, the first bug was found in twelve questions (as opposed to eleven for divide-and-query), however the top-down questions were on average even shorter than the divide-and-query questions. This was due to the fact that the top-down search asked questions about leaf nodes, which divide-and-query wouldn't have asked about. The second bug was found in five questions using top-down search (as opposed to eight for divide-and-query). The reason top-down search performed so well here is because of the relatively shallow EDT involved. Had the EDT been thinner and deeper, and the bug somewhere near the bottom of the EDT, then divide-and-query would have found the bug in pretty much the same number of questions, but top-down search would have asked many more questions to locate the bug.

On the other hand, the questions asked with divide-and-query tended to come from

different, often unrelated, parts of the program. This is not too much of a problem if the user is familiar with the code, but it can make the questions almost impossible to answer if they are not (as we found in case study 1). Using a search strategy like top-down search, or subterm dependency tracking for unknown code is easier to follow, since the dependencies between the nodes are more obvious. For divide-and-query the sequence of questions do not usually follow the flow of execution and so do not coincide with the user's mental model of the program. This can make the questions more difficult to answer, since the user is required to constantly switch mental contexts.

Divide-and-query however remains an essential tool, because of its ability to put an upper bound on the number of questions when the search space is large, even when nothing is known a priori about the location of the bug.

Because we allow the user to switch search strategies between top down and divide-and-query on the fly, the user is free to make use of either depending on the situation.

Chapter 7

Future work

The case studies from the previous chapter, and other experiences we have had, highlight many areas for possible future work. In this chapter we explore some of these problems and offer possible solutions.

7.1 Search strategy improvements

At present we don't use all the information available in the computed dependency chain. Although the current behaviour of zooming in on the binding node has proved very useful in practice, there is more that could be done when the binding node turns out not to be erroneous.

If the user asserts that the binding node is inadmissible by marking one of its inputs then the obvious and useful behaviour is to track the origin of the marked input and ask about its binding node.

If the binding node is not erroneous then our current method of performing a binary search on the path between the binding node and the last erroneous node has not proved particularly useful in practice. Automatically adopting this search strategy when the user asserts that the binding node is correct can be very confusing, especially when the user is unfamiliar with the code they are trying to debug. This is because the subterm may be buried inside a much larger structure higher up the dependency chain. No doubt highlighting the position of the marked subterm in the larger structure would be beneficial here.

Instead of doing a binary search on the dependency chain we suspect a better heuristic would be to first ask about nodes on the dependency chain where the the subterm in near

the top of the argument it appears in. In such nodes the subterm is more easily accessible by the code in the body of the predicate the node refers to, which would seem to indicate that the code is more likely to be buggy, since it would require a smaller modification to allow it to access the subterm directly. Questions in the dependency chain where the subterm is near the top of its argument would also generally be smaller and easier for the user to answer.

Another alternative would be to allow the user to manually browse the dependency chain. This would give the user a better idea of the process that lead to the erroneous atom and would allow them to make a better decision about which nodes in the EDT to explore next.

Currently when the user marks a subterm in a node they also assert that the node is erroneous or inadmissible (depending on whether the subterm is output or input). While often this is useful behaviour, since it reduces the search space, it can on occasion be unhelpful. For example, consider the case where a question is asked about an atom which contains a subterm whose value the user is suspicious of, but the user is not sure that the atom is erroneous or inadmissible. The user might in this case not wish to assert anything about that atom, but may wish to find out where the suspicious subterm came from. Thus a better approach might be to view a marked subterm as merely a hint to the debugger where it should ask the next question, instead of an assertion that the node is erroneous or inadmissible. Even better, allow the user to decide whether marking a subterm in an atom makes an assertion about the correctness of the atom or not.

7.2 Improving resource consumption

The main problem with our approach to building the annotated trace given in chapter 5, and indeed with all the related approaches covered in section 5.6, is the memory leak introduced by recording references to the values of the arguments of all `call` and `exit` nodes we collect. This can become a problem with programs which pass around large, frequently updated, data structures.

We observe that the actual values of the arguments are only needed when the question is to be asked of the oracle. None of the search strategies we have implemented for the Mercury declarative debugger so far care about the argument values of the atoms in the tree (although conceivably this situation could change in the future). Thus for search strategies which examine large portions of the EDT before asking a question, most of

the values referenced by nodes in the EDT will remain unused. Only a small minority are required. One possible solution to the memory leak issue is therefore to simply *not* record any references to any values in the program. When a question needs to be asked about a node, execution can be rewound to before the call corresponding to the node and the call reexecuted to compute its argument values. This reexecution would be relatively quick since no extra work, such as collecting nodes, or computing a histogram, would need to be done (the EDT is already constructed, it is only the argument values which are missing). A variation of this approach would be to store argument values at *some* points in the EDT, much like Plaisted [42] does. If a particular argument value is needed and not stored then we can rewind to the nearest ancestor call whose argument values we *do* know and reexecute from there. On average we would then rewind to a point closer to the call whose argument values we need to calculate which would improve performance. We could guarantee an upper bound on how far we would have to rewind the computation by adjusting the intervals we remember argument values at.

If the garbage collector could be queried about the sizes of values reachable only by nodes in the EDT, which didn't contain subvalues referenced by variables in the program, then we could selectively delete those nodes in the EDT which are holding the most memory. However, this would require substantial work to implement since we currently use a conservative collector [3] which cannot provide such information.

If we used a garbage collector which supported weak references (i.e. references which the garbage collector could recycle if needed), then we could use those to reference argument values in the program from the EDT. The garbage collector could recycle the argument values if memory became tight. This would require a check to see if a referenced argument value had been recycled by the garbage collector, before being accessed.

Another area where our approach currently falls short is with code which relies heavily on backtracking, specifically clauses such as:

```
queen(Data, Out) :-
        qperm(Data, Out),
        safe(Out).
```

where a generator predicate (`qperm/2`) generates possible solutions and a test predicate (`safe/1`) checks if each solution is valid and fails if it isn't, causing the generator to generate more solutions. All `call`, `redo`, `exit` and `fail` events generated by `qperm/2` and `safe/1` will have the same depth. With our current algorithm either *all* these events will be added to the annotated trace during a particular reexecution, or *none* of them will be added.

This can be a problem if there are a large number of these events.

One solution is to permit nodes in the annotated trace whose children are not *all* materialized. So in addition to marking a node as an implicit root, which indicates that *none* of its children are present, we might allow nodes to be marked as the roots of *partially materialized* subtrees too. Before the root of a partially materialized subtree can be considered a bug, all its children would first need to be materialized.

Besides a few artificially constructed examples, we have not found examples of heavy backtracking in practice, simply because most real programs written in Mercury tend to use only deterministic and semi-deterministic code.

Another approach to building the annotated trace would be to do so pro-actively, instead of on demand. In most cases the majority of debugging time is spent waiting for the user's response to queries. This time could instead be spent pro-actively building new portions of the annotated trace. We could adopt a heuristic that assumes the user would answer "no" to most questions. While the user is deciding on their answer we might start materializing implicit subtrees reachable from the current node. If the user did answer "no" then the search could continue downwards without the user having to wait for the subtrees to be materialized. If the user answered "yes" then there would be no penalty, since the part of the tree above the node would already have been materialized.

There are situations where it is hard to avoid materializing almost all of the annotated trace. This situation can arise, for example, if we wish to track a subterm from some point near the end or the program to its origin near the start of the program. It can be the case that the value of the subterm occurs everywhere in the program (perhaps it is that value of some input parameter). In this case the track function from section 4.3, figure 4.10 will cause almost all of the annotated trace for the program to be materialized. There are several solutions to this. One is to simply discard parts of the annotated trace if memory becomes tight. The parts that could be discarded might be the subtrees of calls into which the subterm is passed in and out of unmodified. This solves the space problem, but doesn't address the time problem — the annotated trace for the discarded subtree must still be initially constructed for the track function to work, which takes time. Another approach is to modify the track function so that it can detect the situation where a subterm is passed into a node and then returned from the node unmodified, without it having to explore the subtree rooted at that node. One way to do this is to check if there is a pointer to the same subterm in the input arguments of an atom where the subterm appears in an output argument of the atom. Searching through all input subterms of course might end up taking

longer than just materializing the subtree if the terms are big. One heuristic which could be used to reduce the time taken to perform this check might be to only look in input arguments of the same type as the output argument the subterm appears in and then only check at the same position the subterm appears at in the output argument. Another heuristic would be to check only arguments whose names are the same as the argument the subterm is in, except for a numeric suffix (such as '`ProcInfo`' and '`ProcInfo0`'). This convention would indicate that one of these arguments is an update verson of the other and so if the subterm appeared in any of the input arguments it would most likely appear in the one with the similar name.

7.3 Making questions easier to answer

Making questions easier to answer needs to be the main focus of future research. We have discovered that in practice it is *more* important to make the questions easier to answer than it is to reduce the number of questions. For example consider two possible scenarios where the debugger, employing one search strategy asks only one question to find a bug. The question it asks is, however, very difficult to answer and it takes 2 hours to do so without any interruptions. Suppose that another search strategy takes 120 questions to find the same bug, but each question is very easy and only takes a minute to answer. Which search strategy is better? If the user is able to spend two uninterrupted hours on the problem, then they both find the bug in the same amount of time. However if the user is interrupted, then when they return to their debugging session, they will have to first recall all they had learnt while trying to answer the one hard question, but they will still be able to answer the next easy question in one minute. This means that with interruptions the search strategy which asks one question could take much longer than the search strategy which asks 120 questions. Interruptions are a fact of life in most work environments.

There are several ways to reduce the complexity of the debugger's questions. One solution which needs to be investigated is whether a search strategy which simply favours simpler questions would be more effective. A heuristic which could be used to determine which questions are simpler would be how many characters are required to display the question on screen. Perhaps this could be combined with divide-and-query so that the search would prefer simpler questions which also eliminate many nodes when answered. This could be done by applying the divide-and-query algorithm to the subset of suspects whose complexity is below a certain threshold.

Often questions contain data structures which appear in previous questions, or even in different arguments of the same question. Pointing out these common subterms to the user would mean the user would not have to re-explore the save structures only to discover they have seen the structure before. This would also indicate to the user which terms are different, when the user would have expected them to be the same. An enhancement to this approach would be to show users only the differences between arguments of successive questions (if the differences are small). This would make the questions easier to answer, since it is often only the *changes* a call makes to a data structure which indicate whether the call is correct or erroneous.

Delta debugging techniques [55] could also be applied to the input of erroneous calls to reduce the size of the input, but maintain the erroneous behaviour. The user might have to supply some test which determines whether the behaviour is erroneous or not (if the symptom is a thrown exception then the test is trivial). Delta debugging could then be applied to reduce the size of the question. This would also mean that a different EDT would be searched, since the computation would be different, however the bug found would most likely be the same.

7.4 Handling destructive update

Currently the declarative debugger cannot debug code which updates mutable data structures, such as arrays, since changes to mutable structures cannot be undone and so the program state cannot be rewound.

A simple solutions to this problem is to apply the same solution we currently use for handling code which does I/O (section 3.5). This would, however, mean that we would not be able to look at a mutable value in its entirety. Instead we would only be able to able to see the primitive operations which have been performed on the value. It remains to be seen whether this information will be useful enough to aid with debugging.

Copying mutable values before they are updated is another alternative, although this would be too expensive for big data structures such as large arrays.

7.5 Impure code

Mercury supports the controlled use of side effects through an impurity system [11]. The impurity system allows for impure or semipure predicates. Calls to impure predicates

produce side effects. The output of semipure calls can be different for different calls with the same input. Semipure calls do not however produce any side-effects. Impure goals cannot be reordered with respect to each other. Semipure goals can be reordered inbetween two impure goals, but cannot be reordered with respect to the impure goals. This is because semipure goals may depend on the side-effects of impure calls.

Impurity is often used to implement pure interfaces for libraries written in impure languages, such as C. Impure predicates can be made pure, for example by calling them from a predicate which accepts an I/O state pair. The side-effects are therefore reflected in the I/O state arguments.

Because impure goals produce side effects we cannot apply the same rules for determining the children of a node in the EDT for impure calls as we do for pure goals. Instead the children of an `exit` or `fail` raw EDT node corresponding to any impure call would likely have to be *all* the assertions events on the stratum leading up to the `exit` or `fail`.

This is because any of the impure children could affect the results of subsequently executed children. Even if a child is declared to be pure it may be the case that the purity promise is actually a bug, and we would like the debugger to detect such bugs. Determining which child calls are necessary and sufficient to determine the result of the parent call in the presence of impurity requires further research.

At present impurity does not represent much of a problem. For example in the source code for the Mercury compiler, there are no impure goals. Some library predicates (such as `solutions`) are implemented with impurity, but most of these have been thoroughly tested, so their promised pure interfaces are very likely to actually be pure. However the need to debug impure code will be increasing. This is because of the recent addition of logic constraint programming capabilities to Mercury. Here impure code is used to implement a pure interface to a constraint solver. The impure code used to implement the constraint solvers is usually much more complex than a simple interface to a foreign library, so there would be much more need for good debugging support.

Chapter 8

Conclusion

We have taken an existing top-down declarative debugger for Mercury and made it into a practical debugging tool which is capable of finding real bugs in complex programs.

Our debugger can handle I/O and exceptions which makes it applicable to most Mercury programs. Useful features such as (a) the ability to trust predicates, functions or entire modules, (b) inadmissibility (c) "don't know" responses and (d) customisable term visualisations make the debugger easier use.

The debugger implements new and efficient, algorithms for two previously known search strategies, namely divide-and-query and subterm dependency tracking.

Divide and query is useful for quickly reducing the size of a large search space; that's what it was designed for. In our experience, divide and query is best used when the user is quite familiar with the intended semantics of most of the predicates involved. This is important, because the sequence of questions it asks can be very confusing to anyone unfamiliar with the code. In our experience, for smaller search spaces top down search is much more comfortable to use, even though it asks more questions, because the sequence of questions it asks generally follows the flow of execution of the program. This means that successive questions are clearly and closely related, making them much easier to answer. This effect is due to the cache-like behavior of people's short-term memory; you don't have to explore a possibly large term if you have explored a closely related term a few seconds ago, and you know what their relationship is. The random jumps made by divide and query virtually guarantee that there will be no meaningful relationships between successive questions, and even when there are (typically towards the end, where the suspect set is small) the user typically doesn't know about them. That said divide-and-query can be very useful to quickly reduce a large search space where the questions are simple enough

to answer without any context.

Tracking the origin of a subterm can be even more effective than divide and query at reducing the size of the search space, especially if the subterm is generated far away from where it is marked. The question about the atom which bound the subterm is generally also simpler than its predecessor, since its output is usually smaller than the term that the subterm appeared in when it was marked. In our experience, the sequence of questions generated by subterm dependency tracking is quite easy for the user to understand despite the large jumps it makes in the tree. This is because successive questions are closely related in a way that is meaningful to the user.

The user may use all three algorithms (top down, divide and query and subterm dependency tracking), switching between them and the conventional procedural debugger at will. This allows users to use whichever method they believe is best suited to the problem at hand, and makes them feel more in control. To make the best use of this flexibility, users of course need to understand the strengths and weaknesses of each algorithm.

From our practical experience the user should have as many tools available as possible to assist them in their debugging task. The user should also be able decide what role the declarative debugger plays in the debugging process, because in some cases it is sufficient to look at the value of a particular variable at a certain point in the execution to know what the bug is. It should also be easy for the user to impart any knowledge of the program and bug they might have to the declarative debugger so that it can be as helpful as possible when it is needed.

Ducassé [14] makes the point that different debugging strategies can be complementary if combined in the same system. We have implemented a hybrid approach to debugging making use of features from the declarative and procedural debugger and have found this to be an effective and flexible method in practice. This approach is also validated by the examples we give as case studies and other expriences we have had.

During our implementation of the Mercury declarative debugger it also became apparent that we needed a new technique to control the resources consumed by the annotated trace. The existing techniques in the literature were inadequate in the presence of multiple search strategies.

We have examined three new methods for determining how much of the annotated to build in a single reexecution. Two of these methods rely on approximations of the tree. These fell short for realistic programs with unpredictable tree shapes.

Our third method, which used a histogram to calculate exactly how much of the tree

could be viably generated in a single reexecution, proved much more promising for real programs. The additional space and time cost turned out to be surprisingly small.

The techniques presented for controlling the size of the generated EDT are certainly not specific to Mercury. They could be applied to any declarative debugger where there is a tree that must be searched and a mechanism whereby the necessary information could be gathered for implicit parts of the tree.

By applying the debugger to real examples we have gained valuable insight in to how declarative debugging can be applied in practice and also what its limitations are. Mercury was designed to build large-scale, efficient and reliable systems. We have shown that the Mercury declarative debugger is now up to the task of debugging those same software systems.

Bibliography

[1] Dominic Frank Julian Binks. *Declarative Debugging in Gödel*. PhD thesis, University of Bristol, September 1995.

[2] Inquiry Board. Ariane 5, flight 501 failure report, July 1996. Available from http://www.cs.unibo.it/~laneve/papers/ariane5rep.html.

[3] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[4] Mark Brown and Zoltan Somogyi. Annotated event traces for declarative debugging. Available from http://www.cs.mu.oz.au/mercury/, 2003.

[5] Lawrence Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.

[6] Miguel Calejo. *A framework for declarative Prolog debugging*. PhD thesis, Universidade Nova de Lisboa, March 1992.

[7] M. Cameron, M. Garca de la Banda, K. Marriott, and P. Moulder. ViMer: A visual debugger for Mercury. In *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 56–66, 2003.

[8] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and Tracing Lazy Functional Programs using QuickCheck and Hat. In *4th Summer School in Advanced Functional Programming*, number 2638 in LNCS, pages 59–99, Oxford, August 2003.

[9] Jonathan J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45(6):608–619, 2002.

[10] Neil Deakin and Aaron Andersen. The Mozilla XUL reference and tutorial. Available from http://www.xulplanet.com/, 2004.

[11] Tyson Dowd, Peter Schachte, Fergus Henderson, and Zoltan Somogyi. Using impurity to create declarative interfaces in Mercury. Technical Report 2000/17, Department of Computer Science, University of Melbourne, Melbourne, Australia, 2000. Available from http://www.cs.mu.oz.au/mercury/.

[12] Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. Run time type information in Mercury. In *Proceedings of the 1999 International Conference on the Principles and Practice of Declarative Programming*, pages 224–243, Paris, France, September 1999.

[13] Wlodzimierz Drabent, Simin Nadjm-Tehrani, and Jan Maluszynski. Algorithmic debugging with assertions. In *Workshop on Meta-Programming in Logic*, pages 501–521, 1988.

[14] Mireille Ducassé. A pragmatic survey of automated debugging. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 1–15, London, UK, 1993. Springer-Verlag.

[15] G. Ferrand. Error diagnosis in logic programming, an adaptation of E.Y. Shapiro's method. *Journal of Logic Programming*, 4(3):177–198, 1987.

[16] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimóthy. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, 1(4):303–322, 1992.

[17] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. *SIGSOFT Software Engineering Notes*, 24(6):303–321, 1999.

[18] Pat M. Hill and John W. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[19] Visit Hirunkitti and Christopher J. Hogger. A generalised query minimisation for program debugging. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 153–170, 1993.

[20] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[21] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[22] Robert A. Kowalski and Fariba Sadri. Logic programs with exceptions. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 598–613, Jerusalem, Israel, June 1990.

[23] Bil Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 2003.

[24] John W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[25] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[26] Jan Lönnberg, Ari Korhonen, and Lauri Malmi. MVT: a system for visual testing of software. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 385–388, New York, NY, USA, 2004. ACM Press.

[27] Ian MacLarty, Zoltan Somogyi, and Mark Brown. Divide-and-query and subterm dependency tracking in the Mercury declarative debugger. July 2005. To appear in the proceedings of the Sixth International Symposium on Automated and Analysis-driven Debugging, Monterey, California, September 2005.

[28] M. Maeji and T. Kanamori. Top-down zooming diagnosis of logic programs. ICOT Technical report TR-290, Institute for New Generation Computer Technology, Tokyo, Japan, August, 1987.

[29] Sougata Mukherjea and John T. Stasko. Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. Comput.-Hum. Interact.*, 1(3):215–244, 1994.

[30] B. A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 59–66, New York, NY, USA, 1986. ACM Press.

[31] Lee Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–285, 1992.

[32] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.

[33] Lee Naish. A three-valued declarative debugging scheme. *Australian Computer Science Communications*, 22(1):166–173, January 2000.

[34] Lee Naish and Timothy Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, January 1996.

[35] Henrik Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN international conference on Functional programming*, pages 36–47, Paris, France, September 1999. ACM Press.

[36] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

[37] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, April 1997.

[38] Dino Pedreschi and Salvatore Ruggieri. Verification of logic programs. *Journal of Logic Programming*, 39(1–3):125–176, 1999.

[39] L. M. Pereira and M. Calejo. A framework for Prolog debugging. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium (Volume 1)*, pages 481–495, Cambridge, MA, 1991. MIT Press.

[40] Luis Moniz Pereira. Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, pages 203–210, London, England, June 1986.

[41] Luis Moniz Pereira and Miguel Calejo. Algorithmic debugging of prolog side-effects. In *EPIA 89: Proceedings of the 4th Portuguese Conference on Artificial Intelligence*, pages 151–162, London, UK, 1989. Springer-Verlag.

[42] D. A. Plaisted. An efficient bug location algorithm. In *Proceedings of the Second International Logic Programming Conference*, pages 151–158, Uppsala, Sweden, July 1984.

[43] B. Pope and Lee Naish. Practical aspects of declarative debugging in Haskell-98. In *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–240, 2003. ISBN:1-58113-705-2.

[44] Stéphane Schoenig and Mireille Ducassé. A backward slicing algorithm for Prolog. In *Third International Static Analysis Symposium*, pages 317–331, Aachen, Germany, September 1996.

[45] Ehud Y. Shapiro. *Algorithmic program debugging.* MIT Press, 1983.

[46] Zoltan Somogyi. Idempotent I/O for safe time travel. In *Proceedings of the Fifth International Workshop on Automated and Algorithmic Debugging*, pages 13–24, Ghent, Belgium, September 2003.

[47] Zoltan Somogyi and Fergus Henderson. The implementation technology of the Mercury debugger. In *Proceedings of the Tenth Workshop on Logic Programming Environments*, pages 35–49, Las Cruces, New Mexico, November 1999.

[48] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.

[49] Jan Sparud and Colin Runciman. Complete and partial redex trails of functional computations. *Lecture Notes in Computer Science*, 1467:160–177, 1998.

[50] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In Ralf Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, September 2001.

[51] Mark Weiser. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 439–449, San Diego, California, 1981.

[52] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[53] Rickard Westman and Peter Fritzson. Graphical user interfaces for algorithmic debugging. In *Automated and Algorithmic Debugging*, pages 273–286, 1993.

[54] Andreas Zeller. Isolating cause-effect chains with AskIgor. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 296, Washington, DC, USA, 2003. IEEE Computer Society.

[55] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Softwiare Engineering*, 28(2):183–200, 2002.

[56] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Revised Lectures on Software Visualization, International Seminar*, pages 191–204, London, UK, 2002. Springer-Verlag.