Divide-and-query and subterm dependency tracking in the Mercury declarative debugger

Ian MacLarty, Zoltan Somogyi and Mark Brown
{maclarty,zs,mark}@csse.unimelb.edu.au
Department of Computer Science and Software Engineering
University of Melbourne
Parkville, 3010 Victoria, Australia

ABSTRACT

We have implemented a declarative debugger for Mercury that is capable of finding bugs in large, long-running programs. This debugger implements several search strategies. We discuss the implementation of two of these strategies and the conditions under which each strategy is useful.

The divide and query strategy tries to minimize the number of questions asked of the user. While divide and query can reduce the number of questions to roughly logarithmic in the size of the computation, implementing it presents practical difficulties for computations whose representations do not fit into memory. We discuss how we get around this problem, making divide and query practical.

Our 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 question sequences that are easier for users to answer.

1. INTRODUCTION

Declarative debuggers locate bugs in programs by asking an oracle (usually the user) whether the results of calls made during the execution of a buggy program are correct in the intended interpretation of the program, effectively comparing the actual semantics of the program with its intended semantics. The set of calls executed by the buggy program for the given test case is effectively a giant search space that the declarative debugger explores, looking for a node where the results of corrert subcomputations are combined into an incorrect result.

We have written a declarative debugger for Mercury, a purely declarative logic and functional programming language intended to support the creation of large, reliable programs. While Mercury's strong type, mode and determinism systems work together to catch many common programming errors at compile time, bugs in the program logic still occur. The Mercury declarative debugger takes advantage of Mercury's purely declarative semantics to automate much of the debugging task. Unlike previous declarative debuggers, ours is designed to work for real programs. We have successfully used it to diagnose bugs in the Mercury compiler (which consists of approximately 300,000 lines of Mercury code), as well as bugs in the declarative debugger itself (also a non-trivial program of approximately 8,000 lines of Mercury code).

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's search strategy is therefore to minimize the number of questions asked of the oracle: the fewer questions the user needs to answer, the sooner the bug is found. The classic algorithm for minimizing questions is Shapiro's divide and query algorithm [7]. While conceptually simple, implementing the usual version of this algorithm isn't really feasible when a representation of the computation to be debugged is too large to fit into memory all at once. We have therefore developed a modified version of the algorithm that does scale well to large computations.

While minimizing the number of questions is useful, it isn't a panacea. The time to find the bug is the product of the number of questions asked and the average time required to answer each question. While divide and query minimizes the number of questions, the fact that the questions it asks seem random to the user makes them relatively hard to answer. We have therefore developed another search strategy that tries to minimize the time required to answer questions as well as the number of questions. This strategy, subterm dependency tracking, depends on the user indicating not just the fact that an atom is not correct, but also *which part* of that atom is not correct. The declarative debugger will then track that subterm back to its origin. This strategy can converge on the bug even more quickly than divide and query, and the questions it asks are easier to answer because they are related in a fashion that makes sense to the user.

The structure of the paper is as follows. Section 2 presents the background we require on the Mercury language and on the infrastructure on which the Mercury declarative debugger is built. Section 3 presents an overview of the Mercury declarative debugger itself. Section 4 describes the feasibility problem of divide-and-query as well as our solution, while

section 5 describes subterm dependency tracking. Both sections contain comparisons to some related work. Our conclusion in section 6 evaluates these search strategies and discusses their strengths and weaknesses based on our experience with using them to search for real bugs in real programs.

2. BACKGROUND

2.1 Mercury

Mercury [10] has its roots in logic programming, which is why 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.

Mercury has a Hindley-Milner type system very similar to Haskell's. A mode classifies each argument of a predicate (or function, since Mercury supports functions) 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. This means that for each variable in each procedure, the compiler knows exactly which subgoal (call or unification) in that procedure makes that variable ground.

Each mode of a predicate or function has a determinism, which puts limits on 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. In our experience, few predicates are designed to have more than one solution; most have exactly one. For example, in the Mercury compiler, which is written in Mercury itself, roughly 85% of procedures are det, 14% are semidet, and only 1% multi or nondet.

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.

2.2 Debugger events

The Mercury debugger views the execution of a program as a sequence or *trace* of events; when debugging is enabled, the compiler generates code that gives the runtime system control at each event. The mechanisms involved in doing this are described in [9].

Events can be classified into two categories, *interface* events and *internal* events. Interface events describe the interaction between one invocation of a *procedure* (one mode of a predicate) and its caller, while internal events describe the flow of control inside the call. The four types of interface events supported by the declarative debugger correspond to the four ports in Byrd's box model [2] (the declarative debugger can handle programs that throw exceptions, but we do not discuss that capability in this paper):

`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.

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.

There are also several types of internal events. Their purpose is to mark the boundaries of (possibly) negated contexts and to record the outcomes of decisions about the flow of control. For example, if the program executes an if-then-else, there is an event when control enters the condition. If the condition succeeds, there is an event when control enters the then part; if the condition fails, there is an event when control enters the else part. At all of these internal events , the debugger also has access to the *goal path*, which gives the identity of the subgoal associated with the event (in this case it would specify exactly *which* if-then-else the event relates to).

3. OVERVIEW OF THE DECLARATIVE DEBUGGER

Declarative debugging involves querying an oracle about the validity of the results of calls made during the execution of a buggy program and then using this information to find a bug in the program.

Every `call` event corresponds to an atomic goal in a certain state of instantiation (depending on the mode of the

procedure). This atom has the actual arguments in the input argument positions and distinct free variables in the output argument positions. We refer to this as the *call atom* of the event.

The same view can be taken of `exit` events, although in this case the outputs as well as the inputs will be bound. We refer to this as the *exit atom* of the event. The exit atom is always an instance of the call atom for the corresponding call event.

Using these concepts, it is possible to interpret the events at which control leaves a procedure as assertions about the semantics of the program. These assertions may be true or false, depending on whether the program's actual semantics are consistent with its intended semantics.

The assertion corresponding to an `exit` event is that the exit atom is valid in the intended interpretation. In other words, the procedure generates correct outputs for the given inputs.

Every `fail` event has a matching `call` event, and a (possibly empty) set of matching `exit` events between the `call` and the `fail`. The assertion corresponding to a `fail` event is that every instance of the call atom which is true in the intended interpretation is an instance of one of the exit atoms. In other words, the procedure generates the complete set of answers for the given inputs. (Note that this does not imply that all exit atoms represent correct answers; some exit atoms may in fact be wrong, but the truth of the assertion about the `fail` event is not affected by this.)

If one of these assertions is wrong, then we consider the event to represent incorrect behaviour.

When users encounter an event for which the assertion is wrong, they can start the declarative debugger to diagnose the incorrect behaviour by giving the '`dd`' command to the procedural debugger at that event.

The declarative debugger will ask an oracle about the assertions made by the `exit` and `fail` events generated during the execution of a buggy program. The oracle therefore needs to have knowledge of the intended interpretation of the program. The obvious source of this information is the user, but the oracle may also use other sources, such as a specification, or the user's previous answers.

The oracle may give one of three possible answers to a question from the debugger:

1. *correct* – the assertion made by the event is consistent with the intended semantics of the program.

2. *erroneous* – the assertion made by the event is not consistent with the intended semantics of the program.

3. *inadmissible* – here the oracle makes no judgement about the assertion made by the `exit` or `fail` event, but instead asserts that the inputs at the corresponding `call` event violate a precondition of the procedure involved.

Given the execution trace of a program we construct an *evaluation dependency tree* or EDT which we use to search for bugs. The EDT is an instance of the declarative debugging scheme proposed by Naish [4], in particular its three valued variant [5]. We use the same EDT to diagnose both missing answer bugs and wrong answer bugs.

Each node in the EDT corresponds to an `exit` or `fail` event in the execution trace.

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.

Whether an `exit` or `fail` event is included in the EDT depends on the context in which the event was generated. An event is in a *positive context* if it was generated by a non-negated goal in the body of a procedure which succeeded, or was generated by a goal inside a negation which failed (i.e. the negated goal succeeded). An event is in a *negative context* if it was generated by a non-negated goal in the body of a procedure which failed, or was generated by a goal inside a negation which succeeded (i.e. the negated goal failed). We consider events generated by the condition of an if-then-else to be in a positive context if the condition succeeded and in a negative context if the condition failed.

The rules above are an exact description of what happens in the absence of nested negations. The actual rules we use are more complex, because they have to work even in the presence of nested negations (they are described in detail in [1]). However, the way they work can be understood in terms of a program transformation that replaces the body of each negated goal with a call to a newly defined procedure, thus eliminating nested negations. Our implementation faithfully mimics the tree structure you would get using this transformation but without including any events for the introduced procedures.

In positive contexts, we include the exit events only of non-backtracked-over calls, and for each such call, we include as children only the last exit event before the parent event. We also ignore any backtracked-over nested contexts when looking for children in a positive context. A `fail` event in a positive context cannot be an EDT node, since this would cause the call which generated the `fail` to be backtracked over. Previous solutions to a call in a positive context do not affect the result of the positive context – they cannot make the context negative or change the solution to a parent call. Even if the previous solutions are not in the intended interpretation, they cannot be the cause of the parent solution not being in the intended interpretation. Only the result of the last solution is used in the parent.

In negative contexts we include all the `exit` and `fail` events generated by child calls before the parent event. If a child succeeded, then it may have succeeded with a wrong answer, the correct version of which might have caused the negative context to become positive. If a call failed, then it may have missed a solution which in turn could have caused the negative context to become positive.

For example, consider the following predicate, which checks whether `Struct` contains a `Pairs` element such that all the key/value pairs in `Pairs` in are present in `Table`:

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

Assuming a call to `all_pairs_are_in_table` succeeded, the event(s) generated by the call to `extract_pairs` would be in a positive context, the events generated by calls to `list_member` would be in a negative context and the events generated by calls to `map_search` would be in a positive context (for a call to `all_pairs_are_in_table` to succeed, all

calls to `map_search` would also have to succeed). Suppose `Table` contained the pairs `1 - "one"` and `2 - "two"` and suppose the call to `extract_pairs(Struct, Pairs)` initially unified `Pairs` with `[0 - "zero"]`, but on backtracking unified `Pairs` with `[1 - "one", 2 - "two"]`. The resulting EDT would look like the tree in figure 1.

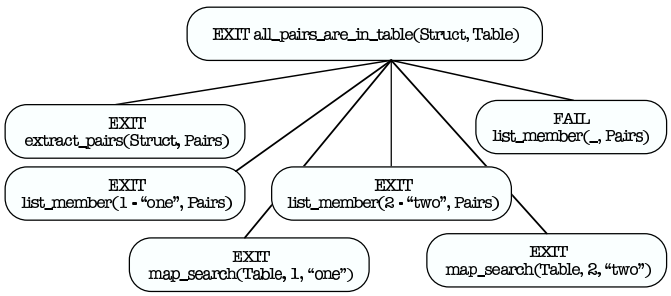

Figure 1: The EDT for a succeeded call to `all_pairs_are_in_table`

Note that since `extract_pairs` is in a positive context, we only include the last `exit` event for it. Since `list_member` is in a negative context, we include all its `exit` and `fail` events.

We construct the EDT on demand, based on a representation of the execution we call the *annotated trace*. 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 a search 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 [9, 8] to repeat that part of the execution, this time recording events to a deeper level. 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. The second difference is that we maintain extra links between nodes. For example, each non-`call` interface event contains a link to the previous interface event of that call and a link directly to the `call` event, while `call` events contain a link to the last `exit`, `redo` or `fail` event of that call. These links allow us to efficiently search through the annotated trace for the child calls which are relevant to any particular node in the EDT. For more details on the structure of the annotated trace, see [1].

We search for bugs in the EDT as we build it. A node in the EDT is buggy if it is erroneous and all its children are either correct or inadmissible. By asking questions about the validity of the assertions made by the events corresponding to the nodes in the EDT, we can eliminate portions of the EDT until we are left with an erroneous node all of whose children are correct or inadmissible.

Nodes are eliminated from the EDT in two ways. First, if the oracle asserts that a node is erroneous, then we only need to search the subtree rooted at the erroneous node – all other nodes can be eliminated. Second, if the oracle asserts that a node is correct or inadmissible, we can eliminate the subtree rooted at that node from the bug search. Although we eliminate the same nodes from the EDT for inadmissible and correct nodes, we might use knowledge of inadmissible nodes to focus the bug search, for example, by concentrating on nodes executed before an inadmissible call. Inadmissible calls are therefore useful because they give us more detailed information we can use to guide the bug search.

We call nodes which have not (yet) been eliminated from the bug search *suspects*. We call the set of suspect nodes in an EDT the *suspect area* of the EDT.

```
middleweight(LastErroneousNode, StepSize):
    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
        return PrevNode
    else
        return CurNode
```

Figure 2: Finding the middle weight node in an EDT

4. DIVIDE AND QUERY

4.1 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 [7]. This algorithm chooses a node in the suspect area of the EDT that divides the suspect area into two parts, each of equal weight (or as close to equal weight as possible) according to some weighting metric. Each time the oracle gives an answer, the weight of the suspect area 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.

Our version of the divide and query algorithm is shown in figure 2. The greedy search works because at each step `CurNode` is guaranteed to be at least as close to the target weight as any of its siblings.

4.2 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. We will then explore some alternative weighting metrics which are easier to calculate, but still good enough to yield effective results in most cases.

4.2.1 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 has shown in [7] 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.

Calculating the weight of an implicit subtree is, however, not as 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. This turns out to be harder than it may at first seem. There may be calls in the implicit subtree which produce multiple solutions. All the `exit` and `fail` events for such calls will be included in the EDT only if they are in a negative context. However we don't know if the context is negative or not until the negation succeeds (if the events are inside a negation), or the parent 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 they turn out to be in a negative context, and need to be included in the weight of the parent. 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.

For example consider the predicate `p`.

```
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 (this uncertainty is indicated by a dotted line in the diagram). 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 3). Suppose the weight of the subtree under this `exit` event is Y.

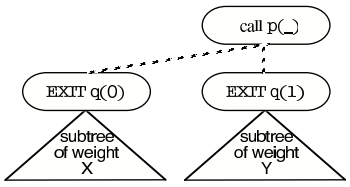

Figure 3: The potential EDT after `q` has produced two solutions.

Figures 4 and 5 show two possible next scenarios. In the scenario in figure 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 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 $X + Y + Z + 1$.

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

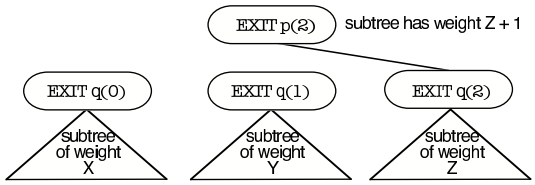

Figure 4: Scenario 1, `q` produces another solution.

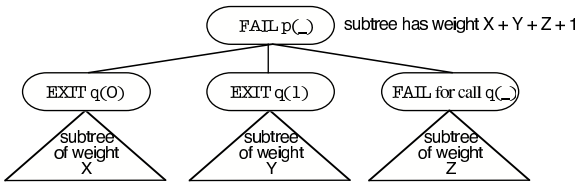

Figure 5: Scenario 2, `q` fails.

weights of their ancestors. If the entire EDT is available in memory, this is easy. 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.

4.2.2 *A practical approximation to the traditional weighting metric*

For implicit subtrees where multi or nondet code is executed we can *approximate* the weight by counting the number of descendant `exit` and `fail` 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 `fail` and `exit` 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. If such an implicit subtree is materialized, we must recalculate its weight and that of its ancestors. However, this is not a significant cost, since in our experience procedures that can succeed more than once are quite rare.

4.2.3 *A biased weighting metric*

Instead of counting the number of `exit` and `fail` 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 sub-

tree even if it is in memory. We have these 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 overapproximation, 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  4: call        event 17: exit
event  7: exit        event 23: redo
event 12: redo        event 45: 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 overapproximation, however, can cause the weights to become inconsistent. For example suppose further that the event number of the parent call to the above procedure had event number 3, the call failed without producing any solutions and the event number of the parent `fail` event 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, which as we mentioned occurs quite rarely in practice.

An interesting property of this weight 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.

This third weighting metric is the one we have implemented in the Mercury declarative debugger.

4.3 Using divide and query

We have successfully used our implementation of divide and query to find two unknown bugs in the declarative debugger itself (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 EDT for the first bug consisted of 893 events. The debugger asked 11 questions before finding the bug. The EDT for the second bug consisted of 166 events and the debugger asked 8 questions before finding the bug.

We can see here the logarithmic relationship between the number of events in the initial EDT and the number of questions it took to find the bug. Because of this we can approximate the number of questions remaining, which makes the debugging process more predictable in this respect. We can (and do) tell the user approximately how many more questions they will need to answer before a bug is found. This type of user feedback is not possible with a top down style search.

On the other hand, the questions asked with divide and query tend to come from different, often unrelated, parts of the program. 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. This is especially true when the search space covers lots of unrelated predicates.

Divide and query however remains an essential weapon in the user's arsenal because of its ability to greatly reduce the search space, 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.

4.4 Related work

Since Shapiro first proposed the divide and query algorithm, there has been no work that we know of aimed at improving the efficiency of the algorithm where the search space is large. A large search space is precisely the scenario under which divide and query is most useful.

Shapiro's 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.

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 node is not `CurNode` at any point in the algorithm in figure 2, 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 the user 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.

5. SUBTERM DEPENDENCY TRACKING

Previous declarative debuggers have asked users to say, for each atom, simply whether the atom is valid, erroneous or inadmissible. However, by accepting only these three answers, they have failed 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 the input argument of the call fail the precondition. 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.

We have included in the Mercury declarative debugger a mechanism that allows users to mark subterms of arguments when browsing the atom that the declarative debugger is asking about. If they mark a subterm of an output argument, they say that the atom is erroneous, and that the marked subterm is wrong, i.e. replacing the marked subterm, and possibly other subterms, with other values could make the atom correct. If they mark a subterm of an input argument, they say that the atom is inadmissible, and that the marked subterm is wrong, i.e. replacing the marked subterm, and possibly other subterms, with other values could make the atom admissible. 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.)

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.

Focusing the search onto the wrong subterm also makes the declarative debugger more understandable, since this behaviour is what the user would intuitively expect. Following the marked subterm also gives users some control in directing the bug search, while still remaining at a high level of abstraction.

Sometimes it is not trivial to specify which part of a term is wrong. Consider a predicate that expects as its input 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]`? Which one should the user mark? The answer is: it doesn't matter very much. Although different answers may lead to slightly different questions, they will all aim the search in very similar directions.

5.1 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 in the executable a representation of the bodies of all procedures, and this representation includes, for each goal, the list of variables bound by that goal. Given the predicate

```
:- pred rational_add(rational::in, rational::in,
    rational::out) is det.

rational_add(HV1, HV2, HV3) :-
    HV1 = r(An, Ad), HV2 = r(Bn, Bd),
    lcm(Ad, Bd, Cd),
    CA = M // Ad, CB = M // Bd,
    Ap = An * CA, Bp = Bn * BA,
    Cn = Ap + Bp,
    HV3 = 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 `+`. (Unlike in Prolog, in Mercury evaluable functions such as `+` can appear anywhere, not just on the right hand side of the `is` operator.)

This works for all predicates whose body is a simple conjunction. However, most predicates have more complex bodies, which 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 can thus reconstruct the sequence or conjunction of goals executed to compute the solution. This sequence can be expressed as what we call a *contour*:

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

```
origin(Head, Conj, Var, SubtermPath):
    find the goal G that produces Var in Conj
    if G is a construction X <= f(Y_1, ...Y_n) then
        Var must be X
        if SubtermPath = [] then
            return ⟨unify(G)⟩
        else
            SubtermPath must be [First | Rest]
            First must be in 1..n
            return origin(Head, Conj, Y_First, Rest)
    else if G is a deconstruction X => f(Y_1, ...Y_n) then
        Var must be one of the Y_is, say Y_k
        return origin(Head, Conj, X, [k | SubtermPath])
    else if G is an assignment unification X := Y then
        Var must be X
        return origin(Head, 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
        return ⟨call(G, k, SubtermPath)⟩
    else
        Var must be an input argument
        let ArgNum be number of that input argument
        return ⟨head(ArgNum, SubtermPath)⟩
```

Figure 6: The origin function

While procedure bodies may contain conjunctions, disjunctions, negations and if-then-elses nested arbitrarily, with the atomic goals being unifications and calls, a contour is always a conjunction in which all conjuncts are either unifications, calls or negated goals. You can boil a procedure body down to a contour by discarding those arms of if-then-elses and disjunctions which did not contribute to the solution being considered. This is basically what the declarative debugger does. When tracking the origin of an output argument in an exit event for a given call, it looks at the internal events of that call and at the interface events of the child calls one level down. By comparing this sequence of events with the code of the procedure involved, seeing which of these events have been backtracked over and which haven't, it can compute the contour leading to the exit event in question.

The marked subterm is identified by 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 $[2, 1, 3]$.

The algorithm for tracking subterm dependencies within procedure bodies is implemented as the `origin` function, shown in figure 6. This algorithm relies on the fact that the compiler converts the bodies of predicates to what we call superhomogeneous form. In this form, all clause 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 $X <= f(Y_1, ..., Y_n)$.
- 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 $X => f(Y_1, ..., Y_n)$.
- 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 $X := Y$.
- 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.

Contours contain only calls, unifications and negated goals, and negated goals cannot bind any variables visible from the outside (this restriction being necessary for the safety of negation as failure). Among unifications, only construction, deconstruction and assignment unifications can bind variables; test unifications cannot. The cases handled by the `origin` function are therefore all the cases.

Consider the `rational_add` example above, in which the predicate body is already a contour, 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(Head, Body, HV3, [1])`, where `Head` and `Body` are the head and body of that clause respectively. The goal that produces `HV3` is `HV3 = r(Cn, Cd)`. Since this is a construction unification and the path isn't empty, we call `origin(Head, 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 denumerator, the declarative debugger calls `origin(Head, Body, HV3, [2])`. This time, the processing of `HV3 = r(Cn, Cd)` leads to the recursive call `origin(Head, Body, Cd, [])`. That in turn tells us that the origin is the third argument of the call to the `lcm` predicate.

This algorithm can be adapted quite simply to handle the dependency tracking needs of inadmissible calls. There are only two differences. First, the head and conjunction we give it as the first two arguments are from the caller of the marked atom, not the predicate involved in the marked atom itself. Second, the conjunction is not a contour, in two respects. One is that while it starts at the start of the relevant procedure body, it ends at a call, not at the end of that procedure body. The other is that the conjunction may go inside negated goals. The conjuncts in such a conjunction may therefore differ as to how many negations they are nested inside, whereas every conjunct of a contour is inside exactly zero negations.

Consider the `all_pairs_are_in_table` from section 3. If the 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 and invoking the `origin` function in the partial contour leading up to that call, stepping one level up in the call tree.

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 in the (full) contour leading up to the `exit` event that computed that atom, stepping one level down in the call tree.

However, even if a call to `origin` 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 take one step down into the body of the call and one step 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 variable will be to the left of the call the algorithm dived into and out of. This guarantees that the algorithm makes progress.

In general, the dependency tracking algorithm may make many steps both up and down in the call tree in its search for the unification that creates the subterm being tracked. There are only two things that can stop it from reaching its objective. 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 bodies in the executable, or if the predicate is defined not in Mercury but in a foreign language. (The Mercury foreign language interface allows Mercury predicates to be defined other languages such as C.) Second, the subterm dependency tracking algorithm will not take a step that would take it above the suspect area, since there is no point in asking questions about nodes there. If necessary, it will take steps that take it *below* the suspect area, into the non-suspect regions rooted at correct 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 correct goal at the root of that non-suspect region as the origin of the subterm.

5.2 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 `origin` repeatedly 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 `origin` made by this algorithm. If the algorithm succeeds, the first node in the dependency chain will be the node that the oracle asserted was erroneous or inadmissible because of an incorrect subterm, while the last node will correspond to the call where the subterm was initially constructed.

Our implementation will then ask the oracle about the validity of the node which bound the incorrect subterm, provided the node wasn't previously eliminated from the suspect area (if the binding node is outside the suspect area, then we ask about the last node on the dependency chain that is in the suspect area). 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 predicate responsible for binding a particular subterm appearing in an erroneous or inadmissible 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, on the other hand, the oracle asserts that the node which bound the incorrect subterm is correct, then it seems likely that the bug lies on a node somewhere on the dependency chain, since these are the nodes through which the subterm was passed; presumably, one of these nodes should have modified the term. We therefore search the nodes on a modified version of chain. The modification looks for cases where the chain dives into a call and then climbs out again, meaning that the wrong subterm was passed around inside that call but wasn't modified. Since such calls are unlikely to be buggy, we omit them from the chain, which we then search either top-down (linear search) or divide-and-query style (binary search).

5.3 An example

Consider the following predicate which calculates the average of a list of floating point numbers by keeping track of the sum of the numbers and how many there are so it can be tail recursive:

```
average([], Sum, N, Sum / float(N + 1)).
average([H|T], Sum, N, Average) :-
        average(T, H + Sum, N + 1, Average).
```

This implementation is buggy, since `average([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], 0.0, 0, 3.0)` is true in this implementation (the correct answer is 3.5). Marking the fourth argument incorrect takes us directly to the bug:

```
average([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], 0.0, 0, 3.0)
Valid?  browse 4
browser> mark
average([], 21.0, 6, 3.0)
Valid?  no
Found incorrect contour:
average([], 21.0, 6, 3.0)
```

Using a divide and query search would result in approximately $\log_2 n$ questions being asked where n is the length of the list. For long lists this could be a substantial number of questions and the questions are likely to take longer to answer. In this case, marking the incorrect subterm in the first question results in only one more question being asked no matter how big the list is.

5.4 Related work

The idea of focusing the search on a marked subterm is not new. It was first proposed two decades ago by Pereira [6], who named it *rational debugging*. However, Pereira's implementation worked by modifying the usual Prolog unification algorithm to keep track of dependencies. This modified unification algorithm has significant overhead, in both space and time, compared to the standard algorithm, and thus cannot be used to replace the standard algorithm. On the other hand, using the standard algorithm in most places and switching to the modified algorithm when tracking subterm dependencies doesn't work either, because they use different data representations.

By contrast, our subterm dependency tracking algorithm requires no changes to the way unifications are done. For us, this is a requirement, because the Mercury runtime doesn't have a general purpose unifier. The only unifications allowed in Mercury programs are one-way *matches*, and all matches are compiled into sequences of primitive operations such as constructions and deconstructions. 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).

Some experimental debuggers for Java (e.g. [3]) can find out where a particular variable was last assigned their equivalent of subterm tracking. However, they implement this capability by effectively recording the entire history of the program execution. This is feasible only for programs with very short run times. Our selective rematerialization of EDT nodes doesn't have this problem.

6. CONCLUSION

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 all 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 part of the tree is small) the user typically doesn't know about them. To alleviate this problem, we are working on changes to the user interface to signal to users that a large term being presented is one they have seen before.

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.

Because we tell the user exactly which unification of which line of code produced the subterm, subterm tracking can also be used to try to understand what a program is doing, even if its behaviour is not necessarily incorrect – the user may simply be trying to understand a piece of code they may not have written themselves.

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.

We would like to thank the Australian Research Council and Microsoft for their support.

7. REFERENCES

[1] Mark Brown and Zoltan Somogyi. Annotated event traces for declarative debugging. Available from `<http://www.cs.mu.oz.au/mercury/>`, 2003.

[2] Lawrence Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.

[3] Bil Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September 2003.

[4] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.

[5] Lee Naish. A three-valued declarative debugging scheme. *Australian Computer Science Communications*, 22(1):166–173, January 2000.

[6] 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.

[7] Ehud Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.

[8] 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.

[9] 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.

[10] 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.