Debuggers in Dynamic Languages

Rocky Bernstein

June 6, 2010

Abstract

I have written debuggers for a number of dynamic languages and found that additional run-time support invariably needs to be added. The state of the art in debugging has not been improving. The paucity of modular testable and debuggable debuggers (many debuggers cannot debug themselves, nor do they come with a test suite for verifying their correctness) is perhaps due to confusion about what is needed at run time and about how to organize a debugger.

So, I suggest a modular program structure for a debugger that I have been using. However first, I describe some of the support features which are commonly missing. These are: (1) better position information for dynamically generated code, e.g., from a method defined by a string variable in a program or coming from a data structure representing some code; (2) verification that source code matches the object code; and (3) better, faster support for breakpoints, "step over" and "step into."

There is a dichotomy between high-level (language level) and low-level (VM instruction) debugging, but there does not need to be. The popular debugger gdb is basically a low-level debugger that tries to support some high-level languages. Debuggers can also go the other way around, but they usually do not. Typically Ruby and Python debuggers, other than the ones I have written, do not allow for VM or byte-code inspection.

Therefore the debuggers are not seen as tools for the compiler writer, for example for testing code generation or optimization techniques; and this is another reason that support for them is not stronger. I believe that debuggers should also be able to debug optimized code as well as unoptimized code.

1 Introduction

I have written debuggers for a number of dynamic languages [BS09, Ber09a, Ber09b, Ber09c, Bera, Berc, Berb, Berd] and have found that additional run-time support invariably needs to be added. Compare debugging capabilities typically available now with those Smalltalk, a language 30 years old [III].

It seems that the state of the art in debugging has been not been improving. The Integrated Development Environments (IDEs) are not worse, but the foundation that they have to interact with is. Test- and behavior-driven development, interactive shells, modular and functional programming, and proving programs correct are all valuable techniques which reduce the need for debuggers and debugging, but they do not eliminate it.

Practical support for debugging is often weak. Consider a typical systems administrator who may not be all that interested in programming languages and programming-language theory. When something goes wrong, he or she may have to troubleshoot and fix a complex, non-modular, untestable program written by someone else. And instead of having a simple, lightweight tool to diagnose the problem, there may be only a complex and heavyweight IDE that poses problems of its own.

At the other, theoretical, end of the spectrum there are authors and teachers of a particular programming language who devote much more time to the programming language's theory and design than to the mundane

task of debugging an existing program. Consequently, in many reference manuals for popular dynamic programming languages like Ruby, Scala, or Python, there is only a little sketchy information on debuggers or debugging.

Perhaps such lack of interest in debuggers is partly because the debuggers are very weak, which is in turn because of the lack of support inside the language: a little bit of a bootstrapping problem. And perhaps lack of support partially stems from confusion about what is needed at run time, and about how to write a debugger in a modular, testable, and debuggable way—it is an unfortunate fact that many debuggers cannot debug themselves.

So, first I outline how to structure and write a modular debugger. Then I describe some of the support features commonly missing from programming languages. Many of these are not specific to debugging or dynamic languages; better error-location reporting is one of these.

2 Run-time support

2.1 Position information

When a debugger is added to a programming language, invariably one of the first things noticed is that there are bugs in how the program reports source locations.

A common example: in languages that allow a statement to span several lines, the location line for the entire statement is recorded where the statement ends rather than where it starts. Generally this is because the location comes from the position information of the scanner (usually a global integer variable containing a line number) when the statement ended; saving information about where the statement started requires additional work. The best practice would be to give a range of locations telling where the statement starts and where it ends.

Programmers tend to tolerate, although they do not appreciate, some fuzziness when the programming language reports a fatal error in the program. However vagueness about whether the debugged program is stopped before or after a particular line of code cannot be tolerated in a debugger. A programmer has to be able to set a stopping point which is both precise and feels natural, that is, which matches his understanding of the source code rather than the machine's understanding of the object code.

I propose that the concept of "position" contains two parts: a container and a place within that container. These correspond roughly to the traditional "file" and "line number." However, in dynamic languages the source code does not always come from a file. It can come from a program object (e.g., a string), or it can be a member of a package of files (e.g., Java jar, or Python egg). So a container should have the ability to refer to other containers. Position inside a container poses a number of other problems. One is that "line" is a bit vague. Perhaps a range of lines would be better, or possibly character offsets, or a range of line- and column-number pairs.

A second problem with reporting the position for a stopped location is that there may be many source-code positions that correspond to a stopped location. With code optimization, such as common-subexpression elimination, a single VM or machine instruction could map to sets of positions in a source program.

There are various ways to solve this problem; for example, all the possible corresponding positions can be listed. The important point is that all the information about the positions within the containers must be preserved.

2.2 Information to verify that source code matches object code

Dynamic languages often provide file-name information for source files, but are missing checksum-like information so that one can verify that what the programmer is looking at matches what the interpreter is running. This problem is more likely to happen if the front-end IDE is different from the one where the program is running ("remote debugging") or when the programming language runs code from a bundled library.

This problem is easy to solve if the compiler runs a checksum as part of the scanning process, and then stores the result in the object.

2.3 Support for stopping inside the debugged program

Many dynamic languages provide debugging support via a trace mechanism which issues a callback to a hook. The debugger writer develops the hook in the language to be debugged and makes a call to register it. In this section, I give suggestions for run-time support to facilitate calling a trace hook inside the debugged program. From the programmer's perspective often this amounts to pausing execution inside program. However strictly speaking, the hook may decide to do other things and not pause. For example it may profile the code or save data about methods called. If the trace hook has to simulate some of the support listed below, the hook may decide not to stop. Not all programming languages have all of the features listed below, although most have some of them.

In the most primitive form, when the runtime does not have a callback hook, debugging can be obtained by having the source program call the debugger or profiler hook. Sometimes the callback hook is performed by instrumenting the source code to add such calls.

But most dynamic languages have some support for issuing a callback to a user-supplied hook within the programming language. The hook gets called immediately before some run-time event, such as calling a procedure or running a language statement. Given this, it is possible to implement "step over," "step out," and breakpoints in the hook itself and that is what many debuggers, such as, pdb, pydb, pydbgr and rdebug, have to do. The greatest slowness is generally seen in handling breakpoints and "step out."

2.3.1 Breakpoint support

Most dynamic languages that implement trace hooks, such as Python, POSIX shells, and Ruby, do not allow setting breakpoints inside the interpreter. A notable exception to this is Perl, ironic since it is one of the older languages. Given this, the hook code must look at location information and determine when a breakpoint has been encountered. Switching back and forth between running program and hook code, and the additional overhead of checking position information, are slow. When debugging a program slows it down, the debugger can not be used to find race-condition bugs. Furthermore, the programmer is discouraged from using a debugger, and then has to devise ways of circumventing flaws in the run-time debugging. A common workaround is to change the source code to put a call to the debugger where the programmer wants to stop. This allows the program to run at full speed until the point when the debugger is needed, making use of dynamic facility of the programming language to pull in code at any time.

However what is really needed is a way to register a stopping point inside the interpreter. Full support for handling conditions under which one needs to stop is not needed. Even support for one-time or temporary breakpoints is not necessary—that can all be done in the higher-level code outside of the interpreter. But some support inside the interpreter is extremely useful in order to handle breakpoints efficiently.

Here are some techniques for allowing for breakpoints inside an interpreter. One of the oldest techniques,

used for example in IBM mainframe assembly code, is to patch over the instruction sequence with a "trap" instruction [Sch07]. One needs to store the overwritten instruction and simulate it when the trap is over. With a virtual machine, a similar and perhaps simpler way is to store a trap bit in the VM opcode. In my experimental Ruby 1.9 debugger, I allocate a parallel byte vector for each instruction sequence on demand where the programmer has set a breakpoint. I did this for simplicity of implementation, but it also has an advantage that since it does not change the instruction sequence, it allows for better security in checking for tampering with the code [Sei].

2.3.2 Frame Support

When stopped in a running program, typically programmers want to know where they are in the source code. Perhaps they want to see and modify variables and objects. One straightforward way to do this is to encapsulate the run-time stack frame as an object. From the stack frame one often can get information such as source location or the frame's calling frame. In some programming languages, such as Python and Ruby, this information is shown when a fatal error occurs.

Some dynamic programming languages like Python provide the frame only as an implementation-specific feature rather than as part of the language standard. Other programming languages are object-oriented or do not have a frame object, so additional variables and functions are provided to get this information. For example, in the POSIX shell Bash, there are global arrays to get caller information. When the debugger ruby-debug is loaded, it installs a kernel method to allow evaluation of expressions in the context of frame on the call stack.

Here is some information typically accessed through a frame:

1. source-code position
2. function name and its parameter names and values
3. ability to evaluate expressions in the context of a frame
4. access to the frame's calling frame.

However providing a frame object is not without some downside. Frames are ephemeral and disappear as the debugged program executes. Some means is needed to ensure that a frame has not disappeared.

In rbdbgr the following heuristic is used. Before we allow access to fields within the frame, there is a check to see whether the frame address lies within the boundaries of the valid frames. On frame creation, certain parts of the frame that we know will never change are copied; some examples are the source location information for the frame, or the name of the method that the frame belongs to if there is one. On access, we check all of these fixed fields to see if they are the same. If not the frame is invalid.

2.3.3 Nested Debugging

When a trace hook is called, the runtime usually sets a flag to indicate that the hook is running so no further callback events should occur. Otherwise upon running the statement inside the trace hook, the runtime would call the trace hook again ad infinitum.

But sometimes while stopped inside a program being debugged, one wants to enter a new expression and debug into that. The simplest way to do this is to allow the "stop trace" flag to be reset. In Python one passes a function and the trace bits are saved and restored. In rbdbgr, the debugger routine to nest debugging handles saving and restoring these flags.

3 The Anatomy of a Debugger for a Dynamic Language

The debuggers I have been working on all have a similar structure that I find works well. In Object-oriented debuggers for dynamic languages here are some of the major components:

- Interface
- Command Processor
- Commands/Subcommands
- Trace Filtering
- Line Caching and Position Remapping

Each of these parts is described below. The last two items however are separate packages as they may be useful in other situations. For example Trace Filtering may also be used in a profiler while Line Caching and Position Remapping are useful for any source-code introspection tool. Both packages would be useful in a code-coverage tool.

3.1 Interface

Sometimes encapsulate debugger input and output is not encapsulated, such as earlier versions of pdb [vRea00]. When this happens there is the temptation to use an ad hoc method to redirect I/O. This causes messy code and ultimately may miss some situations. To handle the different modes of debugging, I find it useful to have an "interface" class which encapsulates I/O.

Some examples of interfaces which I use:

- Debugger and interactive user are in the same process
- Similar to the previous, but debugger handles batch command processing
- Interactive user input but the program to be debugged is in another process
- The opposite of the previous. Debugger is inside the process but the user is at the other end of an out-of-process connection
- Interface used in testing—for example a simple array of command inputs and an array of string of output lines

Interfaces also handle formatting I/O. For example if interaction is via a web service, formatting may be in XML.

My interface classes handle I/O at a slightly higher-level than that for traditional I/O. For example some methods in the interface class prompt for user input or for confirmation before a dangerous action (which in script modes is preset to return either confirmation or denial). There may be special kinds of control I/O when the interface communicates outside of the process, and here solicitation for debugger command input may be separated from other kinds, such as debugged program input.

3.2 Command Processor, Commands and SubCommands

The command processor mediates between the actual commands (or "view" of MVC) and the internal state of the debugger or run-time system (analogous to the "model" of MVC). This might be in the form of a command-line interface, or it might be XML.

In some debuggers, such as perldb or pdb, each command is a method. However in Object-oriented languages it is useful to make each command be its own class, a subclass of a general command class. Doing this makes it simpler to associate commonly needed properties with the command—help for the command, short help, what category of commands this command is in, and so on. Furthermore making each command its own class object facilitates making subcommands for the command. Some examples of commands which have subcommands in gdb-like debuggers are "info," "set," and "show." Subcommands of "info" include "program" (show information about the program debugged), and "args" (argument variables of the current stack frame). By the same token, making each subcommand be a class facilitates giving it properties and adding subcommands to the subcommand. A last benefit of making each command in its own class and putting a small number of these in a file is that it is easy to test individual commands in isolation.

To collect commands, I put each command in its own file. Some debuggers [BS09] group related commands like gdb "up" (move up a stack frame), and "down" (move down a stack frame) in a single file. However I find it simpler just to have one command per file. These commands are collected in a "command" directory. Dynamic languages then can just load all the files located in a directory. This also makes it possible to allow for user customization of commands via a user command directory which is separate from the debugger's base command directory. Likewise commands which require OS-specific or language-level features can be grouped and isolated this way. Lastly if there is a bug in that command one can fix the file and reload it thereby fixing bugs in debugger commands without having to leave the debugger and redo context to get to the point where the bug is seen.

3.3 Trace Filtering

The debuggers I have been working on recently have a layer that handles some basic kinds of filtering to decide whether or not to run a trace hook. This accommodates some common situations which are described below. Also the trace filtering also adds support for chaining hooks. Python and Ruby's API does not allow for hook chaining, even though underlying support is there in Ruby. Hook chaining allows profiling and debugging to use the same underlying mechanism.

One common situation is that there may be a debugger method that needs to set up some of the debugging environment. Since this routine is part of the debugger package rather than the debugged program, one does not want to stop inside it. For example in rbdbgr there is a "debugger" method which one can add to the source code to force an explicit call to the debugger trace hook.

Two ways I have seen this handled are either to arrange to set the tracing bit on as the last instruction of the routine; or to count the number of events that the routine may trigger, and tell the hook to skip that many trace events. Neither of these methods is robust or easy to use. Instead in the trace filter I have a list of method names that can be registered which get ignored.

Another thing a trace filter might do is filter out certain types of events. However this is best done inside the runtime, otherwise tracing slows down. Both Python and Ruby do provide this kind of trace filtering but in both cases the kinds of filters are fixed. Profiling works for calls and returns, while debugging works for all events.

3.4 Line Caching and Position Remapping

Listing source code would be slow if every time one wanted to see some portion of source code the file were opened and that portion extracted. Consequently some dynamic debuggers read in the source code the first time the code is loaded. Python debuggers do this for example using the package "linecache." Ruby provides a mechanism to allow the source code to be saved if a special constant `SCRIPT_LINES__` is defined to be a hash type. The advantage of this over the Python approach is that one can be sure that the array matches what Ruby is running. However in practice one hardly notices any discrepancies.

It may happen, though, that one is not coding directly in the dynamic language, but rather through some sort of macro-expansion mechanism such as a templating system. In Ruby/Rails, Merb [eab] and Erb [eaa] are examples of such templating systems. So it may be helpful to have a way to remap portions of one source text onto another. This would also allow the source text to be changed outside of the debugger using the new location names but have them map to corresponding locations in the running code.

We have also seen that sometimes code is generated dynamically such as from a string or from some structures. Having a position-remapping module facilitates showing such code by extracting the text and saving it in a temporary file so that it viewed though tools IDEs or tools that typically work with files.

4 Similarity between a Debugger and a Command Shell

As mentioned previously, many debuggers for dynamic languages allow for stopping a running program for inspection of a profiler or debugger using a callback hook. When a program is stopped inside a debugger, things are not that different from being in the programming language's interactive shell (if it provides one). In fact many of the debuggers I have worked on allow going into the native shell from inside the debugger (e.g., "irb" for Ruby, "ipython" or the python shell for Python). This allows programmers to use the familiar shell interface while also reducing the need of the debugger writer to provide some of the more involved facilities provided by the programming shell.

5 High-level and low-level debugging

There is a dichotomy between high-level (language level) debuggers and low-level (VM instruction) debuggers. The popular debugger gdb is basically a low-level debugger that tries to add support for some high-level languages. Debuggers can also go the other way around, but they usually do not. Ruby and Python debuggers (other than the ones I have written) do not allow for VM or byte-code inspection. In one case where I requested more access to the lower-level structures, the VM writer resisted because of fear of giving low-level access to an average programmer. Debuggers in dynamic languages, then, are not seen as tools for the compiler writer, i.e., for testing code generation or optimization techniques. This is perhaps another reason why support for debuggers in such languages is weak.

Some debuggers of optimized code "decompile" the optimized code or undo some of the transformations that the optimization has performed [HCU92]. While this very helpful, it also increases the likelihood of Heisenbugs—bugs that disappear when a debugger brought in. Code which has a bug due to a race condition may suffer the Heisenbug effect. So while decompiling optimized code can be very useful, a debugger also needs to debug optimized code. And here what is needed is transparency regarding transformations that have taken place.

Often the debugger is not seen as a vehicle for allowing the programmer to understand what optimization and code improvements have been performed on the program. In Ruby, a compiler-option table is removed

after compilation; it is not kept for dynamic inspection (although this may be fixed in a future release). And recursion elimination is turned off by default for fear that it would confuse programmers. But adding a small amount of information to the object would allow a debugger to report such transformations when they occur, and I think that would be sufficient.

With gdb and gcc, programmers often debug optimized code, where sequential statements might be run in a different order from that given in the source. Unfortunately, code optimizers typically do not record information about the code motion, although this would not be difficult, and it would help a programmer follow execution of the program.

Suppose that whenever an instruction is moved by a code optimizer, information indicating that the instruction is not in its original location is recorded alongside the instruction stream. Then whenever a debugger is stopped on that instruction, it could issue a warning. If the optimizer recorded that the code was moved "up," the debugger would scan forward in the instruction sequence until a branching point for the first instruction that was not moved. This location could be reported as the current position, noting that an instruction from further down is about to be run. A similar procedure would apply if an instruction moved "down," but scanning backward rather than forward.

This example brings up a question regarding the high-level vs. low-level dichotomy: do we try to hide underlying mechanisms or to expose them? There is no one set answer. For code motion, the example above does both. Although programmers generally prefer to think of code in a higher-level way that matches the source code, upon programmer's request, or when not enough information has been recorded to confidently present things in a higher-level way, the full low-level details should be available.

6 Summary

I have tried to outline some common concerns and problems that come in writing a debugger in hopes that language designers keep these in mind when designing new programming languages. I have also given a brief sketch of how to structure writing a debugger that I have found to be modular, scalable and testable. Finally I try to make the case for using a high-level debugger both for debugging optimized code as is and for providing low-level access which may be useful for both programmers and language implementers.

7 Acknowledgments

Many thanks to Stuart Frankel, Yaroslav Halchenko, Barry Hayes, and Ken Zadeck for reviewing drafts of this paper. I have learned a about debuggers reading code, particular Kent Sibilev's ruby-debug and the Python debugger pdb.

The reviewers' comments were also very insightful and helpful.

References

[Bera] Rocky Bernstein. Google code pydbgr project. Available from `http://code.google.com/p/pydbgr`.

[Berb] Rocky Bernstein. kshdb github project. Available from `http://github.com/rocky/kshdb`.

[Berc] Rocky Bernstein. rbdbgr github project. Available from `http://github.com/rocky/rbdbgr`.

[Berd] Rocky Bernstein. zshdb github project. Available from http://github.com/rocky/zshdb.

[Ber09a] Rocky Bernstein. *Debugging with the Bash Debugger*, 4.0-0.3 edition, April 2009. Available from http://bashdb.sourceforge.net/bashdbOutline.html.

[Ber09b] Rocky Bernstein. *Python Library Reference for the Extended Python Debugger*, 0.10.3 edition, April 2009. Available from http://bashdb.sourceforge.net/pydb/pydb/lib/index.html.

[Ber09c] Rocky Bernstein. *Using the GNU Make debugger*, 3.81+dbg-0.2 edition, January 2009. Available from http://bashdb.sourceforge.net/remake/mdb.html.

[BS09] Rocky Bernstein and Kent Sibilev. *Debugging with ruby-debug*, 0.10.3 edition, January 2009. Available from http://bashdb.sourceforge.net/ruby-debug/ruby-debug.html.

[eaa] David Heinemeier Hansson et al. Ruby on rails documentation. Available from http://api.rubyonrails.org/.

[eab] Yehuda Katz et al. Merb. Available from http://www.merbivore.com/.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging opimized code with dynamic deoptimization. In *Proc. of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.

[III] Harry H. Porter III. Smalltalk: A white paper overview. Available from http://web.cecs.pdx.edu/~harry/musings/SmalltalkOverview.html#DebuggingSmalltalkCode.

[Sch07] Werner Schuster. Rubinius internals: Threading, objectspace, debugging. Available from http://www.infoq.com/news/2007/07/rubinius-internals-interview;jsessionid=FDAD5BDC3891A29515DE103D45C57CE2. The current "shotgun" interpreter uses a different approach though., July 2007.

[Sei] Justin Seitz. *Gray Hat Python*. no starch press.

[vRea00] Guido von Rossum et al. pdb – the python debugger. In *The Python Standard Library*. Python Software Foundation, 2.6 edition, June 2000. Available from http://docs.python.org/library/pdb.html.