Usable Live Programming

Sean McDirmid
Microsoft Research Asia
Beijing China
smcdirm@microsoft.com

Abstract

Programming today involves code editing mixed with bouts of debugging to get feedback on code execution. For programming to be more fluid, editing and debugging should occur concurrently as **live programming**. This paper describes how live execution feedback can be woven into the editor by making places in program execution, not just code, navigable so that evaluation results can be **probed** directly within the code editor. A pane aside the editor also **traces** execution with entries that are similarly navigable, enabling quick problem diagnosis. Both probes and traces are refreshed continuously during editing, and are easily configured based on debugging needs. We demonstrate the usefulness of this live programming experience with a prototype.

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

Keywords Live Programming; Debugging

1. Introduction

Programming burdens our minds as we must imagine how code will execute while editing it. To correct errors in our mental simulation, we must stop editing and instead "debug" code to get feedback on how it really executes. Hancock [10] compares this to archery: aiming an arrow (editing code) involves mentally simulating a physical system, while shooting (debugging) provides discrete feedback for the next shot. Good archers learn how to get their aim right the first time—but why not just use a water hose instead?

Live programming makes programming easier by re-executing a program continuously during editing. Back to Hancock's analogy, consider hitting a target with a stream of water: we simply keep correcting our aim until the target is hit where, unlike archery, we receive continuous feedback on where we are shooting. With live programming, we just edit an executing program until its output looks right.

But what does this output look like? In many live programming systems [5, 17, 18, 22, 24, 26, 27], programmers only observe updated visual output while editing; many relevant intermediate results are left unseen. And what about programs like compilers whose original outputs are not even visual? Original program output can be augmented with traces that overview program execution, e.g. we can reason about compiler execution by augmenting it with print statements; but we still cannot see code execute in detail as we could in a step-based debugger.

This paper recasts live programming as editing and **debugging** code at the same time. In our system, places in program execution, not just code, are **navigable** in the editor; i.e. not only can we navigate from a method call to its definition, but we can also *probe* expression values for the call directly in the editor. So we can avoid tediously navigating through execution to find places to probe, a pane aside the editor overviews program execution through *tracing*; trace elements are editor-navigable to execution places where a problem, identified in a trace, can be diagnosed through probing. Probes and traces are refreshed continuously during editing, and are easily configured in code based on debugging needs: traces are formed from print statements, while probes are enabled by annotations. We have implemented this design in a prototype of our YinYang language.

This paper continues with Section 2 describing a base live programming model that is enhanced in Section 3 with probing and tracing. Section 4 discusses our experience with YinYang (videos available); Section 5 presents interesting technical details. YinYang currently does not handle interactive programs that involve time-ordered events: Section 6 describes the challenges of supporting such programs. Related work is presented in Section 7 and Section 8 concludes.

2. Live Programming

In the context of programming, many IDEs provide continuous and/or responsive feedback on the lexical, syntactic, and type safety of our code. However, many "live" visual languages such as VIVA [27], Forms/3 [6], Morphic [15],

Onward! 2013, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509578.2509585

and PureData [22] go beyond this by providing live feedback about how program execute as visual code is edited. For example, a VVVV [18] program is edited by connecting together various data-flow patches; by connecting a rotate patch to a rectangle patch, we can see rendered rectangles rotate in real time as we change the rotation angle.

Beyond live visual languages, two techniques are often used to get timely execution feedback in textual procedural languages. First, REPLs (read-eval-print loop) enable the progressive input of top-level statements, displaying results as statements are written. However, REPLs are intended for quick experimentation and feedback is not available for method definitions. Second, as pioneered in LISP [16] and Smalltalk [8], many IDEs support *fix-and-continue* where code can be changed while the program is executing. However, fix-and-continue only affects future executions; e.g. if we change w.color = red to w.color = blue, the color of old w objects bound will remain red, only future w objects will be blue. In contrast, live visual languages execute changed code as if it was always in its post edit state.

However, fix-and-continue can be augmented as follows to create a more useful live programming experience. Suppose we write a procedure called Render that is called continuously to re-render the display:

```
while (true)
    ClearScreen();
    Render();
def Render() = draw('hello world')
```

Assuming Render executes quickly, editing its code while this program is running will provide responsive execution feedback: as we change draw statements in Render, new results are quickly displayed. This form of basic live programming is supported by and considered useful in Khan Academy's learn programming system [24]. However, this approach to live programming has a few problems. First, no state is preserved between invocations of Render, which is non-interactive. This paper does not solve problems related to interactive programs, which are discussed more in Section 6. Next, a naive implementation will re-execute Render completely on every display frame, which is slow if Render is large; we explore latency issues in Section 5. Finally, the output produced by Render is often not sufficient in helping us write code, which is the focus of this paper.

As described so far, live programming does not replace contemporary debugging techniques. The live output from executing a program continuously is just a picture ("hello world") or, perhaps if Render were like a compiler, obscure binary output. To actually debug code, we need to know more about how the program executes beyond its final output. Additionally, no help is given in mapping output back to code: we must manually figure out how observed outputs are related to the code we are editing. In contrast, contemporary debugging techniques enable code execution understanding through step-based debugging, tracing with print statements, and so on. The next section describes how to merge the responsiveness and continuity of live feedback with the understanding gained from debugging.

3. Making Live Programming Useful

How a program executes is often not obvious from its external behavior, which is true in any complex system; e.g. we cannot know how a car is broken by driving it. On this point, Victor [30] critiques live programming as follows:

> *We see code on the left and a result on the right, but it's the steps in between which matter most. The computer traces a path through the code, looping around loops and calling into functions, updating variables and incrementally building up the output. We see none of this.*

Our live programming system, dubbed YinYang, aims to fix this problem by focusing on debugging, which is missing from many existing live programming systems like [5, 17, 24, 26]. Traditional debugging techniques, like breaking and stepping with a debugger, are not compatible with continuous execution: instruction stepping interferes with continuous live feedback, while a separate debugger interface to navigate and inspect execution necessarily disrupts programmer focus on code editing. YinYang combines editing and debugging not just in time but also in space where updated debug results are conveniently visible while editing. The rest of this section adapts two debugging techniques into YinYang: *probing* that presents in detail how code executes, and *tracing* that overviews execution with print statements.

Probing

A traditional source debugger for a procedural language enables us to observe code execution at a fine-grained level by breaking at or stepping through instructions while inspecting expression values. Unfortunately, using a debugger with live programming would require a focus away from editing, and what do breaking and stepping even mean when the program is continuously re-executing?

To augment live programming with debugging, YinYang *projects* program execution information directly onto code in the editor so that we can keep editing code. Consider a call to a binary search (BS) method that is entered via a REPL into a program as a top-level statement:

```
> e = [1,4,7,8,10,13]
> BS(e, 7, 0, e.L) ⟲
2
>
```

Immediate execution feedback is given because the execution context of the statement is fixed at its point of input in the REPL. To edit and debug at the same time in YinYang, an expression's value has a presence in the editor near the expression so it can be easily observed while editing. To avoid being overwhelmed with useless feedback, programmers have control over what expressions are "probed:" expressions can be preceded with @ operators in the editor to

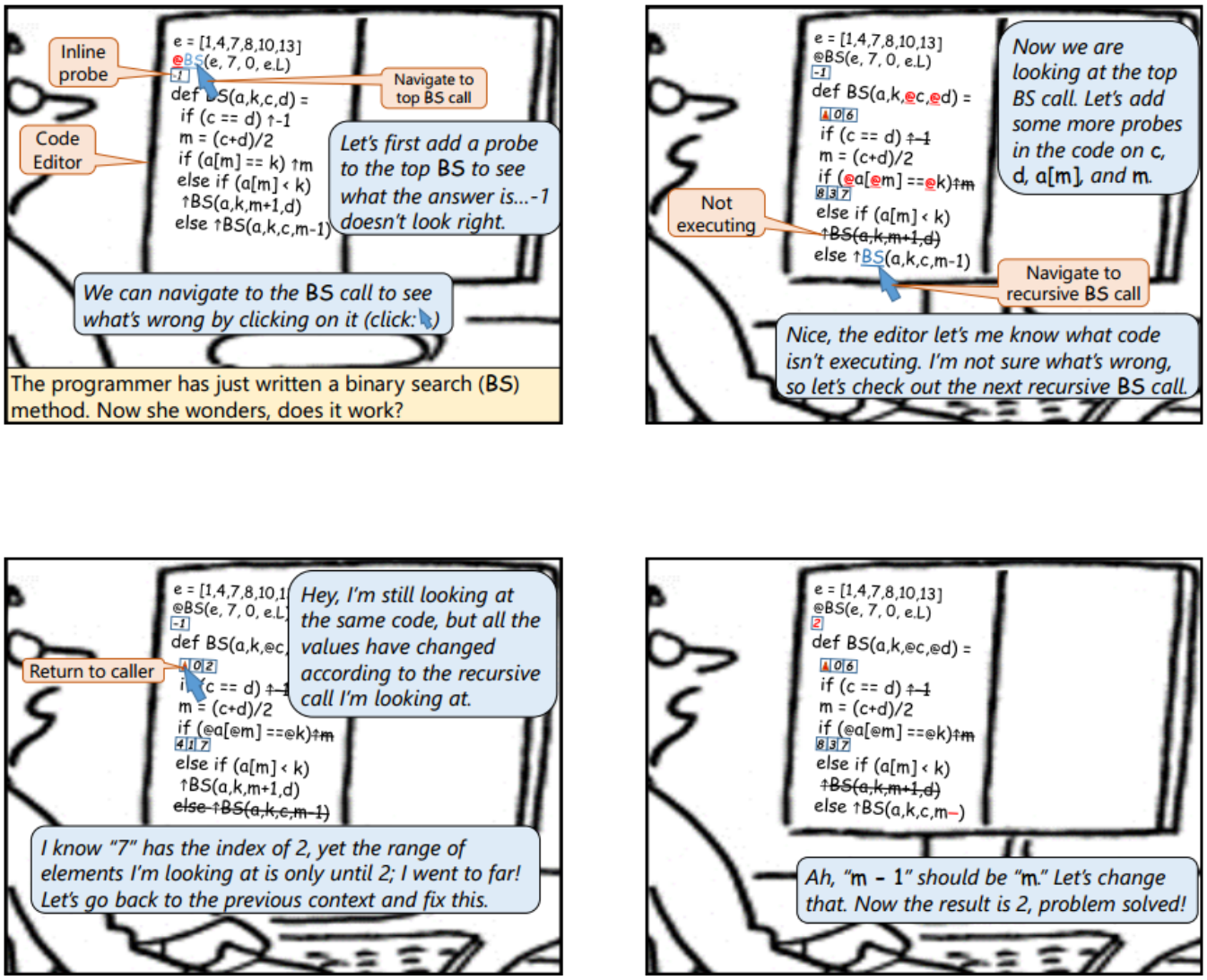

Figure 1. An illustration of live programming with probing alone; read left-to-right; unboxed red/underlined code indicates an edit.

create *probes* that visualize their values; e.g.:

```
@BS(e, 7, 0, @e.L)
2 6
```

Here the probes on both the BS call and e's length appear beneath the line that they are located on; note probes are unlabeled and ordered left to right according to @ order. Probes are continuously maintained as code is edited; e.g. prepending a 0 element to list e causes these probes to immediately go from 2 6 to 3 7.

Many code blocks have ambiguous executions contexts; e.g. methods have multiple callers while loops have multiple iterations. YinYang solves this problem by projecting an execution context according to how we navigate to the code. For a method definition, execution context can be specified with respect to a method call that is top-level or has a projected execution context. Consider the illustration in Figure 1: the top BS method call, which exists in a top-level execution context, can be clicked to navigate to the BS method definition and project the execution context of the call, resulting in the top-right panel. When execution context is projected onto a block of code, whatever code in the block that is not executing in the corresponding control flow is struck out; e.g. at the bottom of the top-right panel, the first recursive BS call is struck out since a[m] (8) is not less than 7.

Because a method definition can have only one execution context projected on it at a time, when navigating to the last BS call of the top-right panel, the new execution context results in different probe values for the bottom-left panel of Figure 1. Method definitions with projected execution context are also annotated with a up-pointing triangle to navigate back to their caller (both in code and execution context), which the programmer does in Figure 1's bottom-left panel to go back to the first BS call, resulting in the bottom-right panel. The projection of execution context onto

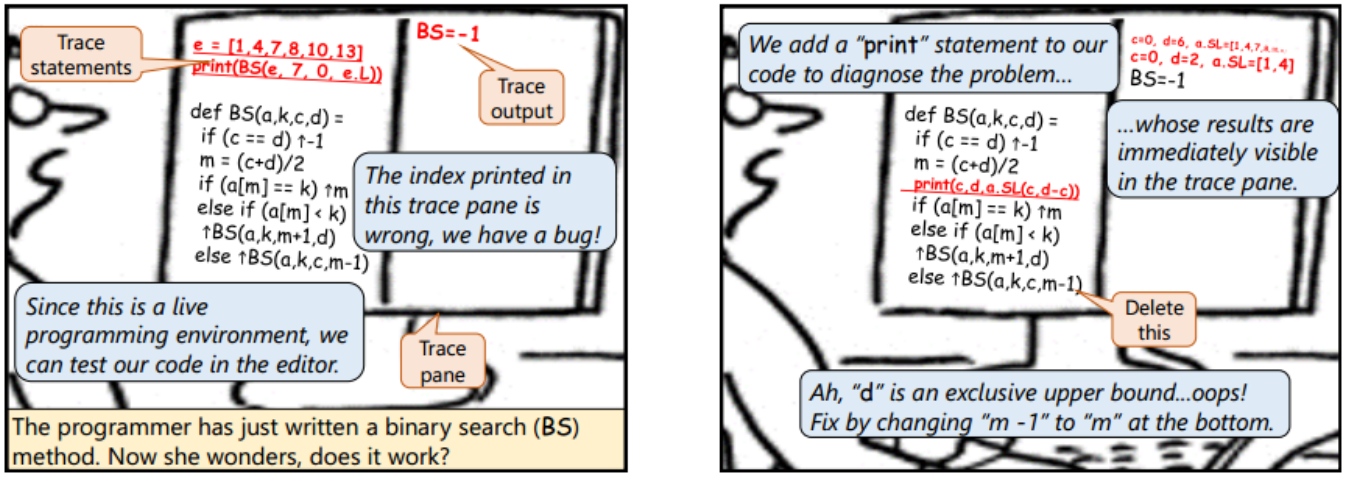

Figure 2. An illustration of live programming with tracing alone.

a block of code does not prevent that code from being edited, and how these edits affect the projected execution context are immediately visible in any probes displayed at the time. For example, as the programmer fixes the bug in the BS method definition, the probe on the top BS call immediately updates to 7's correct index in the list (2). In contrast to method definitions, loops begin the execution context of their first iteration, with arrows to project next or previous iteration execution contexts.

Because program code executes continuously, stepping through code is not meaningful and so the presentation of execution information is spatial. The value of a stateful expression depends exactly on where it occurs and is probed in the execution; the value of an expression at different points in the program's execution can only be seen by replicating and probing the expression in multiple locations; e.g.

```
@x; q(); @x
```

0	42

x has the value of 0 before q is called the value of 42 after as q changes the value of x as a side effect. Likewise, code must be written to probe expressions that are not already in the code; in this way, watch and local variable debugger panes are entirely supplanted by the code editor.

Tracing

> *The most effective debugging tool is careful thought, coupled with judiciously placed print statements—Kernighan [12].*

Finding problems in a program by probing its execution can be like searching for a needle in a haystack. Program execution in YinYang can also be abstracted into a *trace* that overviews what the program is doing through print-like trace statements added to code according to debugging needs. Unlike a program's original output meant for program clients, traces are specifically meant as a programmer-oriented UI that locate problems or map how code is executing. Trace entries immediately appears in a trace pane to the right of the editor as trace statements are added to the executing program; Figure 2 illustrates tracing in use.

Neither probing nor tracing alone provide a complete debugging experience. Probing might find a problem after navigating through much of a program's execution, which is tediously impractical. Tracing, in contrast, identifies problems quickly but hardly tells us why they occurred. We then combine probing and tracing using the insight that the print statements that create trace entries are executed somewhere in the program—this execution place can then be used as an anchor for code execution navigation and probing! Clicking on each line of trace output in Figure 2 will cause navigation in the editor to the code execution place that created that line of output; projecting the correct execution context on BS accordingly. Tracing identifies problems in an overview of the program's execution and then acts as a map to navigate the editor to various code execution locations at which point probing helps to diagnose problems in detail.

Figure 3 demonstrates the synergy between probing and tracing in debugging the binary search method of Figures 1 and 2; note that a longer e list is being used in this example. We start out by adding trace statements and discover that the 28 element is not found in the list. We then navigate directly to the last trace output created in the BS method before the search fails (top-left to top-right panels), where we are able to probe to find out that c and b on the last BS call will both be bound to 16. We correct the problem by realizing that b is exclusive, not inclusive, and so 1 should not be subtracted from m. After we fix the problem (bottom-left panel), the execution context projected onto the code we are editing is gone as it no longer exists in the updated program execution. We navigate from the last BS trace entry to check our results (bottom-left to bottom-right panels), re-projecting an execution context on the BS method's definition.

Probing and tracing together provide almost all of the debugging capabilities that programmers are used to. For

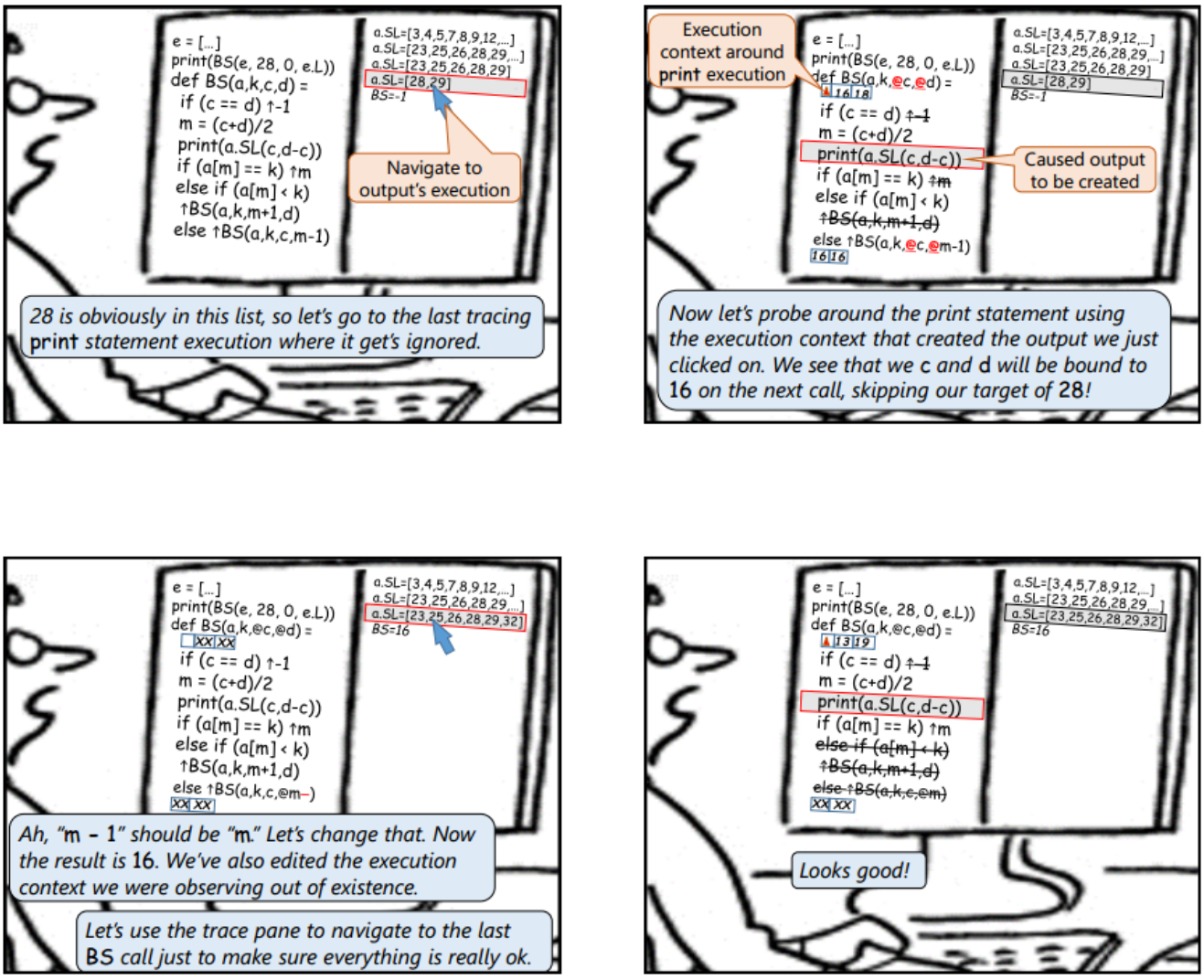

Figure 3. An illustration of live programming with probing and tracing.

example, trace elements can be created instead to make navigable execution locations where we would otherwise use break points, which YinYang lacks. Debugging is then a more spatial, rather than temporal, experience that fluidly corresponds to code editing.

4. Experience

Although Figures 1, 2, and 3 illustrate how YinYang is used, the fluidity of this live programming experience is more dramatic in video form:

sqrt: ***http://www.youtube.com/watch?v=01Xyoh-G6DE***

fib: ***http://www.youtube.com/watch?v=UVf1SJnNb6E***

Note that the language being edited in the video is a custom statically-typed procedural language; however its features are unimportant for the purposes of this paper.

Execution in this prototype is re-done on every keystroke. Although responsive, the feedback might be wrong or missing when code has syntax or semantic errors, or it might be volatile while typing in a construct. Perhaps programmers should sometimes have control over when refresh occurs.

We experimented with multiple designs for probing before settling on the one in YinYang. An earlier prototype projected code onto execution: method calls could be expanded as a projected method definition. However, this design was found to be confusing as projections became deeply nested, while having the same method definition projected in multiple places was disorienting. Our final design reverses this: rather than project code onto execution, execution is projected onto non-duplicated code. While we believe our chosen design is less confusing, more work needs to be done to indicate what execution context is being projected.

YinYang avoids overwhelming us with too much feedback by giving us direct control over probing and tracing through configuration by code—we are not distracted with expression values that we do not care about or trace entries that are not useful. The fact that code is used for debugging

makes its semantics very clear and flexible; e.g. conditions can be used to filter out trace elements while expressions can be written to probe values where (and when) needed. On the other hand, configuration-in-code requires us to litter our code with trace statements and probe annotations, which appears messy while read-only third-party code seems impossible to debug at all. Additionally, debug code might need to access members that should normally be encapsulated; e.g. if we want to probe a private member of an object. However, although debug code appears collocated with normal code as a programmer convenience, it could be stored separately and could be exempted from encapsulation.

Tracing as described so far is essentially printf debugging, which is useful in understanding how a problem was "reached" even if it is somewhat primitive. The synergy between tracing and probing that we identified in the Section 3 clarifies the unique role of tracing during debugging. Although we only use print-style tracing in this paper, traces could also be expandable trees, zoomable graphs, and so on; e.g. graph-style traces are commonly used in compiler development to visualize a large number of relationships. Navigability for such traces still applies; e.g. clicking on a graph node would navigate the code editor to the code and execution context that created the node.

5. Technology

As a live programming language, YinYang depends heavily on incremental computation in both its compiler implementation and programming model. An incremental compiler is necessary to re-parse and type check the code quickly, keeping certain landmarks in the code cohesive between edits to ensure experience continuity. An incremental programming model is necessary so that the program's themselves can re-execute quickly. We solve both problems with an incremental computation framework, called Glitch, that transparently repairs program executions after a change in input...or code.

A program execution in Glitch is decomposed into a tree of *nodes* that can be re-executed independently on a code or input change. Node decompositions are specified by programmers based on their understanding of the program's run-time modularity; e.g. execution nodes for a compiler can be chosen to correspond to nodes of the syntax tree (AST) for the code being compiled. Nodes have direct dataflow dependencies with their parents for arguments and their children for return values, where they are re-executed if these change; e.g. if the code lexer stream that a parse node is called on changes, the node is re-executed to repair the parse tree accordingly. Likewise, nodes are re-executed when the shared state they read, which is traced dynamically, changes; e.g. if a symbol table entry read by a parse node changes.

Unlike in other incremental computation frameworks (e.g. [1]), Glitch code can do imperative operations. These operations are logged on node execution and are automatically undone if they are not re-performed on a node re-execution. Beyond being undoable, imperative operations in Glitch must also be **commutative** as nodes can re-execute in any order in response to arbitrary changes. Only a few operations are easily commutative, e.g. set insertion and aggregation; however, we augment assignment-like operations such as dictionary entry insertions and cell bindings with single assignment semantics, vastly increasing Glitch's expressiveness at the expense that some operations can now fail. In the end, we were able to write YinYang's compiler in a fairly mundane programming style using C# and Glitch.

Beyond Glitch, another key to YinYang's implementation is the use of *execution addresses* that are navigable and coherent across program edits. These addresses are formed from tokens and are added to as methods are called, e.g. token `t` in method `f` called via the token `f'` has the address `f'.t`; or on each loop iteration where a non-token key, such as indices, is used to identify an iteration. Execution addresses are used to record values based on probe annotations; non-probed expression values are not recorded while adding a new probe will always trigger re-execution so its value can be recorded. Probe execution addresses are then reproducible in the code editor so probe values can be rendered inline below their annotations.

The execution address of the call used to create a trace element enables navigation from the element to the execution location in the code editor as described in Section 3. Clicking on a trace element will (a) navigate to the lexical location of the creating call in the editor and (b) use the execution address to set the execution context of any loop iterations or method calls that contain it; the address contains all information needed to set this context. The editor will then re-render the code, using these execution contexts to render any probes in the editor, while producing execution address for further navigation using the structure of the code. Execution addresses are also keys to preserve state between edits so trace elements do not need to be recreated after each edit, avoiding flickering and clobbering any trace pane UI state.

As mentioned, Glitch also forms the basis of YinYang's programming model with the same restrictions that primitive imperative operations be undoable and commutative. YinYang currently gives programmers control of Glitch's incrementalization via a `node` method annotation; the idea being that incremental computation has a cost, and so it cannot be performed transparently, which is a strategy used in other work [4]. However, programmers must deeply understand the granularity and modularity characteristics of the computations being performed by the program. The programmer may lack such understanding, and, in the case of many programs, such understanding may be non-existent.

In any case, change can sometimes have a huge impact on program re-execution time. If significant latency (~50ms) cannot be avoided and is not managed well, live programming would actually reduce programmer productivity as programmers depend on but wait for slow feedback. To mit-

igate this problem, we should explore techniques that guess, reuse outdated results, or can otherwise give useful, if not completely correct execution feedback; e.g. consider a low fidelity "draft mode" used in video editing tools.

6. Dealing with Time

The work presented in this paper does not consider interactive programs involving time-ordered events. Because they involve state that changes over time, such programs do not yet fit into the YinYang model. There are two different styles of how live programming could handle interactivity:

- Edits change program execution in **real-time** leaving results of its past code execution untouched; and
- Events are **recorded** so they can be replayed on code edits, ensuring that the past and present are consistent.

The real-time style of live programming is used by most existing live programming systems that handle interactivity, e.g. [10, 17, 26], with the qualification that some code in the program is executed continuously and so supports editing with live feedback. State changes performed by imperative code are final—code edits cannot change the past and time travel is impossible—while only the "present" state of the program is observed by the programmer. Work on TouchDevelop [5] formalizes this style of live programming by separating a "view," whose code is amenable to live programming, from a "model," whose initialization and mutation code is not; we can conceptualize this as:

```
m = initial;
while (true)
    ClearScreen();
    m = RenderAndHandle(m);
```

Here m is the program's current state while RenderAndHandle is called continuously to both render this state, as a model, and handle any events that occur (implicitly) by mutating the model. This style of live programming has many problems. First, edited code can depend on state that is not allocated during initialization. Second, the state of the program, and hence its behavior, can be inconsistent with the program's edited code; i.e. the edited program would behave differently if re-executed from scratch. Finally, we have no way of debugging state transitions that are unobservable since they happen instantaneously. Still, real-time live programming is useful when changing the past is not sensical; e.g. programming a live performance [26] or physical robot [10].

The recorded style of live-programming turns an interactive program into a batch one by recording input events and rendering multiple frames of output; i.e.

```
while (true)
    frames.Clear();
    m = initial;
    foreach (e in events)
        (f, m) = RenderAndHandle(m, e);
        frames += f;
```

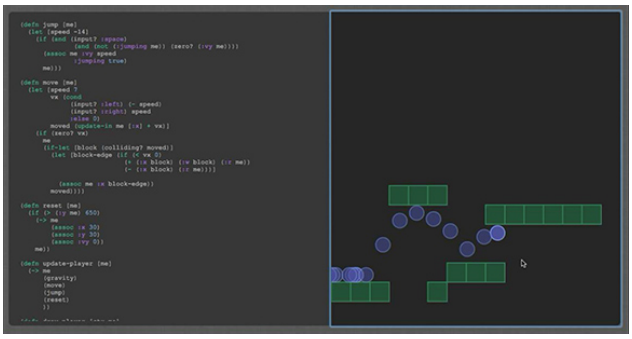

Figure 4. Stroboscopic debugging of a bouncing ball; courtesy of Chris Granger [9].

This program then appears to be amenable to Section 3's probing and tracing. However, consider a physics simulation of one thousand frames (events), where traces are overwhelming with lots of poorly organized or rarely changing trace entries that are re-printed on every frame. How can a trace usably include time? Possible answers include:

- Trace elements for only one time point are shown at a time, and then we **time travel** via a slider;
- Trace elements are grouped by time into visual elements such as **timelines** (Figure 5 bottom left); and/or
- Trace elements are interposed over time using **stroboscopic** visual elements as shown in Figure 4.

Victor explores time travel and strobing in his work [29, 30] while our previous work [11] targeting camera-based programming explores time travel and timelines (Figure 5). Such time-based tracing needs more work before live programming of interactive programs can be very usable.

Latency is also a large problem for recorded-style live programming—live programming a one minute physics simulation involves 3600 simulation frames! Incremental computation is crucial in this case, but if every frame depends on the last, how to do this is not obvious. Spatial relationships in a simulation can be used to incrementalize some computations; e.g. if a change does not interfere with one area, then processing for that area need not be redone. Further work is needed to generalize such techniques.

7. Related Work

The concept of live programming was introduced by Hancock [10] in his Flogo I and II languages, although the role of liveness in programming was previously explored for visual languages [6, 27] as well as in Self's Morphic [15]. As mentioned previously, many systems [18, 22, 24, 26] since have not addressed debugging. SuperGlue [17] is one such system that focuses on the novelty of interacting with a program while editing its code; it was also based on a reactive data-flow programming model that could not express many basic computations, while YinYang is fully procedural.

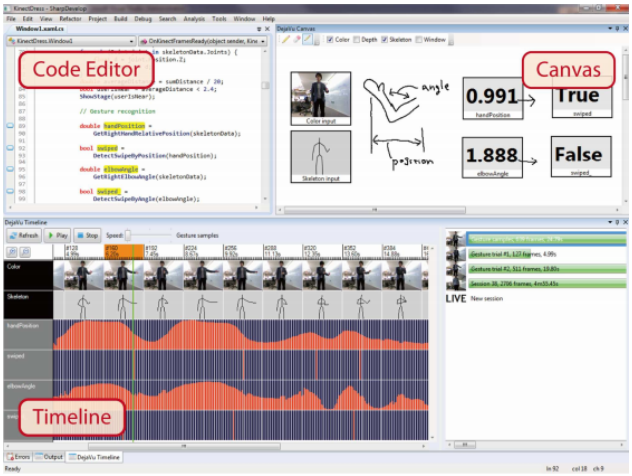

Figure 5. The DejaVu [11] interface for developing interactive camera-based programs; the timeline shows values per-frame.

TouchDevelop [5] likewise does not support debugging but introduces a notion of "navigability" between live rendered UI output and code; YinYang's notion of navigability is directly influenced by this and additionally projects execution context on code navigated to. TouchDevelop also leverages navigability to support bi-directional editing between UI elements and code; e.g. changing the color of a UI element will update code to set its color property, and vice versa. YinYang does not currently support rendered UIs in traces, but otherwise could also support this feature.

Our work is heavily influenced by Victor's recent well-received demos [29] that focus on making programming more friendly; specifically what Victor labels as "making flow visible (and) tangible" in [30] are realized in YinYang. However, whereas Victor leverages instruction stepping in his design, YinYang renders execution spatially, simplifying visualization but also obscuring discrete computation steps. YinYang differs in the same way from previous debuggers that support "time travel" [2, 13, 14, 23, 28]. Our use of replay and light instrumentation is also more pragmatic than approaches that record all execution details [21], or approaches that record all object states [20].

The aforementioned Flogo II [10] supports "live text" that interleaves execution feedback into code being edited, which is similar to YinYang's probing. Unlike YinYang, Flogo code is projected onto expandable program execution. Code Bubbles [3] also can project code onto execution by copying code and debugger watch panes into multiple bubbles on a large canvas; YinYang is much more compact in comparison. Perera [19] shows how execution can be visualized in a more compact way by using slicing to elide irrelevant execution details according to a specified computation. In contrast, we have complete control over probing and tracing but only through manual effort, and the addition of similar slicing in YinYang could help us diagnose problems more quickly. Ressia et al. [25] shifts a live debugging focus away from the execution stack to focus more on individual objects; though YinYang still focuses on execution locations, similar functionality can be obtained via navigable tracing.

YinYang is closely related to Edwards' work [7] on unifying code editing and debugging into an "example enlightened editor" where code is seen side-by-side with a visualization of an example's continuously-refreshed execution. Selection in visualized execution causes navigation in the code where probing is then possible via mouse hover. In contrast, YinYang substitutes visualized execution for programmer-configured tracing and probing, providing programmers with feedback that is more focused and visible.

8. Conclusion

Live programming is emerging as the next big step in programming environments that will finally allow us to move beyond our Smalltalk-era IDEs into a more programmer-friendly future. However, existing live programming experiences are still not very useful—they dazzle us with live feedback but that feedback does not really help us write code! We recasted live programming as a fusion of editing and debugging so that we can leverage live feedback in debugging our code. Many challenges remain in the realms of latency, UI, and interactivity, but this experience is compelling enough that future work will be worth the effort.

Acknowledgments

Thanks to Jonathan Edwards, Jules Jacob, and the anonymous reviewers for providing feedback on this work.

References

[1] U. A. Acar. Self-adjusting computation: (an overview). In *Proc. of Partial Evaluation and Program Manipulation (PEPM)*, pages 1–6, 2009.

[2] S. P. Booth and S. B. Jones. Walk backwards to happiness - debugging by time travel. In *Proc. of Automated and Algorithmic Debugging*, pages 171–183, 1997.

[3] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proc. of ICSE*, pages 455–464, 2010.

[4] S. Burckhardt, D. Leijen, C. Sadowski, J. Yi, and T. Ball. Two for the price of one: a model for parallel and incremental computation. In *Proc. of OOPSLA*, pages 427–444, 2011.

[5] S. Burckhardt, M. Fahndrich, P. de Halleux, J. Kato, S. McDirmid, M. Moskal, and N. Tillmann. It's alive! Continuous feedback in UI programming. In *Proc. of PLDI*, 2013. To appear.

[6] M. M. Burnett, J. W. Atwood Jr, and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *Proc. of the IEEE Symposium on Visual Languages*, pages 126–134, 1998.

[7] J. Edwards. Example centric programming. In *Proc. of OOPSLA*, pages 84–91, Dec. 2004.

[8] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[9] C. Granger. Light Table - a reactive work surface for programming. `www.kickstarter.com/projects/ibdknox/-light-table`, 2012.

[10] C. M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[11] J. Kato, S. McDirmid, and X. Cao. Dejavu: integrated support for developing interactive camera-based programs. In *Proc. of UIST*, pages 189–196, 2012.

[12] B. W. Kernighan. *UNIX for Beginners*. Bell Telephone Laboratories, Inc., 1979.

[13] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4), Dec. 2003.

[14] A. Lienhard, T. GÃőrba, and O. Nierstrasz. Practical object-oriented back-in-time debugging. In *Proc. of ECOOP*, pages 592–615, 2008.

[15] J. H. Maloney and R. B. Smith. Directness and liveness in the Morphic user interface construction environment. In *Proc. of UIST*, pages 21–28, nov 1995.

[16] J. McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[17] S. McDirmid. Living it up with a live programming language. In *Proc. of OOPSLA Onward!*, pages 623–638, October 2007.

[18] Meso group. VVVV - a multipurpose toolkit. `www.vvv.org`, 2009.

[19] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. In *Proc. of ICFP*, pages 365–376, 2012.

[20] F. Pluquet, S. Langerman, and R. Wuyts. Executing code in the past: efficient in-memory object graph versioning. In *Proc. of OOPSLA*, pages 391–408, 2009.

[21] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proc. of OOPSLA*, pages 535–552, 2007.

[22] M. Puckette. Pure Data: another integrated computer music environment. In *Proc. of International Computer Music Conference*, pages 37–41, 1996.

[23] S. P. Reiss. Graphical program development with pecan program development systems. In *Proc. of Practical Software Development Environments*, pages 30–41, 1984.

[24] J. Resig. Redefining the introduction to computer science. `ejohn.org/blog/introducing-khan-cs`, 2012.

[25] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proc. of ICSE*, pages 485–495, 2012.

[26] A. Sorensen and H. Gardner. Programming with time: cyber-physical programming with Impromptu. In *Proc. of SPLASH Onward!*, pages 822–834, 2010.

[27] S. L. Tanimoto. Viva: A visual language for image processing. *Journal on Visual Language Computing*, 1(2):127–139, 1990.

[28] D. Ungar, H. Lieberman, and C. Fry. Debugging and the experience of immediacy. *CACM*, 40(4):38–43, Apr. 1997.

[29] B. Victor. Inventing on principle. Invited talk at the Canadian University Software Engineering Conference (CUSEC), Jan. 2012.

[30] B. Victor. Learnable programming. `worrydream.com/LearnableProgramming`, Sept. 2012.