Python Semantic Investigator: An Interactive Debugger with Reversible State

by

Jeremy Theard Wright

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....
Department of Electrical Engineering and Computer Science
May 28, 2019

Certified by .....
Adam Hartz
Lecturer
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Python Semantic Investigator: An Interactive Debugger with Reversible State

by

Jeremy Theard Wright

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis describes PSI (Python Semantic Investigator), a program designed to help students explore their code by allowing them to see the state of their program at any step along its execution. PSI enables them to move forwards or backwards freely along the timeline of their program. It also enables them to designate variable names or object IDs and jump back to the last time such a variable or object was modified. Doing so is intended to help novice students learn to debug more effectively.

Thesis Supervisor: Adam Hartz
Title: Lecturer

Acknowledgments

I would like to thank both of my advisors, thesis and academic, Adam Hartz and Katrina LaCurts. When I originally switched majors into EECS I could not have imagined that there would be people who would encourage me and help me along the way like they have.

Katrina supported my farfetched dream of making it into the masters of engineering program despite a history of poor academics in course 8, and without that support I wouldn't be writing this right now. Adam helped show me the world of CS education and sparked in me an interest in it. And I became more interested in it than I thought I could about anything.

I would also like to thank Jeremy Kaplan for always being there for me for my entire academic career. If he had not been in my life, I might not have joined the major in the first place. I would also like to thank Kade Phillips, Samantha Briasco-Stewart, and others who made my time in the masters worthwhile.

During my thesis writing, the opinions from Tyler Hamer and Daniel Gonzales were invaluable for their academic perspective. Thanks to them for listening to me bounce chapter ideas off them.

Lastly, I would like to thank my parents, Thomas and Lisa Wright for everything they've done for me that brought me to this point and beyond.

Contents

List of Figures

Chapter 1

Introduction

PSI (Python Semantic Investigator) is a program intended to help first-semester programming students evaluate their code by allowing them to see an effect and then move backwards to find its cause. This thesis explores the design process as well as the implementation details of PSI. PSI extends the concept of a debugger in the hopes of aiding novice programmers in evaluating their code from the angle of a detective by observing an effect and tracing it backwards to its cause.

1.1 Motivation

Introductory programming classes have historically been subject to high failure rates or drop-out rates [3][6]. L. J. Hk et al. found that the biggest difference between students who succeeded and those who did not was programming experience [2]. There are many aspects about programming that are hard to teach due to the fact that programming is an abstract, design-oriented skill. One of the most critical aspects, which PSI is designed to address, is debugging.

Debugging is a critical skill for any programmer. Professional software developers spend 70-80 percent of their time debugging, with the average bug taking several hours to find and fix [4][8]. Novice programming students seem to experience this as well. In this author's experience teaching two terms of a second-semester programming

course and one term of an introductory course, most interactions with students in office hours were debugging sessions.

But debugging is also hard. Even adept programmers commonly make false assumptions while debugging errors [4]. This difficulty, in this author's opinion, extends to novice students as well.

The difficulty arises in part because code can be deeply interconnected. Larger, internally complex programming structures often serve as the blocks from which a program is built, rather than individual lines of code. However, novice programmers often are limited to viewing their program "line by line" rather than as larger meaningful "chunks" [7][5]. This way of thinking can leak into the debugging and rewriting process as well. Unable to see the links that connect behavior of one part of a program to another, students may make "line by line" edits. They may change one line of code and then test the code again then chang one more and test again and so on.

PSI is intended to aid introductory students in the act of debugging by giving them a way to explore their code backwards, sidestepping the issue of hypothesizing the source of errors. It is designed to allow students to see the history of a variable or object, allowing them to identify the point in time where a variable or object took on an erroneous or unexpected value. It does so by extending the concept of the debugger. Traditional debuggers work by giving programmers control over how their code executes, allowing them to pause, observe, and skip around their program. The most commonly used features of debuggers tend to be the ability to halt execution at a chosen place in the program, often through commands like break, continue, step, and until. These features are the first discussed in the tutorials for multiple popular debuggers[1][2][3]. With these features, a developer can mark the place they expect to find a bug and have their program halt there, so they can confirm something went wrong. But in order to use these tools effectively, one must first have a notion of where to set a breakpoint. To do this, one must have a hypothesis as to the location of the bug. Even adept programmers' errors in debugging are often the results of false

[1]https://www.cs.cmu.edu/ gilpin/tutorial/
[2]https://docs.python.org/3/library/pdb.html
[3]https://docs.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour?view=vs-2019

assumptions made in the debugging process, rather than the debuggers they use [4]. Ultimately, a traditional debugger relies on the expertise of the programmer using it.

PSI tackles this issue by adding a program called an investigator on top of a more traditional debugger component. An investigator, as defined in this thesis, is a program that lets users observe and investigate the state of their code's execution over time after that execution has completed. PSI lets the user look at what happened at instruction n in the execution of their code and observe the state of the program in much the same way a debugger would if there had been a breakpoint there. This tool is intended to allow a student to see an error occur and then give them the ability to seek out the bug that caused it by traveling backwards along the timeline to see the history of a specified variable or object over the course of the execution. If the student starts with one variable whose value is wrong, they can have PSI search backwards for the last modification of that variable. They can then make a decision to trace a different variable that contributed to the error, having PSI jump back to the point where that variable was modified. By doing this, PSI does not require the student to have already location of the bug that caused the observed error before execution.

1.2 Previous Work

There are some tools that work on similar problems to PSI. Two closely related projects are the Whyline and Python Tutor which both add to the typical debugging process.

1.2.1 The Whyline

The Whyline is an Alice and Java debugger designed by Andrew Ko and Brad Myers in 2004 with the intent to reduce debugging time for programmers [4] working with graphical programs. In their paper, Ko and Myers broke down the components of debugging into six activities:

- Hypothesizing what runtime actions caused failure;

- Observing data about a program's runtime state;
- Restructuring data into different representations;
- Exploring restructured runtime data;
- Diagnosing what code caused faulty runtime actions;
- Repairing erroneous code to prevent such actions

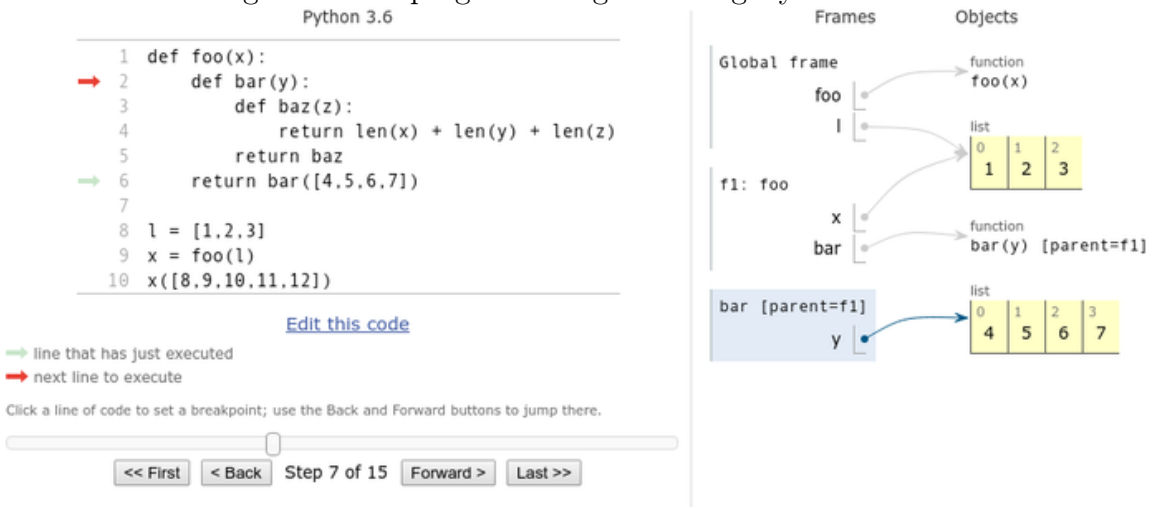
The Whyline was designed to support the hypothesizing component more heavily than other debuggers of the time by creating the concept of interrogative debugging. Upon failure, the Whyline enables programmers to ask questions about their what happend during execution time. Instead of debugging with typical techniques like breakpoints, they could pose questions about events that occured (or didn't occur). For example, a user could ask "Why didn't this event fire?" or "Why did B resize?" in order to test weak hypotheses instantly.

These questions split into two different types: "why did" and "why didn't", handling expected and unexpected runtime events respectively. They broke down the possible answers to these questions into threes:

- False propositions: The user's assumption was false (i.e. they asked "why didn't" about something that very much did).
- Invariants: The action always happens or always doesn't happen
- Data and control flow: Some series of actions lead to the result observed (i.e. some conditional expression evaluated to true)

By dividing possible queries and answers into a few discrete buckets, Ko and Myers were able to develop a computerized system for doing Interrogative Debugging on Alice programs. They had promising results. Comparing six identical debugging scenarios found that debugging time by a factor of 8 for programmers using Whyline (compared to a control set without Whyline).

Figure 1-1: A program being run using Python Tutor



1.2.2 Python Tutor

Another similar technology is Philip Guo's Python Tutor, a web application that lets people write code and run it step by step with detailed visualization of the state of the program at each point in time[4]. This makes the experience of using a debugger significantly more accessible for beginners and also allows users to see how their objects are scoped and arranged, which makes certain bugs, like aliasing, very observable.

Python Tutor's was intended as a teaching aid and visualization tool and, at the time of its publication, had scalability concerns, only being able to visualize several hundreds of execution steps for programs that use little memory [1].

1.3 Approach

Compared to the Whyline and Python Tutor, PSI approaches the problem of debugging in a different way. In the terms of the Ko and Myer's six stages of debugging, PSI is meant to focus on both the observing and diagnosing steps. PSI supports these steps by implementing an investigation mode. During execution of a student's code, PSI tracks the state of the program over time. Once the execution completes, either

[4]https://www.pythontutor.com

via success or an error, PSI transitions into investigator mode. It uses the state at the termination of the program as a starting point, letting students trace back the state of a chosen variable or object.

With PSI, if a student runs into, say, an `AssertionError` for some variable `foo`, they can tell PSI to search for the last place it was modified. If the student finds that one of the variables, `bar`, used to calculate the last value of `foo` had an incorrect or unexpected value, they could then search for the last place that `bar`'s value was calculated. If the error was that textttbar's initial value was wrong, then the last place PSI would stop is at the line of code initializing `bar`.

In contrast, with a typical debugger students would have to step through the code forwards, unable to rewind without starting over. In order to find that `bar`'s initial value was wrong, the student would have to choose a place to halt execution, then check the values of any variables they deem relevant. If they found that a variable had an incorrect value, they would have to restart execution and set an earlier breakpoint in the hopes of finding the moment that the variable took on that incorrect value. As a program becomes longer and more complicated, intelligently placing breakpoints becomes a more complex problem.

1.4 Outline

The remaining chapters of this thesis will be structure to explore the design, implementation, and future goals of PSI. Specifically:

Chapter 2 will discuss the design and implementation of PSI.

Chapter 3 will discuss future work, including extending PSI to other languages.

In addition, Appendix A contains the source code of PSIdb and Appendix B contains code snippets that were used to test the correctness of PSIdb in various situations.

Chapter 2

Design and Implementation of PSI

2.1 High Level Design

PSI's primary capability is the ability to "trace" a variable's history by allowing a student to specify a variable at any point in execution and find the last point that that variable was modified. In order to give the student more knowledge about the program at that point, it also reports the line of code that was being run to change their specified variable. In addition, it reports all of the variable values being used in that line. All of this is in service of helping a student debug code.

In order to do this, PSI has an efficient data structure for encapsulating most of the information used for these purposes called a state timeline. The state timeline exists to allow PSI to determine all of the "state" of a program at a specified step in its execution. State here refers to every frame's local bindings (i.e. variable-value pairs) and the parent-child relationships between them, while a step represents the execution of a single line of code.

PSI's work splits into two parts: generating the state timeline and using the state timeline to trace variables. This work is done by a debugger (referred to as PSIdb) and a new tool, which we will refer to as an investigator, respectively.

2.1.1 Debugger Design

PSIdb subclasses bdb, Python's extensible basic debugger. This gives it many of the typical debugger capabilities discussed in chapter 1.[1] Most importantly, this lets PSIdb take advantage of several hooks that lets us execute code between every executed line of the students code.

PSIdb uses these hooks to build up the state timeline, starting with small building blocks and building up to the completed timeline. The naive method of building the timeline would be to create a list where an element at index i is the complete state of the program at step i. However, this is an inefficient use of memory.

Instead, the state timeline is built out of deltas, which represent changes of state from step i to step $i + 1$. Since a program's state is comprised of multiple frames, each entry in the state timeline actually contains many smaller deltas, one for each frame. For more on the data structures used in building the state timeline, refer to 2.2

Using these deltas, PSI can start with a base state and apply or retract deltas to end up replicating the state at any point in the program's execution, which PSI can use to trace variable histories.

2.1.2 Investigator Design

Once the state timeline is built by PSIdb, the investigator takes over. The primary roles of the investigator are to provide information about a step to the user, to allow the user to determine a variable to trace, and to trace the designated variable.

Providing information on a step to the user is done using two things: the state timeline and the AST of the user's source code. The AST is used to recover the AST node for the line of code being executed during the step, from which the investigator should be able to determine the set of variables that are read or written during execution of the line in question. For a few examples, see Table 2.1.

[1]For more detail on bdb's capabilities, see: https://docs.python.org/3/library/bdb.html

Line of Code	Read Set	Write Set
x = a + b	a, b	x
x.append(c)	x, c	x
y = x.append(a)	x, a	x, y

Table 2.1: Read/write sets for a few lines of code

With the read/write set for a step, the investigator then uses the state timeline to look up the current values of all read variables and supplies this information to the user as well.

Each time PSI stops to present this information, it offers the user a choice to specify a variable to trace. This tells PSI to move backwards along the timeline until it encounters a step where the value of that variable is changed. In doing so, it lets the user follow the history of a variable

2.2 Data Structures

PSI uses a handful of data structures in order to construct a state timeline. The specific code implementing these can be found in appendix A.

2.2.1 RObject

`RObject` is a class that is designed to store a copy of an object while still maintaining the ID (found using Python's `id()` function) of the original. It has no methods and exists solely to wrap an object and an ID. When another copy needs to be made of an `RObject`, the resulting `RObject` has its own unique copy of the object but holds the same ID.

2.2.2 RFrame

In order to be able to determine what has changed before an after a given instruction, PSI cannot directly compare the frame before and after due to aliasing of mutable objects. In order to create copies of a frame and its bound objects, `RFrame`s were created.

An `RFrame` is intended to hold a copy of a frame's contents by taking in a frame (or `RFrame`) as part of its initialization. Like an `RObject`, an `RFrame` contains the ID of the original frame. However, its bindings are copies of those found in the original frame. A given `RFrame` also contains the ID of its parent frame. One of the key features of an `RFrame` is that it can also create copies of an `RFrame` just like it would for a frame. It exposes a similar interface to a frame, allowing one to check membership or retrieve bindings.

2.2.3 FrameDelta

A `FrameDelta` is meant to represent all the changes that occured to a frame before and after a given step. Given two `RFrame`s, one representing the frame before the instruction was executing and one representing the frame afterwards, a `FrameDelta` contains every modification between the two and categorizes them as either additions, deletions, or updates. An example of a `FrameDelta` and its usage can be seen in Figure 3.1.

Additions mark bindings that exist in the 'after' frame that don't exist in the 'before' frame. Each addition contains information on the variable name bound and the new value.

Deletions mark bindings that exist in the 'before' frame that don't exist in the 'after' frame. Each deletion contains information on the variable name not longer bound and the old value.

Updates mark bindings that exist in both frames, but have different values between them. Each update contains information on the variable name and the new and old values bound to it.

A `FrameDelta` also has methods for applying its changes to an `RFrame` in either direction.

2.2.4 FrameStep

A `FrameStep` is an object that contains all of the `FrameDelta`s for a given step. Each `FrameDelta` has a corresponding frame ID, so that the delta is applied to the proper frame. In addition, it also contains the line number of the line being executed.

Similarly to a `FrameDelta`, a `FrameStep` has methods that take a collection of `RFrame`s and applies all of the changes in either direction.

2.3 PSI Implementation

In order to accomplish all the goals described in Section 2.1, PSI needs to:

- Execute a program
- Monitor its execution
- Create a timeline of state
- Traverse that timeline

2.4 Executing and Monitoring a Program

2.4.1 Initializing PSI

When given a program, PSI reads the source code of the program to generate an Abstract Syntax Tree (AST) of the code and begin executing with PSIdb to generate the state timeline.

2.4.2 The PSI Debugger

PSIdb does most of the computationally expensive work of PSI. For every single step of the program execution (that is, for every line of code executed in the program), PSIdb creates a `FrameStep` and adds it to the state timeline. This process is described below.

Initializing PSIdb

PSIdb is a subclass of bdb, but it also needs some additional initialization to be effective. Since it has to be able to compare the state of the program before an after a given step of execution, it needs a way to keep track of the 'before' state. It also needs to store the state timeline as it is being built.

To do this, an instance of PSIdb contains an entry called `last_frame_dict`. `last_frame_dict` is a dictionary containing `RFrame`s representing the state of all relevant frames before a given step, keyed on frame IDs. This entry is initialized to be empty, since there is no relevant state before execution.

PSIdb instances also contain a list called `frame_steps`. This is the list that will represent the FrameTimeline as it is being constructed. Each entry in the list is the `FrameStep` for a given step of execution. This list starts out empty.

For reasons detailed in 2.4.2, PSIdb instances have entries to track the last line number and also to hold on to a stack of line numbers for nested function calls. These entries are called `last_line_no` and `call_stack_linenos` respectively.

Copying Objects

In order to be able to create a `FrameDelta`, PSIdb needs to compare the difference between the value in a binding before and after an instruction is executed. We make the `FrameDelta` by comparing values in the frame before and after the step occurs. In order to avoid any aliasing issues that may arise with mutable objects, PSIdb stores a snapshot of the 'before' frame by creating an `RFrame` copy of it.

Instantiating an `RFrame` takes the local bindings and parent pointer of a frame and produces an `RFrame` which contains a deep copy of every local binding. This means that the `RFrame`'s bindings don't point to the same objects as the frame it was spawned from, removing the aliasing concern. Whenever PSIdb needs create `FrameDeltas` it uses `RFrame`s instead of frames.

In order to deeply copy an object, `RFrame`s use a helper function which takes an object and returns an `RObject` containing a copy of it by recursively copying objects

(e.g. for a list, a new list would be made, with each element of the new list being a copy of the corresponding element in the old list).

Creating and Gathering Deltas

PSIdb uses `FrameSteps` so represent the total changes to state for a step. Before it can build a `FrameStep` it has to build all of the `FrameDeltas` that comprise it.

Given two `RFrames`, a 'before' frame and an 'after' frame PSIdb can form a `FrameDelta` describing all the differences between the two. To supply these frames, PSIdb holds onto the `RFrame` from the last step and uses that as the 'before' frame, using the `RFrame` from the current step as the 'after'.

Any step can change any frames in the program state (not just the local frame), so the `FrameStep` for a given step is comprised of a delta for each frame that changes during that step. Each `FrameStep` also stores the line number of the current step for use in the investigator. The `FrameStep` is then appended to `frame_steps` so it can be part of the state timeline.

Tracking Line Numbers

When users use PSI to debug their code, the investigator supplies information about the line being executed in a given step. As stated in 2.1.2, the investigator can provide the line number, the source at the line, and the read/write set for the current step. In order for PSI to be able to be able to store this information, PSIdb takes some extra care to track line numbers, as any change in state needs to be localizable to a particular line in the code being investigated.

Most of the time, PSIdb just stores the previous line number into the `FrameStep`. However, when a function is called, or returned from, it is less clear what should be considered the 'last line number'. To this end, PSIdb keeps a small call stack, pushing and popping line numbers as functions are called and returned from. These line numbers represent the lines from which each function wall called, which lets PSIdb make sure to credit the correct line with any changes found in the state timeline.

2.4.3 Investigating a state timeline

The process of executing the student's code via PSIdb creates a state timeline which contains information about every change of state that occured during the execution and when it happened. The investigator is designed to use that timeline in order to be able to find points of interest in the code.

Advancing and Reversing State

In order to move along the state timeline, the changes stored in `FrameDelta`s and `FrameStep`s need to be applied to a starting frame. To do this, `FrameStep`s have a pair of methods, `apply_f` and `apply_b` for moving forwards and backwards respectively. These methods take the three different types of changes – additions, deletions, and updates – and apply them in different ways.

`apply_f` adds bindings to the base frame for additions, removes bindings for deletions, and sets values to new values for updates. In contrast, `apply_b` does the reverse. It removes bindings in the base frame for additions, adds bindings for deletions, and sets values to old values for updates. Figure 2.1 demonstrates this for a simple frame and delta.

For advancing the overall state of the program, which involves multiple frames, the `FrameStep` takes a dictionary of frames (keyed on frame ID) and uses its own `apply_f` and `apply_b` methods. These methods take `FrameDelta`s and apply their application methods to the appropriate frames passed in. In doing so, all of the frames are advanced or regressed at once, representing the execution of a single line of code. Figure 2.2 demonstrates this for an example state and `FrameStep`.

Using this feature, a user can request the state of the code at any valid step in the program's execution by going from the state of the program at some other step (for example, at the beginning or the very end) and then applying a series of deltas to it. This can be done from any valid point in time to any other.

Figure 2-1: The `FrameDelta` for a given frame and its use

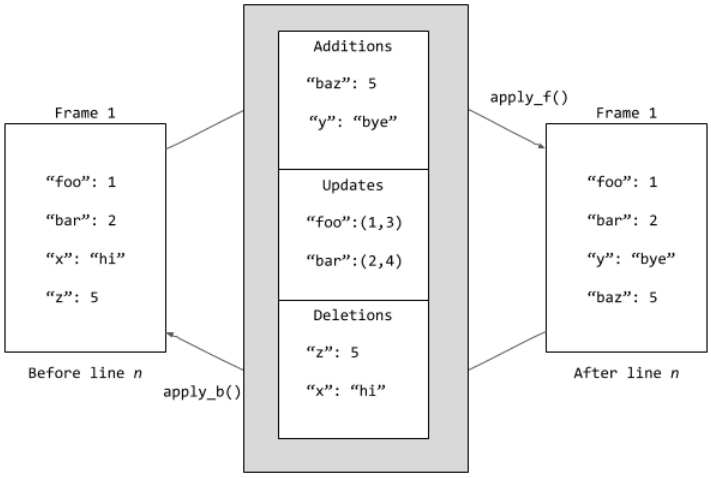


Figure 2-2: The `FrameStep` for a given frame and its use

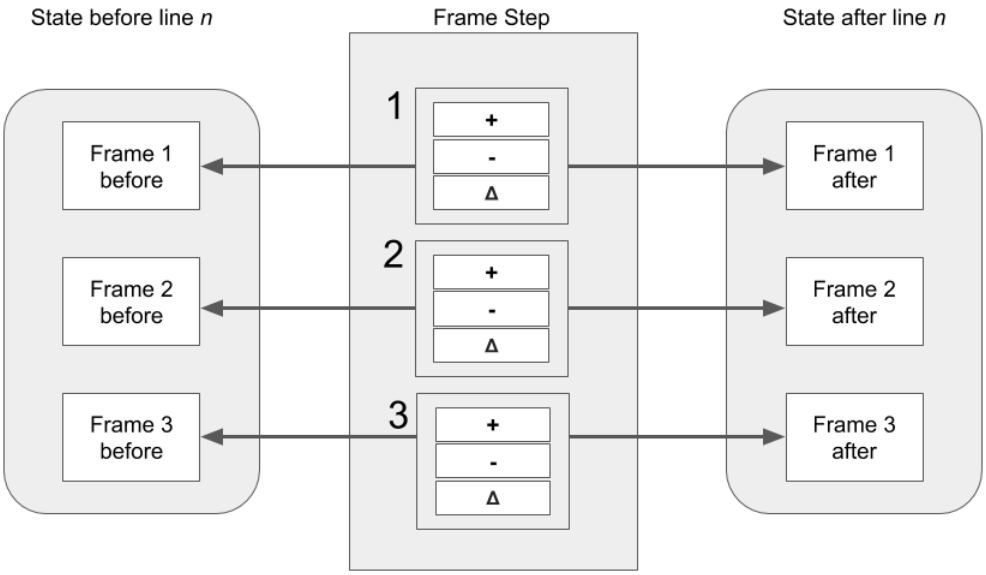
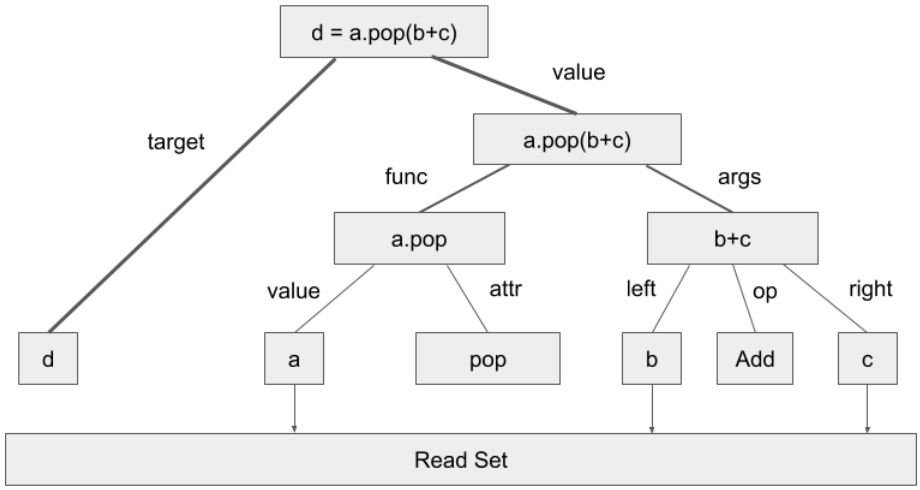


Figure 2-3: How the read set is derived from an AST node. The read set is used to determine which variables' values need to be seen by the user to understand what's going on in a given line of code. Note that target nodes are not added to the read set.



Getting Information From Line Numbers

Getting the state of the program at a particular step only gives some information. It's necessary for students to also know exactly what is executing during that step. PSI needs to be able to tell the student what line of code is being executed, what that line says, and the variables being used.

In order to do this, PSI stores the AST of the student's code. For any given line number (which was stored in the `FrameStep` for a given step), PSI walks through the AST to find the appropriate node. It can then produce the text of that line in the source as well as a summary of what variables read. For an example of an AST node and how the read set is derived from it, see Figure 2.3.

In addition, PSI can do environment lookups to use the variables found in the AST nodes to retrieve and report the values for them. It follows the same environment lookup rules as Python, searching in the most local frame first, and then its parent, and then the parent's parent, and so on. It uses the parent pointer that each `RFrame`

has to do this, letting PSI give users detailed information on where they are in the state timeline.

Tracking Objects and Variables

The primary feature of PSI is the ability to 'trace' a variable or object. By being able to explore the history of a variable or object, the user can trace back errors in their code to the bugs that caused them.

At any point in the investigation, when the user is presented information on the current line, the user can tell PSI to watch a variable or object. If PSI is watching an object, instead of stepping backwards a step at a time, it will keep rewinding the state timeline until it finds a step where the watched variable or object is changed in any way.

Once that condition is reached, PSI will break at that step, presenting the information about that step to the user and allowing them choose another variable/object to trace (or use the same one), allowing the user to explore the history of any variable or object.

Appendix A

PSIdb Source

```
import bdb

import ast
import asttokens

from inspect import currentframe, getframeinfo


class RObject():
    def __init__(self, obj):
        self.o = obj
        self.id = id(obj)

def copy_object(obj):
    primitive = (int, float, bool, str)
    if obj is None or callable(obj) or type(obj) in primitive:
        return RObject(obj)
    collection = (tuple, set, list)
    if type(obj) in collection:
        return RObject(type(obj)([copy_object(i) for i in obj]))
    if type(obj) == dict:
        d = {k: copy_object(obj[k]) for k in obj}
        return RObject(d)
```

```
    return RObject(obj)


class RFrame():
    def __init__(self, fid, f_locals, f_back):
        self.id = fid
        self.f_locals = copy_object(f_locals.copy()).o
        self.f_back = f_back


    def __getitem__(self, key):
        return self.f_locals[key]

    def __contains__(self, key):
        return key in self.f_locals

    @classmethod
    def from_frame(cls, f):
        fid = id(f)
        f_locals = copy_object(f.f_locals.copy()).o
        if isinstance(f, RFrame):
            f_back = f.f_back
        else:
            f_back = id(f.f_back)
        return RFrame(fid, f_locals, f_back)

    @classmethod
    def new(cls, fid, f_back=None):
        return RFrame(fid, {}, f_back)

    def copy(self):
        return RFrame.from_frame(self)

    def __str__(self):
        return str(self.f_locals)

```

```
60
61 class FrameDelta():
62     def __init__(self, adds, upds, dels):
63         self.additions = adds
64         self.updates = upds
65         self.deletions = dels
66
67
68     @classmethod
69     def from_frames(cls, pf, af):
70         additions = {}
71         updates = {}
72         deletions = {}
73
74         pf = pf.f_locals
75         af = af.f_locals
76
77         for a in af:
78             if a not in pf:
79                 additions[a] = af[a]
80                 continue
81             if pf[a] != af[a]:
82                 updates[a] = pf[a],af[a]
83         for p in pf:
84             if p not in af:
85                 deletions[p] = pf[p]
86
87         return FrameDelta(additions, updates, deletions)
88
89
90     def apply_f(self, frame):
91         nframe = RFrame.from_frame(frame)
92         for a in self.additions:
93             nframe.f_locals[a] = self.additions[a]
94         for u in self.updates:
95             nframe.f_locals[u] = self.updates[u][1]
```

```
96          for d in self.deletions:
97              if d in nframe.f_locals:
98                  del nframe.f_locals[d]
99
100         return nframe
101
102     def apply_b(self, frame):
103         nframe = RFrame.from_frame(frame)
104         for d in self.deletions:
105             nframe.f_locals[d] = self.deletions[d]
106         for u in self.updates:
107             nframe.f_locals[u] = self.updates[u][0]
108         for a in self.additions:
109             if a in nframe.f_locals:
110                 del nframe.f_locals[a]
111
112         return nframe
113
114
115     def __str__(self):
116         return "{'+': %s, '-': %s, 'delta': %s}" % (self.additions,
                self.deletions, self.updates)
117
118
119 class FrameStep():
120     def __init__(self, delta_dict, lineno):
121         self.deltas = delta_dict
122         self.lineno = lineno
123
124     def update(self, delta_dict):
125         self.deltas.update(delta_dict)
126
127     def __getitem__(self, key):
128         return self.deltas.get(key, FrameDelta({},{},{}))
129
130     def __setitem__(self, key, value):
```

```
131         self.deltas[key] = value
132
133     def apply_f(self, frames):
134         new_frames = {}
135         for k in frames:
136             new_frames[k] = self[k].apply_f(frames[k])
137         for k in self.deltas:
138             if k not in frames:
139                 new_frames[k] = self[k].apply_f(RFrame.new(k))
140         return new_frames
141
142     def apply_b(self, frames):
143         new_frames = {}
144         for k in frames:
145             new_frames[k] = self[k].apply_b(frames[k])
146         return new_frames
147
148     def __str__(self):
149         dstr = "{"
150         for k in self.deltas:
151             dstr += "%d: %s\n" % (k, str(self.deltas[k]))
152         return dstr + "}"
153
154
155 class rdb(bdb.Bdb):
156
157     def __init__(self):
158         bdb.Bdb.__init__(self)
159         self.frame_steps = []
160         self.last_frame_dict = {}
161         self.master_frame_dict = {}
162         self.last_line_no = 0
163         self.call_stack_linenos = []
164
165     def make_deltas_and_update(self, frame):
166         cur_id = id(frame)
```

```
167         frame_step = FrameStep({}, self.last_line_no)
168 
169         for f, lineno in self.get_stack(frame, None)[0]:
170             # Avoid getting the frame that contains rdb
171             if f.f_locals.get("self", None) == currentframe().f_locals["self"]:
172                 continue
173 
174             fid = id(f)
175             if fid == cur_id:
176                 self.last_line_no = lineno
177             frame_copy = RFrame.from_frame(f)
178             frame_step[fid] = FrameDelta.from_frames(self.last_frame_dict.get(fid,
                    RFrame.new(fid)), frame_copy)
179             self.last_frame_dict[fid] = frame_copy
180 
181         return cur_id, frame_step
182 
183 
184     def user_call(self, frame, args):
185         name = frame.f_code.co_name or "<unknown>"
186         fid = id(frame)
187         return_to_lineno = self.get_stack(frame, None)[0][-2][1]
188 
189         self.call_stack_linenos.append(return_to_lineno)
190         self.master_frame_dict[fid] = RFrame.from_frame(frame)
191         self.last_line_no = return_to_lineno
192 
193     def user_line(self, frame):
194         self.frame_steps.append(self.make_deltas_and_update(frame))
195 
196     def user_return(self, frame, val):
197         self.frame_steps.append(self.make_deltas_and_update(frame))
198         try:
199             self.last_line_no = self.call_stack_linenos.pop(-1)
200         except:
201             pass
```

Appendix B

PSIdb Test Code

Below are the code snippets used for testing PSIdb. Each code snippet was fed in and the resulting state timeline was checked line by line by applying each `FrameStep` in order and comparing to the actual frames at the same step in normal execution.

In the printouts for the test scripts, PSIdb outputs the line of code for each step, the id of the local frame, and the contents of all the relevant frames. The first result comes from initialization, and so has no corresponding line number or AST node.

Here are a few select test snippets and the output that PSIdb output

B.1 A Simple Script

```
1 a = 1
2 b = 2
3 c = a + b
```

Results:

```
None NoneType
curid 140663101745216 parent None
current frames {140663101745216: '{}'}

1 Assign a = 1
curid 140663101745216 parent None
```

```
current frames {140663101745216: "{'a': 1}"}

3 Assign b = 2
curid 140663101745216 parent None
current frames {140663101745216: "{'a': 1, 'b': 2}"}

5 Assign c = a + b
curid 140663101745216 parent None
current frames {140663101745216: "{'a': 1, 'b': 2, 'c': 3}"}
```

B.2 Aliasing

Test snippet:

```
k = 2
a = [[0]*k]*k

for i in range(k):
    a[i][0] = i

b = a[0][0]
```

Output:

```
None NoneType
curid 139784465094800 parent None
current frames {139784465094800: '{}'}

1 Assign k = 2
curid 139784465094800 parent None
current frames {139784465094800: "{'k': 2}"}

2 Assign a = [[0]*k]*k
curid 139784465094800 parent None
current frames {139784465094800: "{'k': 2, 'a': [[0, 0], [0, 0]]}"}

4 For for i in range(k):
    a[i][0] = i
curid 139784465094800 parent None
current frames {139784465094800: "{'k': 2, 'a': [[0, 0], [0, 0]], 'i': 0}"}

5 Assign a[i][0] = i
curid 139784465094800 parent None
current frames {139784465094800: "{'k': 2, 'a': [[0, 0], [0, 0]], 'i': 0}"}

4 For for i in range(k):
    a[i][0] = i
```

```
curid 139784465094800 parent None
current frames {139784465094800: "{'k': 2, 'a': [[0, 0], [0, 0]], 'i': 1}"}

5 Assign a[i][0] = i
curid 139784465094800 parent None
current frames {139784465094800: "{'k': 2, 'a': [[1, 0], [1, 0]], 'i': 1}"}

4 For for i in range(k):
    a[i][0] = i
curid 139784465094800 parent None
current frames {139784465094800: "{'k': 2, 'a': [[1, 0], [1, 0]], 'i': 1}"}

7 Assign b = a[0][0]
curid 139784465094800 parent None
current frames {139784465094800: "{'k': 2, 'a': [[1, 0], [1, 0]], 'i': 1, 'b':
    1}"}
```

B.3 Closure

```
1  def closure_test(x):
2      add3 = closure(3)
3      y = 5
4      z = y + add3(x)
5      return z
6
7  def closure(x):
8      def addx(n):
9          return x + n
10     return addx
11
12 closure_test(3)
```

Results:

```
None NoneType
curid 140440645273664 parent None
current frames {140440641923880: '{}', 140440641924344: '{}', 140440641942040:
    '{}'}

1 FunctionDef def closure_test(x):
    add3 = closure(3)
    y = 5
    z = y + add3(x)
    return z
curid 140440645273664 parent None
current frames {140440641923880: '{}', 140440641924344: '{}', 140440641942040:
    '{}'}

7 FunctionDef def closure(x):
    def addx(n):
        return x + n
    return addx
curid 140440645273664 parent None
```

```
current frames {140440641923880: '{}', 140440641924344: '{}', 140440641942040:
    '{}'}

12 Expr closure_test(3)
curid 140440641923880 parent 140440645273664
current frames {140440641923880: "{'x': 3}", 140440641924344: '{}',
    140440641942040: '{}'}

2 Assign add3 = closure(3)
curid 140440641924344 parent 140440641923880
current frames {140440641923880: "{'x': 3}", 140440641924344: "{'x': 3}",
    140440641942040: '{}'}

8 FunctionDef    def addx(n):
        return x + n
curid 140440641924344 parent 140440641923880
current frames {140440641923880: "{'x': 3}", 140440641924344: "{'x': 3, 'addx':
    <function closure.<locals>.addx at 0x7fbae29ca598>}", 140440641942040: '{}'}

10 Return return addx
curid 140440641924344 parent 140440641923880
current frames {140440641923880: "{'x': 3}", 140440641924344: "{'x': 3, 'addx':
    <function closure.<locals>.addx at 0x7fbae29ca598>}", 140440641942040: '{}'}

2 Assign add3 = closure(3)
curid 140440641923880 parent 140440645273664
current frames {140440641923880: "{'x': 3, 'add3': <function
    closure.<locals>.addx at 0x7fbae29ca598>}", 140440641924344: "{'x': 3,
    'addx': <function closure.<locals>.addx at 0x7fbae29ca598>}",
    140440641942040: '{}'}

3 Assign y = 5
curid 140440641923880 parent 140440645273664
current frames {140440641923880: "{'x': 3, 'add3': <function
    closure.<locals>.addx at 0x7fbae29ca598>, 'y': 5}", 140440641924344: "{'x':
```

```
    3, 'addx': <function closure.<locals>.addx at 0x7fbae29ca598>}",
    140440641942040: '{}'}

4 Assign z = y + add3(x)
curid 140440641942040 parent 140440641923880
current frames {140440641923880: "{'x': 3, 'add3': <function
    closure.<locals>.addx at 0x7fbae29ca598>, 'y': 5}", 140440641924344: "{'x':
    3, 'addx': <function closure.<locals>.addx at 0x7fbae29ca598>}",
    140440641942040: "{'n': 3, 'x': 3}"}

9 Return return x + n
curid 140440641942040 parent 140440641923880
current frames {140440641923880: "{'x': 3, 'add3': <function
    closure.<locals>.addx at 0x7fbae29ca598>, 'y': 5}", 140440641924344: "{'x':
    3, 'addx': <function closure.<locals>.addx at 0x7fbae29ca598>}",
    140440641942040: "{'n': 3, 'x': 3}"}

4 Assign z = y + add3(x)
curid 140440641923880 parent 140440645273664
current frames {140440641923880: "{'x': 3, 'add3': <function
    closure.<locals>.addx at 0x7fbae29ca598>, 'y': 5, 'z': 11}", 140440641924344:
    "{'x': 3, 'addx': <function closure.<locals>.addx at 0x7fbae29ca598>}",
    140440641942040: "{'n': 3, 'x': 3}"}

5 Return return z
curid 140440641923880 parent 140440645273664
current frames {140440641923880: "{'x': 3, 'add3': <function
    closure.<locals>.addx at 0x7fbae29ca598>, 'y': 5, 'z': 11}", 140440641924344:
    "{'x': 3, 'addx': <function closure.<locals>.addx at 0x7fbae29ca598>}",
    140440641942040: "{'n': 3, 'x': 3}"}

12 Expr closure_test(3)
curid 140440645273664 parent None
current frames {140440641923880: "{'x': 3, 'add3': <function
    closure.<locals>.addx at 0x7fbae29ca598>, 'y': 5, 'z': 11}", 140440641924344:
```

```
"{'x': 3, 'addx': <function closure.<locals>.addx at 0x7fbae29ca598>}",
140440641942040: "{'n': 3, 'x': 3}"}
```

Bibliography

[1] Philip J. Guo. Online python tutor: Embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[2] L. J. Höök and A. Eckerdal. On the bimodality in an introductory programming course: An analysis of student performance factors. In *2015 International Conference on Learning and Teaching in Computing and Engineering*, pages 79–86, April 2015.

[3] Päivi Kinnunen and Lauri Malmi. Why students drop out cs1 course? In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 97–108, New York, NY, USA, 2006. ACM.

[4] Andrew J. Ko and Brad A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[5] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. volume 37, pages 14–18, 09 2005.

[6] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '01, pages 125–180, New York, NY, USA, 2001. ACM.

[7] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003.

[8] G. Tassey. The economic impacts of inadequate infrastructure for software testing, 2002.