Annotated event traces for declarative debugging

Mark Brown and Zoltan Somogyi

{dougl,zs}@cs.mu.OZ.AU
Department of Computer Science and Software Engineering
University of Melbourne, Parkville, 3052 Victoria, Australia
Phone: +61 3 9344 9100, Fax: +61 3 9348 1184

Abstract. Many programmers find debugging a frustrating and unproductive activity. Declarative debugging [6, 10] promises to alleviate this problem by automating some of the reasoning used in the debugging process. We have implemented a declarative debugger for Mercury [5]. In the process, we found a problem not addressed in the existing literature on declarative debuggers, which considers programs to consist of clauses (conjunctions of literals): how to handle if-then-elses. The problem is that the condition of an if-then-else should be treated as a negated goal if and only if the condition fails. Negated contexts may cause a declarative debugger to switch from wrong answer analysis to missing answer analysis and vice versa. Since missing answer analysis explores a search space that is subtly different from the space explored for wrong answer analysis, the two kinds of analysis need different kinds of trees to represent the program execution. For the conditions of if-then-elses, the debugger does not know which kind of tree to build until the condition has either succeeded or failed, by which time it is too late. To solve this problem, we have designed a novel data structure, the annotated event trace, which is flexible enough to support both wrong and missing answer analysis. The advantages of this data structure are that it is quite compact, requiring little more space than an ordinary stored event trace, and that the algorithms to build this data structure and to derive from it the information required for two kinds of diagnosis are all simple as well as efficient.

1 Introduction

Logic programming languages have both a declarative semantics and an operational semantics. The declarative semantics views the program as a set of logical statements whose logical consequences give the meaning of the program, while the procedural semantics views the program as a sequence of instructions to be executed by the machine. The declarative semantics is higher level and closer to the way that programmers think than the operational semantics. Ideally, programmers should be able to work entirely with the declarative semantics. Unfortunately, this is not possible with most logic programming language implementations, because the only available debugger works at the operational level.

Until recently, this was also true for Mercury; the first Mercury debugger [11] is a procedural debugger for the Melbourne Mercury implementation. However, we have now built a prototype declarative debugger for Mercury. Declarative debuggers operate at the level of the declarative semantics; they work by asking an oracle (usually the programmer) questions about the intended meaning of the various parts of the program, and comparing this to their behaviour [6, 10]. By tracking inconsistencies between behaviour and intended meaning, such debuggers can locate a point in the program in which the correct results of zero or more subcomputations are combined into an incorrect result. This point is the precise location of a bug in the program source. This technique requires the procedural semantics to be a faithful reflection of the declarative semantics, which means that is only applicable to pure logic programs: to Prolog programs written in Prolog's small pure subset, and to programs written in pure logic programming languages such as Mercury.

The Melbourne Mercury implementation supports declarative diagnosis from within the procedural debugger. Both debuggers view 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 runtime system can then interact with the programmer, allowing the programmer to inspect the state of the computation and to issue commands. Some of these commands tell the debugger to give control back to the program being debugged, and to interact with the programmer again only at a future event that matches a specified condition. One possible condition is "until the current call has either produced a solution or has failed". If the computed answer is wrong, the programmer can issue a command to invoke the declarative debugger to diagnose the wrong answer; if the call has failed without producing an expected answer, the programmer can issue a command to invoke the declarative debugger to diagnose the missing answer.

Wrong answer analysis differs significantly from missing answer analysis. The input to wrong answer analysis is a predicate and all its arguments, both input and output, while the input to missing answer analysis is a predicate and just its input arguments. One can start wrong answer analysis as soon as the wrong answer has been computed, while one cannot start missing answer analysis until *all* answers to the call in question have been computed. When performing wrong answer analysis, the declarative debugger can restrict its attention to the path through the predicate body that execution took in order to produce the wrong answer; when performing missing answer analysis, no such restriction is possible. Although both forms of analysis explore a tree whose nodes are related to calls of the program, and in which a parent/child relationship between two nodes implies a caller/callee relationship between the associated calls, the trees they want to explore are different because the criteria they want to use to select the children of the node associated with a call are different.

A declarative debugger for a logic programming language that permitted only definite clauses could simply build whichever kind of tree was required by the kind of error the diagnoser was asked to debug. A declarative debugger for a logic programming language that also permits normal clauses cannot do that, because negated contexts require the diagnosis algorithm to switch from wrong answer analysis to missing answer analysis and vice versa (this observation was attributed to McCabe in [10, p. 74]). Such a debugger could instead build a single tree with two kinds of nodes. Wrong answer nodes would include as their children only the nodes of the calls along the computation path towards a specific wrong answer of a given predicate invocation, while missing answer nodes would include as their children the nodes of all the calls made by the given predicate invocation.

Unfortunately, even this more complicated approach poses problems for logic programming languages that support if-then-elses as well as simple negated atoms. In Mercury and in Prolog dialects with a sound implementation of if-then-else (e.g. NU-Prolog [12]), `(Cond -> Then ; Else)` has the same declarative semantics as `((Cond, Then) ; (¬∃ Cond, Else))`, which means that `Cond` is in a negated context if and only if it has no solutions. However, operationally, execution enters `Cond` only once. Until `Cond` either succeeds or fails, the implementation cannot know whether the calls inside `Cond` are in a negated context or not, and therefore what kind of tree to build for the calls in `Cond`. One may imagine that this problem could be avoided by transforming if-then-elses into their equivalent form before debugging; this would mean that *both* kinds of tree would effectively be built. While this solution could work for Prolog dialects, it does not work in Mercury because the transformed if-then-elses generally are not mode correct, therefore cannot be executed by the Mercury system.

In the Mercury debugger, we have solved this problem by not materializing either kind of tree at all; what we build instead is an *annotated trace*. When the programmer initiates declarative debugging, the declarative debugger will reexecute the goal in question, but this time in a mode in which the runtime system builds a list which has one node for each event occurring inside the execution of the goal. During its construction, the runtime system annotates this list by linking the nodes of semantically related nodes together with a carefully selected set of extra arcs. Since different kinds of events have different sets of extra arcs, the resulting data structure looks like an irregular kind of skip list. The extra arcs require relatively little extra space, and adding them requires little extra time, so the annotated trace is almost as cheap to build as a simple list of events would be. Nevertheless, the annotated trace is flexible enough to support both wrong answer and missing diagnosis on all the subgoals within the erroneous computation, because the skip links allow the diagnosis algorithm to locate the children of a given node, of either kind, quickly and efficiently. The diagnosis algorithm never fully materializes any kind of tree; it constructs just the portions of interest to it, on demand.

The rest of this paper is organized as follows. Section 2 introduces the ideas of events and event traces, as well as our running example. Section 3 defines the concepts on which the annotated trace is based: the concepts of contexts, strata and contours. Section 4 describes the structure of the annotated trace and gives the algorithm that the Mercury runtime system uses to build the annotated trace. Section 5 describes how the error diagnosis algorithm extracts the information it needs from the annotated trace. Section 6 shows how we handle some of the practical issues that arise in the implementation of our scheme. Section 7 presents our conclusions and lists areas for future work.

This paper assumes familiarity with the basic concepts of Mercury, including types, modes and determinisms; these are described in detail in the Mercury language reference manual [5]. It also assumes familiarity with the basic concepts of declarative debugging.

2 Event traces

The Mercury debuggers view 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 [11].

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 [1]:

`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.

The procedural debugger also supports a fifth type of interface event, the exception event, which occurs when control leaves a procedure and returns to its caller due to an uncaught exception. The declarative debugger is not yet able to analyze computations that raise exceptions; that is future work.

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 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 (a *switch* is a disjunction in which each disjunct unifies a bound variable with different function symbol).

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 not only their names but also their types and storage locations.

At each internal event, the debugger also has access to the *goal path*, which gives the identity of the subgoal associated with the event. For most kinds of internal events, the goal 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. If we view the body of a procedure as a term consisting of primitive goals (unifications and calls) combined with various connectives (conjunction, disjunction, if-then-else, etc), then a goal path is a sequence of components, with each component giving one step from the root of the term to the subterm that represents a goal (which may be primitive or compound). The goal path uniquely identifies this goal. For example the goal path "s2;c2;d1" states that the procedure body is a switch whose second arm is a conjunction in which the second conjunct is a disjunction, and denotes the first disjunct of this disjunction. An internal event which has this goal path associated with it must be a disj event, since its occurrence means that execution is about to enter this disjunct. Goal paths may also include components "?;", "t;" and "e;" for conditions, then branches and else branches, respectively, and "~;" for negations.

We will use the program fragment in figure **??** as our running example in the rest of the paper. We assume that the `main` predicate starts with an if-then-else whose condition is the conjunction `p('a', X), test(X)`, where `test(X)` fails immediately. Then executing the program produces a sequence of events (an event trace) of which the trace in figure 1 is the initial portion.

This example illustrates several aspects of event traces. First, 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 events of `p` (call #2) are interspersed with the events of the calls to `test` (calls #5 and #6) as well as e.g. the calls to `r` (#4 and #8). Second, it is possible for a given procedure invocation to generate `disj` events whose goal paths refer to different disjuncts of the same disjunction; we call such events *parallel* events (e.g. events #6 and #25). On the other hand, `then` and `else` events are not parallel; one cannot get them both for any given entry into the condition (the condition may be entered several times if the code before the condition has more than one solution). Switch events for different arms of the same switch are not parallel either. Third, the sequence of interface events that may occur for call depends on the determinism of the called procedure. The possible sequences of interface events for procedures of the various determinisms are as follows.

nondet:	a call event, zero or more repeats of (exit event, redo event), and a fail event
multi:	a call event, one or more repeats of (exit event, redo event), and a fail event
semidet:	a call event, and either an exit event or a fail event
cc_nondet:	a call event, and either an exit event or a fail event
det:	a call event and an exit event
cc_multi:	a call event and an exit event
failure:	a call event and a fail event
erroneous:	a call event only (followed by an infinite loop or a runtime abort)

3 Contexts, strata and contours

The full trace contains all the information that the declarative debugger may ever need. However, not all of it is needed all the time. At any point in time, the debugger needs information only from

```
:- pred p(character::in, int::out) is nondet.

p(A, D) :-
    q(A, B),
    (
        r(B, C)
    ->
        (
            s(C, D)
        ;
            D = 31
        )
    ;
        not(
            q(B, _)
        ),
        D = 32
    ).

:- pred q(character::in, character::out) is nondet.

q('a', 'a').
q('a', 'b').
q('c', 'c').

:- pred r(character::in, int::out) is semidet.

r('a', 10).

:- pred s(int::in, int::out) is det.

s(N, 3 * N).
```

small subsets of the full trace, although of course in different circumstances, it requires different subsets. In this subsection, we define several concepts that correspond to such subsets. These concepts are the foundation on which the annotated trace is built.

We define the events *inside* the call identified by call number 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 our example, the events inside call #2, the only call to `p`, consist of the events generated by calls #2-5 and #7-9, while the events inside call #3, the first call to `q`, consist only of the events generated by call #3.

When the debugger is looking for a bug in 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 clauses defining 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 contexts come in pairs: `nege`–`negs`, `nege`–`negf`, and `cond`–`else`. (There are also `cond` events which are matched not by an `else`

Event #	Call #	Depth	Type	Procedure	Atom	Path
1:	1	1	`call`	main/2-0		
2:	1	1	`cond`	main/2-0		?;
3:	2	2	`call`	p/2-0	p('a', _)	
4:	3	3	`call`	q/2-0	q('a', _)	
5:	3	3	`swtc`	q/2-0		s1;
6:	3	3	`disj`	q/2-0		s1;d1;
7:	3	3	`exit`	q/2-0	q('a', 'a')	
8:	2	2	`cond`	p/2-0		c2;?;
9:	4	3	`call`	r/2-0	r('a', _)	
10:	4	3	`exit`	r/2-0	r('a', 10)	
11:	2	2	`then`	p/2-0		c2;t;
12:	2	2	`disj`	p/2-0		c2;t;d1;
13:	5	3	`call`	s/2-0	s(10, _)	
14:	5	3	`exit`	s/2-0	s(10, 30)	
15:	2	2	`exit`	p/2-0	p('a', 30)	
16:	6	2	`call`	test/2-0	test(30)	
17:	6	2	`fail`	test/2-0		
18:	2	2	`redo`	p/2-0		
19:	2	2	`disj`	p/2-0		c2;t;d2;
20:	2	2	`exit`	p/2-0	p('a', 31)	
21:	7	2	`call`	test/2-0	test(31)	
22:	7	2	`fail`	test/2-0		
23:	2	2	`redo`	p/2-0		
24:	3	3	`redo`	q/2-0		
25:	3	3	`disj`	q/2-0		s1;d2;
26:	3	3	`exit`	q/2-0	q('a', 'b')	
27:	2	2	`cond`	p/2-0		c2;?;
28:	8	3	`call`	r/2-0	r('b', _)	
29:	8	3	`fail`	r/2-0		
30:	2	2	`else`	p/2-0		c2;e;
31:	2	2	`nege`	p/2-0		c2;e;c1;~;
32:	9	3	`call`	q/2-0	q('b', _)	
33:	9	3	`fail`	q/2-0		
34:	2	2	`negs`	p/2-0		c2;e;c1;~;
35:	2	2	`exit`	p/2-0	p('a', 32)	
36:	10	2	`call`	test/2-0	test(32)	
37:	10	2	`fail`	test/2-0		
38:	2	2	`redo`	p/2-0		
39:	3	3	`redo`	q/2-0		
40:	3	3	`fail`	q/2-0		
41:	2	2	`fail`	p/2-0		

Fig. 1. Event trace

events but by one or more `then` events. These `cond` events are not related to negated contexts 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. (In Mercury, negated goals cannot bind non-local variables, and therefore cannot succeed more than once.)

We define a *stratum* to be the set $C - \bigcup_i c_i$, where C is the set of events from some context, and the c_i are the sets of events from contexts nested in C. 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 such 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, the diagnosis algorithm may treat different body events differently. 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.

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, the debugger is interested only in events that describe how that particular answer was computed; it is not interested in alternative disjuncts or exits that did not lead to the wrong answer. 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.

For answers computed from definite clauses, the contour corresponds to the proof of that answer. However, if the events leading up to a computed answer contain a negated context, then the contour is slightly more complicated: the events in the negated context are skipped, since these events are in a deeper stratum. On the other hand, if the goal associated with a stratum contains a failed negation, then the negated goal succeeded and the events of this goal would have a similar structure to those leading up to a computed answer. These events correspond to forward execution, so the stratum anchored by the `nege`–`negf` pair would contain a contour leading up to the `negf` event.

A contour is always anchored at the start by a `call` or `nege` event. If it relates to a particular computed answer or failed negation, then the contour is anchored at the end by an `exit` or `negf` to match the start. It is possible, though, that the goal associated with the contour's left anchor event failed. This could be due to the success of a negated call, which would generate a `negf` event, a failed call, which would generate a `fail` event, or a failed builtin such as unification, which would not generate any event at all. We consider the sequence of events traversed by forward execution from the left anchor event to the point of failure to also be a contour, with its last event being the `call` event of the failed call, the `nege` event of the negation which failed, or the last event before the failed unification. Failed contours thus do not have a right anchor matching their left anchor.

In our example, the conjunction of goals from which `p('a', 31)` is derived consists of `q('a', 'a')`, `r('a', 10)`, and `D = 31`. and the contour events of that solution are the call and exit events (if any) of those goals (events 4, 7, 9 and 10), and the internal events 8, 11, 19. The contour is anchored by the call and exit events of `p('a', 31)` itself (events 3 and 20). The goal paths of events 11 and 19 record the decisions about the flow of control that caused `p('a', 31)` to be computed. Note that the contour does not include events 12-14; the contour of an exit event never includes any events that had been backtracked over at the time of the exit, whether they led to other exits or not. This bypasses all the backtracked-over events not just in all previous disjuncts (of which in this case there is only one) but also any events that may have followed from their successful solution.

To illustrate this situation, consider figure 2, which describes the events generated by the execution of `p('a', _)` in a graphical form. The contour of the `exit` event of the solution `p('a', 31)` consists of the events in the (almost) horizontal path connecting the `call` port of `p` near the top left corner of the outermost box (event 3), to the second `exit` port of `p`, on the right hand side (event 20). At event 11, the contour branches downwards to event 19, to reflect that the second disjunct has been taken.

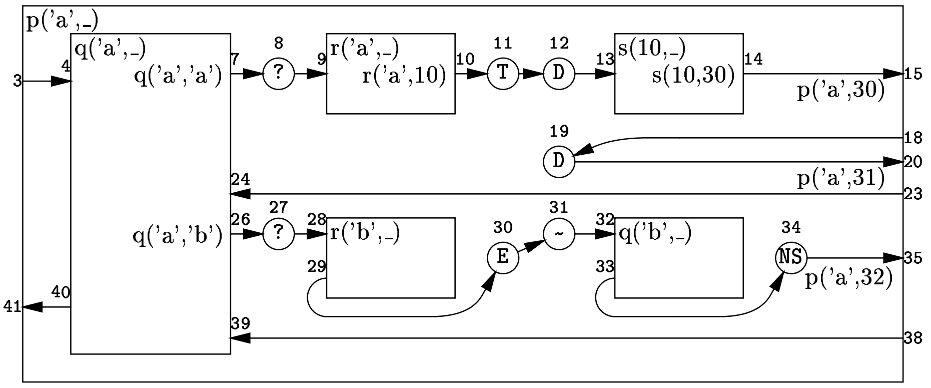

Fig. 2. Box model view of the execution of p('a',_)

4 The annotated trace

4.1 The nodes of the annotated trace

The annotated trace is a data structure that contains one node for each event generated by the goal being debugged. (At least, we will assume so for now; we will assume differently in section 6.2.) Different kinds of events have different kinds of nodes, which contain different sets of 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.
– 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`.
– The nodes of `redo` events contain an `exit` field that points to the node of the `exit` event corresponding to the previous solution produced by this call.
– The nodes of `call` events contain a `most_recent_interface` field that points to the most recent `exit`, `redo`, or `fail` event for the call. It is initialized to `NULL` and destructively updated later; it is the only field in the annotated trace that is so updated. It is also the only field that can point forward in the trace.

– The nodes of `call` and `exit` events have an `atom` field. Besides recording the identity of the procedure and the call number of the specific call to this procedure, the atom field also records the values of the procedure arguments in their state of instantiation at those events. This means that at calls, the field contains just the input arguments, while at exits it contains all the arguments. (Because the `exit` node has a pointer to the `call` node, in practice it need only store the subset of this information that is not available at the time of the call.)

Several of these fields exist to make it possible to find all the `exit` nodes corresponding to a given `call` node, and thus find all the solutions (or all the solutions so far) of the call. When the annotated trace is complete, the `most_recent_interface` field of every `call` node in the trace must point to a `fail` node or an `exit` node. From either kind of node, one can follow first their `redo` field and then the `exit` field of the pointed-to `redo` node, until one finds a `NULL` `redo` field; the set of `exit` nodes encountered during such a search lists all solutions of the call computed until the end of the trace. In our example, for call #3, these fields form a chain consisting of the events 40, 39, 26, 24 and 7. (This alternation of `redo` and `exit` events until one reaches the `call` event is why the lines marked with (*) in figure 4 are guaranteed to respect the semantics of the `redo` field.)

Together with the `call` field, these fields also allow the declarative debugger, when looking at the correctness of a particular call, to quickly find the set of body events for that call. By jumping from a `fail` or `exit` event for an immediate descendant directly to the previous `redo` event of that descendant, or to the `call` event if the descendant had no previous `redo` event, it can avoid all the computations that take place within that descendant. When looking at call #2, for example, the debugger can jump from event 26 directly to event 24, or from event 7 to event 4. By jumping from one of its own `redo` events to the previous `exit` event, (e.g. from event 18 to event 15 or event 23 to event 20), the debugger can avoid all the events generated by code outside the call tree of interest. Such jumps, and the fields that enable them, are essential for the feasibility of the algorithms we will later present. In practice, it is quite common for such a jump to jump over many thousands of events; skipping over that many events one at a time following `preceding` links would be far too slow. In our scheme, the number of events a traversal has to visit to gather all the body events of a call is directly proportional to how many body events the call has.

4.2 Building the annotated trace

Programmers can start declarative debugging from within the procedural debugger by issuing the command "dd" in either one of two situations. First, at an exit event the programmer may notice that the values of the output arguments are not correct; a "dd" at that port will initiate *wrong answer* diagnosis. Second, at a fail event the programmer may notice that the call did not produce a solution the programmer was expecting; a "dd" at that port will initiate *missing answer* diagnosis. The algorithm the runtime system uses to construct the annotated trace is the same in both these cases; only the kind of diagnosis that the runtime system asks the algorithm to perform differs.

The runtime system notes the event number at which the "dd" command was executed (*dd_event*), as well as the call number of the call being debugged (*call_number*). It uses the machinery used to implement the procedural debugger's "retry" command to rewind the computation back to the state it had at the call event of the selected call; this includes the global counters for event numbers and call numbers. Therefore when we restart forward execution, we will get the exact same sequence of events we got before. The difference is that this time, the forward execution will be in the "construct annotated trace" mode. The effect will be as if we had executed the build_annotated_trace algorithm in figure 3, although in actuality, each callback to the runtime system from the program will execute the loop body of that algorithm just once, and the effect of the loop is supplied by the sequence of callbacks. The create_annotated_node algorithm creates a node for every event inside the call tree of the call being debugged up to and including the event at which declarative debugging was initiated. (For the time being, we will assume that the within_depth_limit function always returns true.) If the programmer gives the "dd" command at event 35, the trace would contain events 3-15, 18-20 and 23-35; if the programmer gives the "dd" command at event 40, the trace would contain events 4-7, 24-26 and 39-40.

```
build_annotated_trace($call\_number$, $dd\_event$, $depth\_limit$) returns $trace$ is

$trace$ := NULL
$inside$ := false
Loop
        If $e$ is a call or redo event for call $call\_number$
                $inside$ := true
        If inside and within_depth_limit($e$, $depth\_limit$)
                $trace$ := create_annotated_node($e$, $trace$)
        If $e$ is an exit or fail event for call $call\_number$
                $inside$ := false
Until the event number of $e$ is $dd\_event$
```

Fig. 3. Algorithm for building the annotated trace

The create_annotated_node algorithm itself is shown in figure 4. Most of the algorithm is quite straightforward, consisting of filling in fields in the obvious way, but there are a few subtleties, and most of these (the ones marked with numbered notes) involve searches in contours. In the following, it will be useful to point out a stratum that the contour belongs to. We will say that the search is "confined" to that stratum but this is not strictly true: the search can take one step outside of the stratum, but then it will either terminate or step immediately back inside. To help us explain how these searches work, we again urge the reader to refer to figure 2.

Consider the case of event 15, which is the exit event for `p('a', 30)`, which is in turn one of the exits for call #2. To find the call event of call #2, we must find the start of the contour which leads to this exit event. The search starts at the event before the exit, which can be one of three things: an exit event for an immediate descendant, an internal event of the current call, or, if the procedure body contains only builtins, the call event of the current call. In all three cases, the event is on or just before the contour leading to the exit. (The event before the exit cannot be a redo event for the same call, because the resumption of forward execution after backtracking always results in an internal event, e.g. a disj event.) In this case, event 14 is an exit event of a descendant, call #5. The search (marked (1) in the algorithm) follows the contour by repeatedly applying the `step_left_in_contour` function (figure 5) until it finds the node it is looking for, the call event anchoring the contour. This search ignores any events inside #5, as well as any events subsequent to any previous successes of #5 that have been backtracked over (in this case, there are none), and continues with the event before its call event. After searching disj event 12 and then event 11, the search enters the condition of the if-then-else. Since the condition was successful (there wouldn't be a `then` event for it otherwise), it is part of the surrounding context, which is why search continues with the event before the `then`. After searching exit event 10, cond event 8 and exit event 7, search arrives at event 3, which is the event we are searching for. Given that from nested exit and fail nodes `step_left_in_contour` goes not to the call event but to the event before the call, the algorithm need not even test the call event's call number; the first call event the search finds is guaranteed to be the call event of the right call. Since search stops at that point, `step_left_in_contour` will never be invoked on a call node.

For event 20, the second exit event of call #2, the search jumps directly from event 19, the event for entering the second disjunct, to event 11, the event before the *first* disjunct. Together with jumping over the events inside calls, this guarantees that the number of nodes examined by the search is bounded by the maximum length of a contour. Given that the maximum length of a contour cannot be bigger than the size of the predicate body and usually is significantly smaller, this makes the search feasible.

```
create_annotated_node($e$, $prev\_node$) returns $cur\_node$ is

Allocate a node of the proper type for event $e$, call it $cur\_node$
$cur\_node$.preceding := $prev\_node$
switch on the type of $e$
case call:
        Save call details in $cur\_node$.atom
        $cur\_node$.most_recent_interface := NULL
case exit:
        Save exit details in $cur\_node$.atom
        $c$ := $prev\_node$;
        While $c$ is not a matching call, $c$ := step_left_in_contour($c$)                    (1)
        $cur\_node$.call := $c$
        $cur\_node$.redo := $c$.most_recent_interface                                          (*)
        $c$.most_recent_interface := $cur\_node$
case redo:
        $x$ := find_prev_contour($prev\_node$)
        While $x$ is not a matching exit, $x$ := step_left_in_contour($x$)                    (2)
        $cur\_node$.exit := $x$
        $x$.call.most_recent_interface := $cur\_node$
case fail:
        $c$ := find_prev_contour($prev\_node$)
        While $c$ is not a matching call, $c$ := step_left_in_contour($c$)                    (3)
        $cur\_node$.call := $c$
        $cur\_node$.redo := $c$.most_recent_interface                                          (*)
        $cur\_node$.call.most_recent_interface := $cur\_node$
case swtc:
        Save the goal path of $e$ in $cur\_node$.goal_path
case disj:
        Save the goal path of $e$ in $cur\_node$.goal_path
        If the goal path indicates this is a first disjunct
                $cur\_node$.first_disj := $e$
        Else
                $d$ := find_prev_contour($prev\_node$)
                While $d$ is not a parallel disj, $d$ := step_left_in_contour($d$)            (4)
                $cur\_node$.first_disj := $d$.first_disj
case cond, nege or then:
        Save the goal path of $e$ in $cur\_node$.goal_path
case else:
        $s$ := find_prev_contour($prev\_node$)
        While $s$ is not a matching cond, $s$ := step_left_in_contour($s$)                    (5)
        $cur\_node$.context_start := $s$
        Save the goal path of $e$ in $cur\_node$.goal_path
case negs:
        $n$ := find_prev_contour($prev\_node$)
        While $n$ is not a matching nege, $n$ := step_left_in_contour($n$)                    (5)
        $cur\_node$.context_start := $n$
        Save the goal path of $e$ in $cur\_node$.goal_path
case negf:
        $n$ := $prev\_node$
        While $n$ is not a matching nege, $n$ := step_left_in_contour($s$)                    (5)
        $cur\_node$.context_start := $n$
        Save the goal path of $e$ in $cur\_node$.goal_path
```

Fig. 4. Algorithm for creating a node in the annotated trace

```
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
```

Fig. 5. Algorithm for stepping to the left in the current contour

```
find_prev_contour(this) returns on_contour is

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

Fig. 6. Algorithm for finding a previous contour

While the search for a call matching an exit is confined to the stratum anchored by the exit and its matching call, that is, it touches only events inside the call tree of the call concerned, the search for the exit event matching a redo (marked (2) in the algorithm) is instead confined to an enclosing stratum which contains those two events. Taking the box model view of this situation, we could say the search proceeds outward from the box rather than inward. Consider event 24, the redo for call #3 after the exit `q('a', 'a')`. The search starts at event 23, a redo for call #2, the parent of call #3. The first step is to find any event on a contour which contains the sought-for `exit`; this is performed by the `find_prev_contour` function (figure 6). In many cases the argument to this function is already on the required contour, so it is immediately returned; however, in this case the argument (event 23) is a redo for the enclosing procedure, so the function returns the event which precedes the `exit` matching event 23, event 19. The contour leading to this exit event of the enclosing call (call #2) must contain the sought-for exit (because you cannot backtrack into call #3 from call #2 if call #3 had not previously succeeded). Once we have located the right contour, we again apply `step_left_in_contour` until we find the event we are looking for. Unlike call nodes, many exits can be reached by following this procedure, so in this case we must look for an exit which has the same call #. Of the exit events with the right call #, we are guaranteed to find the most recent one, since `find_prev_contour` will not jump over any events that are inside that call, including the sought-for exit.

The search for a call matching a fail (marked (3) in the algorithm) is similar to the search for an exit matching a redo, except it is confined to the stratum anchored by the call and fail events. For example, for event 41, the fail event for call #2, the search starts at event 40, the fail event for call #3. From there, it skips immediately to event 3, the event we are searching for. The search for the call event matching event 40 is a bit more interesting. It starts at event 39, a redo for call #3, and then jumps to event 25, the event before the previous exit of call #3, thus bypassing all events outside the call tree of call #3 since that exit. Event 25 is on a contour which is anchored on the left by event 4, the call event we are searching for, so we again use `step_left_in_contour` to find the start of the contour: event 25 is the second disj event for call #3 and it is matched by first disj event 6, so the search jumps to event 5, before finding the matching call event, event 4.

The search for an earlier parallel disj event (marked (4) in the algorithm) is similar to the search for a call event that matches a fail event. The search is confined to a stratum which contains both disj events. We can test whether an earlier disj event is parallel to the current one by checking that the goal paths are identical up to but not including the last component. The parallel disj event the search finds is for the immediately previous disjunct of the same disjunction, which may or may not be the first. However, in either case, its `first_disj` field must point to the node of the event denoting entry to the first disjunct.

The search for a cond matching an else and the search for a nege matching a negs are analogous to the search for a call matching a fail (these are marked (5) in the algorithm). The main difference is that (obviously) the events anchoring the stratum are different. Likewise, the search for a nege matching a negf is analogous to the search for a call matching an exit.

5 Diagnosis

5.1 A declarative view

Declarative debugging requires that we reason about the correctness of parts of the execution of a program. Our view of the execution is a sequence of events, so it is useful to take a more declarative view of some of the events in the annotated trace. In particular, we are interested in the events which can anchor the end of a contour or stratum. An `exit` event asserts that the atom associated with it is valid. A `fail` event asserts that every instance of the call atom which is valid is also an instance of one of the solutions. A `negf` event corresponds to a negated goal in the program. It asserts that, given the bindings of variables non-local to the negation at the time of the matching `nege` event, the negated goal is satisfiable. Similarly, `negs` events assert that the negated goal is unsatisfiable; `else` events do likewise for the condition of the corresponding if-then-else. (The remaining event types cannot be the right anchors of contours or strata and thus have no such associated assertions.) An event can be considered correct if the assertion it makes is true in the intended interpretation. Thus a wrong answer is an incorrect `exit` event, and a missing answer is an incorrect `fail` event.

The events with assertions always anchor the end of a stratum. In the case of `exit` and `negf` events, the stratum contains a contour leading up to the event. The event shows that a particular goal in the program succeeded; the information in the contour is sufficient to explain the success of that goal, since alternative solutions and disjuncts are irrelevant to the success. We therefore define the contour to be the *explanation* of the event which anchors it on the right. In the case of `fail`, `negs` and `else` events, the event shows that a particular goal in the program will not produce any more solutions; the failure to do so cannot be explained by any single contour, but only by the entire stratum. We therefore define the explanations of events of those types to be their entire stratum.

If an event e is incorrect but all the events in its explanation are correct, then the part of the program that generated the events and its explanation must contain a bug. On the other hand, if event e is incorrect and one or more of the events in its explanation are also incorrect, then it is likely (though not certain) that the incorrectness of e is caused by the incorrectness of its children; incorrectness propagates to ancestors. In our example, the first solution of the call to `p` anchors the end of a contour which includes the call to `s`, so incorrectness of the `exit` event for `s` could be the cause of incorrectness of the first `exit` event of `p`.

We therefore find it convenient to define a tree, the root of which is any incorrect event in the trace. The nodes of this tree come from the events of the annotated trace that have assertions associated with them. The children of a given node are those nodes which can propagate incorrectness to it: for any event P, the children of P are defined to be all of the events which make assertions in the explanation of P. Since we have a definition of correctness for nodes, this tree is an instance of a scheme of Naish [7]. A bug is a node in the tree which shows where correct premises are combined together to form an incorrect result; in other words, a bug is an incorrect node with no incorrect children. Such a node corresponds directly with a part of the program which is either incomplete, or is inconsistent with the intended interpretation. One advantage of using this scheme is that it simplifies the design of the debugger: it separates the definition of a bug from the search strategy used to find a bug. It is trivial to show that every tree must contain a bug; we do not address the search strategy in this paper.

```
node_children($node$) returns $children$ is

If the type of $node$ is 'exit' or 'negf'
    $children := \emptyset$
    $e := node$.preceding
    While type of $e$ is not 'call'
        If $e$ makes an assertion
            $children := children \cup e$
        $e :=$ step_left_in_contour($e$)
    Loop

If the type of $node$ is 'fail' or 'negs' or 'else'
    $children := \emptyset$
    $e := node$.preceding
    While type of $e$ is not 'call'
        If $e$ makes an assertion
            $children := children \cup e$
        $e :=$ step_in_stratum($e$)
    Loop
```

Fig. 7. Algorithm for finding the children of a node

Figure 7 shows the algorithm for finding the children of a node. We start at the event preceding the node, and repeatedly apply a function that steps back through the explanation. On the way, the events that make assertions are collected. If the goal in question was successful, `step_left_in_contour` (figure 5) is the function we use to traverse the contour anchored by the node. For example if the parent is event 15, a wrong answer node for the atom p('a', 30), the algorithm visits events 14, 12, 11, 10, 8, 7 and 3. On the way it picks out events 14, 10 and 7 (which are all `exit` events) as the children. On the other hand, if the goal in question failed, then we need to traverse the entire stratum, so we call the `step_in_stratum` function instead (figure 8). The differences between this function and the earlier one are the following.

– At a fail event we go to the previous matching redo (if possible) instead of the matching call, which means that the search includes goals to the right of the failed goal.
– At an exit event we again go to the matching redo (if possible) instead of the matching call, which means that the search includes goals following previous solutions of the call.
– At a disj event we go to the preceding node rather than stepping over to the event before the first disj, which means that the search includes goals in and following alternative disjuncts.

These differences ensure that we cover the entire stratum. As an example, consider event 41, which is a missing answer node for the atom p('a', _), with solutions p('a', 30), p('a', 31) and p('a', 32). The children generated by this process are events 40, 34, 30, 26, 14, 10 and 7.

```
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
```

Fig. 8. Algorithm for stepping to the previous event in a stratum

Our implementation actually uses a variation on the tree defined above. Since `else`, `negs` and `negf` nodes correspond to anonymous constructs inside procedures and not to named procedures, it is difficult to verify their correctness by asking the oracle a question. Therefore, for each such node n we remove n from the tree, and attach its children directly to the nearest ancestor of n that is an `exit` or `fail` node. All of the nodes in our tree are thus wrong answer (`exit`) nodes or missing answer (`fail`) nodes. Our debugger does not become unsound by doing this; it merely loses the ability to narrow the location of a bug down to a negated context inside a procedure body.

When finding the children of a wrong answer node, the contour traversal covers internal events which tell us which parts of the body of the procedure were used in the contour that produced that answer. We can collect the goal paths associated with these events; in our example, the goal paths "c2;?;", "c2;t;" and "c2;t;d1;" would be recorded alongside the wrong answer node. The resulting tree is shown in figure **??**. When finding the children of a missing answer node, however, we do not collect goal paths since these cannot help to explain why a stratum unexpectedly fails.

Figure 9 shows the tree for event 41. This time there are both exit and fail events so the tree has both wrong answer nodes and missing answer nodes. The missing answer nodes in this tree are drawn as boxes, with the call atom in the top left corner and the solutions (if any) down the right hand side.

Wrong answer nodes may have missing answer nodes as children and vice-versa. For example, this can happen if a procedure uses negation. In this situation, other debuggers switch from generating wrong answer nodes to generating missing answer nodes, or vice-versa. However, our debugger does not have such a mode switch. The kind of nodes that can be generated from a goal depends only on whether that goal succeeded or failed; it is unaffected by the state the debugger is in outside of that goal. In our example, the children include missing answer nodes for r('b', _) and q('b', _) because the goals containing these calls failed. It is irrelevant that the parent is a missing answer node—these nodes would also have been children of the wrong answer node for p('a', 32).

The strong mode and determinism systems of Mercury allow us to a priori remove some missing answer nodes from the tree. Calls to procedures which are guaranteed to succeed exactly once, i.e. those with determinism `det` and `cc_multi`, do not generate `redo` and `fail` events. Calls to

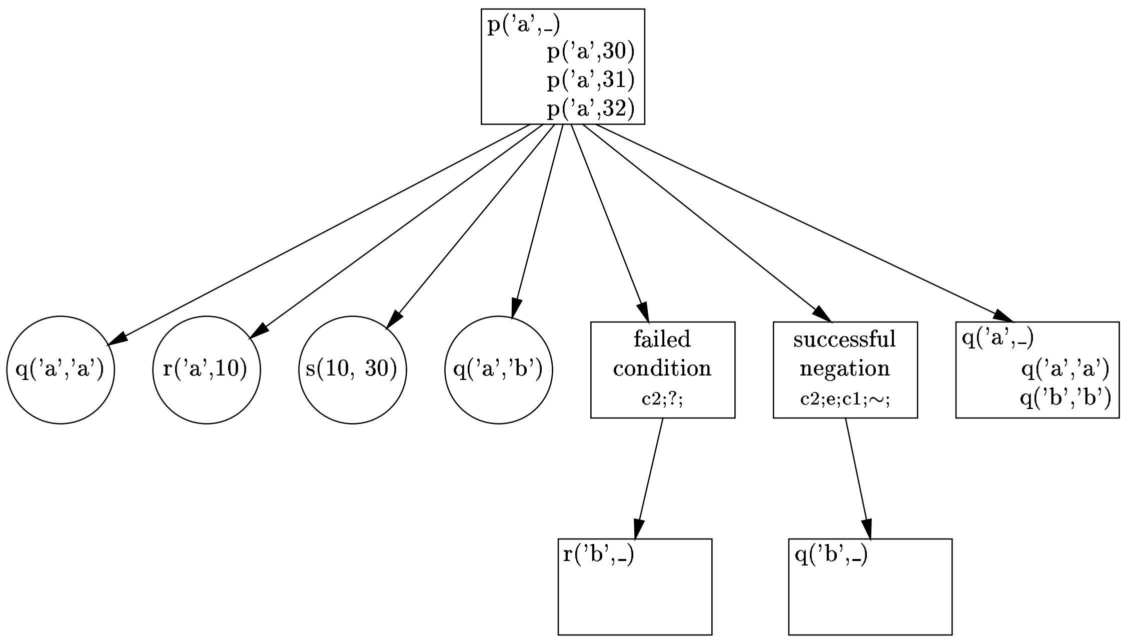

Fig. 9. Declarative view of the annotated trace after event 41

procedures which are guaranteed to succeed at most once, i.e. those with `semidet` and `cc_nondet`, and which do in fact succeed, do not generate `redo` and `fail` events either. This means that there can never be any missing answer nodes for these calls, even though they may be called from a context which ultimately fails. Assuming the determinism declarations in the program are correct, such procedures can have at most one answer, so if there is a missing answer then the computed answer must have been wrong.[1] Since there exists a wrong answer node to cover this case, the algorithm is justified in removing the missing answer node. In figure 9, there is no missing answer node for s because it is declared `det`, and neither is there one for the first call to r, because it is declared `semidet` and it succeeds.

The assumption that we are making here is not a strong one: if the declarations are incorrect, the program will generally get a compile time error because the compiler checks that such declarations are consistent with the program. There is a small possibility, though, that the program contains another bug which makes the program clauses consistent with the incorrect declaration, without making them inconsistent with any other; in these cases, the debugger may give misleading results. Such cases are rare; handling them is future work.

5.2 The diagnoser

After a programmer issues the "dd" command at an incorrect `exit` or `fail` event, the debugger builds the annotated trace then passes it to a front end which performs the diagnosis. Diagnosis consists of asking the oracle a series of questions about the intended interpretation; these answers are used to locate a bug in the materialized part of the tree. A database stores all previous answers, so if a question has already been asked its answer will be retrieved from there; the programmer is never asked the same question twice. For each node in the tree, there is one question: is the assertion made by that node correct? Obviously, the possible answers that the oracle can give include "yes" and "no". However, the oracle can also answer "don't know" which essentially asks the diagnoser to search in a different part of the tree. The diagnoser will do this if possible, but will revisit the question later if necessary.

[1] The case where the computed answer is an instance of a more general answer, but does not cover some other instance, cannot occur due to the strong mode system of Mercury.

Pereira's principle of "rational" debugging, presented in [9], is that the steps taken by a debugger should be both necessary and sufficient to determine the correctness of a node. Our debugger conservatively approximates this: the children of a given node are sufficient but not always necessary to determine whether the node is a bug. For example, even though a successful negation that leads immediately to a computed answer for a goal can never be the source of a missing answer for that goal, we still generate a node for it. Furthermore, if a computed answer is mostly correct but contains some incorrect subterm, knowledge of the binding times of variables can make some children irrelevant. Finding a closer approximation to rational debugging is future work.

When a bug is found, it is displayed to the programmer in a suitable format. If it is a wrong answer node it is called an "incorrect contour", meaning that the program code associated with that contour contains a bug. The contour can be determined from the goal paths collected for that node. If the bug is a missing answer node it is called a "partially uncovered atom". Since the oracle does not actually supply missing answers, but merely points out when incompleteness exists, our debugger cannot report that an atom is totally uncovered.

A future diagnoser could also allow the oracle to answer "inadmissible", meaning that neither "yes" nor "no" is applicable because the goal in question should never have been called. This answer would be different to "don't know", in that the diagnoser would not revisit the question later. Allowing such an answer would make ours a *three-valued* debugger rather than a *two-valued* debugger. This, for example, could provide us with a means for handling exception events. Naish has a scheme to deal with this kind of debugger. The annotated event trace would not need to be changed to accommodate this scheme; only the definition of correctness of events and the definition of bug in the debugging tree would need to be changed. For details, refer to [8].

6 Practical issues

6.1 Repeatability

One of the key assumptions that our algorithm relies on is that the declarative debugger can reset the computation to the state it had at the time of a given call; unfortunately, this is not always true. In order to rewind the computation back to the state it had at the time of a call event, the debugger needs to know the values of the input arguments of that call. When generating debuggable executables, the compiler therefore tries to preserve the values of all the input arguments so that they are available at all events within the lifetime of the call. However, Mercury permits programmers to declare that the mode of an argument is "di" or "destructive input", which means that the storage occupied by that argument may be destructively updated during forward execution. Such destructive update may be performed automatically by the compiler, or, if the predicate is actually implemented using Mercury's foreign language interface, by the programmer writing the C code of the predicate. The former can be easily avoided by turning off the optimization when generating debuggable executables, but the second is a real problem.

One could try to solve the problem by making copies of the values of destructively updated input arguments, in practice this is not feasible, for several reasons. First, it is possible that the type of the destructively updated argument is also defined in C and unknown to Mercury; in that case the Mercury language implementation would not know what algorithm to use to make the copy. Second, the debugger does not know in advance which calls the user may wish to retry, and thus would be forced to make copies of the destructively updated arguments in *all* calls, and to keep those copies for the entire lifetime of the call they came from. Given that programmers are more likely to make a data structure destructively updated if updating it nondestructively is slow (i.e. if the data structure is big), this is a serious problem, especially given that any predicate that processes all elements of the data structure will have each call for each element live at once (turning on debugging defeats tail recursion optimization in Mercury). Therefore this solution can easily require storing thousands of copies of megabyte sized data structures. Even worse, the destructively updated variables that represent the state of the world, which Mercury uses to implement a sound semantics for I/O, cannot be backed up even in principle.

Fortunately, data types implemented outside Mercury are fairly rare. The Mercury standard library supplies one, type `array`, but it also supplies the type `bt_array` which *is* implemented in

Mercury and has the same operations. Therefore when debugging their programs, when performance is not as critical as it can be later, programmers can simply use `bt_array` instead of `array`. The only other frequently used uncopyable type is the I/O state.

However, we have developed a technique that we can use to retry predicates that perform I/O actions that does not require backing up the state of the world. This technique is based on *tabling* the predicates that implement primitive I/O actions. In Mercury, a predicate can perform I/O only if it has two arguments of type `io:state`: the first, whose mode is "destructive input", represents the state of the world before the predicate's I/O operations, while the second, whose mode is "unique output", represents the state of the world after the predicate's I/O operations. However, predicates in Mercury never perform I/O directly. Instead, they call predicates (usually in the Mercury standard library) that have Mercury interfaces but are actually defined in C code.

(This technique is similar in spirit to a technique proposed by Calejo [2], but differs considerably in the details.)

We have a solution for the problem of how to handle I/O states based on tabling the predicates that implement primitive I/O actions. and that when the proposed solution to I/O states is implemented,

Therefore for the time being, the retry operation will work only if destructive update does not appear anywhere in the call tree of the procedure being debugged.

We estimate that about 95% of the predicates that people write fall into the subset of Mercury that our declarative debugger can handle. While this is

```
:- pred io__read_char_code(io__input_stream, int, io__state, io__state).
:- mode io__read_char_code(in, out, di, uo) is det.
% Reads a character code from specified stream.
% Returns -1 if at EOF, -2 if an error occurs.

:- pragma foreign_proc("C",
    io__read_char_code(File::in, CharCode::out, IO0::di, IO::uo),
    [will_not_call_mercury, tabled_for_io],
"
    CharCode = mercury_getc((MercuryFile *) File);
    IO = IO0;
").

:- pred p(int, string, io__state, io__state).
:- mode p(in, out, di, uo) is det.

p(In1, ..., InM, Out1, ..., OutN, S0, S) :-
    :- $pragma(...).

p(In1, ..., InM, Out1, ..., OutN, S0, S) :-
    increment_io_action_counter(UpdatedCounter),
    lookup_io_action_table(UpdatedCounter, ActionTableSlot),
    (if
        table_has_no_block(ActionTableSlot)
    then
        $pragma(...), % Original procedure body
        table_create_block(ActionTableSlot, N + M + 1, Block),
        table_assign_to_block(Block, 0, <<info_about_p>>),
        table_assign_to_block(Block, 1, In1),
        ...
        table_assign_to_block(Block, M, InM)
        table_assign_to_block(Block, M + 1, Out1),
        ...
        table_assign_to_block(Block, M + N, OutN)
else
    table_pickup_block(ActionTableSlot, Block),
    table_restore_from_block(Block, M + 1, Out1),
    ...
    table_restore_from_block(Block, M + N, OutN),
    S = S0
).
```

6.2 Limiting memory usage

One of the design objectives of Mercury was to make it easier to construct large programs. Large programs tend to have long run times and thus generate many events; the Mercury compiler, which is a Mercury program, generates several hundred million events when compiling large files. Given that each node in the trace requires about a dozen bytes of memory, annotated traces derived from that many events would require amounts of storage that are currently infeasible.

To avoid this problem, the within_depth_limit function from figure 3 doesn't always return true. If the call *call_number* is at depth *call_depth*, then within_depth_limit will return true for an event e at depth d if and only if either $d < \mathit{call\_depth} + \mathit{depth\_limit}$, or e is an interface event and $d = \mathit{call\_depth} + \mathit{depth\_limit}$. Therefore at depths between *call_depth* and $\mathit{call\_depth} + \mathit{depth\_limit} - 1$, the annotated trace has the same information as before, but it has only summaries for calls at depth $\mathit{call\_depth} + \mathit{depth\_limit}$. The diagnosis algorithm may still be able to identify the exact cause of the problem, but now it is also possible that it may not. However, in the latter case it will have narrowed the bug's location down to a single call (call it c) at depth $\mathit{call\_depth} + \mathit{depth\_limit}$. The debugger will therefore rewind the program state back to the call event of c (or to any event before that), and reexecute the build_annotated_trace algorithm again, but this time to build an annotated trace for c. This means passing c as the value of the *call_number* parameter, and the number of the exit or fail event of c corresponding to the problem isolated by the diagnosis algorithm as the value of the *dd_event* parameter.

Not only does this algorithm avoid the requirement to store the entire annotated trace all at once, for events below correct calls at the depth limit boundary, which may be a large majority of all events, it never creates a node at all. The price it pays for this is the time taken to reexecute parts of the program, possibly several times. Increasing the value of *depth_limit* reduces the number of reexecutions at the cost of increasing the storage requirement, not only for the nodes of the evaluation tree, but also for the data structures which are referred to from the extra nodes whose storage therefore cannot be recycled by the garbage collector. This is a classic space/time tradeoff.

The usual preferred choice in such situations is the maximum value for *depth_limit* that results in acceptable memory consumption, because this minimizes the number of recomputations required. In our prototype debugger, *depth_limit* is fixed. Although it would be trivial to allow programmers to set its value, they would have no good basis on which to do so. One possible way to adjust *depth_limit* automatically would be to start with a value that is known to be on the high side, and to have the debugger decrement it when it detects that its own memory consumption is too high. In order to free up memory, the decrement would have to be coupled with freeing the parts of the annotated trace that would not have been created had *depth_limit* had the smaller value from the beginning. This may need to be done several times. Of course, allocating a node, filling in its fields, and then later freeing it is a waste of time; the bigger the discrepancy between the initial and the final value of *depth_limit* the bigger this waste is. One could try to reduce the waste by deriving a good estimate of a likely good value for *depth_limit* from profiling information, about the typical branching factors of predicates, about the typical numbers of internal events they generate, and about the typical amount of otherwise garbage-collectable memory pinned down by keeping saved copies of their arguments.

6.3 Debugging the debugger

Our prototype implementation is largely written in Mercury, so the question naturally arises: can we use the debugger to debug itself? The main complication with doing this is that the underlying infrastructure of the procedural debugger does not (yet) support the simultaneous debugging of multiple threads. Our prototype effectively runs in a parallel thread to the program being debugged, so it is not accessible to the procedural debugger under normal circumstances. The solution we have adopted is that, whenever the debugger is to be applied to itself, we save the annotated trace in a file, and then run the prototype in a separate process with that file as input. The problem with this approach is that there are conflicting requirements: for reasons of efficiency the usual implementation needs to avoid deep copying of data terms, whereas writing to a file requires this to happen.

We use Mercury's typeclasses to create an abstract data type that models the annotated trace. Being abstract, the data is not tied down to a particular representation, so we can achieve the best of both worlds by creating different instances of the abstract data type for different circumstances. The normal instance requires the analyser to be in the same process, and represents the data directly without the need for deep copying; the alternative instance sacrifices efficiency for functionality, using deep copying but allowing the data to be written to a file. This alternative representation is used whenever the programmer wishes to apply the debugger to itself; in all other cases the normal representation is used.

7 Conclusion

Annotated event traces efficiently store a representation of execution that is practical for declarative debugging. The annotations are relatively compact, and the algorithm to compute them is both simple and fast. And since the annotations on each event link it to semantically related events in the execution, the debugging algorithm can quickly and easily construct on demand the portions of the debugging tree of interest to it. This ability to efficiently retrieve semantically connected information could also have uses other than those presented here. One future possibility is to implement a graphical debugger which visually represents execution with some variation on the box model, or in the style of the Transparent Prolog Machine [4], using the annotated trace to store the information internally.

Contours are a central concept in the annotated trace. For an event in the trace, the contour describes how forward execution reached that particular event in the body of the relevant procedure. Contours are useful because they provide an explanation for events, which we use as a basis for our debugging tree, and because they narrow down the space which needs to be searched to find other events which are semantically related. This is essential for the feasibility of our algorithms.

Our work is related to the leaf failure tracking algorithm of Ducassé [3], which identifies particular event patterns related to failure in the body of Prolog goals. The main difference is that our system generates comprehensive explanations of both success and failure, whereas leaf failure tracking searches through failures to find ones that are likely to show why a goal unexpectedly fails. Leaf failure tracking aims to help programmers understand how a bug symptom (failure) has been produced; on the other hand, our aim is to find the precise location of a bug related to a symptom (failure or success), so we must perform a far more complete analysis of the trace.

Earlier declarative debuggers (for example, Calejo's framework [2]) treat wrong answers and missing answers uniformly when traversing the debugging tree, but not when defining the tree. In contrast, annotated event traces are defined independently of the kind of diagnosis required—they are flexible enough to provide either kind. This is necessary in order to support the operational semantics of Mercury's if-then-else: the goal in the condition is considered to be negated if and only if it fails, so until execution reaches the end of a condition the debugger does not know whether it will be providing the explanation of a negated goal or not.

We have implemented a debugger for Mercury based on annotated event traces. One of the advantages of using Mercury is that it has strong mode and determinism systems, which allow the execution algorithm to be optimized in many cases (for example, deterministic code). This is reflected in our debugging tree: we a priori remove some missing answer nodes from consideration. Our implementation uses a depth bound to limit memory consumption, which means that the algorithm can be scaled up to large examples at the cost of having to re-execute parts of the program. It supports some flexibility in the way that debugger questions can be answered by the oracle. In future we plan to increase the flexibility by basing the debugging tree on a three-valued logic rather than a two-valued logic. This would not require any change to the annotated trace; we could move to a three-valued logic simply by changing the definition of correctness for individual events, and changing the definition of a bug in the debugging tree. The latest version of our debugger is freely available as part of the Melbourne Mercury implementation, which can be found at `http://www.cs.mu.oz.au/mercury`.

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

References

1. Lawrence Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.
2. Miguel Calejo. *A framework for declarative Prolog debugging.* PhD thesis, Universidade Nova de Lisboa, March 1992.
3. Mireille Ducassé. Analysis of failing Prolog executions. In *Proceedings of Journées Francophones sur la Programmation Logique*, Lille, France, May 1992.
4. M. Eisenstadt and M. Brayshaw. The Transparent Prolog Machine (TPM): An execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277–342, 1988.
5. Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.
6. John W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.
7. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.
8. Lee Naish. A three-valued declarative debugging scheme. Technical Report 97/5, Department of Computer Science, University of Melbourne, Melbourne, Australia, April 1997.
9. Luís Moniz Pereira. Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, pages 203–210, London, England, June 1986.
10. Ehud Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1983.
11. 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, December 1999.
12. James Thom and Justin Zobel. NU-Prolog reference manual. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.