The implementation technology of the Mercury debugger

Zoltan Somogyi and Fergus Henderson

{zs,fjh}@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. Every programming language needs a debugger. Mercury now has three debuggers: a simple procedural debugger similar to the tracing systems of Prolog implementations, a prototype declarative debugger, and a debugger based on the idea of automatic trace analysis. In this paper, we present the shared infrastructure that underlies the three debuggers, and describe the implementation of the procedural debugger. We give our reasons for each of our main design decisions, and show how several of these decisions are rooted in our experience with the debugging of large programs working with large data structures.

1 Introduction

One of the main aims of the Mercury project has been to make it easier to write correct programs. The high level and declarative nature of the language eliminate several classes of errors; for example, in Mercury one cannot free memory prematurely, clobber a pointer, or accidentally include a side-effect in a predicate. The strong type, mode, and determinism systems of the language further guarantee that several other classes of errors, e.g. passing an unbound variable to a predicate that expects a ground term, or forgetting to handle one of the function symbols that an input argument can be bound to, will be caught and pinpointed by the compiler. As a result, most bugs in a Mercury program are caught and corrected before the first executable is produced.

Of course, the compiler cannot catch *all* errors; some errors will still need to be tracked down by the programmer. In Mercury, the availability of support for this task in the form of a debugger is more important than it would be in another language: whereas in C or Prolog, one can insert diagnostic `printf`s or `write`s into the program at arbitrary points, in Mercury this is possible only in predicates that have an `io__state` parameter passed to them, which in most programs is only a small minority of all predicates. (It *is* possible to abuse Mercury's C interface to get around this limitation, but strictly speaking the Mercury implementation does not guarantee that such tricks will work in the future, especially at high optimization levels.)

At the start of the Mercury project, we as the developers of the Mercury compiler, which is written in Mercury itself, compensated for the absence of a Mercury debugger by writing all our code in the intersection of Mercury and Prolog (initially NU-Prolog, later SICStus Prolog), and using a Prolog system to execute and to debug our programs. However, as Mercury matured, this became a more and more onerous limitation, because it prevented us from exploiting the new features we were adding to Mercury, such as functions, type classes and existential types. We therefore placed a high priority on building a native Mercury debugger, and started work on one as soon as we got the resources.

Initially we were aiming for a simple procedural debugger, similar to the tracing systems of Prolog implementations. Mireille Ducassé and Erwan Jahier at IRISA/INSA at Rennes started working on Opium-M, an adaptation of the Opium trace analysis system for Mercury, about the same time that we started work on our procedural debugger. This fact strongly influenced our design approach, because we wanted to make sure that if a Mercury program were compiled with debugging enabled, the resulting executable could be debugged either with our procedural debugger or with Opium-M. The two debuggers are therefore built on top of the same ideas and the same basic infrastructure. Later on, we started work on a prototype declarative debugger, which is implemented as an extension of our procedural debugger. Therefore instead of one debugger for Mercury, we ended up with three, which, despite disparate functionalities and user interfaces, share considerable amounts of code with each other.

This paper describes the Mercury procedural debugger and its infrastructure; it does not cover Opium-M or the declarative debugger. For information on Opium-M, see [11]; for information on our prototype Mercury declarative debugger, see [3].

The rest of this paper is organized as follows. Section 2 introduces the ideas of events and event traces, and describes how the Mercury compiler arranges for events to happen; this is the basic infrastructure on top of which all the various flavours of Mercury debuggers are built. Section 3 describes the main features of the Mercury procedural debugger, the unconventional debugging strategies they support, and their implementation. Section 4 presents our conclusions.

Due to space limitations, this paper does not cover all aspects of the debugger. The full version of this paper, [12], discusses the features we did not have room to cover here (e.g. interactive queries), and presents measurements of the impact of enabling debugging on both the sizes of executables and on their speed of execution.

We assume familiarity with the basic concepts of Mercury, including types, modes and determinisms; these are introduced in the Mercury tutorial [1] and described in detail in the Mercury language reference manual [9]. Knowledge of the main ideas of the Mercury execution algorithm [13] would also be an advantage, but it is not essential, except for one fact, which is that Mercury generates separate code for each mode of a predicate. We call each mode of a predicate a *procedure*; the entities that programmers debug are procedures, not predicates.

2 Event traces

The debuggers of most compiled languages are based on the idea that

- the compiler annotates the generated object code in various ways, e.g. recording which instructions implement which lines of code and in what register or stack slot a given variable is stored, and
- the debugger uses these annotations to make sense of the machine state and to allow it to modify the code of the running program, e.g. to insert breakpoints.

Although this approach is quite powerful, it has two significant drawbacks. First, its implementation requires much low-level work. Second, a large fraction of this work has to be repeated for each port of the language to a new platform.

Portability has always been one of the main objectives of the Mercury implementation; it is the reason why the Mercury compiler translates Mercury into C and not into object code. Adopting the traditional approach to the implementation of the Mercury debugger would have forced us to abandon this objective, and allocate scarce programmer resources either to writing one or more object code generators, or to extending an existing portable compiler such as gcc to record debugging information about the original Mercury program. Both of these would be quite big jobs for our small research team, both stray a long way outside our area of focus (logic programming), and we would be very unlikely to get money for either from our usual funding sources.

One alternative we considered was writing an interpreter for a bytecode form of the high-level intermediate representation produced by the compiler front-end. Unfortunately, it is not really possible to implement interfacing between compiled code and interpreted code in a way that is portable across operating systems as well as architectures. We therefore had to reject this approach too.

The approach we have chosen to follow is based on the idea of viewing the execution of a program as a sequence or *trace* of events. At each event, the compiler inserts extra code that calls the debugger; the debugger may then suspend program execution temporarily to allow the programmer to inspect the state of the program and to issue commands to the debugger. For example, the trace-based debugger for C named cdb [8] does this. Alternatively, the trace could be analyzed at least partly automatically, as in the trace-based Prolog debugger named Opium [6].

The designer of a trace-based debugging system can choose whether the entire code of the debugger should be linked with the executable being debugged, or whether the bulk of the debugger should be in a second process, communicating with a small kernel inside the debuggee over interprocess communication channels such as pipes or sockets. The advantages of the first approach, which is taken by the Mercury procedural and declarative debuggers, are simplicity and higher performance. The advantages of the second approach, which is taken by Opium-M [11], are flexibility and higher robustness in the face of wild pointer errors.

2.1 Event types

The selection of event types (also called *ports*) is an important decision in trace-based debuggers, since such debuggers can only give control to the user at events. Traditional Prolog debuggers have given control to the user at the four ports in Byrd's box model [4]:

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 still has some more possibly successful alternatives.

These ports are clearly important in understanding the behavior of the program because they describe the interaction of a procedure with its callers. However, these *interface* ports are often not sufficient, because they do not provide information about the internal execution of the procedure. For example, if the programmer inspects the values of procedure p's arguments at p's exit port, and finds them to be incorrect, he or she will want to know which clause of p computed that result. While the programmer can often deduce the identity of this clause from the events generated by the procedures called by p, there are cases in which this is quite difficult (e.g. p may have several clauses that all call the same procedures with arguments that are large data structures differing only in small details). This is why the Opium system also used unify events, which occurred just after the successful unification of a call's actual arguments with the head of a clause of the called predicate.

A similar port would not work in Mercury, for two reasons. First, Mercury does not make any distinction between unifications in the head and those in the body; the unifications implied by non-variable terms in the head are subject to mode reordering just like all computations on the body. While one could redefine the port type to say e.g. that unify events occur after all possible unifications against *input* arguments have succeeded, this definition would not be convenient for programmers, since some such unifications may be moved after calls in the clause body. Second, while the prevalence of first argument indexing in Prolog strongly encourages programmers to write predicates as sequences of clauses, with most clauses being conjunctions of literals and only a few if-then-elses and explicit disjunctions thrown in, this is not true for Mercury, which has permitted free composition of goals by logical connectives from the beginning.

Instead of creating a unify port, we create event types corresponding to every kind of decision about the flow of control. These *internal* event types are the following.

then	A *then* event occurs when execution reaches the start of the then part of an if-then-else.
else	An *else* event occurs when execution reaches the start of the else part of an if-then-else.
disj	A *disj* event occurs when execution reaches the start of a disjunct in a disjunction.
switch	A *switch* event occurs when execution reaches the start of one branch of a switch (a switch is a disjunction in which each disjunct unifies a bound variable with different function symbol).

Every internal event is associated with a *goal path*, which gives the identity of the subgoal that execution is about to enter 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 "c2;?;d1" states that the procedure body is a conjunction in which the second conjunct is an if-then-else whose condition 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.

2.2 Event implementation

Having decided what the event types are, the next question an implementor must answer is how to arrange to give control to the debugger at each event. The simplest approach is to use a source to source transformation, as shown by Ducassé and Noyé [7]:

```
Head :-
        ( trace(call, Head)
        ; trace(fail, Head), fail
        ),
        Body,
        ( trace(exit, Head)
        ; trace(redo, Head), fail
        ).
```

Unfortunately, this transformation breaks one of the invariants that the Mercury execution algorithm depends on, namely that a procedure declared to be det (i.e. to have exactly one solution) or semidet (i.e. to have either zero or one solution) will not leave behind any nondet stack frames (Mercury's equivalent of Prolog choice points) when it exits. Trying to get around this by simply saying that in debugging mode, all procedures are considered to be nondet or multi (and thus allowed to have more than one solution) would not work. In Mercury, I/O is only allowed in a det context, so this change would in effect disallow I/O.

A simpler approach, which we adopt, is to vary the transformation depending on the determinism of the procedure. We transform det procedures to

```
Head :-
        trace(call, Head),
        Body,
        trace(exit, Head).
```

and semidet ones to

```
Head :-
        (
                trace(call, Head),
                Body
        ->
                trace(exit, Head)
        ;
                trace(fail, Head), fail
        ).
```

For nondet and multi procedures (procedures that can succeed more than once) we use a variant of Ducassé and Noyé's transformation, which has the same operational semantics but leads to better code in Mercury, because the Mercury compiler generates more efficient code for nested disjunctions than for conjoined disjunctions:

```
Head :-
        (
                trace(call, Head),
                Body,
                ( trace(exit, Head)
                ; trace(redo, Head), fail
                )
        ;
                trace(fail, Head), fail
        ).
```

Our overall scheme preserves the required invariants, and has the advantage that programmers do not have to see events that are not appropriate for a procedure's determinism (e.g. given that the compiler proves that det and semidet procedures have at most one solution, it does not make sense to ask them to produce another solution). This approach does make the task of automatically analyzing a trace, as in Opium [6], a bit more complex, since the trace analyzer now has to understand three different behaviors for traced predicates. However, this is a small disadvantage, and we are willing to live with it. The Mercury debugger is therefore based on this idea.

However, the Mercury compiler does not implement the insertion of the calls to the trace system as a Mercury-to-Mercury source transformation. There are two reasons for this. First, there is no *simple* extension of this scheme that allows the trace system access to the values of live variables that are not arguments at internal events and at exit events. Second, calls to the trace predicate would need to allocate a memory cell to hold the term being passed as the second argument, and more memory to hold a description of the *type* of the term, and yet another cell to package the two together as a value of the `univ` type. (Different calls to the trace predicate will have different type arguments, which in Mercury have different representations; without the type description, the trace predicate couldn't do anything with the term.) This is a higher level of overhead than we are willing to accept.

The Mercury compiler therefore handles tracing during code generation, by including calls to the trace system in the C code in generates. The inserted code fragments look like this:

```
{
        Code    *MR_jumpaddr;
        MR_jumpaddr = MR_trace(
                &mercury_data__layout__mercury__main_2_0_i14,
                MR_PORT_DISJ, "c2;?;d1", 3);
        if (MR_jumpaddr != NULL) GOTO(MR_jumpaddr);
}
```

`MR_trace` is the function in the Mercury runtime system that implements the debugger. Its return value will almost always be NULL, which means that most of the time execution will continue normally. (The only exception is when the debugger has just executed a retry command, which we will discuss in the next section.)

`MR_trace` has four explicitly passed arguments.

- The first argument is a pointer to a data structure created by the Mercury compiler that we call a *label layout* structure, which describes the names, locations and types of all variables that are live at the given trace point. Describing locations is quite simple; a variable is either in the nth abstract machine register, or at an offset m in the procedure's stack frame on either one of Mercury's two stacks. Describing types is considerably more complex; type constructors may nest to an arbitrary depth, and variables may have types that are not fully known statically (i.e. their types may include a type variable). For the details on how the Mercury implementation represents types and how it ensures that system components such as the debugger can always find out the concrete type of any value, see our paper on runtime type information in Mercury [5]. The label layout structure also contains a pointer to another compiler-generated data structure, the *proc layout* structure, which contains various items of information about the procedure (its name, arity, determinism, etc).
- The second argument specifies the type of the current event.

- The third argument specifies a goal path, which uniquely identifies a subgoal in the body of the current procedure. This argument is only meaningful if the current event is an internal event, in which case it identifies the subgoal that control is about to enter. For example, in the code fragment above, it specifies that control is about to enter the first disjunct of the disjunction which is located in the condition of the if-then-else that is the second conjunct of the procedure body.
- The fourth argument specifies the number of the highest numbered abstract machine register in use at the point of the call to `MR_trace`. `MR_trace` saves the original values of all the registers up to this number on entry and restores them on exit; this is necessary because `MR_trace` calls Mercury code to perform some of its functions, and thus may clobber the abstract machine registers. The alternative, saving and restoring *all* the abstract machine registers, is not a good idea because there are more than a thousand of them. (In the Mercury abstract machine, all arguments are passed in registers, so an implementation must have lots of registers if it wants to support -possibly automatically generated- predicates with high arities.)

The values of these arguments will differ in different calls to `MR_trace` from within a given procedure invocation. Some other data items `MR_trace` may want to access will always be the same for a given procedure invocation. These data items are the call sequence number (which uniquely identifies a procedure invocation), the event number of the last event before the call event that created the invocation, and the depth of the invocation. To reduce the size of calls to `MR_trace`, these items are not passed as arguments. (We have found that that it is important to keep down the amount of memory needed by calls to `MR_trace` [12]; Hanson and Raghavachari make a similar point in [8].) Instead, they are stored in the stack frames of traced procedures at fixed offsets, and `MR_trace` accesses them there. These stack slots are filled in when a traced procedure is entered, with the help of three global variables in the runtime system. The value in the event number global variable is simply copied to the slot holding the event number before the call event, while the global variables holding the call sequence number and the depth are incremented before they are assigned to the corresponding slots. To allow the depth field to be filled in properly, traced procedures reset the global depth counter to the value stored in their own depth slot immediately before they make a call.

2.3 What should be traced?

The Mercury implementation gives a substantial degree of control to the programmer about the set of events that a compiled Mercury program will generate. Some of this control is exercised at compile time; two options tell the compiler not to emit code for internal events and for redo events respectively. One can get along without them in many cases, and the programmer may not wish to pay the space or time costs they incur. Some of this control is exercised at run time. If a programmer builds an executable containing debugging information from (say) a C program, he or she expects to be able to run that program not only under the control of a debugger, but also on its own, without involving a debugger. We allow this choice via a boolean variable, `MR_trace_enabled`, in the runtime. The first thing `MR_trace` does is consult `MR_trace_enabled`; if it is false, `MR_trace` returns immediately. This test is centralized in `MR_trace` instead of being wrapped around every call to `MR_trace` because there are *lots* of calls to `MR_trace`, and the cost in program size would be very large [12]. Since the test in `MR_trace` will virtually always be in the cache, while with the alternative approach many of the tests dispersed through the program will not be, our approach can also be superior purely on execution time grounds.

By default, `MR_trace_enabled` is in fact initialized to false, but the initialization routine included in every Mercury program sets it to true instead if the environment variable MERCURY_OPTIONS includes the right flag. The Mercury debugger, mdb, is in fact nothing more than a shell script that sets this flag in MERCURY_OPTIONS before executing a given command. All the actual work of the debugger is done by functions in the Mercury runtime system that are called from `MR_trace`.

Most programs contain significant numbers of procedures that the programmer believes are unlikely to contain an error. For example, most programmers trust the list manipulation predicates in the list module of the Mercury standard library, both because they are relatively simple and because they have been used many times before, which means that any flaws are very likely to have been found and fixed long ago. Events inside the call trees of trusted procedures are by definition not interesting to the programmer. However, the interface events (call, exit, redo and fail) of such calls may still be interesting, because they provide information about the operation of their caller.

In [7], Ducassé and Noyé propose that the code of such trusted procedures should not generate trace events; instead, calls to those procedures should be wrapped in code to generate the interface events of the call. We rejected this approach for two reasons. First, trusted procedures tend to be utility procedures, which means that they are called from many places; incurring the space overhead of the interface events once per call site instead of just once for the procedure definition is unlikely to be a good tradeoff. Second, if a procedure changes status from trusted to untrusted or vice versa, this scheme would require the recompilation of all modules that call that procedure, and this requirement interferes with separate compilation.

The approach we have adopted allows programmers to specify their level of trust in the code in a module by selecting a *trace level* for the module. (We could also allow programmers to define different trace levels for different procedures in a module, but as programmers we have never really felt a need for this level of flexibility, and providing it would require a significantly more complex user interface than the simple compiler option we now use.)

none A procedure compiled with trace level *none* (which is the default trace level) will never generate any events, and contain no debugging information.

deep A procedure compiled with trace level *deep* will always generate all the events requested by the programmer. Unless the programmer explicitly turns off internal events or redo events, this will be all possible event types.

shallow A procedure compiled with trace level *shallow* will generate interface events if it is called from a procedure compiled with trace level *deep*, but it will never generate any internal events, and it will not generate any interface events either if it is called from a procedure compiled with trace level *shallow*. If it is called from a procedure compiled with trace level *none*, the way it will behave is dictated by whether its nearest ancestor whose trace level is not *none* has trace level *deep* or *shallow*.

Procedures compiled with trace level deep are compiled as we have discussed above, with the addition that before every call, they set `MR_from_deep`, a global boolean variable in the Mercury runtime system, to true. Procedures compiled with trace level shallow set this same boolean variable to false before their calls. They also save the value of this variable on entry in a stack slot reserved for this purpose, and consult it later. Conceptually, such procedures should call `MR_trace` only if the saved value of `MR_from_deep` is true. In practice, for reasons of program size and cache effectiveness that are identical to the reasons why we put the test for `MR_trace_enabled` in `MR_trace`, we also put the test of the saved value of `MR_from_deep` into `MR_trace`. We do this by including a field in every proc layout that says whether the procedure is shallow traced, and if yes, which stack slot holds the saved value of the flag, and making `MR_trace` consult this field and act on its value immediately after it has found `MR_trace_enabled` to be true. The code for filling in the other three debugging stack slots, which should increment the global call sequence number only if `MR_from_deep` is true, cannot *usefully* be centralized in this way, so for this action the test of `MR_from_deep` is done in each procedure.

3 Procedural debugging

The procedural debugger we have built on top of the trace system has several interesting features, but is mostly conventional. This is by design; our starting point was that we wanted a debugger that combined the best features of debuggers for imperative languages (e.g. gdb) and the debuggers built into Prolog systems. We also wanted to try to make programmers used to other debuggers to feel at home to the maximum extent possible. We have therefore made the debugger highly customizable.

The two key system components that implement customizability are aliases and the configurable help system. Aliases allow a programmer to map e.g. the command “s” to either “skip” to the end of the predicate (if they are used to Prolog) or to “step” to the next event (if they are used to gdb). The help system then allows the programmer to document the command “s” as if it were a native part of the debugger, for use by other programmers of a similar background.

3.1 Examining the program state

One class of debugger commands allows the programmer to examine the state of the program at the event. One command lists the names of the variables that are live at the event; this list is taken straight from the RTTI information contained in the data structure pointed to by the first argument of `MR_trace`. Another command prints the values of all these variables, or just the value of the variable with a given name. This command uses the RTTI data structures to interpret the machine state, including the values of the Mercury abstract machine registers the last argument of `MR_trace` specifies which registers are live at the trace point, so that `MR_trace` can save their values on entry and restore them on exit.

This precaution is necessary because several functions of the Mercury debugger are implemented in Mercury itself, and these can and will overwrite those registers. (They will also use the Mercury stacks, but modifications of the regions beyond the current stack tops, unlike modifications of the registers, will not affect the program when it resumes execution.) In particular, the code for printing Mercury terms and browsing inside them is implemented in Mercury. Actually, printing and browsing are done by the same code invoked in two different ways. This code respects parameters such as the maximum depth to which a term should be printed as well as the maximum size (width times height) that its printed representation may occupy, omitting subterms as necessary to keep within these limits (which are of course configurable). There is no command that a user can give in the debugger to print the value of a variable that does not respect these same limits. We enforce this condition because of our experience with Prolog systems that do not. One of the programs we use the debugger on is the Mercury compiler itself, some of whose data structures are quite big: several megabytes of data, whose printable representation can exceed a hundred megabytes in size. An accidentally issued command that prints such a data structure in full can take more than an hour at the usual terminal display speeds, which means that the programmer has no better course of action than to abort the program and start it again from scratch.

The variables the programmer can inspect at an event include not only the variables in the procedure invocation to which the event belongs (which may be in stack slots or in (saved copies of) registers), but also the variables in the ancestors of that invocation (which can only be in stack slots). Such *uplevel printing* is possible because whenever that procedure is compiled at a trace level other than none, the compiler generates a proc layout for that procedure and a label layout for each return point, i.e. code address where execution resumes after a call, in that procedure. (The generation of label layouts for return points can be suppressed by a compiler option.)

Besides procedure identification information (module name, predicate/function name, arity, and mode number), the proc layout also records the procedure's determinism and hence which stack its stack frame is on, and if it is on the det stack, also gives the size of its stack frame and the location of the saved return address within that frame. (Frames on the nondet stack have the saved return address and the link to the caller's frame in fixed slots.) The label layout of a return point records the names, types and locations of the variables that have been saved across that call, and includes a pointer to the proc layout of the procedure containing the return point. The Mercury runtime system has a mechanism for mapping a code address (such as a return point) to the label layout, if any, for that address. By successively looking up the return address in its slot in the current stack frame, finding its label layout structure and the corresponding proc layout structure, and then conceptually popping off the callee's stack frame, the debugger can reach the stack frame of any given ancestor. The fact that a procedure that lives on the det stack can call another procedure that lives on the nondet stack, and vice versa, complicates the traversal code, but the determinisms in the proc layout structures allow us to resolve the resulting ambiguity. The only case when the stack traversal has to stop prematurely is when it encounters the stack frame of a procedure that was compiled without debugging, and which therefore does not have either kind of layout structure.

Of course, a programmer cannot meaningfully say e.g. "print the variable List in the third ancestor of the current call" unless they know what the third ancestor is. This is one reason why we provide a stack trace command that prints out the identities of all the ancestors. Given that recursion can be very deep in Mercury (sometimes reaching more than ten thousand levels), we have found it necessary to compress stack traces. We use a simple run-length encoding, in which sequences of nested invocations of the same procedure are represented by naming the procedure just once but giving the repetition count. This only works for self-recursion and not for mutual recursion, but we have found that many cases of mutual recursion are irregular (e.g. `p` calls `q` calls `r` calls `p` calls `r`), which makes it difficult to find a representation for them that is easily understood and unambiguous as well as short.

3.2 Controlling forward execution

Debugger commands that examine the state of the program do their job and print a prompt for another debugger command without returning from `MR_trace`. Other debugger commands cause `MR_trace` to return and forward execution to resume. They do so after specifying the condition that a following call to `MR_trace` has to satisfy before that `MR_trace` prints a prompt and allows user interaction again. For the `step` and `goto` commands, this condition is reaching an event with a given event number; the `step` command specifies this event number as an offset from the current event, while the `goto` command specifies it directly. The `goto` command is particularly useful in finding the causes of runtime aborts (calls to the Mercury `error` predicate, which corresponds to an assertion failure in languages such as C). When an executable compiled with debugging aborts, it prints the number of the last event executed. If this number is (say) 987,654,321, the programmer can reexecute the program and goto e.g. event number 987,654,000, and start inspecting the program at just before the abort. When debugging programs that do not abort, event numbers also allow programmers to do a real binary search; they can go to the exact middle point of the suspect region (which is initially the entire trace), check whether everything is all right, and replace the suspect region by its left or right half depending on the answer. These two techniques are not supported by any other debugger we know of, for any language.

The `finish` command stops at the next exit or fail event at the specified depth. By default, the depth is the depth of the current event, so `finish` goes to the end of the call associated with the current event, but one can also specify that e.g. one wants to go to the end of the third ancestor of the current call. This fixes a problem present in many Prolog debuggers, which is that while there is a way to step *over* a call, there is no way to go to the end of a call once one has accidentally stepped *into* that call.

3.3 Controlling breakpoints and trace output

Like most debuggers, the Mercury debugger allows programmers to put breakpoints on a given procedure. The breakpoint can apply to the call event only, to all interface events, or to all events in the named procedure, at the programmer's option. Checking whether an event matches a given breakpoint must therefore include checking the type of the event as well as checking whether the procedure associated with the event is the same as the one associated with the breakpoint. For speed, this latter check is done by comparing the addresses of their procedure layouts; this works because procedure layouts are static data structures. (The procedure layout of an event is the procedure layout pointed to by the label layout structure whose address is passed as the first argument of `MR_trace`.)

For all forward execution commands, there is a question as to what the debugger should do at events that do not match the condition of the current debugger command but which do match a breakpoint; should it stop forward execution, or should it go on until the condition of the debugger command is satisfied? In the Mercury debugger, the answer depends on the *strictness* of the current command. If the current command is non-strict, execution will stop and user interaction will begin; if the current command is strict, execution will continue. Every forward movement command (`step` etc) has a default strictness which can be overridden by an option supplied by the programmer, and there is a forward movement command, `continue`, which has no stopping condition other than encountering a breakpoint.

3.4 Implementing retry

Like Prolog debuggers but unlike debuggers for imperative languages, the Mercury debugger has a `retry` command that allows the programmer go back to an earlier point in the program execution, to the call event of the current procedure invocation or of any of its ancestors. The proc layout of every procedure compiled with debugging includes pointers to the start of the code of the procedure and to the label layout structure of the procedure's call event, which states what variables are supposed to be where at the time of the call. The debugger looks up the current locations of those variables in the stack frame of the invocation to be retried (possibly including the saved registers when retrying the current invocation), and copies them to the places they had occupied at the time of the call. Normally, the lifetime of a variable ends immediately after its last use, and the value of the variable will not be kept after that point. However, to support `retry`, the compiler tries to extend the lifetimes of all input arguments so that they cover the entire lifetime of a call to any procedure compiled with debugging. The compiler will succeed in doing so unless the procedure body destructively updates some of the input arguments. The `retry` command will fail if invoked from a location at which some of the input arguments have already been so updated. Since I/O states are destructively updated, this means that it is rarely to possible to use `retry` on predicates that do I/O. We are currently implementing a proposed solution to this problem.

Besides resetting the registers containing the input arguments (Mercury passes all input arguments in registers), the `retry` command must also reset the rest of the machine state, including the stack pointers and the return address register; the stack traversal process tells us the appropriate values to reset them to. Mercury currently uses the Boehm conservative garbage collector [2] and its associated allocator, so there is no heap pointer to be reset. There are also no logic variables to reset to unbound; since the mode system knows what variables are unbound at any given program point, compiled code does not look at unbound variables, which means that their contents do not matter. The machine state also includes part of the debugger state, meaning the global variables holding the next event number, the next call sequence number, and the call depth; the values to reset these to are taken from the fixed stack slots of the invocation being retried. (Other parts of the debugger state, e.g. the list of current breakpoints, are persistent.) Once the machine state has been reset, we can reexecute the call. We do this by returning the address of the entry point of the retried procedure from `MR_trace`; since this address will not be NULL, the code that calls `MR_trace` will jump to that address. The code inside `MR_trace` cannot perform the jump itself, because if it did, `MR_trace` would never return, and its stack frame could never be reclaimed.

4 Conclusion

The mechanisms used by the Mercury system to generate and process event traces have proven to be quite versatile. They have been used as the basis of no less than three debuggers: the procedural debugger described in this paper, the prototype declarative debugger described in [3], and Opium-M [11], a debugger based on the idea of running queries to analyze the trace generated by a buggy program. The same mechanisms have also been used as the basis of a generic facility for monitoring the executions of Mercury programs [10].

We have found the time and space costs of enabling debugging in Mercury programs to be within acceptable limits. Just enabling debugging but not running the program under the debugger causes about a 50-75% increase in runtime; this is not much higher than the slowdown one expects when compiling a C program with "-O0 -g". Even when the Mercury program is run under the debugger, its speed is comparable to the speed of one of the faster Prolog implementations (SICStus fastcode), and the executable sizes are also similar. On the usability side, we have been using the procedural debugger as our only tool for debugging Mercury programs for about a year now. Since we work all the time with a large, complex, long-running Mercury program that manipulates multi-megabyte data structures, we have made sure that the debugger works well on such programs. The techniques we have reported that allow the debugger to do this should be applicable to the implementations of other languages as well.

We would like to thank Mireille Ducassé and Erwan Jahier for several fruitful discussions about tracing systems and for first planting the idea of building a Mercury debugger based on event traces, and Tyson Dowd for his work on the Mercury RTTI system. We would like to thank the Australian Research Council and the European Community for their support.

References

1. Ralph Becket. Mercury tutorial, 1999. Available from `<http://www.cs.mu.oz.au/research/mercury/tutorial>`.
2. Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.
3. Mark Brown and Zoltan Somogyi. An evaluation tree for declarative diagnosis in Mercury, 1999. In preparation.
4. Lawrence Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.
5. Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David Jeffery. Run time type information in Mercury. In *Proceedings of the 1999 International Conference on the Principles and Practice of Declarative Programming*, pages 224–243, Paris, France, 1999.
6. Mireille Ducassé. Opium: an extendable trace analyser for Prolog. *Journal of Logic Programming*, to appear in 1999.
7. Mireille Ducassé and Jacques Noyé. Tracing Prolog programs by source instrumentation is efficient enough. In *Proceedings of the JICSLP 98 Workshop on Implementation Technologies for Programming Languages based on Logic*, pages 46–58, June 1998.
8. D.R. Hanson and M. Raghavachari. A machine-independent debugger. *Software – Practice and Experience*, 26:1277–1299, 1996.
9. 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.
10. Erwan Jahier and Mireille Ducassé. A generic approach to monitor program executions. In *Proceedings of the Sixteenth International Conference on Logic Programming*, Las Cruces, New Mexico, 1999.
11. Erwan Jahier and Mireille Ducassé. Opium-M 0.1 user and reference manuals. Technical Report PI-IRISA-1999 No 1234, IRISA, Rennes, France, March 1999.
12. Zoltan Somogyi and Fergus Henderson. The implementation technology of the Mercury debugger. Technical Report 99/30, Department of Computer Science, University of Melbourne, Melbourne, Australia, November 1999.
13. 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.