

Dynamic Contextualization with Live Probes

ANONYMOUS AUTHOR(S)

Programmers routinely step away from their code to consult consoles and debuggers, paying a cognitive cost to connect emitted runtime data to source code locations, and *vice versa*. We present Live Probes, a system for selectively surfacing intermediate runtime samples alongside code in the Hazel live programming environment. Each sample is an observed value together with its dynamic context, including its environment and call stack. The programmer and system together control which terms are probed (manual or automatic), which samples are shown (via a *dynamic focus*, a path in the call tree that orients every sample around one execution of a function), and how each sample is shown (using dynamic folding or leveraging graphical views that programmers can define for their own types). Feedback updates continuously while code is typed, and incomplete edits do not disturb the feedback around them. We report a 15-participant user study comparing probes to print debugging and examining the benefits and drawbacks of always-on live values while writing and debugging code. The study finds that programmers value each major affordance, while exposing a generalizable design tension between moving the dynamic focus to supply values while writing code and keeping it still while debugging.

1 Introduction

Programming is decision-making, and decisions require context. When programmers read, write, and debug code, however, dynamic context that might inform their decisions (such as the values that flow through a function, the call stack at those values, and control flow decisions) is often distant from the code itself, emitted in an overwhelming form, or unavailable because the code is being actively edited. When context is not readily accessible in these ways, it incurs costs due to limited human attention [32], working memory [13], and time.

Consider *print debugging*, a common approach to dynamic contextualization and a frequent first resort [5]. This requires marking up the syntax, fiddling with string conversion, running the program, and inspecting an often overwhelming log in a separate console, with no direct connection to the code being debugged. *Stepping debuggers* provide richer interfaces for inspecting dynamic state, and some are integrated directly into code editors [31, 51], but these provide dynamic context only for a single moment at a time, requiring tedious stepping to develop context over time. Both provide dynamic context only after an explicit run of a complete program. Figure 1 previews the alternative this paper develops: runtime values shown beside the code, updated on every edit.

1.1 Design Problems in Dynamic Contextualization

The two aforementioned approaches have long dominated practice, but there are a number of other approaches described in the literature to address their limitations. Let us summarize some key design problems in dynamic contextualization and mention notable pieces of prior work that have progressed toward addressing them (with a more in-depth discussion of related work in Section 6).

The indirection problem. When dynamic context is accessible only via tools external to the code editor, e.g. consoles, sidebars, or log files, the programmer must expend time and attention to correlate the information in these tools with source code locations [32].

A long line of research has advocated for inline displays to address this problem. We are most directly inspired by the system of *probes* introduced in YinYang [47] and developed further in Babylonian-style programming [68], where individual program terms are equipped, inline in the code editor, with sampled concrete values. Emacs also supports inline display of values provided by a stepping debugger session [12]. Other approaches include Seymour [34] and Projection Boxes [43], which display tables of values taken by variables (but not general terms) beside the code.

A Dynamic context while writing

```

let grove_name(path: String): String = ③ "/moon/fern" ... ①
  let pieces = string_split("/", path) in ③ ["", "moon", "fern"] ... ②
  nth(pieces, ①) ③ if ① == 0 then x else nth(xs, n - 1) ... ③
in
test? grove_name("/moon/fern") == "moon" end;
test? grove_name("/night/moss") == "night" end;
test? grove_name("/sun/rose") == "sun" end

```

These three tests supply the calls that produce these samples.

① A probe underlines a term and shows one of its ③ samples at the end of the line.

② The split list sample helps choose an index by revealing an empty first element.

③ nth runs until the hole stops its if condition, whose test against 0 shows it counts from zero.

B Detecting an argument-order error

```

let grove_name(path: String): String = ③ "/moon/fern" "/night/moss" "/sun/rose"
  let pieces = string_split(path, "/" ) in ✓ ③ ["/"] ["/"] ["/"]

```

Double-clicking a sample switches to *Many* mode. All three paths yield ["/"], suggesting reversed `string_split` arguments. Both arguments are strings, so type hints alone cannot resolve their order.

C Checking an index choice

```

nth(pieces, 0) ③ "" ...
in
testx grove_name("/moon/fern") == "moon"

```

→

```

nth(pieces, 1) ③ "moon" ...
in
test✓ grove_name("/moon/fern") == "moon"

```

Changing 0 to 1 provides immediate feedback, from the probe and test, even before the closing `)` is entered.

Fig. 1. A user writes `grove_name`, which extracts the first directory name from an absolute path. Inline samples reveal argument-order and indexing errors while the code is still incomplete.

The term selection problem. Which terms should be selected for sampling? Print debugging requires the programmer to manually select terms, somewhat tediously. Stepping debuggers allow the programmer to explore variable values at the current position in the call stack, though more complex terms are trickier to inspect. Omniscient (also called time-travel or reversible) debuggers collect *all* dynamic information in the form of a trace, but this can incur a high runtime cost and still requires choosing a single moment in time [4, 44, 66]. Recent work on automatic printing [39] inserts print statements after every statement in a user-selected line range, and perhaps surprisingly even this coarse heuristic was found to be useful in practice, suggesting value in more automated and heuristic approaches to term selection.

The coherence problem. When many values have been sampled, which should be shown together? Print logs interleave output ordered by collection time, so it is difficult to see all values sampled from a single term across time, i.e. these related samples are not spatially coherent. Values sampled from different expressions in a function body may come from different calls, i.e. they may not be temporally coherent [71]. In a print log, with a print at the start of a recursive function and another at its end, each call's two lines are separated by the output of every call beneath it. Stepping debuggers show only one moment in time, so they do not have the temporal coherence problem,

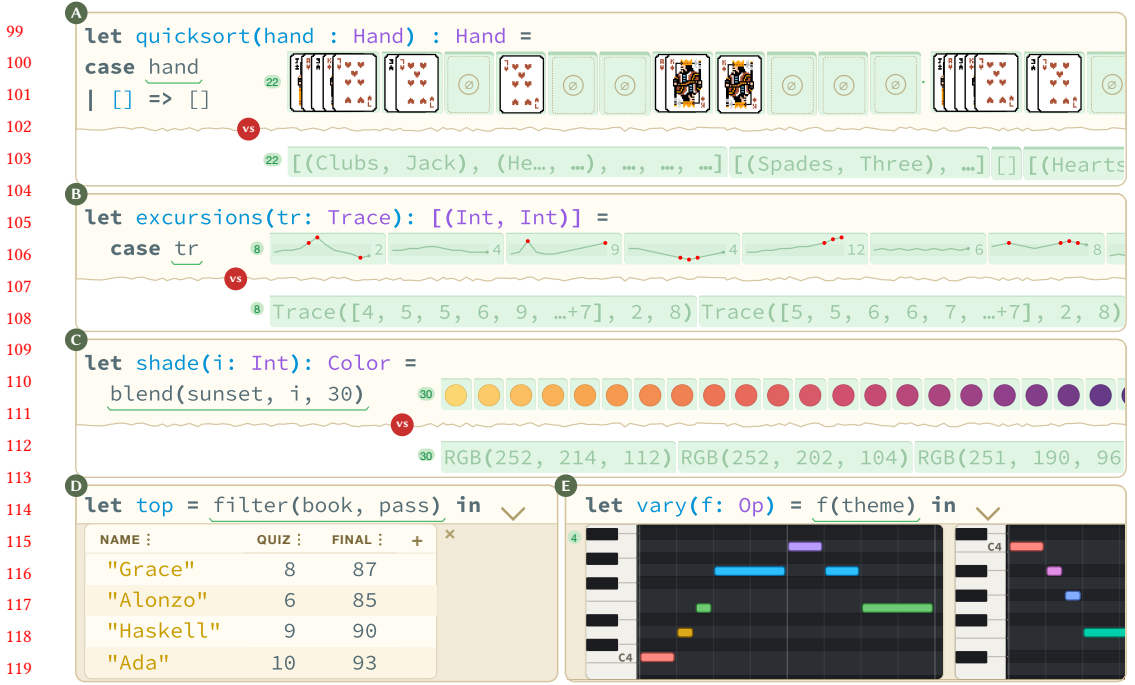


Fig. 2. Probe samples drawn by views. In **A** to **C**, each sample appears twice: drawn by its view above the tear, and as text below it. **A** Recursive calls to quicksort sorting a hand of playing cards (a crossed circle is an empty hand). **B** Temperature traces of eight shipments, as sparklines: the band is the allowed range, red dots indicate readings outside it, and the number is the last reading. **C** The colors of a sunset ramp from `blend`, as swatches. **D** Students' grades, as a table in the probe's drawer. **E** Piano rolls in the drawer, one per transformation `f`: the theme, then its inversion. The cards and the table are built-in views; the sparklines, swatches and piano rolls are user-defined.

but they also do not sample values changing over time, leaving the programmer to develop this understanding by manually stepping through breakpoints.

The multiplicity problem. When many samples for a single term have been collected, how many should be shown at once? In some cases, the programmer may want to see many values to get a sense of the shape of the data flowing into a probed term, while in other cases, the programmer may only want to focus on a single value.

The value detail problem. How should values be displayed? Many research tools are presented with examples consisting of small values, e.g. numbers or small lists. In practice, however, programmers often work with larger values, which may not fit on the screen and some of which may be difficult to comprehend even when shown fully. Some tools provide domain-specific GUIs for displaying values of particular types [9, 56].

The gap problem. All of the above presuppose a program complete enough to run, but most editor states are not [62]. Error-correcting parsers partially close the gap, and Hazel's semantics for incomplete programs closes it more systematically [60, 61]. Where code is incomplete, some systems let the programmer supply the values it cannot compute [17, 42]. A related question, for any live system, is *continuity across states*: how feedback should change as the program does, which Perera

treats as a semantic concern [63]. Our design addresses a related practical requirement, *feedback stability* (Kanon’s steadiness, after Hancock’s ‘steady frame’ [23, 58]): preserving the association between code and its dynamic context, without unnecessary disruption during incomplete edits. Small edits should have small effects unless they must have large ones: a one-character edit that completes a variable name may legitimately change feedback throughout the program, but beginning a new definition should not disturb feedback for the surrounding code merely because the editor reinterprets that code mid-edit.

1.2 Dynamic Contextualization with Live Probes

This paper introduces *Live Probes*, the first system that minimizes all six of these problems in a single coherent design, and unlike many of the prior systems that have provided design inspiration, evaluates this design with an in-depth human subjects study involving both code writing and debugging tasks. Live Probes is implemented in the Hazel system, described further in Section 2.

Central to the system are *probes*, which collect runtime *samples* from any probed expression or pattern and display them directly adjacent to the probed code, minimizing the *indirection problem*. Figure 1 demonstrates the system through the construction of `grove_name`, which extracts the first directory name from an absolute path. In A, three included unit tests supply argument values even though the function body is incomplete; these appear beside the parameter (callout 1). The `string_split` sample (callout 2) reveals the easy-to-miss fact that the split list begins with an empty string, informing the choice of index on the next line. Each probe collects samples from all three test calls but initially displays only one. All three probes display their sample from the same call, which addresses the *coherence problem*. This coherence is maintained by a *dynamic focus*, a path in the call tree that distinguishes one execution (Figure 3), detailed further below.

Callout 3 shows that feedback continues even during syntactic incompleteness, minimizing the *gap problem*: with its index still a hole, `nth` shows a *partial result*, evaluated until the hole blocks the `if`, which here hints that `nth` counts from zero.

In B, the programmer has instead reversed the arguments to `string_split`, producing an unexpected value. Double-clicking a sample switches from *Single* to *Many* mode, showing that all collected `string_split` samples have the same value. Since both arguments have type `String`, types alone would not expose this mistake. The switch demonstrates our approach to the *multiplicity problem*: the programmer can cheaply switch to a multi-sample view when a single sample is uninformative. When a sample is selected (red border), the stronger green highlighting on other probes’ samples indicates that they are part of the same call, promoting *coherence*.

Subfigure C shows immediate feedback guiding an edit. Changing index `0` to `1` produces the expected directory name and a passing test before the closing parenthesis is entered, further emphasizing that the system minimizes the *gap problem*.

Additional features of Live Probes, described further below, address the problems of term selection, multiplicity, and value detail through progressive disclosure:

- *Term selection*: Control over which terms are probed, from placing one probe with a single gesture to placing them automatically across a whole definition or program, steered by line breaks, similar to a REPL (read-eval-print loop).
- *Multiplicity*: Control over which executions of a probed term are shown, from one focused execution to all collected samples, including filtering attention to a single *pinned* call.
- *Value detail*: Control over how much of each sample is shown, from a structurally abbreviated, resizable value at the end of the line to a full-width pretty-printed drawer below the line, with environment and other ancillary information available via a dropdown on each sample. Values of some types can instead be drawn by views written for that type, such as a sparkline or a table (Figure 2), and programmers can write their own (Section 3.6).

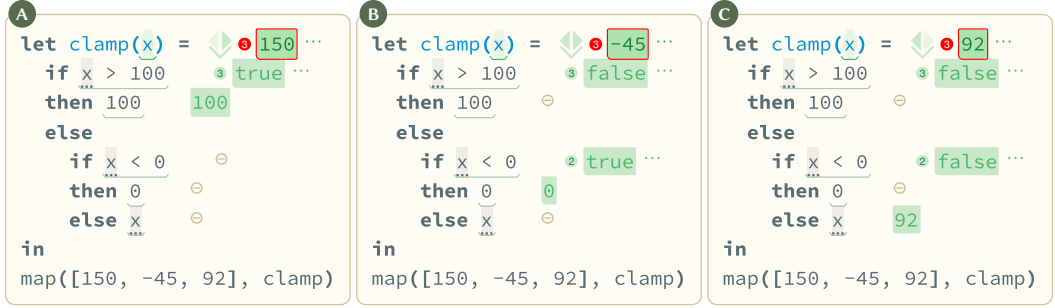


Fig. 3. Dynamic focus in *Single* display mode. Here a probe has been automatically placed on every line of the `clamp` function definition, which is called three times below. The three subfigures show three different focus configurations, aligned to the three function calls. On each line, an underline indicates the probed term, and a sample value for that term is shown at the end of the line. Although only one sample per probe is shown, count badges (3, 2) show the total number of samples for that probe. The $\ominus$ symbol indicates that there are samples at a probe, but none are shown as they do not align with the current focus, here indicating conditional branches not reached for a given call. A user can navigate between these three views by (for example) selecting the green `150` sample in A (outlined in red once selected), and then pressing the right arrow key to navigate to B and C.

2 Background

Live Probes builds on three parts of the Hazel live functional programming environment (<https://hazel.org>): live evaluation of incomplete programs (Section 2.1), a structure editor that edits like text (Section 2.2), and livelits [59], GUIs defined and situated within the program (Section 2.3).

2.1 Evaluating Incomplete Programs

Hazelnut Live [60] introduced live evaluation in the presence of typed holes—incomplete programs that maintain meaningful type and dynamic semantics. Hazel evaluates around holes, so a program produces *partial results* even while incomplete, providing a foundation for live feedback during program construction. A partial result is a term evaluated as far as its holes allow, with the holes left in place (an indeterminate form, in Hazelnut Live’s terms). We say ‘value’ for both, and ‘partial result’ where the difference matters.

Live Probes relies on Hazel’s feedback totality: every editor state elaborates to a well-typed term, with empty holes for incompleteness and error marks for inconsistency, so every intermediate state during writing has a semantics. Hazelnut Live [60] collected dynamic information for holes (hole closures) and for the program as a whole. Live Probes collects it for any expression or pattern the programmer probes, recording a trace of evaluation from which samples are derived (Section 3.1).

2.2 Editing Incomplete Programs

The original totality properties underlying always-on feedback relied on the fact that Hazelnut, the calculus Hazel started from, was a structure editor, with edits operating directly on the AST [61]. Many programmers find tree edits baroque and unintuitive, so later work sought the same guarantees with free-form text entry. Hazel’s editor descends from `tylr` [54], a tile-based structure editor offering a text-like editing experience. Every intermediate state parses, with holes inserted where a term or an operator is missing, and syntactic completion heuristics [54, 55] supply the missing delimiters, so that partially typed forms parse as their most plausible completion. As a result, every edit state has a parse, statics, and dynamics. Here, we extend Hazel’s editor with mechanisms focusing on stability of intermediate feedback (Section 3.7).

2.3 Live GUIs and Augmented Code Views

Livelits [59] are user-defined GUIs, drawn inside the code view, that fill typed holes in Hazel: a livelit stays in the program as a graphical literal, and can show live feedback based on the program's runtime values. Probes are driven by similar infrastructure, gathering and putting runtime information in the code view, though they augment instead of replace textual syntax. Hazel Lab [3] introduced *rich probes*, domain-specific views of probed values, with a table view for lists of records; it left user-definable views, and their relation to livelits, to future work (Section 3.6).

3 Design

We begin by describing probes and their associated samples (Section 3.1). The next subsections follow the introduction's problems of term selection, multiplicity, and value detail: which terms are probed (3.2); which executions are shown, as determined by the dynamic focus (3.3), together with call pinning, stepping into calls, and the stack focus bar (3.4); and how much of each sample is shown (3.5), including views that draw a sample by its type (3.6). The final subsection describes how feedback, probes, and the focus are maintained as the program is edited (3.7).

3.1 Probes and Samples

A probe is placed on a term (any expression or pattern) with a context menu or keyboard shortcut, and its samples appear directly to the right of the line, minimizing indirection. Probes are persistent syntax annotations: once placed, a probe stays on its term as the program is edited, even while the term itself is being edited (Section 3.7). A sample presents a value taken on by that term during execution, or a partial result where a hole stopped its evaluation (Section 2.1); see Figure 1 A. Each sample is a value together with its dynamic context, including its environment, its range of reduction steps, and its call stack. A probe often has several samples; Section 3.3 describes which of those are shown.

There can be at most one probe on a line. Adding a probe replaces the one already on its line, and when an edit brings two probes onto one line, the rightmost is kept. We chose this limit deliberately, to keep each sample easy to match with its probed term. Section 3.5 describes the handling of values too large for the end of the line.

When no samples are shown for a probe, one of several icons is shown, including $\emptyset$ if the expression never ran, $\ominus$ in *Single* mode if none of its samples belongs to the focused execution (Section 3.3), and $\otimes$ if a pin hides them (Section 3.4).

The probes described so far are *simple probes*, placed one at a time by the programmer. Section 3.2 describes multi-probes and autoprobing, which place probes automatically.

3.2 Which Terms? Multi-Probes and Autoprobing

Simple probes give exact control over which terms show samples, but placing them one at a time is slow when the programmer wants broader dynamic context.

We have developed a logic for automatic placement, for when a programmer wants pervasive feedback while writing, or wants to follow a whole function's behavior line by line without placing probes by hand. Autoprobing is intended especially for in-place experimentation and debugging, to help form hypotheses as much as to answer questions about the program.

When enabled, probes are automatically and dynamically placed on heuristically selected terms of each definition, one per line, as seen in Figure 4. This mode enables the user to flexibly steer where probes are placed without the need for an additional control surface, demonstrating intent via line break placement. This steering via line breaks was inspired by REPL-driven development. Here, though, line breaks only choose which terms show values. Evaluation does not wait for them:

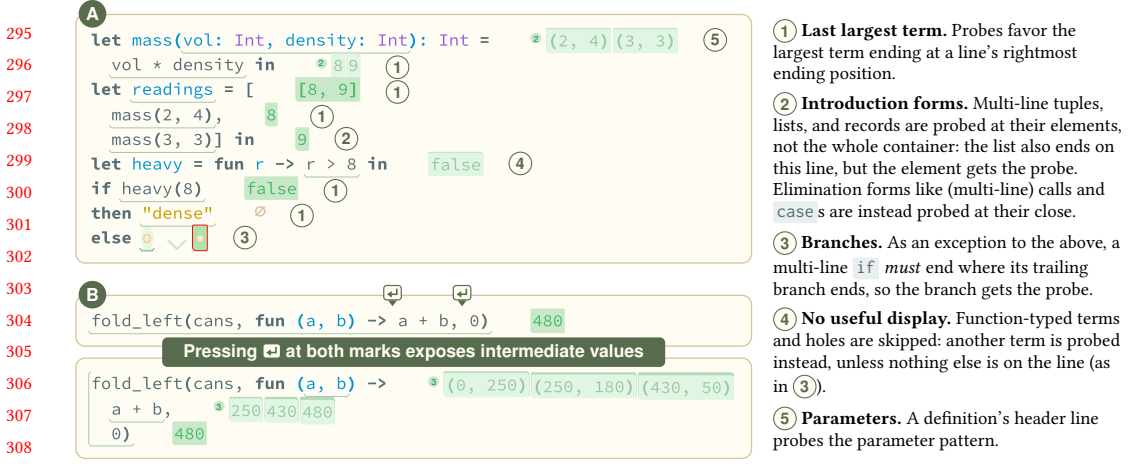


Fig. 4. Autoprobe placement. Probed terms are decorated with hooked underlines, with a left-hand rail spanning multi-line terms, such as the three-line fold in **B**. The placement heuristics relevant to each line in **A** are numbered at the end of each line and explained on the right. The hole's sample at the bottom of **A** is selected (outlined in red). In **B**, `cans` is `[250, 180, 50]`: on one line, the fold shows only its value; inserting two line breaks 'pops the hood' of the fold, exposing the callback's per-iteration arguments and running totals, aligned by call.

values appear and update as the programmer types. The aim is to provide the exploration-oriented and feedback-driven experience of a REPL [20, 36, 38, 41] to try out unfamiliar functions or compose a data transformation pipeline, but without switching away from the main editor, without having to finish a function definition, and driven by the live values already flowing through the program.

Actuation. Placement can be automated at two levels:

- **Function-level multi-probe:** On a function definition, the normal way of adding a probe yields a **multi-probe**, placing a probe on every line of the definition. This lets the programmer see the function's intermediate values line by line.
- **Whole-program autoprobings:** The system automatically multi-probes every definition, test, and top-level expression in the program.

Placement heuristics. Unlike per-line displays that show every variable of every line [40, 43], multi-probes pick at most one term per line, defaulting to each line's *last largest term*. This means finding the last position on the line where a term ends, and, if more than one nested term ends at that position, probing the largest term (① in Figure 4 **A**). As a result, a term that spans multiple lines competes only on the line where it ends, so a multi-line function application is probed at its closing parenthesis. Probes are restricted to expressions and patterns; case rules, module declarations, and labels are not terms of either sort and are never selected. Hooked underlines mark the probed terms (Figure 4), but with a probe on every line, which term a sample belongs to is not always plain. Clicking a sample highlights its term in green, as for `string_split` in Figure 1 **B**.

The base placement rule has a few refinements, numbered ②–⑤ in Figure 4 **A**. A multi-line introduction form like a list, tuple, or record is probed via its elements rather than as a whole, since it simply puts its elements together. A multi-line elimination form like a function application or `case` is probed where it closes, since its result is something new. A multi-line `if` is an exception: unlike `case`, which closes with `end`, an `if` ends exactly where its trailing branch ends, so its result, which comes from *whichever* branch ran, would appear on that branch's line; the probe

goes on that branch instead. Functions and holes have no informative display, so they yield to the next term ending on their line. But when nothing else is left, they are still probed, since the sample count still shows how often they were reached. On a function's header line, the probe goes on the parameters, so argument values appear at their definition.

When pressing Enter gives an intermediate expression its own line, it gets its own probe. In Figure 4 ⑤, a one-liner fold only shows its result; breaking the line after the callback's arrow and after its body shows each iteration's arguments and running result.

Automatic placement can be overridden. The probe shortcut hides an automatic probe and shows it again, and placing a simple probe on another term of the line replaces the automatic one. In our study, autoprobing was the placement mode participants most often reported using while writing, though line-break steering divided them (Section 5).

3.3 Which Executions? The Dynamic Focus and Alignment

Placement decides which terms show samples; the dynamic focus decides which executions they come from, by marking a place in the program's execution. In *Single* mode it acts as a filter: each probe shows its sample from the focused execution. In *Many* mode it acts as a reference frame: each probe shows a row of samples, styled by how each relates to the focus. Refocusing is generally an explicit user action, clicking on or arrowing to a sample, but can happen automatically when the program is edited (see Section 3.7).

Single vs. Many. A probe shows either one sample at a time or a row of up to 30, a window onto its samples that slides as the focus moves past its edge. *Single* mode is for attending to one execution, and *Many* mode for scanning across calls. In *Single* mode, a probe whose samples all belong to other executions shows a *not-aligned* marker (⊖); the sample count badge indicates how many samples are hidden.

In *Single* mode, the editor has to decide which sample each probe shows. If each probe decided separately, the samples on screen could come from different calls: a condition could show `false` next to a value from the branch that runs only when it is `true`. This is the coherence problem from the introduction. Instead, all samples shown are *aligned* to the dynamic focus. In the common case, the focus is one call to a function, and each probe in the function shows its sample from that call. Each panel of Figure 3 shows a different call of `clamp`. The arrow keys move the focus to the previous or next call, and all of the function's probes change together. A debugger similarly coordinates value displays via a selected stack frame [52]; Section 6.2 compares how inline displays make this choice.

Nested calls. In Figure 5, `map` applies an outer function literal to each row of `[[1, 2], [3, 4]]`, and that outer function applies an inner function to each element of its row. Choosing a call of the outer function (say the one for `row = [3, 4]`) still leaves two calls of the inner function to choose from (say `n = 3` and `n = 4`). So the focus is a path in the call tree, with one call at each level. Clicking a sample (the *focal sample*, outlined in red) sets the path to the calls that were executing when that sample was recorded (i.e., the call stack captured with it), and each probe shows its sample from those calls. The focus starts on the deepest of these calls, the one that recorded the focal sample. The arrow keys can then step through the other samples of the same probe. In Figure 5, the programmer clicks `10`, the earliest value of `n * 10` (①), and presses `→` twice to reach `30` (②). The focus is now the inner call for `n = 3` inside the outer call for `row = [3, 4]`, and the line for `row` has changed to `[3, 4]` to match. The stack focus bar above the editor lists the calls on the path (Section 3.4).

The path can stop short of a given probe's calls. If the focus is on the top level and the path holds nothing below it, nothing says which calls of the two functions to show. The inner probes then show one call per function, chosen from the top down: the first call of each function inside the

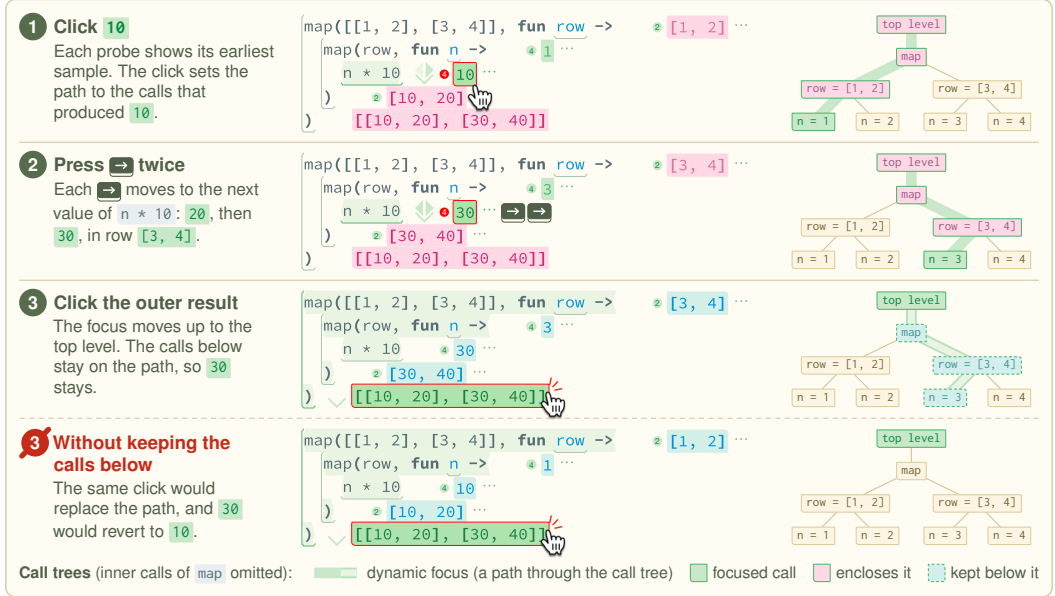


Fig. 5. The dynamic focus is a path in the call tree, shown in each tree above, from the top level down. ① In the initial state, each probe shows its earliest sample. The programmer clicks 10, the earliest value of $n * 10$. ② Pressing → twice steps to 30, in `row = [3, 4]`. Each probe now shows its sample from the outer call for `row = [3, 4]` and, inside it, the inner call for `n = 3`. ③ Clicking the outer result moves the focus up to the top level. The calls below stay on the path, so 30 stays. ④ Hypothetical: if the click instead replaced the path, the inner probes would show their earliest samples again, and 30 would revert to 10, losing context.

calls already chosen (for a recursive function, its outermost call), here `row = [1, 2]` and, inside it, `n = 1` (Figure 5 ④). A line that no chosen call reached shows $\ominus$, even when another call reached it.

Moving up and back down. Programmers click samples to inspect them, and clicking a sample from an enclosing call should not cause the loss of a previously focused call below it. So clicking a sample from a call on the path, or that call's entry in the stack focus bar, *moves the focus up the path* to that call and *keeps the calls below it*. In Figure 5 ③, clicking the outer result moves the focus to the top level, and the deeper probes keep their samples: 30 remains shown. This matches debugger behavior, where selecting a caller in the call stack pane shows that call without discarding the frames below it [52]. Only moving to a call that is not on the path replaces the path: for example, clicking [3, 4] and pressing ← moves the focus to the outer call for `row = [1, 2]`, and the inner probes start again from `n = 1`. The calls kept below the focus are exactly the history needed to keep the programmer's place, since they record exactly which samples the deeper probes show. Under recursion, the path holds several calls of one function; each of its probes shows the sample from the call the focus is on, and the bar can move the focus back down to the deeper calls.

Many mode. In *Many* mode, the focus serves as a reference frame rather than a filter, with sample background and font color showing how each sample relates to it (Figure 6). The *Simple* scheme highlights the samples from the focused call and fades the rest. The *Hybrid* scheme relates every sample to the focal sample in time (step reduction order), treating each evaluation as an interval, from when the expression starts evaluating to when it produces its value. Because evaluations nest, any two are either disjoint or one lies inside the other, so four relations suffice: a sample's

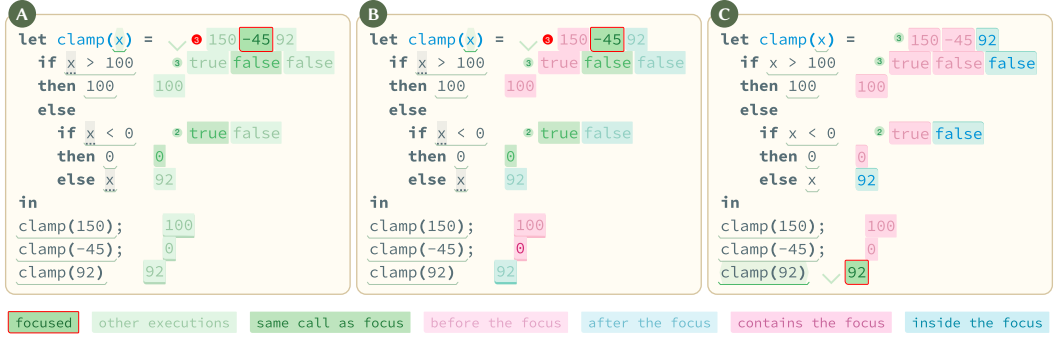


Fig. 6. Dynamic focus in Many mode. Each probe shows all of its samples in execution order (the definition line shows each call's argument; the call sites at bottom show each call's result). **A** and **B** focus the same sample, the argument `-45`, in the two color schemes. **A** shows the Simple scheme: the samples of other calls, such as `150`, are faded, and the other samples of the focused call, such as `true`, stay saturated. **B** and **C** show the Hybrid scheme, which colors each sample by the relation of its evaluation span to the focus. In **B**, the other samples of the focused call, such as `true`, get the same-call color; the earlier call's samples, such as `150`, come before the focus, and the later call's, such as `92`, come after it; and the call's result `0` at call site `clamp(-45)` contains it. **C** focuses the result `92` at call site `clamp(92)`: the earlier calls' samples, such as `-45`, come before it, and the samples recorded during that call, such as `false`, lie inside it.

evaluation can come before the focal sample's, after it, contain it, or lie inside it. The Hybrid scheme gives each of those relations its own color. What makes the scheme a hybrid is that it also assigns a fifth color to the other samples from the focused call, flattening their temporal sequence so that the samples from one call are easy to pick out. In Figure 6 **A** and **B**, the focus is on the parameter sample `-45`, first in the Simple scheme and then in the Hybrid scheme. In both, the other samples of that call stand out. In **B**, the call's result at the call site contains the focus, and the samples of the other calls are colored by whether they come before or after it; this picks out the call that led to the focal sample. In **C**, the focus is on the result of `clamp(92)`: the samples recorded during that call lie inside it, and the earlier calls come before it. To reduce the amount of explanation required, participants in our study used the Simple scheme; we have not evaluated the Hybrid scheme with users. Under simulated red-green color vision deficiency, the Simple scheme stays legible, while the Hybrid scheme's pink and blue converge (Section 5.6). Color is not the only cue, though: samples from calls that enclose the focused call sit slightly higher on the line, and samples from calls inside it sit slightly lower.

3.4 Pinning and Stepping into Calls

Debugging often means following a failing case through a chain of functions, and three features built on the dynamic focus support this: **pinning** filters every probe's samples to one call; **stepping into** a call descends into the called function with that call focused; and the **stack focus bar** can be used to backtrack up the call stack.

Pinning a call sample hides, in the other probes, every sample that was not recorded during that call, reducing distraction when following one case through the code. A probe whose samples are all hidden by the pin shows $\odot$. Inside the pinned call, a probe can still have several samples, for example from a callback called several times. Samples from the calls that enclose the pinned call stay visible, showing how the call was reached. While a call is pinned, the focus never moves on its own (Section 3.7).

Step into, named for the debugger command, does three things on a call sample: it pins that

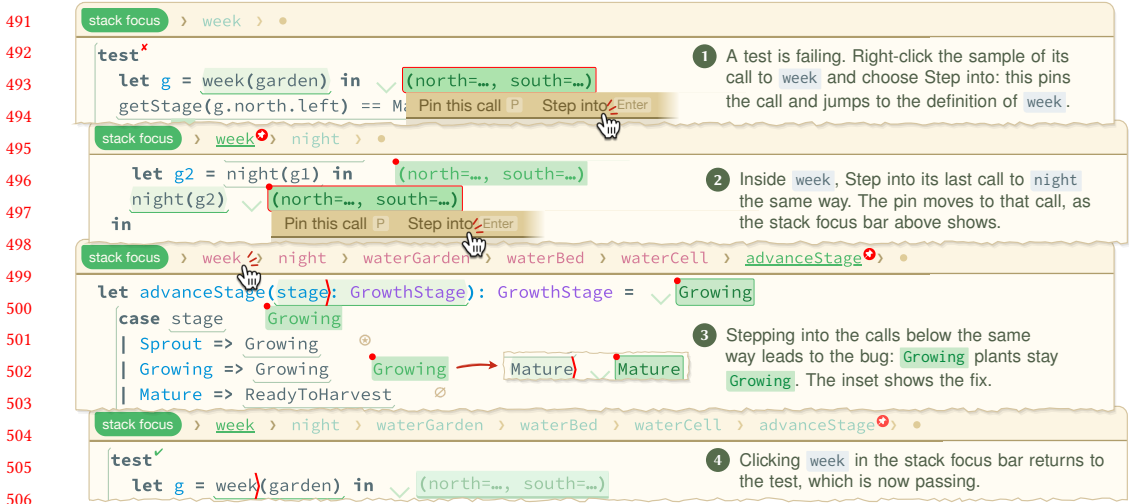


Fig. 7. Pinning and backtracking in a study debugging task. A failing test expects a plant to be **Mature** after a week. Stepping into a call pins it, so every probe shows only samples from that call, not those of other calls, which would multiply on the way down. The stack focus bar lists the calls from the test down to the focus, and its pin icon marks the pinned call. The programmer drills down through a series of calls, then can backtrack with the bar: the calls below the focus stay on the path, faded, so the path back down is kept.

call, jumps the caret to the called function's definition, and probes that definition. The programmer arrives inside the function with the relevant execution already displayed. On a recursive call, it moves the focus one level down the recursion.

Stack focus bar. Stepping back out uses the stack focus bar above the editor, which lists the calls on the focus's path, outermost first. The focused call is highlighted, the calls below it are faded, and a pinned call is marked. Clicking an entry moves the focus to that call and jumps to the place where it was called, keeping the calls below it on the path (Section 3.3); clicking the entry of a call one has stepped into returns to its call site. The bar can be used to backtrack across functions while debugging. In Figure 7, the programmer has followed a failing test down to the call of `advanceStage` that returns the wrong stage. Clicking `week` in the bar moves the focus up to the test's call of `week` and jumps there; the calls below `week` stay on the path, faded in the bar.

3.5 How Much Detail? Displaying Samples

Samples vary in size, from a single number to a list of records many times wider than the line. Samples start small, abbreviated to fit the end of the line, and the programmer can expand any one of them in several ways, described below. Values of some types can instead be drawn graphically (Section 3.6).

Abbreviation. Inline display is valuably terse, but simply truncating longer values obscures their structure. Instead, abbreviation keeps the value's outer structure, preserving closing delimiters of e.g. lists and tuples, while eliding inner children with an ellipsis and a count of the elements left out. The engine recursively divides a character budget by a semantically driven scheme: record fields split the budget evenly (as each field name carries structural information), while list elements deliberately do not (list elements share a type, so one element communicates value-wide structural information). Samples can be resized via shift-drag or shift-arrow-keys (Figure 8 ①), with sizes persisting across re-evaluation.

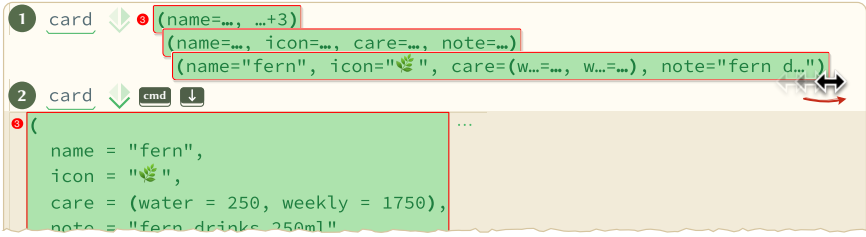


Fig. 8. Resizing and the drawer: ① A sample of `card` at three widths, each drawn over the last, widened by shift-dragging its edge (or Shift+→); ② Opened in the drawer by clicking the down chevron (or Cmd+⌵).

End-of-line vs. drawer. The end-of-line position suits small values, and facilitates showing many intermediates over a range of lines. In contrast the *drawer* (below the line display) takes the full width of the code and arbitrary height (with a separate scroll frame after a fixed number of lines), admitting multi-line pretty-printed value representations (Figure 8 ②). Such a mechanism is necessary to support large values while staying close to the syntax, but brings with it design challenges. For example, scroll stability: sample sizes can change as the programmer edits code, which might cause drawers above the edit to change height, requiring scroll compensation logic.

Sample Dropdown. Right-clicking a sample opens a dropdown with auxiliary context. It shows the dynamic environment, filtered to references actually present in the syntax (formally, the co-context of the probed expression). For probes on applications, it also shows the final argument values. This dropdown also surfaces debugging affordances including pinning and step-into (Section 3.4).

Keyboard and mouse. Most of the interface can be driven from the keyboard. A shortcut moves from the code into the probe on the caret’s line. There, the left and right arrows move between the probe’s samples, the up and down arrows move between probes, and other keys resize a sample, open its drawer, switch between *Single* and *Many*, and pin or step into a call.

3.6 Rich Probes

Abbreviation (Section 3.5) is a generic approach to summarization of a value’s text representation. But many types have more natural representations than text, and some of those stay legible at sizes where text cannot. For example, a color becomes a swatch, or a trace of readings becomes a sparkline of fixed width independent of the number of readings (Figure 2 A–C). Probes that draw samples this way are *rich probes* [3]: a sample whose value has a *view* is rendered by it instead of as text. A view at most two lines tall takes the place of the sample’s text at the end of the line, where the sample’s focus styling still applies to it. A taller view, such as a table or a piano roll, is drawn below the probed term when the probe’s drawer is open (D, E), and on the line its samples are shown as text. Values without a view stay text, so views and text mix freely across a program’s probes.

In *Many* mode, where each sample gets its own view, a probe’s views read as small multiples [76, Chapter 4]: in Figure 2 B the traces of eight shipments stay comparable where their text has room for only the first few readings, and in A the hands at several of quicksort’s calls are visible at once.

Choosing a view. Hazel provides some views, such as the table of Figure 2, and programmers can define their own as *livelits* (Section 2.3), like the sparklines, swatches, and piano rolls. A *livelit* is a GUI for a type: in code it serves as a literal of that type, and in a probe it acts as a view for that type’s values. A *livelit*’s view applies automatically to the samples of a term whose type has the same name as the type the *livelit* produces. Giving a value a named type is the programmer’s explicit choice, whereas a value whose type merely has the same structure, such as a pair of integers,

could mean anything. The sample dropdown's *View as* list is more permissive: besides text, it offers every view whose type fits the probe's, such as the table for any list of records, which is not chosen automatically. A global setting turns automatic views on or off.

User-defined views. A livelit is represented by its model (state), which for a literal is stored in the syntax. A color picker's model is a color, along with the positions of its handles and whether it is open. A probe sample is only a value, so to serve as a view, a livelit defines one more function, which builds its state from a value: for the color picker, the color with the picker closed. The view is also told whether it is read-only, so that it can hide its controls in a probe, and how much room it has (end-of-line limits height to two lines, and the drawer is unlimited). In a probe, a view only displays. Unlike a livelit in code, or Hazel Lab's table, which rewrites the probed expression [3], it does not edit the program. Letting views edit code is future work.

3.7 Stability While Editing

Hazel re-evaluates the program on each keystroke, and most edit states are incomplete. We target *feedback stability* through these states: as the programmer inserts new syntax, feedback for the code before and after the incomplete spot should stay steady and accurate, and the editor should not reinterpret that code until the new syntax is complete enough for its effect to be clear.

Keeping surrounding code stable. We added two rules to Hazel's editor to keep an incomplete edit from prematurely changing the meaning of the code around it. First, complete forms are *anchored*: a delimiter that belonged to previously complete syntax does not re-pair with a newly typed one unless doing so reduces the number of incomplete forms, rather than merely moving them around. Second, an incomplete form is completed at the first line indented no deeper than the one where it begins, so a definition typed between two others leaves feedback on the code below it unchanged. Hazel's syntax does not depend on indentation: delimiters decide how code nests, and only where one is missing does this rule fall back on indentation as a sign of the programmer's intent. Enter indents each new line to fit the code's structure at that moment, but the editor does not re-indent incomplete code afterward, so the indentation this rule reads is the programmer's own layout.

The editor also shows the completion it uses. Each pending delimiter appears between the lines, just above where it would go, at a place recomputed from the current code after every edit, whereas tylr draws pending delimiters as persistent *ghosts* among the characters, which cannot be deleted directly [55, Sections 2.4 and 4.2].¹ Here the code edits like plain text, while the programmer can still see what the feedback is computed against.

Keeping probes, samples, and the focus attached. The editor preserves the identity of tokens and terms across edits, so a probe stays on its term even while the term itself is being edited. Each keystroke re-runs the program and produces new samples, which the editor matches to the previous run's by the term that recorded them and the calls on the stack, much as Managed Time keys each execution by its syntax, call path and loop iterations, so the key survives re-execution after edits [48]. Through that match, per-sample display settings, such as a resized sample's width, carry over, and the dynamic focus (Section 3.3) stays on its calls as long as they still occur.

Automatic refocus on an edit. While the dynamic focus is generally under manual programmer control, the editor itself refocuses in one situation, trading stability for responsiveness (Figure 9). The focus moves only when the line being edited would otherwise show $\ominus$, and *never* if a call is pinned, since a pin is the programmer's explicit choice of call.

¹In tylr's user study, at least five of nine participants tried to delete ghosts directly, and five felt that pending delimiters got in the way while modifying code [55, Section 5.2].

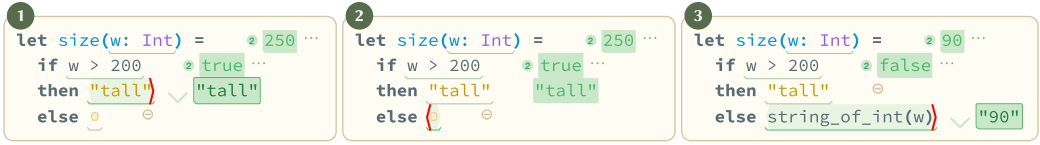


Fig. 9. Automatic refocus on an edit. ❶ The focus is on the call for `w = 250`, which takes the `then` branch, so the `else` hole shows $\ominus$. ❷ Moving the caret (red) to the hole changes nothing. ❸ Typing `string_of_int(w)` moves the focus to the call for `w = 90`, the first call that reaches the `else` branch: the new code shows `"90"`, and the `then` line now shows $\ominus$.

While moving the focus automatically helps provide context while writing, it can get in the way while debugging. Here, the focus is usually the case under investigation, and when it moves on its own, every probe changes at once. In our study, one participant took such a change for an effect of their own edit (Section 5.4).

4 Implementation

Live Probes is implemented in Hazel, a web application written in OCaml and compiled to JavaScript, so the editor and the evaluator both run in the browser. Hazel’s evaluator is an interpreter that has not been optimized for speed.

Sample collection. Before evaluation, the probes become a map from persistent expression ids to statically derived *capture specifications*, each listing the variables to record with a sample: the free variables of the probed term. During evaluation, the evaluator emits a stream of events. A *span* opens when a probed expression starts evaluating and closes when it produces a value, and when a probed expression is a call, a call event records the call’s frame and argument. Samples are built by folding over this stream.

Sample identity is content-derived: a sample’s id hashes its syntax id and call stack, ingredients that are stable under edits in a structure editor. Per-sample display settings (Section 3.7) are stored under this id, so they re-attach after each re-evaluation and lapse when no matching sample exists. Each call of a recursive function, including each iteration of `map` or `fold` (both recursive in Hazel), runs one frame deeper on the call stack, so its samples get their own ids. The focus refers to its sample by the same ingredients plus the evaluation step at which the sample’s span opened; when an edit rebuilds a call site on the stack, the editor falls back to that step to find the sample again.

In the browser, evaluation runs asynchronously, in slices, so editing does not wait for it. Results stream back to the editor after each slice, so samples are received even in the case of non-termination, and unchanged top-level definitions are not re-evaluated.

Performance. On the study’s programs, of up to about 300 lines, autoprobings the whole program made evaluation 1.3 to 4.5 times slower in headless measurements; since evaluation runs asynchronously, this delays samples, not editing. In principle collection is linear in the number of samples, since values are immutable and call stacks persistent, so both can be shared rather than copied. The prototype copies captured values and rescans a probe’s earlier samples on each new one; in an experiment, removing these two costs made collection a steady multiple of evaluation without changing the samples recorded, except under deep recursion, where per-sample work on the call stack still grows with depth. We do not yet bound the number of samples; adding a bound is straightforward, with one wrinkle: it must keep the samples the focus aligns to.

Editor and language requirements. The design makes use of four features that Hazel has and most editors and languages lack. Its structure editor gives every term a persistent identity, which

keeps probes and samples attached across edits (Section 3.7); a text editor could approximate this by matching each new syntax tree to the previous one, though a term in the middle of an edit may not match its earlier version. Typed holes and syntactic completion let incomplete programs run, so they still produce samples (Section 2). The language is pure, so re-running the program on every edit is safe. And livelits let programmers define GUIs for their own types within the program, which probes reuse as views (Section 3.6).

5 User Study

The study asked how well the design addresses the six problems of Section 1 in use, through two questions: how do participants use probes while writing and debugging, and where does the design get in their way?

5.1 Study Design

Study format. We ran 15 facilitated sessions where programmers used Live Probes to write and debug programs in Hazel. An interactive tutorial introduced the language, the editor, and probe controls, followed by tasks with supplied starter code and tests. Sessions were advertised as approximately two hours, remote over Zoom. We recorded participants' screens and conversation. Selected writing and debugging tasks were assigned without probes, providing a basis for reflections on print-based inspection. Thirteen participants returned an exit survey.

Participants. We recruited through Twitter, Bluesky, Mastodon, and the Future of Coding Slack community. Participation was voluntary and uncompensated, with eligibility requiring prior functional-programming experience. The intake form included two questions on basic functional programming. All 15 participants reported prior functional-programming experience and 4–30 years of programming experience overall (median 10). Fourteen reported an age, ranging from 19 to 50 (median 27). Gender responses were eight male, two female, two non-binary identities, and three participants did not report gender. The study was approved by our institution's IRB.

Tasks and facilitation. The task sequence alternated debugging existing programs with writing or extending functions. Writing tasks included extracting a path component, computing running sums, cleaning log entries, and adding a grid-update operation, requiring approximately 2–10 lines of new code in the reference solutions. Debugging tasks used programs of approximately 60–315 lines, mostly GUI apps in the Elm Architecture style [14]. Participants were directed to read the main types and find the failing test before investigating. Supplied tests specified expected behavior and provided calls from which probes could collect samples. The task materials encouraged experimentation with library functions and allowed participants to choose among manual probes versus autoprobing. Participants were asked to think aloud. The facilitator answered language and library questions, discussed participants' observations, and sometimes intervened to help them progress to ensure wide task exposure. Task coverage and use of the print contrast varied between sessions.

Analysis. We reviewed all session transcripts, facilitator notes, and returned surveys, using recorded segments to contextualize selected remarks and interactions (see the next paragraph for AI tool use). Below we present an interpretive synthesis of this material, analyzing how participants used probe features and assessed the system's usefulness. We organized observations around recurring uses and difficulties, grouped by the problems of Section 1. A quantitative summary covers the 13 returned exit surveys.

Use of AI tools. We used AI agents in this work, mainly Claude Code (Anthropic) and Codex (OpenAI). Claude Code was used to refine and extend the first author's initial implementation, in

sessions the first author directed and reviewed, making about half of the commits overall. It also drafted some study tasks and materials; the first author revised these, piloted the tasks, and chose the participants. In the analysis, agents wrote per-session analyses from facilitator notes, transcripts, and video frames, checked by a second agent, and proposed episodes and quotations reported here based on author queries. Agents also wrote some of the custom data views in Section 3.6, as well as scripts that drew survey charts and captured editor figures, and reconstructed the study figures. An author checked every quotation against the audio, extracted figure captures against the recording, and survey counts against the responses.

5.2 Indirection: Values Beside the Code

Checking each transformation. While building a string-cleaning pipeline, (P06) described developing the code alongside its intermediate results: “It was really useful to develop it while I was looking at the outputs. It was kind of this test-driven development, but on steroids.” (P15) explained what seeing each stage saved them from doing: “I’d have to mentally imagine the transformations, and then try to guess if it was working or if it failed earlier in the pipeline or later.”

Quick and direct experimentation. The log-cleaner task also evoked small experiments with unfamiliar string functions. (P10) wrote a separate call to `string_replace`² within the unfinished cleaner, using the short input “abc” and varying the replacement pattern while inspecting its result. (P07) identified “Figuring out the behavior of horribly-documented string functions by example” as a helpful use of autoprobe and described probes more generally as “Every line is a little REPL”.

Matching output to its source with prints. During the harvest-streak task without probes, (P09) printed the ledger and action at each fold iteration, then added a print inside the function processing each harvest. The resulting console output interleaved actions from successive iterations with the harvest values. (P09) later described the work of matching printed values to the program as “additional mental load”. (P05) encountered a related difficulty during the print-only timer task where they commented out tests to reduce the output to the calls of interest. In the survey, (P05) explained that filtering the output had not eliminated the work of relating it to the source: “Even then, I had to do the mental work of associating logged values with where they came from in the code.”

5.3 Term Selection and Value Detail: Choosing What to Show

These episodes concern term selection, multiplicity, and value detail: which terms show samples, how many, and how much of each.

Automatic and manual placement. In the survey, respondents were divided on which placement they mostly used: six autoprobe, four manual, and three a mix while debugging, and eight, three, and two while writing. Neither mode alone would have suited everyone. Automatic placement provided a popular starting point for inspection, with (P05) describing probing each line of a chosen function with the definition multi-probe as “a nice way to ‘pop open the hood’ without knowing what I was looking for”. (P13) distinguished background feedback from deliberate inspection: “I found it useful to leave autoprobe on ‘in the background’, so I could use it as immediate feedback, without thinking about it much”, while sometimes placing a probe manually to examine a specific variable or expression. But the overview also made choosing difficult for (P07): “I need the auto-show-probes-by-default mode because I don’t yet have a sense of where to surgically place them, but then there are so many”.

²The replacement function inherited its argument order, quirks and all, from `js_of_ocaml`.

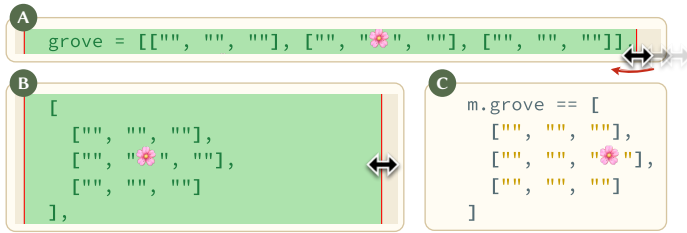


Fig. 10. [P02] drag-resizes a computed grid from a single row (A) to three rows (B), making the misplaced flower easier to compare with the expected grid (C). A and B show the `grove` field of the model in the drawer, before and after dragging its right edge, which re-wraps the value at the new width; C is the failing test's expected grid. Reconstructed from the session recording.

[P03] valued the automatic intermediate strings while writing the log cleaner, but preferred manual placement more generally: “Most of the time I liked placing them manually so that I knew exactly what I was probing.” Their survey response connected this uncertainty to the combined state of automatic and manual placement, pinning, and the Single or Many mode: they were often unsure which samples the current configuration would show.

Steering placement with line breaks. Targeting feedback through line breaks tied inspection to changes in source layout. [P01] described line breaks as quicker than manual probe placement, concluding, “I think I could live without manual placement”. For [P03], however, “adding a single line break to otherwise well-formatted code often leaves it looking off and harder for me to scan visually”. [P03] also complained of having to revert the line breaks after inspection to restore their preferred layout.

Making a comparison visible. Sample presentation could make a particular comparison easier: during crop planting, [P02] narrowed an output sample until its nested lists matched the expected 3 × 3 layout (Figure 10). Asked about the resize, they explained that they wanted it “in the same sort of layout as this one”. [P01] similarly inserted line breaks in a log-cleaner test so that the computed string appeared directly above the expected string. [P02]’s grid is a layout the value’s type implies, and [P07] asked for such presentations in general, wanting “a more structured viewer for objects” and “ways to lay out and navigate the different conceptually 2D data”; the type-specific views of Section 3.6, added after the study, let programmers define presentations like these.

Presentation remained a limitation for larger values and sample sets. [P10], who gave one of the two lowest ratings for focusing on relevant samples, explained, “With big strings, with many samples it was too much sometimes.” [P13], who gave the lowest rating for choosing how much detail to see, reported, “when I had more than 3 or 4 values, I couldn’t even see them.” [P15], who also rated it low, wanted less shown by default: “I don’t want to see everything. I want to see what I want to focus on. Only if I’m lost will I browse all available info.”

5.4 Coherence: One Execution, Kept While Editing

Relating samples across executions. In the crop planting task, [P13] inspected a failing test in which a flower appeared in the middle column instead of the rightmost column. Samples at the inner conditional showed `false, true, false`, matching the misplaced flower. [P13] changed the comparison from the row to the column, the samples switching to `false, false, true`, and the flower appearing in the expected position. Asked afterward what had helped, they said they had matched the pattern of booleans to the flower’s position before working out what the condition

checked: “It immediately drew my attention”.

Following one execution. Pinning a call let participants follow its execution through samples displayed beside the relevant code. After stepping into a failing call, [P11] welcomed the resulting branch-local view: “this is actually really nice. I can literally immediately ignore all of [the other code] that’s going on”. The other branches no longer showed samples, allowing them to focus and fix the error. In the debrief after the growth task, [P09] placed a probe in a deep helper, several calls below an existing pin. The new probe still showed samples from that pinned call; they did not have to step through each intervening call. They could inspect the pinned case “by just putting down a probe like five levels down in the call stack”. They then checked that a new probe also follows a selected sample: they selected a sample, placed another probe, and saw it highlight the sample from that call: “it’s already highlighting at the right part, so I think that that makes sense”. [P05] navigated the same way: “when there was a lot of values probed it felt okay to navigate by focusing on things and seeing what other things would be highlighted”.

Expecting a selection to stay. In the timer task, [P08] selected the callback call whose input was 60, then added a line break to see the result of `format(t)`. The selection jumped to the call with input 20. [P08] said, “I expect this 60 to stay 60 and it doesn’t. It switches to 20.” The jump came from a prototype bug, since fixed: the rule in Section 3.7 would have kept the focus. The reaction shows that stability matters for dynamic context as well as for code: an execution the programmer has chosen should hold while they edit.

Distinguishing a program change from a context change. While editing a growth-stage branch, [P13] saw samples disappear from other branches as the focus shifted to an execution of the branch being edited. This changed which executions were represented, as well as updating values for the edited program. In their survey, [P13] explained the confusion: “I thought it was because of the change I made, but it was actually because of automatic alignment. If I didn’t know about it, I would have probably investigated the change, and would have been left very confused.” The line-break episode above illustrates a prototype bug, but preserving a case also creates a broader tension in the focus policy. During debugging, preserving the selected case supports comparison before and after an edit. During writing, moving focus into newly written code can supply feedback that the previously selected case would not reach. Kanon reports the mirror image of [P13]’s confusion, a stale view misread as a run in which an edit had no effect [58]; both suggest that the editor should show when what is on screen changes, or fails to change, for a reason other than the edit.

5.5 The Gap: Partial Results

Inspecting an unfinished computation. Partial results let [P06] check the accumulating structure of a running-sum solution before implementing the missing arithmetic. With the previous-sum calculation left as a hole, they browsed callback samples showing the current input element and growing list of partial additions (Figure 11 A). [P14] instead left the callback itself as a hole, and the fold’s sample showed it applied to each element in turn (B).

Limits of partial results. Whether partial results proved useful depended on the question being considered. Reflecting on the running-sum task, [P06] explained, “I don’t want to think about the values yet, I’m just thinking about the problem.” When they wanted to inspect a function argument, however, “at that point, I did find it useful.”

Partial results also differed in how readily participants could interpret them. At one point during running sum, [P04] replaced the fold’s initial accumulator with a hole. This prevented the callback from unpacking the (tuple) accumulator, leaving nested anonymous function applications in the sample. They explained, “Once you get past two function calls deep, you start to kind of forget

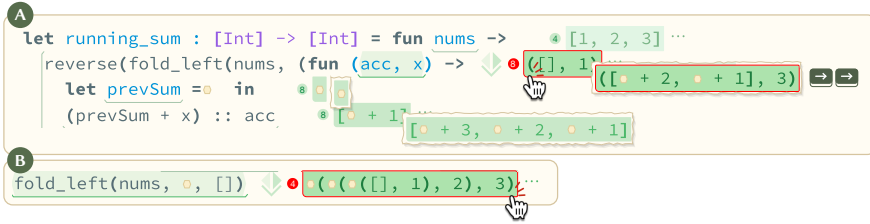


Fig. 11. **P06** browses the callback’s samples without editing the program. They click the first iteration’s sample **A** and press **→** twice to reach the third, whose changed samples are laid over the first’s. Although `prevSum` remains a hole, the selected arguments and partial lists expose what accumulates across iterations. **B** In another session, **P14** left the fold’s callback as a hole: with `[]` as the initial accumulator, the sample shows the missing callback applied to `1`, `2`, and `3` in turn. Reconstructed from the session recordings.

Indirection	Connect values to code	4	5	5	5	6	6	7	7	7	7	7	7	7	7	7
	Keep attention on code	5	5	5	5	5	6	6	6	7	7	7	7	7	7	7
What to show	Autoprobe chose useful expressions	4	4	4	4	5	5	6	6	6	6	6	6	6	7	7
	Line-break steering felt natural	1	2	4	4	5	5	6	6	6	6	6	7	7	7	7
	Focus on relevant samples	3	3	4	5	5	5	5	5	6	6	6	6	6	7	7
	Choose the amount of detail	2	3	3	3	4	4	5	5	6	6	6	6	7	7	7
While writing	Check work incrementally	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7
	Helped more than distracted	5	5	6	6	6	6	7	7	7	7	7	7	7	7	7
Overall	Would use in my own work	2	5	5	6	6	6	6	6	7	7	7	7	7	7	7

Fig. 12. Participant agreement with propositions from the exit survey ($n = 13$). The first two groups bear on the problems of indirection, term selection, multiplicity, and value detail (Section 1); the others concern feedback while writing and overall fit. Each numbered tile is a rating from 1 (strongly disagree) to 7 (strongly agree), sorted independently within each row.

what’s going on.” Where **P06** could inspect a growing list of additions, **P04** had to navigate an unwieldy stack of nested lambda applications.

5.6 Overall Assessment and Limitations

Strong support for live feedback. Our sessions suggest that useful feedback combines ready and direct access to values with control over their selection and presentation, including preserving a chosen execution during investigation. On indirection, comparative ratings for seeing values and connecting them to code favored probes over print statements (13/13 “better” or “much better”) and conventional debuggers (11/13; one respondent corrected both on follow-up, having misread the scale). All but one survey respondent rated connecting values to their code at least 5 on the seven-point agreement scale, as all did for keeping attention on the code. All 13 rated incremental checking 6 or 7, and all rated help over distraction at least 5. Of the features, respondents most often rated very useful the inline value itself (12 of 13), pinning and stepping into calls (10 and 9), and the alignment of samples to one call (8). Assessments of controlling the view and using probes in participants’ own work were more varied (Figure 12); the one low rating for using probes in their own work came from **P01**, who doubted that autoprobe would help with the IO-driven server programs they usually debug.

Reliability and fit with existing practice. First-choice preferences were more divided: six chose probes, six said it depended on the task, one chose a debugger, and none chose print statements. P06 would use probes for “every task I use print statements for now”, with a possible exception for logging, but “only so long as using them is as accessible and reliable as using print statements”.

Limitations. Sessions involved short-term, facilitated use with supplied tasks and tests. Instruction, task-specific help, and prototype bugs limit comparisons. Author presence may have influenced participant reactions. Retrospective explanations do not definitively establish what participants noticed without commenting on it. Longer-term use and later changes remain to be evaluated.

Accessibility. We have not designed or evaluated Live Probes for accessibility. Many mode’s Hybrid scheme separates its relations mainly by hue, and under red-green color vision deficiency two of them converge. We have not examined how inline samples reach screen readers, or tested the keyboard interactions with assistive technology.

6 Related Work

Live programming environments [7, 28, 73, 74] have long shown runtime values beside the code that produced them [23, 77], one of the editor augmentations Sulir et al. survey [72]. Showing values there reduces indirection; Log-it reduces it from the other side, keeping print output in a log but linking each entry back to the code [32]. We compare prior systems by how they handle the other problems of Section 1: Section 6.1 takes term selection, Section 6.2 multiplicity and coherence, Section 6.3 value detail, and Section 6.4 the gap. Throughout, Live Probes aims to make each choice of what to show cheap to change from the code itself, to relate every value on screen to one chosen execution, and to keep showing values while code is incomplete.

6.1 Inline Values and Term Selection

Showing runtime values alongside the code that produced them has a long history. ZStep, a stepper, shows the value of the expression being evaluated in a floating window aligned with it [46]. Example-Centric Programming shows a whole run of an example in a pane beside the editor, updated as the code or the example changes [15], and in Subtext, as in a spreadsheet, the values are part of the program itself: it executes constantly as it is edited, and every reference displays its value [16]. McDirmid’s YinYang made early use of the term *probe* in this context: a programmer marks an expression and its values appear beside it [47].

Later systems differ in how many terms show values without being asked. Theseus shows the least: only how many times each function was called, until the programmer clicks a count to see the calls [45]. In YinYang, Babylonian-style programming, the Newspeak IDE, Light Table’s watches, and a debugger’s inline watches, the programmer marks each term that shows values [6, 21, 31, 68, 70]. Other systems show values everywhere: a value or the variables of every line [40, 71, 78], every subexpression [10], or a plot of every numeric variable [33]. Marking takes a gesture per value. Showing everything takes space, as when Omnicode’s plots grew too wide for the screen in pilot tests once users created many variables [33], and it still leaves gaps where a line holds more than one value. Krämer et al.’s editor showed the values assigned on each line, yet participants still added statements whose only job was to show a value, much like print statements. The most frequent cause was a line such as `arr.reverse().push(5)`, for which the editor showed nothing because it could not tell whether to show `arr`, the result of `arr.reverse()`, or the value of the whole line [40]. Hoffswell et al. likewise note that a value at the end of a line is ambiguous when the line holds several terms [27]. Like the Quokka editor plugin, which shows the values the programmer marks by default and can show one for every line [67], Live Probes offers both approaches: one gesture probes a single term or every line of a definition, and with autoprobing turned on, every line of the program shows a value.

Projection Boxes is the closest to our framing: beside each line it shows a configurable projection of the program's runtime state, a table of the program's variables at that line with one row for each time the line ran [43]. A projection can choose which lines show boxes and which variables each box shows, but a box shows variables, so a value appears only once the code assigns it to one. Its author names functional languages as a direction for future work, with where to place boxes as a likely challenge. A line of functional code may assign no variables and hold several nested expressions, so Live Probes places probes on terms rather than variables, and a probe's underline shows which term its value belongs to (Section 3.1). Autoprobing gives a quick default, one probe per line on the largest term that ends last, steerable with line breaks (Section 3.2). Krämer et al.'s editor could not choose among the values of a chain of calls. With autoprobing, a line break before each step (in Hazel, before each `|>`) is enough: each line's probe shows that step's result, as with the fold in Figure 4. Where the default picks the wrong term, a probe placed by hand replaces it.

6.2 Alignment and the Dynamic Focus

When code runs many times, any display of runtime values must choose which execution's values to show, and keep that choice consistent across all the values it shows. A paused debugger achieves this for free. Everything shown comes from the paused moment, with the variables pane and any inline values following the selected stack frame [51]. Selecting a caller in the call-stack pane moves the view up a level but leaves the frames below it in place, because execution is suspended in them [52]. But a recorded run lacks a distinguished point in execution, forcing a choice. Omniscient debuggers choose a moment and show the program's state at that moment [4, 30, 44, 46, 53, 66]. Pernosco, a commercial omniscient debugger, calls this choice its *focus*: a moment plus a stack frame [64]. Inline displays mostly choose a *dynamic extent* instead, the stretch of a run during which one call or loop iteration is active, and show the values computed anywhere in that stretch.

Inline displays differ in how much of the program one selection covers. Seymour shows one call and leaves the bodies of other functions blank, including the function that called it [34]. Lamdu shows one call of each function, with a function written inside another limited to calls within the outer one's chosen call [10, 11]. Maniposynth likewise keeps one chosen call per function, picked from a grid of that function's calls [26]. The Babylonian editor runs code on examples, concrete calls the programmer attaches to functions, and by default its probes show every example's values, each labeled with its example; a slider narrows the probes inside a loop or function to one iteration or call, leaving probes elsewhere unfiltered [68]. Example-Centric Programming also works from examples, and one call selected in its trace pane drives the whole editor [15]. Projection Boxes makes no selection: every box shows every time its line ran, and the tool offers no way to narrow boxes to chosen iterations [43]. In YinYang the choices are chained: each method body shows the call through which the programmer navigated to it, and bodies not navigated into show nothing [47]. The dynamic focus is instead one selection for the whole program: a path of nested calls that every probe follows (Section 3.3). A paused debugger's call stack is also a path of nested calls, but each caller on it is stopped where it made the next call. In the dynamic focus, the probes in each function called on the path show that call's samples, including those recorded after it made the next call.

Systems also differ in how the programmer reaches an execution, and each change of selection takes work. YinYang moves down one call per click, or jumps to the execution behind an entry in a trace of printed output [47]. Lamdu and the Babylonian editor step a separate control for each function or loop. Example-Centric Programming selects in a trace pane beside the code, where reaching a nested call takes one click per level, and Seymour in an overview of the run below the code, where any call is one click away. RuntimeSamp follows the caret [71] (Section 6.4). Omniscient debuggers step through time or select an item in one of their panes; in Pernosco, clicking a line of the program's output moves the focus to the moment that line was printed [64]. In Live Probes, the

samples are the controls. In Single mode, where each probe shows one sample, the arrow keys step a probe through its samples, as Lamdu's arrows do, but each step moves the whole path, so every probe follows. In Many mode, where a probe shows its samples side by side, clicking any of them makes its call stack the path, however deep the call. Clicking a sample of an enclosing call moves the focus up to that call but keeps the deeper calls on the path, so the probes below keep their samples and stepping back down returns to them. YinYang's author left open how to show which execution the code is displaying [47]. In Live Probes, the stack focus bar names the calls on the path, a probe on a function's parameters shows which call of it is shown, and the focal sample is outlined in red (Section 3.4).

In Many mode, the Hybrid color scheme also shows how every other sample relates to the focal sample: from the same call, before it, after it, from a call containing it, or from inside it (Section 3.3). So when a call site is on screen, its samples show whether the focus lies inside that call. CrossCode, the closest precedent, relates whole calls rather than values: in a visualizer beside the code, each call is a box marked as not yet started, running, or finished at one execution cursor [25]. Pernosco highlights the lines run in the current loop iteration and, more lightly, those run in the call's other iterations, much as our Simple scheme highlights the focused call's samples and fades the rest [65]. Babylonian/S shows which chain of calls produced each value, with a bar under the value colored by its path, and lists the paths in a sidebar [69]. Our colors instead relate each value to the chosen path.

6.3 Value Detail and Rich Views

Earlier work has raised the question of how to show a large value in the little space beside the code: RuntimeSamp settles on each object's string form, truncated [71], and Projection Boxes' author expects larger data to need partial previews or aggregate plots [43]. Most systems that show values beside the code show a short form there, and a fuller form only on request, over the code or apart from it. Babylonian probes print an object on one line, and clicking it opens an inspector [68]; debuggers' inline values similarly leave larger values to a hover, a popup, or the variables pane [31, 51]. Hoffswell et al.'s in situ visualizations show one property of an object inline, and the whole object on mouse-over [27]. Lamdu cuts each annotation by fixed rules, an array to ten items and a list to its first, and offers no fuller view [10, 11]. Omnicode plots only numbers, deriving a collection's size and, for a collection of numbers, its sum; selecting a line opens Python Tutor's drawing of the stack and heap as boxes and arrows, at each step where the line ran [22, 33]. Notebooks show a cell's result below the cell, where there is room to show it at length [38], and Clerk shows the first items of a large collection there with a count of the rest, loading more on request [35]. Live Probes gives every probed term three levels without leaving the code: an abbreviation that keeps the value's outer structure and counts what it leaves out, a drawer that shows the focused sample in full below its line, pushing the code below it down, and a dropdown with the sample's environment (Section 3.5). The closest analog to the drawer is the result view Xcode playgrounds open below a line, which shows the line's latest value, all its values, or a graph [2]. Both ensure large values stay beside the expressions that computed them, rather than after a whole cell (as in a notebook), in a hover or popup covering code, or in a separate pane the programmer must match back to the code (Section 1).

Programmer-defined views of runtime values are long established. INCENSE chooses a display by a variable's declared type [56], debugger formatters match a type's name [18, 50], and notebooks look for display methods on a value's class [75] or take the first viewer whose predicate accepts the value [35]. Glamorous Toolkit's moldable development goes furthest: a class defines views of its objects in a few lines each, which the inspector offers as tabs and the programmer can edit from inside it [9, 57], and a debugger for a domain can add its own views and steps, such as running a parser until it next fails [8]. These views appear in panes or outputs apart from the code, each drawing a value the programmer has chosen to inspect. Hazel Lab draws probed tables beside the code, with a built-in view chosen by the value's shape, and leaves views that programmers can define to future work [3].

Other systems embed GUIs that edit programs [1, 29, 37]; in Lorgnette, a programmer-defined projection attaches a GUI to each fragment of code that matches a pattern, such as a color literal, and can ask the debugger for a value where its fragment runs [19]. Vital, like CLIM before it [49], goes both ways: a Haskell datatype’s stylesheet draws its values, and editing a drawn value rewrites the source [24]. Livelits are GUIs that programmers use as literals, filling typed holes [59]. We extend livelits to serve as views, which makes Hazel Lab’s views programmer-definable (Section 3.6); a view appears where a sample does, at the end of the line or in the drawer below it. A view applies automatically where the probed term’s static type is the livelit’s type, by name, and the livelit is in scope, so it draws only values the program has given that type, and a nearer definition shadows an outer one; the sample’s *View as* list offers, on request, every view whose type is consistent with the term’s.

6.4 Feedback While Editing

Hazel gives every editor state a result, so probes keep showing values while code is incomplete (Section 2); Maniposynth similarly keeps evaluating past holes by passing placeholder values along [26]. When code cannot run, each projection box keeps its last values, and a dot on the box turns from green to orange, then red, to show they are out of date [43]. YinYang’s author found its feedback unstable while a construct was half typed [47]. Our stability rules keep an incomplete edit from changing the meaning of the code around it while it is typed (Section 3.7). Asked whether values changed in distracting ways, only one of 13 participants named changes while editing, and all 13 agreed that values appearing as they wrote helped more than distracted (Figure 12).

Systems that keep the programmer’s place across edits recognize the same execution in the new run much as we do, by a location in the code and the calls leading to it [15, 47, 48, 58]. Kanon follows a location by shifting its stored text offsets as characters are inserted and deleted, and loses track of it on cut and paste or undo [58]; Hazel’s structure editor keeps a term’s identity while it is edited, and undo restores the program with its earlier identities (Section 3.7). After an edit, most of these systems hold the selected execution: YinYang keeps a call or iteration while it still exists, Seymour keeps its call, and Lamdu keeps each function’s chosen call by its number in the run, falling back to the first call when no call has that number [10, 11, 34, 47]. Kanon keeps the call or iteration the programmer chose, and the caret picks only which statement’s state it draws [58]. RuntimeSamp instead shows, for the caret’s line, an execution in which that line ran; it re-picks at every caret move, so it never holds a selection, and it discards its data on any edit [71]. Holding still leaves the programmer to refocus by hand on code the held call does not reach, and following the caret gives up the selected case: the tension of Section 5.4. Projection Boxes’ formative study saw programmers alternate between coding and debugging, wanting less information while coding and more while debugging [43]; our sessions suggest that the alternation also bears on the selection, which should follow the programmer while writing and hold still while debugging. Our editor holds the focus, with one exception that shares RuntimeSamp’s aim: when an edit would leave the edited line showing no sample, it moves the focus so that the line shows one. Otherwise the focus moves only when the programmer clicks a sample, and never under a pin (Section 3.7).

7 Conclusion

Live Probes shows runtime samples beside the code that produced them, with inline control over which terms are probed, which of their executions are shown, and how much of each sample is shown. A dynamic focus, set by clicking any sample, relates every sample on screen to one execution, and feedback stays steady through incomplete edits. In our study, participants used inline samples to judge progress and correctness. The study also exposed a tension between a focus that moves to supply values while writing and one that holds still while debugging.

Data-Availability Statement

Our artifact is the implementation of Live Probes described here: a fork of the open-source Hazel editor, which runs in a web browser (link removed for double-blind review; its source and a compiled web client are included as supplementary material). It includes a version of the probes tutorial that study participants used, and examples of user-defined views, among them those shown in Figure 2. It is a later build than the one used in the study, with the fixes and changes made since. We will submit it for Artifact Evaluation. Recordings and transcripts are not shared, pursuant to the participant agreement.

References

- [1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding interactive visual syntax to textual code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 222 (Nov. 2020), 28 pages. doi:10.1145/3428290
- [2] Apple. [n. d.]. View Results of an Executed Statement. Xcode 10.2 Help. <https://help.apple.com/xcode/mac/10.2/en.lproj/devab506297c.html>
- [3] Alexander Bandukwala and Cyrus Omar. 2026. Interactive Data Analysis with Lively Typed Tables. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 369 (Oct. 2026), 29 pages. doi:10.1145/3839501
- [4] Earl T. Barr and Mark Marron. 2014. Tardis: affordable time-travel debugging in managed runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) (OOPSLA '14). Association for Computing Machinery, New York, NY, USA, 67–82. doi:10.1145/2660193.2660209
- [5] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) (ICSE '18). ACM, New York, NY, USA, 572–583. doi:10.1145/3180155.3180175
- [6] Gilad Bracha. 2021. Enhancing Liveness with Exemplars in the Newspeak IDE. In *Proceedings of the Seventh Workshop on Live Programming (LIVE 2021)*. <https://liveprog.org/live-2021/Enhancing-Liveness-with-Exemplars-in-the-Newspeak-IDE/>
- [7] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. 2013. It's alive! continuous feedback in UI programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) (PLDI '13). Association for Computing Machinery, New York, NY, USA, 95–104. doi:10.1145/2491956.2462170
- [8] Andrei Chiş, Tudor Girba, and Oscar Nierstrasz. 2014. The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In *Software Language Engineering*. Springer International Publishing, 102–121. doi:10.1007/978-3-319-11245-9_6
- [9] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Girba. 2015. The moldable inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA) (Onward! 2015). Association for Computing Machinery, New York, NY, USA, 44–60. doi:10.1145/2814228.2814234
- [10] Yair Chuchem and Eyal Lotem. 2018. Scaling the REPL Experience. Talk on Lamdu (<https://github.com/lamdu/lamdu>) at the Workshop on Live Programming (LIVE 2018). <https://2018.splashcon.org/details/live-2018-papers/7/Scaling-the-REPL-Experience>
- [11] Yair Chuchem and Eyal Lotem. 2026. Lamdu. Source code repository, commit 38ef56a. <https://github.com/lamdu/lamdu>
- [12] CIDER. 2026. Debugger. CIDER documentation. <https://docs.cider.mx/cider/debugging/debugger.html> Accessed 14 October 2026.
- [13] Will Crichton, Maneesh Agrawala, and Pat Hanrahan. 2021. The Role of Working Memory in Program Tracing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) (CHI '21). Association for Computing Machinery, New York, NY, USA, Article 56, 13 pages. doi:10.1145/3411764.3445257
- [14] Evan Czaplicki. 2016. The Elm Architecture. An Introduction to Elm. Retrieved September 28, 2026 from <https://guide.elm-lang.org/architecture/>
- [15] Jonathan Edwards. 2004. Example centric programming. *SIGPLAN Not.* 39, 12 (Dec. 2004), 84–91. doi:10.1145/1052883.1052894
- [16] Jonathan Edwards. 2005. Subtext: uncovering the simplicity of programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) (OOPSLA '05). Association for Computing Machinery, New York, NY, USA, 505–518. doi:10.1145/1094811.1094851
- [17] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: interactive program synthesis with control structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 153 (Oct. 2021), 29 pages. doi:10.1145/3485530
- [18] Free Software Foundation. 2026. Pretty Printing. In *Debugging with GDB: the GNU Source-Level Debugger*, Tenth Edition,

- for GDB Version 19.0.50.20261002-git. <https://sourceware.org/gdb/current/onlinedocs/gdb.html/Pretty-Printing.html>
Accessed 2 October 2026.
- [19] Camille Gobert and Michel Beaudouin-Lafon. 2023. Lorgnette: Creating Malleable Code Projections. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) (UIST '23). Association for Computing Machinery, New York, NY, USA, Article 71, 16 pages. doi:10.1145/3586183.3606817
 - [20] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.
 - [21] Chris Granger. 2013. Light Table 0.5.0. <https://chris-granger.com/2013/08/22/light-table-050/>
 - [22] Philip J. Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) (SIGCSE '13). Association for Computing Machinery, New York, NY, USA, 579–584. doi:10.1145/2445196.2445368
 - [23] Christopher Michael Hancock. 2003. *Real-Time Programming and the Big Ideas of Computational Literacy*. Ph. D. Dissertation. Massachusetts Institute of Technology. <http://hdl.handle.net/1721.1/61549>
 - [24] Keith Hanna. 2002. Interactive visual functional programming. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (Pittsburgh, PA, USA) (ICFP '02). Association for Computing Machinery, New York, NY, USA, 145–156. doi:10.1145/581478.581493
 - [25] Devamardeep Hayatpur, Daniel Wigdor, and Haijun Xia. 2023. CrossCode: Multi-level Visualization of Program Execution. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) (CHI '23). Association for Computing Machinery, New York, NY, USA, Article 593, 13 pages. doi:10.1145/3544548.3581390
 - [26] Brian Hempel and Ravi Chugh. 2022. Maniposynth: Bimodal Tangible Functional Programming. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)* (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222), Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:29. doi:10.4230/LIPIcs.ECOOP.2022.16
 - [27] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) (CHI '18). Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3173574.3174106
 - [28] Joshua Horowitz. 2026. Technical Dimensions of Live Feedback in Programming Systems. In *PLATEAU Workshop*. doi:10.1184/R1/31814011
 - [29] Joshua Horowitz and Jeffrey Heer. 2023. Engraff: An API for Live, Rich, and Composable Programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) (UIST '23). Association for Computing Machinery, New York, NY, USA, Article 72, 18 pages. doi:10.1145/3586183.3606733
 - [30] Michael James. 2014. Time Travel Made Easy. <https://elm-lang.org/news/time-travel-made-easy>.
 - [31] JetBrains. 2026. Examine suspended program. IntelliJ IDEA 2026.2 documentation. <https://www.jetbrains.com/help/idea/examining-suspended-program.html> Accessed 11 October 2026.
 - [32] Peiling Jiang, Fuling Sun, and Haijun Xia. 2023. Log-it: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) (CHI '23). Association for Computing Machinery, New York, NY, USA, Article 594, 16 pages. doi:10.1145/3544548.3581403
 - [33] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) (UIST '17). Association for Computing Machinery, New York, NY, USA, 737–745. doi:10.1145/3126594.3126632
 - [34] Saketh Kasibatla and Alex Warth. 2017. Seymour: Live Programming for the Classroom. Web essay presented at the Workshop on Live Programming (LIVE 2017). <https://harc.github.io/seymour-live2017/>
 - [35] Martin Kavalár, Philippa Markovics, and Jack Rusher. 2023. Clerk: Moldable Live Programming for Clojure. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Tokyo, Japan) (Programming '23). Association for Computing Machinery, New York, NY, USA, 22–31. doi:10.1145/3594671.3594682
 - [36] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) (CHI '18). Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3173574.3173748
 - [37] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) (UIST '20). Association for Computing Machinery, New York, NY, USA, 140–151. doi:10.1145/3379337.3415842
 - [38] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and

- Jupyter Development Team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Schmidt (Eds.). IOS Press, 87–90. doi:10.3233/978-1-61499-649-1-87
- [39] Minhyuk Ko, Omer Ahmed, Yoseph Berhanu Alebachew, and Chris Brown. 2025. AutoPrint: Judging the Effectiveness of An Automatic Print Statement Debugging Tool. In *2025 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 379–384. doi:10.1109/VL-HCC65237.2025.00049
- [40] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How live coding affects developers' coding behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 5–8. doi:10.1109/VLHCC.2014.6883013
- [41] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The road to live programming: insights from the practice. In *Proceedings of the 40th International Conference on Software Engineering (Gothenburg, Sweden) (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1090–1101. doi:10.1145/3180155.3180200
- [42] Sorin Lerner. 2020. Focused Live Programming with Loop Seeds. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (Virtual Event, USA) (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 607–613. doi:10.1145/3379337.3415834
- [43] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proc. 2020 CHI Conference on Human Factors in Computing Systems*. ACM, 1–7. doi:10.1145/3313831.3376494
- [44] Bil Lewis. 2003. Debugging Backwards in Time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, M. Ronsse and K. De Bosschere (Eds.). Ghent, Belgium. arXiv:cs/0310016 doi:10.48550/arXiv.cs/0310016
- [45] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Toronto, Ontario, Canada) (CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2481–2490. doi:10.1145/2556288.2557409
- [46] Henry Lieberman and Christopher Fry. 1998. ZStep 95: A Reversible, Animated Source Code Stepper. In *Software Visualization: Programming as a Multimedia Experience*, John T. Stasko, John B. Domingue, Marc H. Brown, and Blaine A. Price (Eds.). MIT Press. <https://web.media.mit.edu/~lieber/Lieberyary/ZStep/ZStep-SoftViz/ZStep-SoftViz.html>
- [47] Sean McDirmid. 2013. Usable live programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Indianapolis, Indiana, USA) (Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 53–62. doi:10.1145/2509578.2509585
- [48] Sean McDirmid and Jonathan Edwards. 2014. Programming with Managed Time. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Portland, Oregon, USA) (Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/2661136.2661145
- [49] Scott McKay. 1991. CLIM: the Common Lisp interface manager. *Commun. ACM* 34, 9 (Sept. 1991), 58–59. doi:10.1145/114669.114675
- [50] Microsoft. 2025. Create custom views of C++ objects. Visual Studio documentation, Microsoft Learn. <https://learn.microsoft.com/en-us/visualstudio/debugger/create-custom-views-of-native-objects> Accessed 2 October 2026.
- [51] Microsoft. 2026. Debug code with Visual Studio Code. Visual Studio Code documentation. <https://code.visualstudio.com/docs/debugtest/debugging> Accessed 11 October 2026.
- [52] Microsoft. 2026. View the call stack and use the Call Stack window in the debugger. Visual Studio documentation, Microsoft Learn. <https://learn.microsoft.com/en-us/visualstudio/debugger/how-to-use-the-call-stack-window> Accessed 11 October 2026.
- [53] Juan Monetta. 2026. FlowStorm debugger User's Guide. https://flow-storm.github.io/flow-storm-debugger/user_guide.html Accessed 11 October 2026.
- [54] David Moon, Andrew Blinn, and Cyrus Omar. 2022. tyrl: a tiny tile-based structure editor. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development (Ljubljana, Slovenia) (TyDe 2022)*. Association for Computing Machinery, New York, NY, USA, 28–37. doi:10.1145/3546196.3550164
- [55] David Moon, Andrew Blinn, Thomas J. Porter, and Cyrus Omar. 2025. Syntactic Completions with Material Obligations. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 404 (2025), 3669–3695 pages. doi:10.1145/3763182
- [56] Brad A. Myers. 1983. INCENSE: A system for displaying data structures. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques (Detroit, Michigan, USA) (SIGGRAPH '83)*. Association for Computing Machinery, New York, NY, USA, 115–125. doi:10.1145/800059.801140
- [57] Oscar Marius Nierstrasz and Tudor Girba. 2024. Moldable Development Patterns. In *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices (EuroPLOP '24)*. Association for Computing Machinery, New York, NY, USA, Article 18, 14 pages. doi:10.1145/3698322.3698327
- [58] Akio Oka, Hidehiko Masuhara, and Tomoyuki Aotani. 2018. Live, synchronized, and mental map preserving visualization for data structure programming. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Boston, MA, USA) (Onward! '18)*. ACM, 72–87.

- doi:10.1145/3276954.3276962
- [59] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proc. 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM, 511–525. doi:10.1145/3453483.3454059
- [60] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL (2019), 1–32. doi:10.1145/3290327
- [61] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proc. 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 86–99. doi:10.1145/3009837.3009900
- [62] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *Summit on Advances in Programming Languages (SNAPL) (LIPIcs, Vol. 71)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 11:1–11:12. doi:10.4230/LIPIcs.SNAPL.2017.11
- [63] Roly Perera. 2019. The meaning of a program change is a change to the program’s meaning. *CoRR* abs/1908.00898 (2019). arXiv:1908.00898 <http://arxiv.org/abs/1908.00898> Presented at the Second Workshop on Incremental Computing (IC 2019) at SPLASH 2019.
- [64] Pernosco. [n. d.]. Basic interface. Pernosco documentation. <https://pernosco.co/about/basics/> Accessed 11 October 2026.
- [65] Pernosco. 2026. Visualizing control flow. Pernosco documentation. <https://pernosco.co/about/control-flow/> Accessed 13 October 2026.
- [66] Guillaume Pothier, Éric Tanter, and José Piquet. 2007. Scalable omniscient debugging. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) (*OOPSLA ’07*). Association for Computing Machinery, New York, NY, USA, 535–552. doi:10.1145/1297027.1297067
- [67] Quokka.js. [n. d.]. Advanced Logging. Quokka.js documentation. <https://quokkajs.com/docs/features/advanced-logging/> Accessed 11 October 2026.
- [68] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (2019), 9:1–9:39. doi:10.22152/programming-journal.org/2019/3/9
- [69] Patrick Rein, Christian Flach, Stefan Ramson, Eva Krebs, and Robert Hirschfeld. 2024. Broadening the View of Live Programmers: Integrating a Cross-Cutting Perspective on Run-Time Behavior into a Live Programming Environment. *The Art, Science, and Engineering of Programming* 8, 3, Article 13 (2024), 38 pages. doi:10.22152/programming-journal.org/2024/8/13
- [70] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-based Live Programming as a Use Case for Context-oriented Programming. In *Proceedings of the 11th ACM International Workshop on Context-Oriented Programming* (London, United Kingdom) (*COP ’19*). Association for Computing Machinery, New York, NY, USA, 17–23. doi:10.1145/3340671.3343358
- [71] Matúš Sulír and Jaroslav Porubán. 2018. Augmenting source code lines with sample variable values. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) (*ICPC ’18*). Association for Computing Machinery, New York, NY, USA, 344–347. doi:10.1145/3196321.3196364
- [72] Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubán. 2018. Visual augmentation of source code editors: A systematic mapping study. *Journal of Visual Languages & Computing* 49 (Dec. 2018), 46–59. doi:10.1016/j.jvlc.2018.10.001
- [73] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *J. Vis. Lang. Comput.* 1, 2 (June 1990), 127–139. doi:10.1016/S1045-926X(05)80012-6
- [74] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. IEEE, 31–34. doi:10.1109/live.2013.6617346
- [75] The IPython Development Team. 2026. Integrating your objects with IPython. IPython 9.17.1 documentation. <https://ipython.readthedocs.io/en/stable/config/integrating.html> Section “Rich display”. Accessed 2 October 2026.
- [76] Edward R. Tufte. 1990. *Envisioning Information*. Graphics Press, Cheshire, CT.
- [77] Bret Victor. 2012. Inventing on Principle. <https://worrydream.com/InventingOnPrinciple/> CUSEC talk, January 20, 2012. Video recording.
- [78] Bret Victor. 2012. Learnable Programming. Web essay. <https://worrydream.com/LearnableProgramming/>