Medic: Metaprogramming and Trace-Oriented Debugging

Xiangqi Li
University of Utah, USA
xiangqi@cs.utah.edu

Matthew Flatt
University of Utah, USA
mflatt@cs.utah.edu

Abstract

Modern programmers enjoy a wealth of high-level and graphical tools for understanding and debugging programs. Nevertheless, programmers often resort to the simple and the time-honored technique of inserting `print` statements into programs to reveal progress and to expose intermediate values. This *trace debugging* (a.k.a. *printf debugging*) technique persists because it has many advantages. Traditional trace debugging also has several drawbacks, including the need to modify the source program and the need for additional tools when trace output becomes too voluminous. **Medic**, our new debugging and program-exploration tool for Racket, augments the traditional examination of control and state with output processing, metaprogramming, and visualization features. Medic allows programmers to leverage the benefits of trace debugging while addressing many of its drawbacks.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Metaprogramming, trace debugging, visualization

1. Introduction

Many different techniques can help a programmer find bugs or understand the execution of a program. A programmer can set breakpoints, step through the program, check the call stack and values of local variables, and add `print` statements to produce traces of program states and values. That last technique of adding `print` statements is called *trace debugging* or *printf debugging*.

Trace debugging offers many benefits compared to other techniques. Print statements are lightweight, and they are convenient for exposing specific values of interest or examining specific behaviors of programs. However, trace debugging in practice has changed little since the early days of programming. Programmers continue to work with text-based, linear traces, while a variety of data visualizations are possible, and textual output is awkward for representing tree and graph-structured relationships among data. Traditional trace debugging is good for generating data with local (to the implementation) filters, but it offers little support for processing the output data in a non-local way.

The necessity of modifying a program's source to insert `print` commands creates other obstacles. Debugging code is mingled with the original code, which obscures the original code during debugging; after debugging is complete, further work is needed to recover the clear, original source. Removing debugging code wholesale, meanwhile, discards work that is useful for understanding or further debugging of the code in the future. Finally, code written for debugging purposes rarely enjoys the organizational and reuse properties of most other code, because the debugging code must be tightly bound to the details of the main implementation.

Medic addresses these two sets of trace-debugging problems through a pair of complementary strategies. The first strategy is to separate debugging code from the main implementation through a metaprogramming language that can weave—in the sense of aspect-oriented programming (Kiczales et al. 1997)—`print` commands into the main program. The second strategy is to provide a set of `print` replacements and post-processing tools that better organize and present the printed output. While the code-weaving and data-visualization features of Medic are not new in isolation (Czyz and Jayaraman 2007; Lienhard et al. 2009; Stamey et al. 2005; Usui and Chiba 2005), Medic demonstrates how to combine those pieces into a more effective debugging tool.

Medic's language for describing and integrating `print` statements into a program is similar to AOP systems and the advising facility in PILOT (Teitelman 1966), but tailored to the specific needs of trace debugging. Similarly, with specific data-visualization techniques in mind, Medic supports four kinds of traces:

- **Log traces**: Similar to traditional `print` statements, log traces are linear and text-based, but they include

FPW'15, October 26, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3905-6/15/10...$15.00
http://dx.doi.org/10.1145/2846656.2846658

conveniences for recording the *source context*, *function behavior*, and filterable *layers* in trace output.

- **Graph traces**: A tracing graph presents a new means of recording *relationships* among traced values, ultimately allowing programmers to visually understand the spatial relationship between trace elements. Graph traces are complementary to text-based traces, which cannot easily represent connective relationships.
- **Aggregate traces**: Aggregate traces enable grouping multiple trace elements together. Programmers can *scrub* through the accumulated steps of data—seeing one step of data at one time—and enable *data comparisons* by specifying the two comparing data points.
- **Timeline traces**: Each individual trace element is represented by a *timeline*, which shows an overview of the evolution of values over time. A timeline slider enables step-wise exploration in an abstract execution time view or a clock-based view, and a tooltip window allows displaying an individual value.

In the remainder of this paper, we describe the design of Medic's trace-oriented metaprogramming language and its four kinds of traces and their visualizations. In each section on a specific tracing facility, we show how the tracing facility is useful for finding bugs and understanding program behavior. Finally we discuss the implementation of the debugging system, including the interpretation of the Medic language, the weaving of debugging code, and the generation and representation of traces.

2. A Trace-Oriented Metaprogramming Language

Medic's metaprogramming facilities start with the ability to describe the placement of debugging instructions. Programmers must specify the *locations* in the source program where the debugging instructions are added. We provide three categories of *scope* specification of locations: module-level (through the `(in #:module` *module-name match-form ...*`)` form), function-level (through the `each-function` and `(`*f ...*`)` forms), and expression-level (through the `at` form).

After the scope of location is identified in the source program, debugging code can be inserted either on entry to the identified code or on exit using the `[on-entry` *source-expr ...*`]` or `[on-exit` *source-expr ...*`]` form. While some AOP systems constrain inserted code so that it preserves modularity and encapsulation properties of the original program, Medic imposes no constraints, as is appropriate for debugging purposes. Medic thus allows an inserted form to access any identifier that is visible in the source program at the insertion point.

Medic also provides a means of modularity and abstraction. A Medic program consists of different *layer*s, where a layer modularizes debugging code for a specific functionality and groups traces produced by the `log` form. A layer is declared by `(layer` *layer-id layer-form ...*`)` with an optional `#:enable` specification that can enable or disable a layer's output. Within a layer, we can enable abstraction and abstraction and parameterization of a fragment of code, which can be either run-time code or a fragment of metaprogramming specification.

Besides allowing access to elements of a source program in debugging instrumentation, Medic reflects additional information for use in certain debugging forms. The function-name variable is bound to the enclosing function or a function being called. For example, instead of tediously annotating each function `f`, `g`, etc., with `(print "f function entered")`, `(print "g function entered")`, etc., Medic lets a programmer write

```
[each-function [on-entry @log{@function-name function entered}]]
```

The string-template notation here using @ is an alternate representation of an S-expression, and is the same as used for Scribble (Barzilay 2009; Flatt et al. 2009). Through `(with-behavior` *f template*`)`, a programmer can define the logging behavior, represented by *template*, of the *f* function. Inside *template*, we allow the @arg and @ret escapes to access a function's argument and return value.

3. Trace Debugging

Medic supports four kinds of tracing (i.e., variants of `print`) with associated visualizations, each of which is useful for different debugging tasks. All traces are directed to an interactive graphical user interface, a *trace browser*. The trace browser consists of four panes that correspond to the four kinds of traces: a Log pane, a Graph pane, an Aggregate pane, and a Timeline pane.

The syntax and semantics of each tracing form are as follows:

- **Log forms.** `(log` *e*`)`, `(log` *form v* `...)`, `@log{`*template*`}`: Add a log entry in the Log pane. For the second `log` form, where *form* is a string containing `~a`s, `log` substitutes the value from the *v*s corresponding to the positions of `~a`s. The last `log` form allows the use of @expr inside *template*.
- **Graph forms.** `(node` *v* `[`*node-label color*`])`, `(edge` *from to* `[`*edge-label color from-label to-label*`])`: The `node` form creates a node associated with *v* in the graph visualization, and `edge` creates an edge between the node associated with *from* and the node associated with *to*. The square brackets specify optional arguments. When there exists no explicit node creation from *from* or *to*, `edge` creates a corresponding node.
- **An aggregate form.** `(aggregate` *v* `...)`: Add an aggregate entry in the Aggregate pane.
- **Timeline forms.** `(timeline` *v*`)`, `(assert` *pred*`)`, `(same?` *v*`)`: Add a timeline entry in the Timeline pane.

```
                                              f.rkt
(define (f x y)
  (+ (sqr x) (sqr y)))
```

```
                                         f-medic.rkt
(layer layer1
  (in #:module "f.rkt"
      (with-behavior f
           @{f: @x squared plus @y squared is @ret} )
      [on-exit (log (f 3 4))
               (log (f 4 5))]))
```

Figure 1: Showing the behavior of data

```
                                                   find-path-medic.rkt
(layer left-path
       (in #:module "find-path.rkt"
           [at (if left-p _ _)
            [on-entry
             (log "left branch: ~a, ~a" (cadr t) left-p)]]))

(layer right-path
       (in #:module "find-path.rkt"
           [at (if right-p _ _)
            [on-entry
             (log "right branch: ~a, ~a"
                  (caddr t) right-p)]]))
```

Figure 2: Showing the layer of interest

3.1 Log Tracing

Log tracing with Medic's `log` form is similar to traditional trace debugging with `printf`, but Medic's logging facilities simplify the construction and browsing of printed output. The `log` form not only produces traces sequentially in execution order, but it also augments traces with useful context information.

For example, suppose that the value of `x` is 3. The expression `(log x)` produces a trace entry

```
x = 3
```

in the Log pane of a trace browser. Unlike the traditional `print` statement, which merely prints out `x`'s value, `log` produces extra context information about the value: the name of the variable under inspection. This automatic addition of context by `log` relieves the programmer of tedious string-templating work for simple logging output.

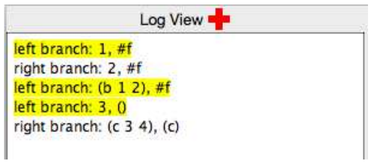
The `log` form recognizes function calls as well as variable references, and it cooperates with Medic statements that adjust the format of output for function calls. Using `(with-behavior` *f template*`)`, a programmer can control the way that log output is written for `f` calls. This centralized control provides an alternative to duplicating template constructions at each logging site. Furthermore, the `with-behavior` form provides access to the result of the function calls, as well as the arguments, which allows the `log` form to more completely expose the behavior of the function.

For example, suppose that we write a program in the module `"f.rkt"`. We can log the behavior of calls to the `f` function by writing a Medic program in `"f-medic.rkt"` as shown in Figure 1. The `with-behavior` clause in `"f-medic.rkt"` changes the output produced by `log` whenever a function call to `f` is logged. Instead of printing just the result of calling `f`, `log` displays the customized behavior of `f`—"`f: @x squared plus @y squared is @ret`"—with `@x`, `@y`, and `@ret` replaced by arguments' values and the return value of `f` function call. After starting a debugging session by running `"f-medic.rkt"` and `"f.rkt"`, we get the following log entries (traces showing the behavior of data are highlighted in blue):

```
f: 3 squared plus 4 squared is 25
f: 4 squared plus 5 squared is 41
```

One of the most difficult aspects of trace debugging is determining the right amount of data to log. Logging too little data defeats the point, but logging too much data makes the interesting information difficult to find, and the right amount of logging is not always clear from the start. Similar to the layers in Adobe Photoshop, Medic's *layers* help programmers organize output so that layers of output can be selected interactively, which helps balance the needs of showing enough information and limiting the amount of information to inspect.

Figure 2 shows an example of defining layers for traces. To see the traces produced by `(log "left branch: ~a, ~a" (cadr t) left-p)`, the programmer can click the "Log View" button, which opens a layer-view window listing the layers of traces: `left-path` and `right-path`. After selecting the `left-path` layer, the Log pane updates the display of traces immediately, highlighting the traces that belong to the selected layer:



```
left branch: 1, #f
right branch: 2, #f
left branch: (b 1 2), #f
left branch: 3, ()
right branch: (c 3 4), (c)
```

Seeing traces in layers enables a programmer to make better comparisons of relevant trace elements among all mixed traces, while the execution order of traces is preserved.

3.2 Graph Tracing

Traces produced by `log` are linear and text-based. They print primitive values in a typical form, and by preserving the execution order of traces, they enable analysis of the evolution of a value in a program. Textual traces, however, provide a poor view of certain relationships among trace elements that could become immediately apparent in a graph view. With conventional logging tools, converting textual output

doubly-linked-list-medic.rkt
```
(layer dlist
       (in #:module "doubly-linked-list.rkt"
           [[at (values next (cons (get-field datum temp) lst))
                [on-entry
                 (when next
                   ; visualize the temp node's next link
                   (edge temp next "" "Red" (get-field datum temp)
                         (get-field datum next)))
                 (when prev
                   ; visualize the temp node's previous link
                   (edge temp prev "" #f (get-field datum temp)
                         (get-field datum prev))))]]]))
```

Figure 3: Graph tracing

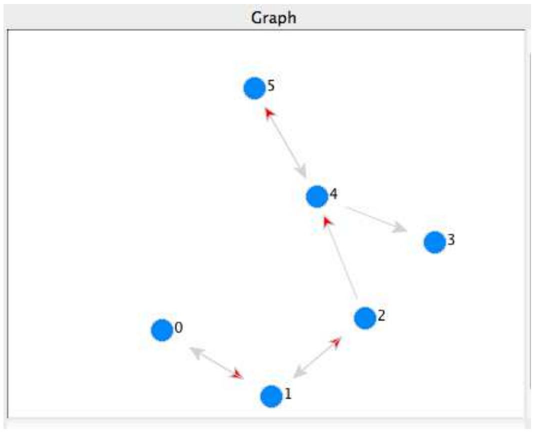


Figure 4: The graph view of a doubly linked list

to a graph view requires additional tools, careful formatting of output to fit the tools' input formats, and isolation of that output from other debugging output.

Medic directly supports the construction of *graph* output to help programmers see the otherwise hidden relationships among values. To generate a graph, instead of using `log`, the programmer uses the `node` and `edge` forms. To create a simple and aesthetically pleasing visualization, Medic uses force-directed algorithms to layout the output (Eades 1984; Fruchterman and Reingold 1991).

To illustrate Medic's graph tracing facilities, suppose that we have a correct implementation of a doubly linked list with support for common accessing, inserting, and removing operations. The `remove` method takes an argument, `i`, and removes the `i`th element from the list starting from index 0. We can create a bug in the `remove` implementation by commenting out the line of code that updates the previous link of a node, `temp-next`, to point to the node, `temp-prev`, when the node, `temp`, is to be deleted.

To test the broken library, we add ten numbers from 0 to 9 to the doubly linked list `dlist` and then remove five successive elements 3, 4, 5, 6, 7 from the list by calling `(send dlist remove 3)` five times. We can first use

fact-iter.rkt
```
(define (fact x a)
  (if (zero? x)
      a
      (fact (sub1 x) (* x a))))
(fact 3 1)
```

fact-iter-medic.rkt
```
(layer fact
       (in #:module "fact-iter.rkt"
           [(fact) [on-entry (aggregate x a)]]))
```

Figure 5: Aggregate tracing

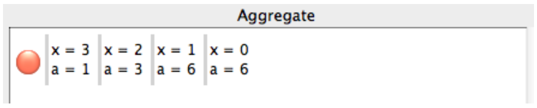
`log` to print out the elements at each step, and we notice we get a faulty list after the removal operation—the final list should be the sequence 0, 1, 2, 8, 9, instead of a sequence of 0, 1, 2, 4, 5. However, the tracing log gives us little insight into the cause of the problem. If we use `edge` to visualize the doubly linked list (see Figure 3), we can see the problem instantly. As shown in Figure 4, the doubly linked list is broken with a unidirected edge between nodes 2 and 4.

In our example, the test and `edge` declarations were part of the metaprogram. When the library might have its own tests, Medic's metaprogramming facilities can be used to weave `node` and `edge` declarations into the library's implementation to track down the source of a test failure.

3.3 Aggregate Tracing

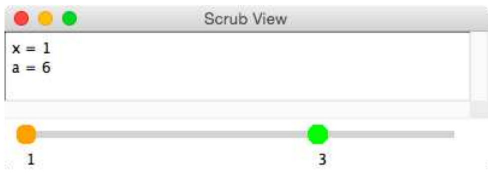
A programmer can use linear traces with multiple values in each entry to detect a relationship between values and how they change together. A manual inspection of linear output, however, can make those changes difficult to extract from the layout and noise of traces. Medic's `aggregate` form presents trace output in a way that makes related output values easier to inspect and compare.

Consider the source and Medic programs shown in Figure 5. In the source program, the `x` and `a` values change together across calls to the function, and inspecting them as a pair can help a programmer understand how they work together. Specifically, using `(aggregate x a)` produces a result that is more organized than a linear trace:



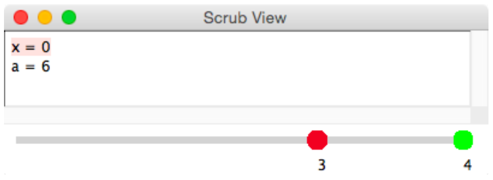
```
x = 3 | x = 2 | x = 1 | x = 0
a = 1 | a = 3 | a = 6 | a = 6
```

When `(aggregate x a)` is evaluated many times, as in `(fact 1000 1)`, the resulting large number of traces must be pruned to expose the values at each step and enable comparison at different steps. Clicking the red button to the left of the `aggregate` trace view opens a scrub-view window, which allows the programmer to inspect the traces step-by-step (currently at step 3):



```
x = 1
a = 6
```

The scrub view provides two slider handles; the window displays the current step of traces indicated by the second slider handle, but it compares that value to the one selected by the first slider handle. For example, moving the orange slider handle to step 3 and right-clicking on it turns the slider red, which marks the step for later comparison. Then, moving the green slider handle to step 4 compares the values at step 4 to the values at step 3. The difference between two steps is highlighted in pink:



```
x = 0
a = 6
```

3.4 Timeline Tracing

When traces involve changes over time, programmers need to see the overview of data in a temporal fashion. There are a few possible ways to present traces such as using a slider to "time travel," using "timeline" views, and using "stroboscopic" views (McDirmid 2013). Inspired by the timeline view of data of Victor (2012), which helps programmers understand data with visual context, instead of "peeking through a pinhole," Medic provides three forms for timeline tracing: `timeline`, `assert`, and `same?`. These three forms all generate traces with a view similar to a timeline view, where each trace element is arranged along the vertical axis, while the changing values of each trace element are displayed along the horizontal axis. By default, the timeline view's horizontal axis corresponds to an abstract execution time reflecting the order of logged events, but not the delays between events. A clock-based view is available in a separate window.

As a further refinement over aggregate tracing, the timeline view automatically determines a graphical presentation mode for some logged values. For a given element `v` in a trace, if all occurrences of `v` are *number*s, a line plot is rendered on the timeline (where each point is arranged in the square unit according to its numeric value). For *boolean* values, each square unit represents a value with false values colored red and true values colored blue. For other data types, a textual form is displayed.

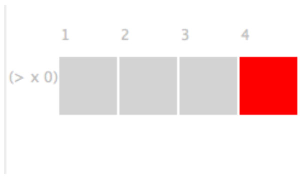
To illustrate, for the programs shown in Figure 6, the left panel of Figure 7 presents the resulting timeline view. The timeline slider on the top can step through the timeline traces showing multiple values with the same horizontal coordinates at the same time (see the right panel of Figure 7). To examine an individual value, we can click the corresponding square unit, which shows the current value as a tooltip.

```
count.rkt
(define (count-length v count)
  (if (null? v)
      count
      (count-length (cdr v) (+ count 1))))
(count-length (cons 8 (cons 9 '())) 0)
```

```
count-medic.rkt
(layer count
       (in #:module "count.rkt"
           [(count-length)
            [on-entry
             (timeline count)
             (timeline v)
             (timeline (null? v))]]))
```

Figure 6: Timeline tracing

A timeline element produced by `(assert pred)` is similar to `pred` as a boolean result, but true values are deemphasized by coloring them in gray, while false elements are highlighted in red. For example, with `(assert (> x 0))` and when values of `x` over time are 3, 2, 1, and 0, the assertion fails on the fourth value of `x` producing the following timeline:



Although comparisons between two elements of a trace are sometimes useful, a comparison of one trace element with its initial value is more often useful. A programmer could change (through metaprogramming) the source program to propagate the old version, but Medic makes the comparison considerably simpler through a `same?` form in a timeline trace. The `same?` form always produces true for the initial trace. Afterwards, the result is true only if the trace element's value is the same as the initial trace; the `same?` predicate compares values like Racket's `equal?`, but is extended to perform a deeper comparison by traversing opaque structures and objects. Like `(assert pred)`, only false values produced by `(same? v)` are highlighted in red in the timeline.

Clicking the "Time View" button opens a variant of the timeline view as shown in Figure 8. A programmer can slide through time to see which events take place at a particular time, showing not only the relative order for events of interest but also the gaps between events. The programmer can scale this view to different time granularities (second, mil-

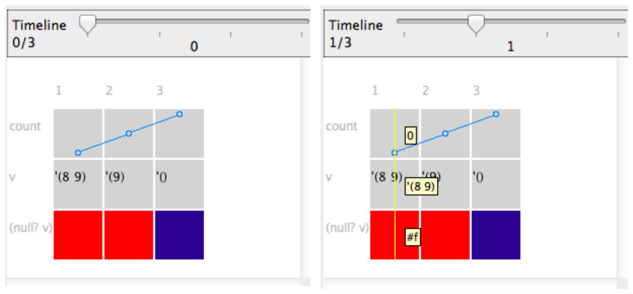


Figure 7: Timeline traces for timeline form

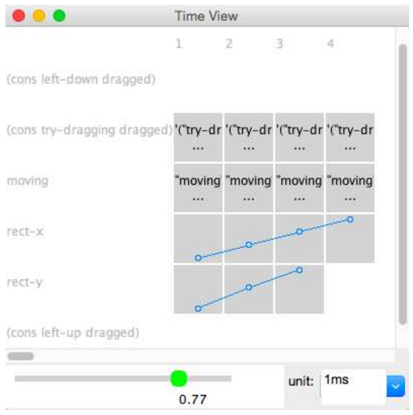


Figure 8: Time view

lisecond, or tenths of a millisecond) to explore events at different scales.

4. Implementation

The implementation of Medic leverages Racket's ability to support a completely new language, such as `#lang medic`, plus Racket's ability to macro-expand a program written in any language to a common core language. This combination means that Medic can offer a specialized language for writing metaprograms, and those metaprograms can introduce debugging annotations on programs written in any language that compiles to recognizable core forms, such as definitions and functions.

A debugging session starts by interpreting a Medic program, which describes debugging instructions in terms of a source program and logging instrumentation to add to the program. Interpretation extracts debugging instructions and their related debugging information, such as source location and trace layer id, into debugging tables that drive an instrumentation phase. In the instrumentation phase, logging forms are woven into the source program (a full model of the instrumentation process available at `http://www.cs.utah.edu/~xiangqi/html/documents.html`) . The instrumented program is then compiled and run, and generated traces are piped to the front-end trace browser. The trace browser processes traces and presents them in a visual way with interactive exploration.

5. Experience

To gauge Medic's effectiveness for debugging, we have used it for our own debugging tasks. In addition, we debugged student solutions for a programming assignment, which let us try Medic on a larger selection of programs and with several kinds of realistic errors made by different programmers.

Student programs, logging, and graph traces. To try Medic on student programs, we ported programs previously written in Java for implementing a vector abstract data type. The assignment's underlying data structure made the implementation of the interface especially error-prone. We performed line-by-line ports of the students' work from Java to Racket.[1] Using structures and functions instead of methods was straightforward, since the assignment and solutions did not rely on inheritance. The assignment was originally graded by a driver program using print statements to report test results. We wrote an equivalent driver program in Medic, and the medic driver proved more effective in meaningfully detecting errors in students' implementations. Medic's support for reuse of debugging metaprograms made debugging multiple students' programs faster and easier.

To debug some faulty implementations, we used `log` to examine variable values. After recognizing an incorrect value of a suspect variable, we narrowed the problem to a specific function, and we wrote another `layer` form with several function-level `log` statements. We ran the program again, and the *layer* feature of log traces helped us focus on the traces produced by the second trial of `log` statements without the distraction of output from other irrelevant traces. As we debugged the code more, many traces were produced, and we resorted to layer filtering, which helped us discover why two identical trace entries were shown in different places.

In other cases, we used `node` and `edge` to expose the hidden state of the test results. The vector structure was similar to a doubly-linked list, where each vector element has a previous and a next reference, and the homework required returning a new vector if an interface function must modify the vector. Graph traces allowed us to check whether a vector is properly constructed by examining the vector's head, tail, and references between vector elements. Most importantly, object identity, which is hard to explore with

[1] Many Java programs can run directly in Racket by installing the `"profj"` package and prefixing the programs with `#lang profj/full`. Unfortunately, the macro-driven compilation process from Java to Racket mangles the source too much for S-expression-based locations to be effective. In the near future, we plan to refine Medic's support for expression locations to provide a more direct path to debugging Java programs.

traditional `print` statements, was clearly exposed by graph traces; if a student failed to create a new vector as suggested by a method's signature, we could easily see that the original vector shown in the Graph pane had been modified, instead of seeing a different vector alongside the original.

Aggregate traces. In an unrelated project, we experimented with a Markov decision process problem to determine which discount value would result in the convergence of state A and E utilities. For each state, the value iteration algorithm computes the Q-state values, and it updates the utility value by choosing the maximum Q-state value at each step. Since we wanted to know the utility values of both states at each step, we used Medic to add an `aggregate` statement to show the values of discount, step number, state-A utility, and state-E utility. The program produced an overwhelming number of aggregate traces, but Medic's scrub view allowed us to scrutinize values at each step. For an occurrence of abnormal patterns of utility values, we isolated the cause of the problem by adding a `same?` statement to ensure that the state-utility table held a correct initial value for each state; the trace then revealed the bug of neglecting to reset the table on entry to the loop iterating through possible discount values. After fixing the bug, we restarted the program and scrubbed through the aggregate traces again. By using the data comparison feature of the scrub view, we managed to compare values at two arbitrary steps and look for unchanged values.

Timeline traces. In another use of Medic, we investigated a buggy GUI application that failed to move objects in response to a mouse drag in a window. Three kinds of mouse events are involved, and a state variable, `to-move?`, determines the behavior of the window when a mouse-drag event is received. We added a `timeline` statement at the button-press and left-button event handlers to ensure that each one updated `to-move?` correctly. We also added several `timeline` statements in the mouse-drag event handler to check whether the event was fired and to see the coordinates of an object. After running the instrumented program, the Timeline pane displayed timeline traces, which helped us to see that each mouse-event handler was called; since the Timeline pane does not explicitly show the time of events, however, we could not know whether two events were happening at the same time. By opening a time-view window, we were able to slide through the execution time of traces and see what events were triggered at a particular time. Overall, the timeline views proved far more effective than plain textual logging, since traces for relevant events would be scattered through the logged output.

Performance. For our debugging tasks, Medic added a negligible overhead to compilation and execution times. The runtime overhead of Medic is linear in the amount of tracing. Medic's space overhead depends on the logged values; since Medic saves each logged value, logging annotations potentially change the space complexity of a program. For both time and space, programmers must be aware of these costs, but the cost model is straightforward.

6. Related Work

Our work is related to several areas: aspect-oriented programming for weaving debugging code into source programs, programmable debugging, visual debugging techniques, and data-visualization techniques to help programmers to better understand output traces.

The AOP paradigm expresses cross-cutting concerns in a separate modular unit, an *aspect*, which allows tracing without modification of source programs. Most AOP systems (Dutchyn 2012; Kiczales et al. 2001) provide limited join-point models and data access, while an aspect-oriented system called Bugdel (Usui and Chiba 2005) tries to tackle the limitations. PILOT (Teitelman 1966) provides an operation of advising that can modify the source program through modifying the interfaces between procedures. Medic shares Bugdel's support for invasive weaving and unrestricted references, but provide more trace-oriented support.

Several debugging systems separate debugging code from the source program and treat debugging as a programmable and reusable activity, like MzTake (Marceau et al. 2006), RAIDE (Johnson 1977), Dalek (Olsson et al. 1990), and Acid (Winterbottom 1994). In comparison, Medic is tailored to trace debugging and more specialized visualization tools and higher-level constructs for instrumenting the source program.

JIVE (Czyz and Jayaraman 2007; Gestwicki 2004; Girgis et al. 2005) is a declarative and visual debugger for Java, using object diagrams and sequence diagrams to visualize runtime states. The Compass debugger (Lienhard et al. 2009) presents an object flow diagram in a fisheye view to assist tracking down the flow of objects. DDD (Zeller 2004) offers various visual ways of examining data such as 2-D and 3-D plots, and Misha (Papoylias 2010) provides an API for visualizing language-oriented data types including numbers, strings, references and containers. Many existing tools provide timeline views, like DejaVu (Kato et al. 2012), the playground of Swift (Swift 2014), and the timeline display generator (Karam 1994).

However, Our work considers a more general setting involving many programming language paradigms where log, graph, aggregate, and timeline views of data are essential, and synthesizes many of these ideas into an effective debugging tool.

7. Conclusion

We have presented a debugging tool, Medic, that employes a trace-oriented metaprogramming language to describe the task of debugging and incorporates four kinds of enhanced tracing statements to improve the trace debugging experience. We showed the usefulness of the enhanced tracing statements in various common debugging settings, and we demonstrated a more programmer-friendly and visual experience with the Medic debugging system.

There are three promising directions to further enhance the trace debugging experience. Currently our metaprogramming language is a starting point for facilitating a source program's manipulation, but a more sophisticated model will be added to support efficient debugging. We also plan to add more visualization support for trace output such as navigation between trace elements in a trace browser and associated source locations and live programming features. Finally some other debugging techniques such as setting breakpoints, stepping, and reverse execution can be combined to explore trace output more effectively.

Because of the recent popularity of domain-specific languages (DSLs) and the sparse debugging support for DSLs, we'll also investigate facilitating construction of DSL debuggers by extending the work in Medic to evolve the system into a debugging framework for DSLs.

8. Acknowledgments

We would like to thank Sean McDirmid for sharing his direction and work in the area of debugging. In addition, we appreciate the support of the members of the Racket community and unnamed reviewers for their feedback and suggestions on the work.

Bibliography

Eli Barzilay. The Scribble Reader. In *Proc. Wksp. on Scheme and Functional Programming*, 2009.

Jeffrey K. Czyz and Bharat Jayaraman. Declarative and Visual Debugging in Eclipse. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications Wksp. on Eclipse Technology Exchange*, pp. 31–35, 2007.

Christopher J. Dutchyn. AspectScheme–Aspects in Higher-Order Languages. In *Proc. Workshop on Scheme and Functional Programming*, 2012.

Peter Eades. A Heuristic for Graph Drawing. *Congressus Numerantium* 42, pp. 149–160, 1984.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. Intl. Conf. on Functional Programming*, pp. 109–120, 2009.

Thomas M.J. Fruchterman and Edward M. Reingold. Graph Drawing by Force-Directed Placement. *Software—Practice & Experience* 21(11), pp. 1129–1164, 1991.

Paul V. Gestwicki. Interactive Visualization of Object-Oriented Programs. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 48–49, 2004.

Hani Z. Girgis, Bharat Jayaraman, and Paul V. Gestwicki. Visualizing Errors in Object Oriented Programs. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 156–157, 2005.

Mark Scott Johnson. The Design of a High-Level, Language-Independent Symbolic Debugging System. In *Proc. ACM '77 Annual Conf.* , pp. 315–322, 1977.

Gerald M. Karam. Visualization Using Timelines. In *Proc. ACM SIGSOFT Intl. Sym. on Software Testing and Analysis*, pp. 125–137, 1994.

Jun Kato, Sean McDirmid, and Xiang Cao. DejaVu: Integrated Support for Developing Interactive Camera-Based Programs. In *Proc. 25th annual ACM Sym. on User Interface Software and Technology*, pp. 189–196, 2012.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proc. European Conf. Object-Oriented Programming*, pp. 327–353, 2001.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. European Conf. Object-Oriented Programming*, pp. 220–242, 1997.

Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. Flow-Centric, Back-In-Time Debugging. In *Proc. TOOLS Europe*, pp. 272–288, 2009.

Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The Design and Implementation of a Dataflow Language for Scriptable Debugging. *Automated Software Engineering* 14(1), pp. 59–86, 2006.

Sean McDirmid. Usable Live Programming. In *Proc. Sym. on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 53–62, 2013.

Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. Dalek: A GNU, Improved Programmable Debugger. In *Proc. Usenix Technical Conf.* , pp. 221–232, 1990.

Nick Papoylias. High-Level Debugging Facilities and Interfaces: Design and Development of a Debug-Oriented I.D.E. In *Proc. 6th Intl. IFIP WG 2.13 Conf. on Open Source Systems*, pp. 373–379, 2010.

John W. Stamey, Jr., Bryan T. Saunders, and Ryan Watts. Aspect-Oriented Debugging. In *Proc. Intl. Conf. on Aspect-Oriented Software Development*, 2005.

Swift. 2014. `https://developer.apple.com/swift/`

Warren Teitelman. PILOT: A Step Toward Man-Computer Symbiosis. PhD dissertation, Massachusetts Institute of Technology, 1966.

Yoshiyuki Usui and Shigeru Chiba. Bugdel: An Aspect-Oriented Debugging System. In *Proc. Asia-Pacific Software Engineering Conf.* , pp. 790–795, 2005.

Bret Victor. Learnable Programming. 2012. `http://worrydream.com/LearnableProgramming/`

Phil Winterbottom. Acid: A Debugger Built From A Language. In *Proc. Usenix Annual Technical Conf.* , pp. 211–222, 1994.

Andreas Zeller. Debugging with DDD. First edition. GNU Press, 2004.