Debugging Probabilistic Programs: Lessons from Debugging Research

Henry Lieberman
CSAIL
MIT
Cambridge, MA 02139
lieber@media.mit.edu

Yen-Ling Kuo
CSAIL
MIT
Cambridge, MA 02139
ylkuo@mit.edu

Valeria Staneva
CSAIL
MIT
Cambridge, MA 02139
valeria.staneva@gmail.com

Probabilistic programming, like much of machine learning, has a dirty little secret: probabilistic programs are difficult to debug. Improvements in debugging technology hold the most promise for improving the practicality of machine learning. We need to democratize the technology by making it more accessible to beginners.

Probabilistic programming is an elegant formulation of machine learning tasks and methods. It enables bringing concepts from conventional programming into the machine learning workflow. But it also brings unique challenges, such as thinking about values that vary over time.

It isn't the fault of the Probabilistic Programming movement itself. Interactive development environments for probabilistic programming are usually based on the underlying implementation language, which share similar kinds of problems. Shockingly, most of today's IDEs simply offer the same set of debugging tools that appeared in the earliest programming languages: breakpoints, stack and data inspection, function trace, variable monitoring, and (if you're very, very lucky), a reasonably accurate single stepper.

Debugging of probabilistic programs is still in its infancy. Here, we present some lessons from research in debugging of general-purpose programs, that we think hold promise for future probabilistic programming environments.

First, the idea of a *reversible stepper*. Single-steppers are common in programming environments, but most have fatal flaws that prevent them from becoming practical tools. Dynamic control over the level of detail presented is a crucial facility.

More specific to probabilistic programming, is the fact that the concept of a scalar "value", which appears in most programming languages, isn't stable. It may take on different numerical constant values over time. We propose to replace that notion with the idea of a *probabilistic value* (or *probval*), and propose a visualization technique to represent it in the programming environment.

Instrumentation and Localization

There are two kinds of cognitive tasks in debugging: *instrumentation* and *localization*. Instrumentation is the process of finding out what the behavior of a given static description is. Examples are trace, breakpoints, and print statements.

Localization is the process of isolating which static description is "responsible" for some given undesirable behavior without prior knowledge of where it might be. Among traditional programming tools, a *stepper* is potentially the most effective localization tool, since it interactively imitates the action of the interpreter, and the program can, in theory, be stepped until the error is found.

We've got you coming and going

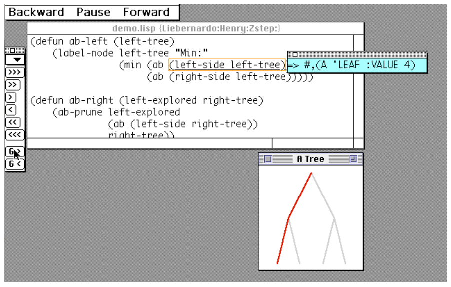
Programming is the art of constructing a static description (program) that results in some dynamic behavior. Debugging is understand the *correspondence* between code and behavior, in both directions. It is the process of breaking down this correspondence into smaller parts when the larger correspondence does not meet expectations.

Traditional steppers have a fatal interface flaw: they have poor control over the level of detail shown. They offer the user a choice of whether or not to see internal details, before each evaluation of the current expression. If the user says yes, they risk wading through much irrelevant information. If they attempt to speed up the process by skipping over (presumably working) subparts, they risk missing the precise location of the bug. This leaves the user in the same dilemma as the instrumentation tools -- they must have a reasonable hypothesis about where the bug might be before they can effectively use the debugging tools!

The solution is to provide a *reversible control structure*. It keeps a complete, incrementally generated history of the entire workflow. The user can confidently choose to temporarily ignore the details of a particular expression, secure in the knowledge that if the expression later proves to be relevant, the stepper can be backed up.

The downside is that histories cost time and space. But our slogan is: *there's nothing slower than a program that doesn't work yet!* It's also important to keep the history of the developer's investigation of the program. Because machine learning programs may be tried again and again with varying datasets, subsets of a common dataset, and varying parameters, much can be learned from selectively returning to previous states in the investigation. This even opens up the possibility that a machine learning algorithm can use the histories of investigation themselves as data, learning from the developer's actions!

A crucial problem in designing an interface for program debugging is maintaining the visual context. If the item and its visual context are spatially or temporally separated, a new cognitive task is created for the user -- matching up the item with its context.



```
(defun ab-left (left-tree)
    (label-node left-tree "Min:"
              (min (ab (left-side left-tree)) => #,(A 'LEAF :VALUE 4)
                   (ab (right-side left-tree)))))

(defun ab-right (left-explored right-tree)
    (ab-prune left-explored
             (ab (left-side right-tree))
             right-tree))
```

The ZStep reversible stepper

Some of these ideas explained were first implemented in our ZStep reversible stepper for Lisp. Code display, data display, and graphical output display representations were kept in sync at all times [Lieberman and Fry 97].

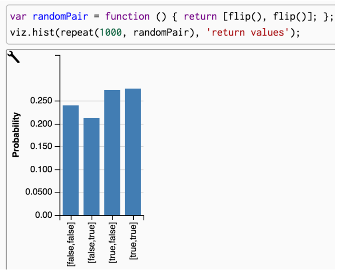
Probabilistic Values

One of the key problems is that deterministic programming languages operate on single values, whereas probabilistic programming languages operate stochastically on sets. Conventional practice is to generate multiple values for observation. Look over the shoulder of a practitioner, and you are often likely to see lines and lines of similar looking printouts fly by at warp speed. More careful work generates graphs of these values, with conventional line, scatter plots, and histograms.

We introduce the idea of a *probabilistic value* (*probval*), which can take on different values at different times. This is like the idea of *stochastic variable* or *random variable* in other systems, but we want to push these visualization and dependency tracking facilities into the values themselves, and better integrate them into the programming environment.

Languages like WebPPL have a set of visualizations which are applicable to conventional datatypes like vectors, but these visualization procedures must be explicitly invoked. The book, *Probabilistic Models of Cognition* [Goodman and Tenenbaum 16] presents an online tutorial that includes a routine that displays probabilistic values, e.g. as histograms. But visualization should be automatically invoked, in a READ-EVAL-INSPECT loop, replacing the traditional READ-EVAL-PRINT loop. It can automatically invoke visualizations for observing that value from different points of view, including conventional histograms and graphs. It should have interactive operations for tracing dependencies forward and backward through the workflow, available in one click from any context where that value appears.

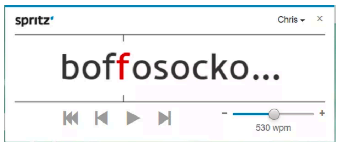
We need operations that allow you to replace one underlying set with another. We need to be able to create various subsets of the set, filtered by various predicates. We need to go forward and backward in the stream, and look at the stream at various levels of detail. It should be possible to easily substitute one sampling procedure for another in the context of any probabilistic value. User interaction with sampling procedure selection and control is also useful in other contexts, such as sampling in a hypothesis space.



```
var randomPair = function () { return [flip(), flip()]; };
viz.hist(repeat(1000, randomPair), 'return values');
```

Tutorial with explicit histogram visualization

One visualization that we think is promising is the use of *Rapid Serial Visual Presentation*, or *RSVP* [Potter 14]. This allows visual observation of rapid streams of data in a very small and bounded screen space. RSVP, invented by Molly Potter at MIT, was developed to read streams of text in a small space at very fast speeds. Skilled readers of RSVP can read text at faster rates than conventional left-to-right reading of static text, because it does not require the saccadic movement of the eye. Even if the stream goes by so rapidly that you can't fully pay attention to every item, visual memory allows you to get an overall impression quickly, exactly the goal of sampling.



An RSVP reader. Eye focus is on the central red letter.

Practical use of an RSVP interface requires dynamic control over the speed of presentation. RSVP can be adjusted to an appropriate speed for an individual reader or individual context. An adaptation effect makes it easier to tolerate higher speeds as you "get used to it", even over a single session. Reversibility, and forward and backward skip operations, are useful to allow for momentary lapses of attention. There is even the potential to automatically adapt speed in accordance with input from an eye tracker.

Seymour Papert, a pioneer in computers and education, used to say, “bugs are your friends”. Far from being something shameful, bugs can be an impetus for learning more about your program. And for learning more about yourself. Debugging shouldn’t be painful. It should help you get to know your friends better.

REFERENCES

[Goodman and Tenenbaum 16] N. D. Goodman and J. B. Tenenbaum (2016). Probabilistic Models of Cognition (2nd ed.). Retrieved 2017-11-16 from https://probmods.org/

[Lieberman & Fry 97] Henry Lieberman and Christopher Fry, ZStep: A Reversible, Animated, Source Code Stepper, in *Software Visualization: Programming as a Multimedia Experience*, John Stasko, John Domingue, Marc Brown, and Blaine Price, eds., MIT Press, Cambridge, MA, 1997.

[Potter 14] Potter, M.C., Wyble, B., Hagmann, C.E., & McCourt, E.S. (2014). Detecting meaning in RSVP at 13 ms per picture. Attention, Perception, and Psychophysics. [2] F.N.M Surname, Article Title, https://www.acm.org/publications/proceedings-template.