Filling Typed Holes with Live GUIs

Cyrus Omar
University of Michigan
Ann Arbor, MI, USA
comar@umich.edu

David Moon
University of Michigan
Ann Arbor, MI, USA
dmoo@umich.edu

Andrew Blinn
University of Michigan
Ann Arbor, MI, USA
blinnand@umich.edu

Ian Voysey
Carnegie Mellon University
Pittsburgh, PA, USA
iev@cs.cmu.edu

Nick Collins
University of Chicago
Chicago, IL, USA
nickmc@uchicago.edu

Ravi Chugh
University of Chicago
Chicago, IL, USA
rchugh@cs.uchicago.edu

Abstract

Text editing is powerful, but some types of expressions are more naturally represented and manipulated graphically. Examples include expressions that compute colors, music, animations, tabular data, plots, diagrams, and other domain-specific data structures. This paper introduces *live literals*, or *livelits*, which allow clients to fill holes of types like these by directly manipulating a user-defined GUI embedded persistently into code. Uniquely, livelits are *compositional*: a livelit GUI can itself embed spliced expressions, which are typed, lexically scoped, and can in turn embed other livelits. Livelits are also uniquely *live*: a livelit can provide continuous feedback about the run-time implications of the client's choices even when splices mention bound variables, because the system continuously gathers closures associated with the hole that the livelit is filling. We integrate livelits into Hazel, a live hole-driven programming environment, and describe case studies that exercise these novel capabilities. We then define a simply typed livelit calculus, which specifies how livelits operate as live graphical macros. The metatheory of macro expansion has been mechanized in Agda.

CCS Concepts: • **Software and its engineering → General programming languages**.

Keywords: live programming, macros, typed holes, GUIs

ACM Reference Format:
Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454059

PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454059

1 Introduction

Text-based program editors are flexible and powerful user interfaces, so it is little wonder that they remain dominant decades after the teletype. However, textual user interfaces are not the best tool for every computational job.

Consider, as a simple example, a record type classifying RGBA-encoded colors. It is possible to select a particular color by entering an expression of this type in a text editor, e.g. `{ r: 57, g: 107, b: 57, a: 92 }`. The problem with this *de facto* textual user interface for color selection is that it offers no live feedback about which color has been expressed and limited editing affordances for choosing a different color. Analagous critiques apply to strictly textual user interfaces for countless other data structures, e.g. audio filters, vector graphics, board game states, tabular numeric data, geospatial data, neurobiological circuits, and mathematical diagrams.

Practitioners in domains where manipulating data of types like these is a central activity have largely eschewed general-purpose programming environments in favor of specialized graphical end-user applications, e.g. image and video editors, music composition software, level design tools, and bespoke GUIs written by lab technicians. This is in large part because these applications take seriously the need for live feedback, domain-specific non-textual data representations, and direct manipulation affordances, e.g. checkboxes, sliders, color palettes, visual timelines, interactive plots, and maps.

The tragedy is that these applications have limited support for abstraction and composition. It is difficult, for example, to bind a color to a variable for use in multiple locations in an otherwise directly constructed game map, or to define functional combinators to compute portions of an otherwise directly constructed musical composition. Some applications include *ad hoc* abstraction mechanisms for the most common such use cases, e.g. named color swatches, but users cannot themselves define new affordances, either programmatic or graphical, nor compose affordances in ways that the application developer did not anticipate. For example, users cannot make even simple changes like replacing a numeric text box with a slider, much less more ambitious changes like importing an alternative visual interface for expressing geospatial data queries into a civic database front-end.

This paper aims to bridge the gap between programmatic and direct manipulation user interfaces by designing a programming environment that is able to surface a GUI when entering an expression of a type for which it is useful, while retaining full support for symbolic program manipulation and the abstraction and composition mechanisms available in modern general-purpose languages, both in the spaces between these GUIs and compositionally within these GUIs.

1.1 Background

We are not the first to integrate GUIs into programming. The prior work most relevant to this paper is the Graphite system for Eclipse for Java, demonstrated in Fig. 1a [37]. Graphite allows a library provider to associate a GUI, called a *palette*, with a type (via a Java class annotation), here `Color`. Wherever an expression of this type is needed, i.e. wherever there is a *hole* of this type (as determined by Eclipse's online parser and typechecker), the environment offers the client the option, via the code completion menu, to generate it using the palette. Once the user presses `Enter` to indicate that the interaction is finished, the palette generates a Java expression to fill the hole, here `new Color(57, 107, 57, 235)`. Several other systems, such as the **mage** system for Jupyter notebooks [24] behave analagously. Projectional editing, the visual macro system in Racket [3], and a number of other designs also confront this general problem of integrating GUIs with symbolic code in broadly similar ways.

Omar et al. [37] evaluated Graphite by surveying 473 developers and Kery et al. [24] evaluated **mage** by interviewing 9 developers. Both studies found that participants viewed the proposed mechanism favorably and would use a suitable GUI some or all of the time. This and other prior work also collectively showcase a wide variety of use cases [3, 24, 37], and the Graphite survey solicited dozens of additional use cases from participants, which the authors systematically taxonomize [37]. We take these extensive empirical findings as evidence for, and a showcase of, the value of this broad class of mechanisms for integrating GUIs with code.

1.2 Contributions

We turn our attention in this paper to a number of fundamental technical deficiencies that limit both GUI providers and clients using these prior mechanisms. To address these, we introduce a system of *live literals*, or *livelits*, demonstrated in Fig. 1b. Livelits are unique in achieving all of the following properties. (Sec. 6 describes which subset of these are achieved by prior systems, including those just mentioned.)

Decentralized Extensibility. Providers define livelits in libraries. Clients invoke livelits by name. Livelit names, e.g. `$color`, are prefixed by `$` (pronounced "lit"), to distinguish them from variables. We call this *decentralized extensibility* to distinguish it from systems that are not extensible or that can only be extended via editor extensions or editor generation.

Persistence. Livelit invocations are expressions, i.e. they are persistent elements of the syntax tree. They operate as graphical literals, rather than as the ephemeral code generation GUIs of Graphite and **mage**. We define a pure model-view-update-expand architecture (a variation on Elm's model-view-update architecture [11]) where only the model needs to be persisted. The dynamic meaning of a livelit is determined by a macro expansion step.

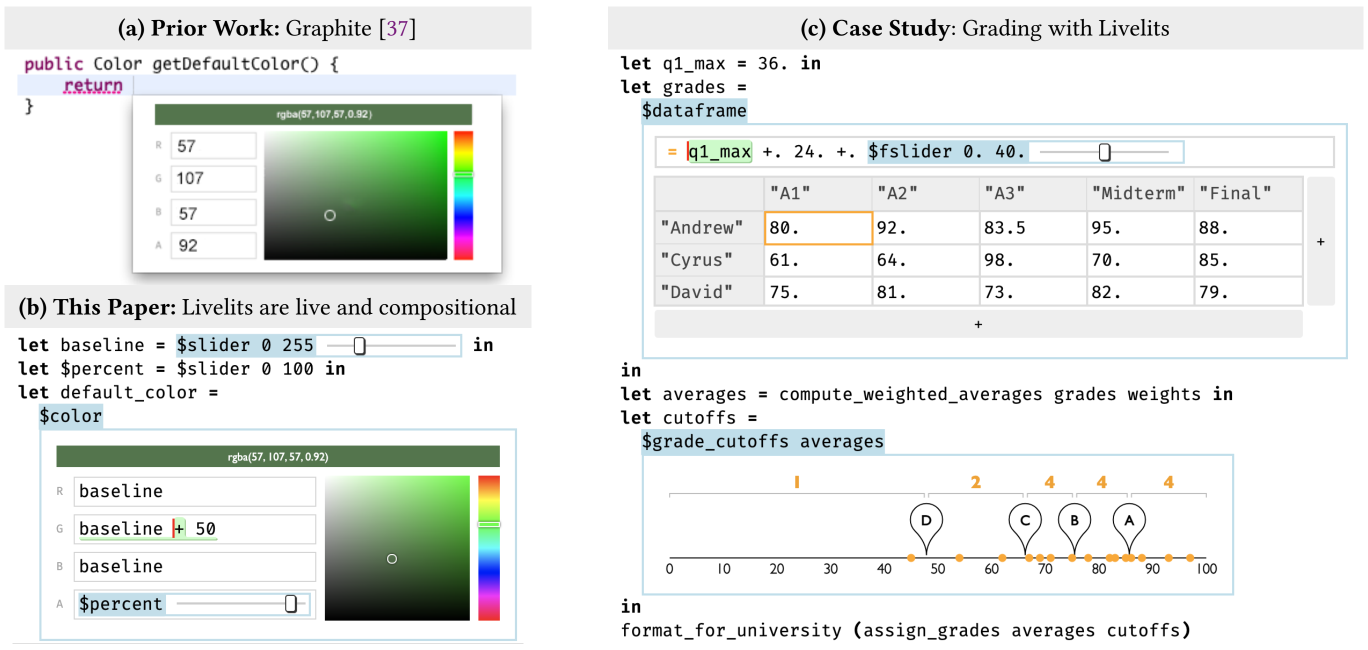
Compositionality. Livelit GUIs can embed sub-expressions, which we call *splices* (after Omar and Aldrich [33]). Fig. 1b demonstrates splicing: the RGBA components are each splice editors, so the client can define a variable, `baseline`, to relate the color components (here, to explore greens by offsetting the green component past the baseline) and use a slider livelit inline to specify the alpha component.

Crucially, composition is governed by a binding discipline that ensures (1) **capture avoidance**, i.e. that variables that the client uses in splices, like `baseline`, are lexically scoped to the livelit invocation site (so they cannot inadvertently capture expansion-internal bindings, which can therefore be left abstract); and (2) **context independence**, i.e. that the livelit can be invoked and operate in any lexical context (so the client need not worry about naming conflicts or manage hidden library dependencies). This binding discipline is a form of "macro hygiene" [2, 9, 33], though note that we take a particularly restrictive approach to hygiene compared to, for example, the Racket macro system [14]: new binding and control flow constructs intentionally cannot be expressed to allow clients to understand splices as function parameters (and indeed this is how they are internally handled).

Parameterization. Livelits can also take parameters directly, forming parameterized families. For example, `$slider` in Fig. 1b is parameterized by the slider's bounds. Parameters operate like splices, differing in that they can be partially applied in livelit abbreviations. For example, Fig. 1b partially applies `$slider` to `0` and `100` to define a `$percent` slider.

Typing. Each livelit must specify an expansion type, and parameters and splices must also specify types, so livelits are compatible with type-driven development. Together with the binding discipline, this allow clients to reason abstractly, i.e. without inspecting the expansion or livelit implementation.

Liveness. Uniquely, livelits can evaluate splices throughout the editing process (i.e. in a *live* manner [45]) to provide feedback related to run-time behavior. For example, in Fig. 1b, displaying the selected color requires evaluating the RGBA component splices to numeric values. Evaluation occurs in a run-time environment (i.e. closure) determined by leaving the hole being filled by the livelit temporarily unfilled and then evaluating using a two-phased variant of the semantics for Hazelnut Live [35]. Live evaluation is supported even for livelits that appear inside a function: multiple function calls can lead to multiple closures that the client selects between.


(a) Prior Work: Graphite [37]

```
public Color getDefaultColor() {
    return
}
```

(b) This Paper: Livelits are live and compositional

```
let baseline = $slider 0 255          in
let $percent = $slider 0 100 in
let default_color =
  $color
```

(c) Case Study: Grading with Livelits

```
let q1_max = 36. in
let grades =
  $dataframe
```

```
= q1_max +. 24. +. $fslider 0. 40.
```

	"A1"	"A2"	"A3"	"Midterm"	"Final"
"Andrew"	80.	92.	83.5	95.	88.
"Cyrus"	61.	64.	98.	70.	85.
"David"	75.	81.	73.	82.	79.

```
in
let averages = compute_weighted_averages grades weights in
let cutoffs =
  $grade_cutoffs averages
```

```
in
format_for_university (assign_grades averages cutoffs)
```

Figure 1. Introductory Examples

Outline. Sec. 2 introduces livelits from the perspective of client programmers. Sec. 3 then considers the livelit provider's perspective by introducing livelit definitions with a detailed example. Sec. 4 defines the *typed livelit calculus*. We have mechanically specified the central mechanism, livelit expansion, and proven the associated metatheorems in Agda. This calculus serves to capture the essential nature of livelits independent of the particularities of syntax, GUI frameworks, and other orthogonal design details, because we believe livelits can be integrated into a wide variety of programming systems. Sec. 5 provides a more detailed account of our implementation of livelits. Our primary implementation, used in the screenshots in the paper, is integrated into Hazel, a live functional programming environment designed around hole-driven development. We have also prototyped livelit editing within a standard text editor. Additionally, we discuss factors that must be considered when integrating livelits into languages with side effects. Sec. 6 compares livelits to related work using the design properties outlined above as a rubric. We conclude in Sec. 7 after a discussion of present limitations and several directions for future work.

2 Livelits by Example

In this section, we will detail the livelits mechanism by way of two domain-specific case studies: a course grade assignment case study in Sec. 2.1 and an image transformation case study in Sec. 2.5.3. These case studies have been implemented in Hazel, a browser-based live programming environment for a variant of Elm [10]. Elm is an industrial pure typed functional language in the ML family used for client-side web development. We assume basic familiarity with ML.

2.1 Case Study: Grading with Livelits

Consider this familiar scenario: an instructor needs (1) to record numeric grades for various assignments and exams, and (2) to visualize and perform various computations with these numeric grades in order ultimately to assign final letter grades. (In fact, this case study is not contrived: one author is using Hazel to compute grades this semester.)

The most common end-user application for this task is the spreadsheet, because it allows an instructor to (1) record grades using a natural tabular interface, (2) visualize this data in one of a finite number of plot styles, and (3) perform basic computations, with results updated live. However, these affordances are limited. It is difficult to package common operations into reusable libraries, interact with the data using domain-specific visualizations, and perform complex, unanticipated operations (e.g. preparing the data in an idiosyncratic format demanded by the university registrar).

General-purpose programming languages *can* handle these scenarios, but users lose the ability to directly manipulate data and visualizations and receive live (in-editor) feedback.

Livelits are able to address this tension. Fig. 1c shows a Hazel program where the instructor alternates between programmatic and direct manipulation in several situations.

First, the instructor defines a value `grades` that records the grades for each student using a livelit, `$dataframe`, that implements a tabular user interface. The formula bar allows the selected cell to be filled with an arbitrary Hazel expression, here a floating point expression that adds together individual problem scores, one of which is expressed using another livelit, `$slider`. The table itself displays not the expression itself but rather its value, `80.`, just as in a spreadsheet.

Next, the instructor computes `averages` for each student by applying `compute_weighted_averages`, a helper function defined in a library (not shown) shared between courses.

After that, the instructor wants to "eyeball" reasonable letter grade `cutoffs` by directly manipulating a domain-specific livelit, `$grade_cutoffs`, that provides draggable "paddles" superimposed on a live visualization of the distribution of `averages`, which is provided as a livelit parameter. The value of `cutoffs` is a labeled 4-tuple containing each cut-off.

Finally, the instructor programmatically assigns grades to students based on these `cutoffs` by calling `assign_grades` and `format_for_university`, again shared functions.

2.2 Livelit Expansion

Livelit invocations are expressions that are given meaning by expansion. For example, the expansion of Fig. 1c is:

```
let q1_max = 36. in
let grades = Dataframe (
  ["A1", "A2", "A3", "Midterm", "Final"],
  [("Andrew", [q1_max +. 24. +. 20.,
                92., 83.5, 95., 88.]),
   ("Cyrus", [61., 64., 98., 70., 85.]),
   ("David", [75., 81., 73., 82., 79.]),
   (* ... *) ]) in
let averages = compute_weighted_averages
  grades weights in
let cutoffs = (.A 86., .B 76., .C 67., .D 48.)
in format_for_university
  (assign_grades averages cutoffs)
```

The client can inspect this expansion in Hazel via a toggle (not shown). Ideally, however, reasoning about types and binding should not require the client to inspect the expansion nor the livelit implementation. After all, function clients do not need to look inside function bodies to reason about types and binding. Instead, in the words of Reynolds [43], "type structure is a syntactic discipline for maintaining levels of abstraction". Livelits maintain this discipline by several means, described next in Sec. 2.3-2.4.

2.3 Expansion Typing

Livelit definitions declare an *expansion type*. The declarations of the livelits in Fig. 1c, eliding their implementations, are:

```
livelit $dataframe at Dataframe {...}
livelit $grade_cutoffs(averages: List(Float)) at
  (.A Float, .B Float, .C Float, .D Float) {...}
livelit $slider (min: Int) (max: Int) at Int {...}
```

The expansion type of `$dataframe` is `Dataframe`, which classifies tabular floating point data together with string row and column names (see the expansion above). The expansion type of `$grade_cutoffs` is a labeled product of grade cutoffs (field labels are written `.label` rather than `label:` in Hazel). Hazel displays the information in the livelit declaration when the cursor is on the livelit's name, just as it displays typing information in other situations (not shown) [38].

2.4 Compositionality

Livelit invocations are compositional: they can embed subexpressions in the form of parameters and splices.

2.4.1 Parameters. Livelit can declare a finite number of parameters of specified types. For example, `$grade_cutoffs` above declares one parameter, the averages to be plotted, of type **`List(Float)`**. Parameters are applied using function application notation as seen in Fig. 1c or using the pipelining (i.e. reverse function application) operators, `<|` and `|>`, which allow multiple livelits to form dataflows (not shown).

Livelit abbreviations can partially apply parameters. For example, we can partially apply the first parameter of `$slider` to define a parameterized unsigned slider livelit:

```
let $uslider = $slider 0 in ...
```

Only livelits with no remaining parameters can be invoked, so writing `$uslider` in expression position will display as a "missing livelit parameter" error. In Hazel, erroneous expressions are automatically placed inside holes and do not prevent other parts of the program from evaluating [35]. A livelit invocation can also indicate that no expansion is available with a custom error message, e.g. due to non-sensical bounds, which is also displayed to the user (not shown).

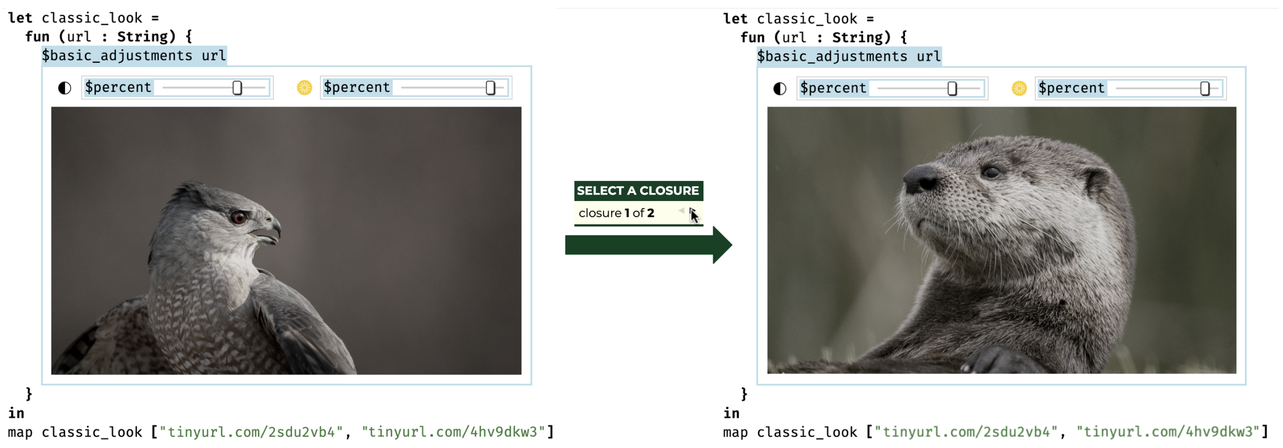
2.4.2 Splices. Spliced sub-expressions, or *splices*, appear directly inside the livelit GUI. Splices can be filled with Hazel expressions of any form, including other livelit invocations. For example, each cell in the `$dataframe` GUI in Fig. 1c has a corresponding splice. The formula bar at the top allows the user to edit the splice corresponding to the selected cell, and all of Hazel's editing affordances are available when the client does so. Unlike parameters, the number of splices can change as the user interacts with the livelit, e.g. when adding or removing rows or columns in a `$dataframe`.

The livelit provides an expected type for each splice when it is created. For example, the splices for the row and column keys in Fig. 1c have expected type **`String`**, and the remaining cells have expected type **`Float`**. Hazel displays and uses the expected type when the cursor is on the splice [38].

2.4.3 Binding Discipline. Ensuring that clients can reason about binding while leaving expansions abstract requires enforcing two key properties: *capture avoidance* and *context independence* [2, 9, 33].

Capture Avoidance. Splices and parameters are passed into the expansion as function arguments, so they cannot capture any bindings internal to the expansion. All bindings are lexically scoped to the livelit invocation site.

Consider an alternative where splices could directly appear anywhere within the expansion, e.g. in the body of a generated function or **`let`** binding. Naïvely, this could cause inadvertent capture of the bound variables in the expansion by a free variable in the parameter or splice. For example, consider a livelit that generates an expansion of the following seemingly innocuous form:



```
let classic_look =
  fun (url : String) {
    $basic_adjustments url
      $percent      $percent
  }
in
map classic_look ["tinyurl.com/2sdu2vb4", "tinyurl.com/4hv9dkw3"]
```

```
let classic_look =
  fun (url : String) {
    $basic_adjustments url
      $percent      $percent
  }
in
map classic_look ["tinyurl.com/2sdu2vb4", "tinyurl.com/4hv9dkw3"]
```

Figure 2. Case Study: Image Transformation. The image shown is determined based on the selected closure.

```
let len = strlen <splice1> in
if len > 0 then Some (<splice2> + len) else None
```

Here, `<splice2>` appears under the binding of `len`. If the client has filled `<splice2>` with an expression that refers to a client-side binding of `len`, these references would naïvely be captured. This would not occur in `<splice1>`, because the **`let`** is not recursive. This breaks abstraction and is notoriously difficult to debug, both for the livelit provider, who has no way to predict which variables a client will use, and the client, who does not know which variables the provider used.

To avoid this situation, parameters and splices must be placed in the expansion in a capture-avoiding manner: variables in splices always refer to the bindings visible to the client, rather than bindings that are hidden inside the expansion. In other macro systems, e.g. that of Omar and Aldrich [33], capture avoiding substitution is used. To simplify reasoning about live evaluation, discussed below, we take an even more restrictive approach by passing splices in as function arguments. In an eagerly evaluated language, this ensures that splice evaluation is not conditional on computations invisible to the client (such as the length check above), and it also ensures that evaluation will not inadvertently occur multiple times or not occur at all.

The trade-off is that this prevents the expression of new binding constructs and even control flow constructs as livelits. We take this as a strength, in that it simplifies reasoning for client programmers. However, this is an independent design decision: more permissive designs that nevertheless maintain the critical capture avoidance property are possible.

Context Independence. The example expansion above used a library function, `strlen`. Naïvely, this expansion would break if placed in client contexts where `strlen` is not bound, or bound to an unexpected value. To avoid requiring clients to determine and satisfy these invisible dependencies, the livelits mechanism enforces *context independence*: generated expansions are valid in any context. Dependencies are bound relative to the livelit definition site (see Sec. 3.2.5).

2.5 Live Evaluation

Livelits have the ability to evaluate a splice or a parameter in order to provide better feedback about run-time behavior to the client. The `$dataframe` livelit uses this facility to display the evaluation result for each cell, like a spreadsheet. The `$grade_cutoffs` livelit uses this facility to plot the grades, which were passed in as a parameter, on the number line.

2.5.1 Closure Collection.

The subtlety is that evaluation in Hazel is defined for closed expressions as usual, but parameters and splices can be open, i.e. refer to surrounding variables. To provide a environment that binds these variables, Hazel performs **closure collection** in two phases.

In the first phase, *proto-closure collection*, Hazel replaces each livelit with a uniquely numbered hole and then evaluates the program using the semantics for evaluating programs with holes developed by Omar et al. [35]. Evaluation proceeds around these holes, producing a result containing corresponding hole closures, i.e. holes with environments.

For example, there is one closure for `$dataframe` in Fig. 1c. It contains the value of `q1_max` and the other variables in scope. These values can be used to evaluate splices that use these variables, such as the cell selected in Fig. 1c.

Similarly, the closure for `$grade_cutoffs` in Fig. 1c includes the necessary `averages` variable, but its value depends on `grades`, which is determined by `$dataframe`. If we stop after proto-closure collection, no useful value will be available: `averages` will be *indeterminate*, because `$dataframe` has been replaced with a hole [35]. For this reason, there is a second phase of closure collection, *closure resumption*, where any livelit holes in the collected livelit closures are *resumed*, i.e. the hole is filled with the expansion and evaluation resumes.

2.5.2 Indeterminate Results.

Even after closure resumption, some elements of the closure may remain indeterminate, e.g. due to holes that are not filled with livelits. When a livelit requests an evaluation result, it must be able to handle these indeterminate results. For example, if there were missing grades, then `$grade_cutoffs` would have degraded functionality: it would display only the list elements that are values on the timeline, skipping indeterminate elements. We will return to how this occurs in Sec. 3.2.3.

2.5.3 Case Study: Live Image Filters. Our next case study considers the situation where multiple closures are collected, and the workflows it enables.

We interviewed a photographer who described a typical workflow: they use the Lightroom application to apply a set of adjustments across all photos in a collection before making individual adjustments. Many photographers do this one photo at a time, though this photographer had recently learned how to use Lightroom's saved presets to apply adjustments to multiple photos at once. However, they remained dissatisfied by the workflow. They wanted to be able to see how the shared settings affected multiple photos as they tweaked them, without having to save and reapply the preset. They also wanted to be able to change the applied preset even after making individual adjustments. Finally, they expressed interest in parameterizing presets, and in automating parts of the post-production process.

Motivated by this interview, we prototyped a collection of photo filter livelits. One of these, `$basic_adjustments`, is demonstrated in Fig. 2. This livelit contains two splices of type **`Int`**, one to adjust the contrast, and the other to adjust the brightness. In this example, we have filled those splices with `$percent`, but as above we could enter any expression of type **`Int`**, e.g. a variable. The livelit shows a live preview of the transformed image. The expansion generates calls to a browser image processing framework, not shown.

This livelit is used within a function, `classic_look`, that creates a "preset" filter. This function is mapped over a list of images (loaded by URL) at the bottom of the figure. The provided URL is passed into to livelit as a parameter.

Because the livelit appears inside a function applied (by `map`) twice, there are now two closures associated with the livelit. Hazel allows the programmer to select between the closures when the cursor is on the livelit expression via the sidebar toggle, shown in the middle of Fig. 2. This allows the client to see how the filter being designed will affect a number of example images by quickly toggling between closures. The underlying expansion remains abstract, i.e. it refers to the image via the `url` variable.

We showed this and similar examples to the photographer we had interviewed. They expressed enthusiasm for this approach despite having only limited programming experience (with Python). They made the fair point that it would take substantial effort to match Lightroom's breadth of filters, but stated that this approach could be more powerful than Lightroom's point-and-click interface while retaining many of its benefits (specifically mentioning sliders). Although this was only a single interview, it is consistent with the body of evidence summarized in Sec. 1.1.

3 Livelit Definitions

We will now take the perspective of a livelit provider. Fig. 3 defines `$color` from Fig. 1b, which is our prototypic example of a livelit definition. We omit certain incidental details and use unimplemented syntactic sugar, including Haskell-style **`do`** notation [30] and quasiquotation [5], for presentation.

Livelit definitions are scoped and packaged like any other definition. Each *definition* consists of a *declaration* and an *implementation*. In Hazel, a "template" declaration and implementation is generated as soon as the user types "**`livelit `**".

3.1 Livelit Declarations

Line 2 of Fig. 3 is `$color`'s declaration, which defines its name and its expansion type, `Color`, defined on Line 1. Both are required. Livelit parameters can also optionally appear here as shown in the declarations in Sec. 2.3. The declaration is part of the client interface, as discussed in Sec. 2.3-2.4.1.

3.2 Livelit Implementations

The curly braces delimit the livelit's implementation, which is not intended to be seen by clients. Each livelit implementation must define types `Model` and `Action`, values `init`, `view`, `update`, and `expand` with the signatures specified in Fig. 1b, and a **`context`**, which is a listing of definition-site bindings that the livelit can make use of as described below. All of these are included in the generated template. These constitute a variation on the pure functional model-view-update architecture popularized by Elm [11]. We add a fourth component, expansion generation. In addition, we use a monadic framework (*a la* Haskell [30]) to provide a pure interface between the livelit and the editor: monadic commands, like `new_splice` discussed below, are executed by the editor, and **`do`** notation provides syntactic sugar for monadic bind.

3.2.1 Model. The state of a livelit's GUI is determined by its model value. Line 3 of Fig. 3 specifies the corresponding *model type*, here a labeled 4-tuple of *splice references*, one for each of the four splices that appear in the GUI in Fig. 1b. The model is how the GUI state is persisted in the syntax tree, so the system requires that the model type supports automatic serialization (so functions cannot appear in models).

The `init` value on Line 8 determines the value of the model when the livelit is first invoked in the editor. It is a command in the **`UpdateCmd`** monad, further discussed in Sec. 3.2.4, that returns the initial model value after generating four new splices using the `new_splice` command:

```
new_splice : (Typ, Maybe(Exp))
               -> UpdateCmd(SpliceRef)
```

This command creates a splice of the given type and, optionally, its initial contents. It returns a splice reference, which uniquely identifies that splice. In this section, bolded types are defined in the standard library. The **`Typ`** and **`Exp`** types encode the syntax of Hazel's types and expressions and we use quasiquotation, e.g. `` `0` ``, as the introduction forms [5].

```
1  type Color = (.r Int, .g Int, .b Int, .a Int)
2  livelit $color at Color {
3    type Model = (.r SpliceRef, .g SpliceRef,
4                  .b SpliceRef, .a SpliceRef)
5
6    context { }
7
8    let init : UpdateCmd(Model) = do
9      r <- new_splice(`Int`, Some(`0`))
10     g <- new_splice(`Int`, Some(`0`))
11     b <- new_splice(`Int`, Some(`0`))
12     a <- new_splice(`Int`, Some(`100`))
13     return (r, g, b, a)
14
15   type Action =
16   | ClickOn(Color)
17
18   let view : Model -> ViewCmd(Html(Action)) =
19     fun model -> do
20       (* determine a color to display *)
21       r_res <- eval_splice(model.r)
22       g_res <- eval_splice(model.g)
23       b_res <- eval_splice(model.b)
24       a_res <- eval_splice(model.a)
25       let cur_color : Color =
26         case (r_res, g_res, b_res, a_res)
27         | (Some(Val(IntLit(r))),
28            Some(Val(IntLit(g))),
29            Some(Val(IntLit(b))),
30            Some(Val(IntLit(a)))) ->
31              Some((r, g, b, a))
32         | _ ->
33           (* indeterminate color shown as X *)
34           None
35       in
36
37       (* generate splice editors *)
38       let size = FixedWidth(20) in
39       r_editor <- editor(model.r, size)
40       g_editor <- editor(model.g, size)
41       b_editor <- editor(model.b, size)
42       a_editor <- editor(model.a, size)
43
44       (* ... now we can render the UI ... *)
45
46   let update :
47       Model -> Action -> UpdateCmd(Model) =
48     fun model (ClickOn c) -> do
49       set_splice(model.r, IntLit(c.r))
50       set_splice(model.g, IntLit(c.g))
51       set_splice(model.b, IntLit(c.b))
52       set_splice(model.a, IntLit(c.a))
53       return model
54
55   let expand : Model -> (Exp, List(SpliceRef)) =
56     fun model -> (`fun r g b a -> (r, g, b, a)`,
57       [model.r, model.g, model.b, model.a])
58 }
```

Figure 3. Example Livelit Definition

To ensure *context independence*, the system checks that the splice type and initial content are valid assuming only the parameters and explicitly specified **context** on Line 6. Here, the context is empty because **Int** is a built in type and the initial expressions are integer literals. We use an explicit context, rather than implicitly capturing all bindings at the definition site, to ensure that private bindings are not unintentionally leaked to clients [33].

3.2.2 Action. Line 15 defines the `Action` type for the `$color` livelit, which specifies a single user-initiated action: clicking on a color using the right half of Fig. 1b. Actions are emitted from event handlers (e.g. click handlers) defined in the computed `view`, Sec. 3.2.3, and actions are consumed by the `update` function, Sec. 3.2.4, causing a change to the model.

3.2.3 View. The `view` function computes the view given the model and access to the commands in the **ViewCmd** monad (which is distinct from the **UpdateCmd** monad). The computed view is a value of type **Html**(`Action`). This type provides a simple immutable encoding of an HTML element, where the type parameter is the type of actions that are emitted by event handlers that can be attached to elements, e.g. `on_click` and so on. We elide the details of the particular user interface in Fig. 1b, but note that livelit implementations can themselves invoke other livelits, e.g. the `view` function could use livelits for expressing user interface widgets and layouts (not shown). Instead, we focus on three mechanisms exposed by **ViewCmd**: live evaluation, splice editors, and result rendering.

Live Evaluation. As discussed in Sec. 2.5, the view can depend on the result of evaluating a splice or a parameter under the closure the client has selected. (Parameters, not shown in this example but discussed in Sec. 2.4.1, operate like splices and have type **SpliceRef** within the livelit definition.) The interface between the view and the live evaluator is via the following command:

```
eval_splice : SpliceRef -> ViewCmd(Maybe(Result))
```

The `None` case arises when evaluation is not possible, e.g. because no closures are collected or because no value has been collected for a variable used in the splice. In practice, there is an implicit hole at the end of each cell in Hazel, so livelits will have at least one closure containing results for at least the top-level bindings. Function argument values may not be available if that function has not been applied.

If available, **Result** distinguishes two possibilities:

```
  type Result = Val(Exp) | Indet(Exp)
```

The `Val` case arises when evaluation produces a value, whereas the `Indet` case arises when evaluation results in an indeterminate expression, i.e. an expression that cannot be fully evaluated due to holes in critical positions [35].

Lines 26-34 determine a color to display in the color preview if all four splices evaluate to integers. Otherwise, there is not enough information to determine a color. The livelit indicates this situation by disabling the color preview.

Livelits can attempt to offer feedback even when the result is indeterminate, because indeterminate expressions might nevertheless contain useful information. For example, a livelit that previews a sequence of notes as audio might be able to handle a list of notes where certain notes are missing, i.e. holes, by playing silence. This behavior is highly domain-specific, so each livelit provider must decide whether and how indeterminate results are supported.

Splice Editors. The view includes an editing area for each splice. These editors must support all of Hazel's editing services. To support this, the `view` function can request an editor with a given dimension (in character units) for a given splice:

```
editor : (SpliceRef, Dim) -> ViewCmd(Html(a))
```

The result is an opaque **`Html`**`(a)` value that the remainder of the function can place where needed. When the livelit is rendered, this part of the tree is under the control of Hazel. The **`Dim`** parameter currently supports only a fixed character width, with overflow causing scrolling, but in the future we plan to offer to offer more flexible layout options.

Result Rendering. Some livelit views needs to include a rendered evaluation result. For example, each of the cells in the `$dataframe` livelit in Fig. 1c show the evaluation result for the corresponding cell. Only the formula bar at the top is an editor. To support this, the `view` function can use the `result_view` command, which mirrors the `editor` command:

```
result_view : (SpliceRef, Dim)
            -> ViewCmd(Maybe(Html(a)))
```

3.2.4 Update. When the user triggers an event in a livelit view, it emits an `Action`. The system responds by calling the `update` function to determine how this action should affect the model and, in some cases, the splices.

In Fig. 3, we see that the `$color` livelit responds to the `ClickOnColor` action by invoking the `set_splice` command to overwrite the current splices with integer literals determined based on which color the user clicked on:

```
set_splice : (SpliceRef, Exp) -> UpdateCmd ()
```

As with `new_splice` described in Sec. 3.2.1, the system maintains context independence by checking the expression against the splice type and only allows use of the specified **`context`**.

When the model is updated, a new view is computed. The system then performs a diff between the old and new view in order to efficiently perform the necessary imperative updates to the editor's visual state. Changes to splices can also cause the view to be recomputed, because the view might evaluate the splices. The **`UpdateCmd`** monad does not itself have the ability to request evaluation (`eval_splice`), because the model should not depend directly on which closure the user has selected. Of course, the `view` might emit result-dependent actions when appropriate.

3.2.5 Expansion. The ultimate purpose of a livelit is to fill the hole where it appears by generating an expansion, i.e. an expression of the expansion type, here `Color`. The `expand` function determines the expansion given the model. The two are necessarily distinct: the model encodes the GUI state, and so usually tracks ephemeral widget state, like which tab is active in a tabbed GUI, as well as splice references, which refer only indirectly to the spliced expressions themselves, discussed below. This information is not needed at run-time as part of the expansion. Only the model is persisted when a Hazel program is saved, because the expansion can be regenerated by calling `expand` whenever needed.

The expansion can include spliced expressions, but the system does not make these expressions available directly as values of type **`Exp`**. Instead, the expansion must treat splices parametrically. In particular, `expand` returns an encoded *parameterized expansion*, of type **`Exp`**, paired with a list of **`SpliceRef`**s (which can come from both splices and parameters as discussed above). The parameterized expansion is a function (here curried) that takes an argument for each listed **`SpliceRef`**. That argument is of the corresponding splice type, which was provided when the splice was initialized. The return type of the parameterized expansion is the expansion type. So here, the parameterized expansion for `$color` must be a function of type **`Int`** `->` **`Int`** `->` **`Int`** `->` **`Int`** `-> Color` because the splices in the splice list were each of type **`Int`**.

This parameterization strategy makes enforcing the binding discipline described in Sec. 2.4.3 straightforward. Context independence is maintained by allowing the parameterized expansion to depend only on bindings given in the explicit **`context`**, which the system maintains bound relative to the definition site. The splices can depend only on the client site typing context. The use of function application ensures that splices are capture avoiding.

Note that Hazel does *not* statically check the definition of `expand` to ensure that the encoded parameterized expansion has the necessary type in the specified **`context`**. Instead, the parameterized expansion is only *validated* at each livelit invocation site, with errors reported to the client. A *typed quotation* system as in, e.g., MetaOCaml [25], could be adapted to allow for definition-site verification, as discussed in [33], but note that the type of the quotation depends on the type of each splice in the splice list, which has arbitrary length in general (e.g. the number of splices in the `$dataframe` livelit depends on how many rows and columns the user has added).

4 A Simply Typed Livelit Calculus

In order to specify the semantics of livelits independently of the specifics of the Hazel environment and the web platform, we now specify a *simply typed livelit calculus*.

Fig. 4 specifies the syntax of the livelit calculus. Programs are written as *unexpanded expressions*, $\hat{e}$, which are *expanded* to *external* (or *expanded*) *expressions*, e, before being *elaborated* to *internal expressions*, d, for evaluation. All three sorts are classified by the same types, τ.

$$
\begin{array}{rrcl}
\mathsf{Typ} & \tau & ::= & \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid t \mid \mu(t.\tau) \\
\mathsf{UExp} & \hat{e} & ::= & x \mid \lambda x.\hat{e} \mid \hat{e}_1\ \hat{e}_2 \mid \ldots \mid \llparenthesis\rrparenthesis^u \mid \$a\langle d_{\text{model}}; \{\psi_i\}_{i<n}\rangle^u \\
\mathsf{EExp} & e & ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \ldots \mid \llparenthesis\rrparenthesis^u \\
\mathsf{IExp} & d & ::= & x \mid \lambda x.d \mid d_1\ d_2 \mid \ldots \mid \llparenthesis\rrparenthesis^u_\sigma \\
\mathsf{Splice} & \psi & ::= & \hat{e} : \tau
\end{array}
$$

Figure 4. Syntax of types, τ, unexpanded expressions, $\hat{e}$, external expressions, e, and internal expressions, d. Here, x ranges over variables, u over hole names, and $\$a$ over livelit names. We write $\{\psi_i\}_{i<n}$ for a finite sequence of $n \geq 0$ splices, and σ for finite substitutions of $n \geq 0$ internal expressions for variables, $[d_i/x_i]_{i<n}$. We elide standard forms related to product, sum, and recursive types [18].

We include partial functions, products, sums, and recursive types, all in their standard form [18], but this specific type structure is not critical. Any language expressive enough to encode its own abstract syntax would be a suitable basis.

4.1 Background: External and Internal Language

The external and internal languages are straightforward adaptations of the external and internal languages of *Hazelnut Live*, a typed lambda calculus that assigns static and dynamic meaning to programs with holes, notated $\llparenthesis\rrparenthesis^u$ where u is a *hole name* [35]. We omit non-empty holes (which internalize type inconsistencies [36]) and type holes (which operate like the unknown type from gradual type theory [36, 44]). These mechanisms are orthogonal to livelits and are included in our implementation.

External expressions, e, are governed by a typing judgement of the form, $\Gamma \vdash e : \tau$, where the typing context, Γ, is a finite set of typing assumptions of the form $x : \tau$ [18].

The internal language is a contextual type theory [32], i.e. the typing judgement is of the form $\Delta; \Gamma \vdash d : \tau$ where the hole context Δ is a finite set of hole typing assumptions of the form $u :: \tau[\Gamma]$ which mean that hole u must be filled with an expression of type τ under Γ. We need a hole typing context only for the internal language because, although hole names are assumed unique in the external language, they can be duplicated during evaluation of internal expressions.

External expressions elaborate to internal expressions, d, according to the declarative elaboration judgement $\Gamma \vdash e \rightsquigarrow d : \tau \dashv \Delta$. (Hazelnut Live specifies elaboration algorithmically [35].) The main purpose of elaboration is to initialize the substitution σ on each hole closure, $\llparenthesis\rrparenthesis^u_\sigma$, which takes the form $[d_i/x_i]_{i<n}$ and serves to capture the substitutions that occur around the hole during evaluation. The key rule for elaboration at type τ is:

$$\texttt{Elab-Hole}\quad \frac{}{\Gamma \vdash \llparenthesis\rrparenthesis^u \rightsquigarrow \llparenthesis\rrparenthesis^u_{\mathsf{id}(\Gamma)} : \tau \dashv u :: \tau[\Gamma]}$$

The substitution is initially the identity substitution, $\mathsf{id}(\Gamma)$, i.e. the substitution that maps each variable in Γ to itself, because no substitutions have yet occurred. For example,

$$\vdash (\lambda x.\llparenthesis\rrparenthesis^u)\ \underline{5} \rightsquigarrow (\lambda x.\llparenthesis\rrparenthesis^u_{[x/x]})\ \underline{5} : \mathsf{nat} \dashv u :: \mathsf{num}[x : \mathsf{nat}]$$

During evaluation, $d \Downarrow d'$, the closure's substitution accumulates the substitutions that occur. For example, the internal expression above evaluates as follows:

$$(\lambda x.\llparenthesis\rrparenthesis^u_{[x/x]})\ \underline{5} \Downarrow \llparenthesis\rrparenthesis^u_{[\underline{5}/x]}$$

Rather than restating the remaining rules, we simply state the key governing metatheorems and defer to the prior work and our Agda mechanization (Sec. 4.2.3) for the details [35].

Elaboration preserves typing.

Theorem 4.1 (Typed Elaboration). *If $\Gamma \vdash e : \tau$ then $\Gamma \vdash e \rightsquigarrow d : \tau \dashv \Delta$ for some d and Δ such that $\Delta; \Gamma \vdash d : \tau$.*

Evaluation of a closed well-typed expression with holes results in a final (i.e. irreducible) expression of the same type.

Theorem 4.2 (Preservation). *If $\Delta; \cdot \vdash d : \tau$ and $d \Downarrow d'$ then d' final and $\Delta; \cdot \vdash d' : \tau$.*

4.2 Expansion

The novelty of the livelit calculus is entirely in its handling of unexpanded expressions, $\hat{e}$, which are given meaning by typed expansion to external expressions, e, according to the judgement $\Phi; \Gamma \vdash \hat{e} \rightsquigarrow e : \tau$ defined in Fig. 5. Unexpanded expressions mirror external expressions but for the presence of livelit invocations. The rules for the mirrored forms like `EVar` and `EFun`, both shown in Fig. 5, are straightforward.

4.2.1 Livelit Contexts. Livelit definitions are collected in the livelit context, Φ, which maps livelit names $\$a$ to livelit definitions of the following form:

$$\mathsf{livelit}\ \$a\ \mathsf{at}\ \tau_{\text{expand}}\ \{\tau_{\text{model}}; d_{\text{expand}}\}$$

Here, τ_{expand} is the expansion type, τ_{model} is the model type, and d_{expand} is the expansion function, which generates an expansion given a model. We omit the logic related to view computations and actions, which are tied to a particular UI framework and have only indirect semantic significance.

Definition 4.3 (Livelit Context Well-Formedness). A livelit context Φ is well-formed if and only if for each livelit definition, $\mathsf{livelit}\ \$a\ \mathsf{at}\ \tau_{\text{expand}}\ \{\tau_{\text{model}}; d_{\text{expand}}\} \in \Phi$, we have $\vdash d_{\text{expand}} : \tau_{\text{model}} \to \mathsf{Exp}$.

Here, Exp stands for a type whose values isomorphically encode external expressions. The isomorphism is mediated in one direction by the encoding judgement $e \downarrow d$ and the other by the decoding judgement $d \uparrow e$. Any scheme is sufficient, so we leave it as a matter of implementation. The simplest approach is to define Exp as a recursive sum type, with one arm for each form of external expression (cf. [34]).

For simplicity, we assume that the livelit context is provided *a priori* and therefore that the expansion function is already closed and fully elaborated. In practice, the livelit

$$\texttt{EVar}\quad \frac{x : \tau \in \Gamma}{\Phi; \Gamma \vdash x \rightsquigarrow x : \tau} \qquad \texttt{ELam}\quad \frac{\Phi; \Gamma, x : \tau_{\text{in}} \vdash \hat{e} \rightsquigarrow e : \tau_{\text{out}}}{\Phi; \Gamma \vdash \lambda x.\hat{e} \rightsquigarrow \lambda x.e : \tau_{\text{in}} \to \tau_{\text{out}}} \qquad \cdots$$

$$\texttt{ELivelit}\quad \frac{\begin{array}{c}\textsf{livelit } \$a \textsf{ at } \tau_{\text{expand}}\ \{\tau_{\text{model}}; d_{\text{expand}}\} \in \Phi \\ \vdash d_{\text{model}} : \tau_{\text{model}} \\ d_{\text{expand}}\ d_{\text{model}} \Downarrow d_{\text{encoded}} \qquad d_{\text{encoded}} \uparrow e_{\text{pexpansion}} \\ \vdash e_{\text{pexpansion}} : \{\tau_i\}_{i<n} \to \tau_{\text{expand}} \\ \{\Phi; \Gamma \vdash \hat{e}_i \rightsquigarrow e_i : \tau_i\}_{i<n}\end{array}}{\Phi; \Gamma \vdash \$a\langle d_{\text{model}}; \{\hat{e}_i : \tau_i\}_{i<n}\rangle^u \rightsquigarrow e_{\text{pexpansion}}\ \{e_i\}_{i<n} : \tau_{\text{expand}}}$$

Figure 5. Expansion

context would be controlled by a definition form in the language that allows the definition to itself invoke livelits. This would require a staging mechanism, because we need to evaluate expansion functions in prior definitions to be able to expand subsequent definitions. There are a number of ways to support the necessary staging in practice, e.g. via explicit staging primitives [13], by requiring that these definitions appear in separately compiled packages [33], or by using live programming mechanisms such as those in Hazel to evaluate "up to" each definition before proceeding [35].

4.2.2 Livelit Expansion. Unexpanded expressions include livelit invocations:

$$\$a\langle d_{\text{model}}; \{\psi_i\}_{i<n}\rangle^u$$

Here, $\$a$ names the livelit being invoked. Livelits can be understood as filling holes, so u identifies the hole that is, conceptually, being filled. The current state of the livelit is determined by the current model value, d_{model}, together with the splice list, $\{\psi_i\}_{i<n}$. Each splice ψ_i is of the form $\hat{e}_i : \tau_i$, where $\hat{e}_i$ is the spliced expression itself (unexpanded, so it may contain other livelits) and τ_i is the type of that splice, as determined when the livelit definition first requested the splice (as discussed in Sec. 3.2.1). Note that we do not formally model the edit actions that change the model or splice list here; we focus on a single snapshot of the editor state. We leave an action semantics for livelits, following the Hazelnut action semantics [36], as future work.

Rule `ELivelit` performs livelit expansion. Its premises, in order, operate as follows:

1. **Lookup.** The first premise looks up the livelit definition in the livelit context.
2. **Model Validation.** The second premise checks that the model value, d_{model}, is of the specified model type, τ_{model}.
3. **Expansion.** The third premise applies the expansion function, d_{expand}, to the model value, d_{model}, producing the *encoded parameterized expansion*, d_{encoded}, which, by the definitions and theorems given above, is of type Exp.
4. **Decoding.** The fourth premise decodes d_{encoded}, producing the *parameterized expansion*, $e_{\text{pexpansion}}$. The isomorphism between encodings and external expressions ensures that decoding cannot fail.
5. **Expansion Validation.** The fifth premise checks that parameterized expansion is a function that returns a value of the expansion type, τ_{expand}, when applied (in curried fashion, though this is not critical) to the splices, whose types, $\{\tau_i\}_{i<n}$, are given in the splice list. Validation can fail, in which case expansion fails. (In Hazel, validation failure is instead marked by a non-empty hole and an appropriate error message is shown.)
   We ensure that the parameterized expansion is *context independent*, i.e. that it cannot depend on the particular bindings available in the call site typing context, Γ, by requiring that the parameterized expansion be closed. Consequently, any necessary helper functions used in the expansion must be provided by the client via a splice.
   In Sec. 3, we discussed how the use of an explicit definition site **`context`** eliminates this client burden. The explicit context can formally be modeled as just a value (tupled, typically) that can be passed as an additional argument to the parameterized expansion alongside the splices, so we omit it from the calculus, but see [33] for a full treatment.
6. **Splice Expansion.** The sixth premise inductively expands each of the spliced expressions in the same context as the livelit invocation itself.

The conclusion of the rule then applies the parameterized expansion to the expanded splices. By applying the splices as arguments, we maintain *capture avoidance* – splices cannot capture variables bound internally to the expansion because beta reduction performs capture-avoiding substitution.

The typed expansion process is governed by the following metatheorem, which establishes that the expansion is indeed an external expression of the indicated type (i.e. the expansion type, in the case of livelit invocations).

Theorem 4.4 (Typed Expansion). *If* $\Phi; \Gamma \vdash \hat{e} \rightsquigarrow e : \tau$ *then* $\Gamma \vdash e : \tau$.

When composed with the Typed Elaboration theorem and the type safety of the internal language, we achieve end-to-end type safety: every well-typed unexpanded expression expands to a well-typed external expression, which in turn elaborates to a well-typed internal expression, which in turn evaluates in a type safe manner.

4.2.3 Agda Mechanization. We have mechanically proven these theorems using the Agda proof assistant, building on the Agda mechanization of Hazelnut Live [35]. This is the first mechanization of the literal macro expansion process (previous work on textual literal macros had only paper proofs [33]). The mechanization is available in the artifact and online at the following URL:

https://github.com/hazelgrove/hazelnut-livelits-agda/

4.3 Live Feedback via Closure Collection

In order to support live feedback, a livelit needs to be able to ask the system to evaluate expressions under one of the closures associated with the livelit. This mechanism was introduced by example in Sec. 2.5.1. In this section, we will formalize the process of efficiently collecting closures.

4.3.1 Proto-Environment Collection. We begin by generating an alternative expansion, called the *cc-expansion*, where each livelit invocation expands to an empty hole applied to its splices. In other words, a hole appears in place of the parameterized expansion (but not splices). On the side, we generate a *cc-context*, Ω, that maps each livelit hole to the *elaboration* of its parameterized expansion, $u \hookrightarrow d_{\text{pexpansion}}$. The key rule for cc-expansion, $\Phi; \Gamma \vdash_{\text{cc}} \hat{e} \rightsquigarrow e : \tau \dashv \Omega$, is:

`CCLivelit`

$$\frac{\begin{array}{c}\{\Phi; \Gamma \vdash_{\text{cc}} \hat{e}_i \rightsquigarrow e_i : \tau_i \dashv \Omega_i\}_{i<n} \\ \Phi; \Gamma \vdash \$a\langle d_{\text{model}}; \{\hat{e}_i : \tau_i\}_{i<n}\rangle^u \rightsquigarrow e_{\text{pexpansion}}\ \{e'_i\}_{i<n} : \tau_{\text{expand}} \\ \vdash e_{\text{pexpansion}} \rightsquigarrow d_{\text{pexpansion}} : \{\tau_i\}_{i<n} \to \tau_{\text{expand}} \dashv \\ \Omega = \cup_{i<n}\Omega_i \cup \{u \hookrightarrow d_{\text{pexpansion}}\}\end{array}}{\Phi; \Gamma \vdash_{\text{cc}} \$a\langle d_{\text{model}}; \{\hat{e}_i : \tau_i\}_{i<n}\rangle^u \rightsquigarrow (\!|\,|\!)^u\ \{e_i\}_{i<n} : \tau_{\text{expand}} \dashv \Omega}$$

We then elaborate and evaluate the cc-expansion. The result will contain some number of hole closures for each livelit hole. We call these the proto-closures and their environments the proto-environments for that livelit hole.

Definition 4.5 (Proto-Closure Collection)**.** If $\Phi; \cdot \vdash_{\text{cc}} \hat{e} \rightsquigarrow e : \tau \dashv \Omega$ and $\vdash e \rightsquigarrow d : \tau \dashv \Delta$ and $d \Downarrow d'$ and $u \in \text{dom}(\Omega)$ then $\text{protoenvs}_{\Phi}(\hat{e}; u) = \{\sigma \mid (\!|\,|\!)^u_\sigma \in d'\}$.

4.3.2 Closure Resumption. A proto-environment for a livelit hole might itself contain a proto-closure for another livelit hole, which is problematic for the reasons detailed in Sec. 2.5.1. Consequently, the second step of closure collection, called *closure resumption*, is to fill any livelit holes that appear in the proto-environments for other livelit holes. We do so by filling them using the parameterized expansions gathered in Ω and then resuming evaluation where appropriate. Formally, this involves the hole filling operation $[\![d_1/u]\!]d_2$ for Hazelnut Live (which derives from the metavariable instantation operation of contextual modal type theory [32, 35]). This operation fills every closure for hole u in d_2 with d_1. Unlike substitution, hole filling is not capture-avoiding. Instead, the environment on each of these closures is applied to d_1 as a substitution, i.e. the delayed substitutions captured in the environment are realized. In this case, however, the parameterized expansion is necessarily closed due to the context independence discipline we maintain in Rule `ELivelit`, so hole filling amounts to syntactic replacement.

Formally, we begin by defining an operation $\text{fill}_\Omega(\sigma)$ which acts on proto-environments to fill the livelit holes.
Definition 4.6 (Livelit Hole Filling)**.**

1. $\text{fill}_\Omega([d_1/x_1, \ldots, d_n/x_n]) =$
   $[\text{fill}_\Omega(d_1)/x_1, \ldots, \text{fill}_\Omega(d_n)/x_n]$
2. $\text{fill}_\Omega(d) = [\![d_{\text{pexpansion}}/u]\!]_{u \hookrightarrow d_{\text{pexpansion}} \in \Omega} d$

This first step may cause certain expressions to become non-final, because the filled hole is no longer blocking evaluation. We therefore define an operation $\text{resume}(\sigma)$ that resumes evalution for all closed expressions in σ. (The only open expressions that might remain are the initial variables from the identity substitution generated by elaboration. Some closures appear under binders in the final result, so these variables will not have yet recorded a substitution.)
Definition 4.7 (Environment Resumption)**.**

1. $\text{resume}([d_1/x_1, \ldots, d_n/x_n]) =$
   $[\text{resume}(d_1)/x_1, \ldots, \text{resume}(d_n)/x_n]$
2. $\text{resume}(d) = d'$ if $\text{FV}(d) = \emptyset$ and $d \Downarrow d'$
3. $\text{resume}(d) = d$ if $\text{FV}(d) \neq \emptyset$

Finally, we can produce the final set of environments by filling and resuming the proto-environments.

Definition 4.8 (Environment Collection)**.** If $\Phi; \cdot \vdash_{\text{cc}} \hat{e} \rightsquigarrow e : \tau \dashv \Omega$ then

$$\text{envs}_\Phi(\hat{e}; u) = \{\text{resume}(\text{fill}_\Omega(\sigma)) \mid \sigma \in \text{protoenvs}_\Phi(\hat{e}; u)\}$$

This same fill and resume operation can be used to avoid recomputation when evaluating the fully expanded version of the user's program. If the editor has already performed environment collection, then it can simply continue from where it left off by filling and resuming the remaining top-level livelit holes (those that do not appear in a proto-environment).

The correctness of the mechanisms described in this section rest on the fact that evaluation commutes with hole filling in the pure setting.

Theorem 4.9 (Post-Collection Resumption)**.** *If* $\Phi; \cdot \vdash_{cc} \hat{e} \rightsquigarrow e_{cc} : \tau \dashv \Omega$ *and* $\vdash e_{cc} \rightsquigarrow d_{cc} : \tau \dashv \Delta$ *and* $d_{cc} \Downarrow d'_{cc}$ *and* $\text{resume}(\text{fill}_\Omega(d'_{cc})) = d_1$ *and* $\Phi; \cdot \vdash \hat{e} \rightsquigarrow e_{full} : \tau$ *and* $\vdash e_{full} \rightsquigarrow d_{full} : \tau \dashv \Delta$ *and* $d_{full} \Downarrow d_2$ *then* $d_1 = d_2$.

Proof. The key observation is that filling the livelit holes in the cc-expansion gives the full expansion, i.e. $\text{fill}_\Omega(d_{\text{cc}}) = d_{\text{full}}$. Resumption is simply evaluation for closed expressions. By commutativity of hole filling, established in the prior work [35], we can delay hole filling until d_{cc} has first been evaluated to d'_{cc}. □

In an imperative language with non-commutative side effects, resumption is not sound. We conjecture that one can specify an alternative evaluation mode where the full expansion of each livelit invocation is evaluated in the order that corresponding livelit hole is encountered during evaluation, with the closure recorded in a memory. This would ensure that side effects happen in the same order and only once. Alternatively, an imperative language might use a different mechanism to make environments available to livelits, e.g. by environment snapshotting *a la* Lamdu [29]. This is more limited in situations where a livelit invocation appears in a function which has yet to be applied (where collected livelit closures essentially capture the function's closure).

Livelits invocations in branches that are not taken or that appear under unused variables do not have closures collected for them with our approach (though in practice there is an implicit "trailing hole" after each cell in Hazel which means the latter situation is infrequent). Livelit invocations that determine which of several branches are taken prevent substitutions conditional on those branches from being recorded in downstream closures because resumption operates only within the proto-closures. This design gives clients the ability to control closure collection in branches that are not of interest (and therefore control the cost of closure collection), but may sometimes cause unexpected behavior. We leave further exploration of this design space to future work.

5 Implementation

Implementing livelits requires tight integration between a rich editor, a type checker, and a live evaluator capable of evaluating incomplete programs and gathering hole closures.

5.1 Hazel

Hazelnut Live is the foundation of the Hazel programming environment, and Hazel has support for all of the necessary mechanisms, so it was natural to choose Hazel for our primary implementation. Hazel is implemented in OCaml and compiled to JavaScript using the `js_of_ocaml` compiler [39].

The livelit definition mechanism described in Sec. 3 is implemented in Hazel, albeit without some of the syntactic sugar in Fig. 3. This, together with the fact that Hazel lacks a mature GUI widget libraries as of this writing, makes complex examples tedious to implement within Hazel, so we also added the ability to define livelits using JavaScript or OCaml. These are loaded when Hazel is compiled. As Hazel evolves, we expect to need to define such "primitive" livelits less frequently, using them mainly for livelits that would benefit from access to established JavaScript or OCaml libraries.

Uniquely, every editor state in Hazel is semantically meaningful: it has a type, it can be evaluated, and it can be transformed in a type-aware manner. This implies that livelits remain fully functional at all times, even when the program is incomplete or erroneous. Hazel achieves this "gap-free" liveness guarantee by automatically inserting explicit holes as necessary while the user edits the program. Formally, Hazel is a type-aware structure editor [36], rather than a text editor, although the developers are aiming to maintain a text-like experience (this effort is orthogonal to our own). To maintain this guarantee, Hazel inserts empty holes where there are missing terms, and non-empty holes as markers of errors. To maintain this invariant in the presence of livelits, new non-empty holes are needed to mark each of the failure modes suggested by the premises of the `ELivelit` rule in Fig. 5. In particular, non-empty holes mark (1) the invocation of an unbound livelit; (2) the invocation of a livelit with a model value of the wrong type; (3) a run-time error during evaluation of `init`, `update`, or `expand` functions (which would manifest as run-time holes); (4) expansion validation failure (i.e. that the parameterized expansion is of the wrong type given the splice list and expansion type). Errors in view generation are not considered semantic errors (they display as error messages where the livelit GUI would have appeared.)

5.2 Text Editor Integration

Livelits do not require the use of a structure editor. We have also developed a proof-of-concept implementation of livelit interaction in a textual program editor, Sketch-n-Sketch [21]. The livelit GUI appears in a pop up window when requested by the user. Interactions with this GUI cause the serialized model in the text buffer to be changed, which updates the view. This proof-of-concept is not at feature parity with the Hazel implementation, but it demonstrates that a syntax-recognizing text editor [3, 4, 23] is sufficient to support livelits, albeit with gaps in availability when there are syntax errors. There are other systems that integrate visual syntax into an otherwise textual editor as well, notably Dr. Racket's recent visual macro system [3], discussed in the next section.

5.3 Layout

Whether implemented in a structure editor or a text editor, livelits present interesting layout challenges. Hazel uses an optimizing pretty printer based on the work of Bernardy [6] to determine layout. This system relies fundamentally on character counts. Consequently, our implementation asks each livelit to specify dimensions in terms of character counts rather than pixels. Livelits can be laid out either as inline livelits, like `$slider`, which are one character high and appear inline with the code, or as *multi-line livelits*, which occupy up to the full width and a specified number of lines.

One might also consider a number of other layout options, e.g. inline-block literals *a la* Wyvern [34], pop-up livelits, livelits pinned to sidebars, and livelits that are rendered on a separate canvas or document while still formally being located within an underlying functional program.

This latter option would be particularly interesting for end-user programming scenarios: users with limited programming experience could interact with a collection of livelits laid out separately in the popular "dashboard" style, without necessarily even being aware that their interactions are actually edits to an underlying typed functional program.

5.4 Integration into Imperative Languages

Our focus in this paper was on pure languages. Side effects pose a challenge on two fronts.

If they occur in a livelit implementation, as is possible for livelits implemented using Hazel's support for Javascript-based livelits, then the state exists at edit-time. To ensure that the GUI's state persists and remains valid between reloads, it may then be necessary to persist the relevant portions of the edit-time state alongside the program, as in Smalltalk systems [15], or else provide for a state re-initialization protocol that is called when a program is re-loaded from persistent memory. It may also be necessary to more carefully control the access that a livelit has to the editor itself: many participants in the Graphite survey indicated that they did not want palettes, which were able to make network requests at edit-time, to be able to access proprietary or sensitive source code [37]. In any case, the expansion should ideally be a pure function of the model even in an imperative language, or run as an isolated process each time, to ensure that program behavior does not depend on expansion-time editor state.

If side effects occur in livelit expansions, then the closure collection mechanism requires more care to avoid unsoundness, as described in Sec. 4.3. In addition, continuous live evaluation as in Hazel is intrinsically problematic in an imperative setting because running code on each keystroke can cause arbitrary side effects, e.g. network requests or I/O.

These problems are surmountable with suitable compromises, e.g. limiting continuous evaluation for code with external side effects, and we look forward to efforts to integrate livelits into imperative languages.

6 Related Work

As detailed in Sec. 1.1, the Graphite system developed the idea of filling typed holes using a type-specific user interface, and it was the starting point for our work [37]. Subsequent work on **mage** further explores this design space [24]. This prior work engaged in substantial qualitative evaluations, which due to the fundamental similarities between the two systems is as relevant to our design as theirs. However, the prior work left a number of core technical issues unresolved, as summarized Sec. 1.2. Livelits resolve these in large part by bringing together ideas from other recent work.

In particular, recent work on type-specific languages (TSLs) [34] and typed literal macros (TLMs) [33] explored similar ideas of user-defined literal forms with support for hygienic splicing, with the former using type-directed dispatch similar to Graphite and the latter supporting decentralized extensibility via explicit naming as in our approach. However, these systems operate in a purely textual setting and have no support for live feedback. The typed expansion judgement central to the typed livelit calculus in this paper is structured similarly to the corresponding judgements in the formal systems describing TLMs and TSLs, and the reasoning principles are closely related. However, the approach to capture avoidance we take is both more restrictive and cleaner by its use of function application rather than direct insertion of splices (and could perhaps be ported to those formalisms). Splices also operate quite differently in our work, because they are placed automatically and structurally delimited in the user interface, rather than placed by the client and then parsed out of the text by a custom parser charged with retaining provenance information. This substantially complicates the design of splices in that work.

This structural delimitation of splices is reminiscent of work on *language boxes* [42], which focused on combining different notations, primarily textual, using structural delimiters inserted explicitly using a special-purpose editor.

Recent work on a interactive visual macro system in Dr. Racket is quite similar in spirit to our work [3]. However, it supports only a limited form of splicing via a pop up text editor that does not support full compositionality as described here, where livelits can appear within other livelits. Splicing is not strictly hygienic, though Racket's scope management machinery can be used by macros to voluntarily maintain a binding discipline [14]. There is not any consideration of typing. Parameterization and partial application is also impossible, because invocation is via an editor command rather than a syntactic mechanism. Finally, there is no support for liveness: macro evaluation occurs in the editor environment, not the run-time environment of the program being written, so splice evaluation would not work correctly (except in trivial cases where the splice was, e.g., a closed expression, or used only standard library constructs). That said, we conjecture that it is possible to implement many of the mechanisms we describe in this paper in some form as a layer atop these existing mechanisms in Dr. Racket.

There has been a long line of research on *projectional editing*, where the user edits graphical representations (projections) of code constructs [26, 31, 40, 41]. Livelits are a form of projectional editing. Many of the oldest systems offer only a fixed set of projections and interaction techniques. More recently, language workbenches with support for projectional editing like Citrus [26] and MPS [47] have made it easier to define new projectional editors. However, these systems generate entire editors, whereas livelit definitions are decentralized in libraries and lexically scoped. Furthermore, livelit definitions are governed by a type and binding discipline. Finally, livelits are uniquely live, building on hole closure tracking from Hazelnut Live [35].

A number of notebook systems, including Mathematica, Jupyter [16], and others, do support the insertion of simple widgets like sliders that respond live to changes. These systems inspired our approach but they are not compositional: only constant values can be constructed, as with Graphite.

Related to projectional editors are a variety of systems that generate visualizations from code [15, 27, 28, 46]. Livelits differ from these systems in directionality: visualization systems generates visualizations from values, whereas livelits generate expressions where there would otherwise be a hole.

Conal Elliott's work on tangible functional programming [12] similarly explored a system allows editing a graphical representation of code in compositional ways, but the editing representation itself is fixed as a series of connected windows. Only the visualizations are customizable.

The Vital programming environment for Haskell [17] supports type-specific stylesheets that can be used to create custom user interfaces for both editing and displaying values. The editors support splices, called *cells*, that can contain Haskell code. Results are computed in a live manner, though there is no support for evaluating incomplete programs. The visualizations themselves cannot provide live feedback. Moreover, the system does not enforce any hygiene or typing principles: the user is entirely responsible for syntactic and semantic correctness.

7 Discussion and Conclusion

The arithmetical symbols are written diagrams
and the geometrical figures are graphic formulas.
— David Hilbert [22]

Diagrams have played a pivotal role in mathematical thought since antiquity, indeed predating symbolic mathematics [7]. Popular computing and creative tooling, too, has embraced visual representation and direct manipulation interfaces for decades. Programming, however, has remained stubbornly mired in textual user interfaces. Our hope with this paper is to demonstrate that principled, mathematically structured programming is not only compatible with live graphical user interfaces, but that the combination of these two holds promise for the future of programming.

This paper's contributions are in advancing the expressive power of the mechanism in several directions, most notably in terms of compositionality and liveness. These technical contributions are a complement to the empirical findings by Omar et al. [37] and others. That said, the two case studies we considered in this paper were motivated by real world problems. As the implementation matures, we plan to introduce enthusiasts in a wide variety of problem domains to livelits and continue these empirical evaluations.

Mechanisms for deriving simple livelit definitions from type definitions, perhaps similar to Haskell's `deriving` directive or the GEC toolkit [1], or from `to_string` functions [20], may prove fruitful in the future.

The livelits mechanism as described in this paper operates only on expressions, but livelits might be useful for generating other sorts of terms, such as types, patterns, and entire modules. Prior work on literal macros has explored this [33]. Note that Racket's visual macro system generates arbitrary syntax, so it can already be used in this manner, albeit with no sort-specific semantic guarantees.

The strict binding discipline has, we believe, substantial benefits—programmers will inevitably encounter unfamiliar livelits, and the reasoning principles that we enforce are likely to help them "reason around" the situation. However, it may be useful in certain circumstances to relax these, with the editor alerting the user to the unusual situation.

Another direction for future work has to do with pushing edits from computed results back into livelits. For example, a slider expands to a number, which may then flow through a computation. Bidirectional evaluation techniques may allow the user to edit a number in the result of a computation and see the necessary change to a slider in the program [8, 19].

Programming and authoring have much in common. Documents often contain structured information, and programs are written to manipulate structured information. Another future direction for livelits is as the basis for a programmable authoring system, where the non-symbolic elements on the page are revealed to be code after all, albeit code generated by a livelit invocation that presents a more natural editing experience. Taking this further, a networked collection of these documents could form a powerful computational wiki. We present this paper as a foundation for such explorations.

Acknowledgements

We thank Pavel Panchekha and anonymous referees at TyDe 2019, POPL 2021 and PLDI 2021 for thoughtful feedback that has improved this paper. This material is based upon work supported by the National Science Foundation under Grant No. 1814900 and 1817145.

References

[1] Peter Achten, Marko C. J. D. van Eekelen, Rinus Plasmeijer, and Arjen van Weelden. 2004. GEC: A Toolkit for Generic Rapid Prototyping of Type Safe Interactive Applications. In *Advanced Functional Programming, 5th International School, AFP 2004, Revised Lectures (Lecture Notes in Computer Science, Vol. 3622)*. Springer, 210–244. https://doi.org/10.1007/11546382_5

[2] Michael D. Adams. 2015. Towards the Essence of Hygiene. In *POPL*. 457–469. https://doi.org/10.1145/2676726.2677013

[3] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 222:1–222:28. https://doi.org/10.1145/3428290

[4] Robert A. Ballance, Susan L. Graham, and Michael L. Van de Vanter. 1992. The Pan Language-Based Editing System. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (1992), 95–127. https://doi.org/10.1145/125489.122804

[5] Alan Bawden. 1999. Quasiquotation in Lisp.. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), January 22-23, 1999. Technical report BRICS-NS-99-1.* University of Aarhus, 4–12.

[6] Jean-Philippe Bernardy. 2017. A pretty but not greedy printer (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (2017), 6:1–6:21. https://doi.org/10.1145/3110250

[7] Florian Cajori. 1993. *A history of mathematical notations.* Vol. 1. Courier Corporation.

[8] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *PLDI*. https://doi.org/10.1145/2908080.2908103

[9] William D. Clinger and Jonathan Rees. 1991. Macros That Work. In *POPL*. 155–162. https://doi.org/10.1145/99583.99607

[10] Evan Czaplicki. 2018. An Introduction to Elm. (2018). https://guide.elm-lang.org/. Retrieved Apr. 7, 2018.

[11] Czaplicki, Evan. 2018. Elm Architecture. (2018). https://guide.elm-lang.org/architecture/. Retrieved Apr. 7, 2018.

[12] Conal Elliott. 2007. Tangible Functional Programming. In *ICFP*. https://doi.org/10.1145/1291151.1291163

[13] Matthew Flatt. 2002. Composable and compilable macros: you want it when?. In *ICFP*. https://doi.org/10.1145/581478.581486

[14] Matthew Flatt. 2016. Binding as sets of scopes. In *POPL*. https://doi.org/10.1145/2837614.2837620

[15] Adele Goldberg. 1984. *Smalltalk-80 - the interactive programming environment.* Addison-Wesley.

[16] Philip J. Guo. 2013. Online Python Tutor: embeddable web-based program visualization for CS education. In *The 44th ACM Technical Symposium on Computer Science Education (SIGCSE)*. 579–584.

[17] Keith Hanna. 2002. Interactive visual functional programming. In *ICFP*. https://doi.org/10.1145/581478.581493

[18] Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). https://www.cs.cmu.edu/~rwh/plbook/2nded.pdf

[19] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Symposium on User Interface Software and Technology (UIST)*.

[20] Brian Hempel and Ravi Chugh. 2020. Tiny Structure Editors for Low, Low Prices! (Generating GUIs from toString Functions). In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. https://doi.org/10.1109/VL/HCC50065.2020.9127256

[21] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *International Conference on Software Engineering (ICSE)*.

[22] David Hilbert. 1902. Mathematical problems. *Bull. Amer. Math. Soc.* 8, 10 (1902), 437–479.

[23] Joseph R. Horgan and D. J. Moore. 1984. Techniques for Improving Language-Based Editors. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM, 7–14. https://doi.org/10.1145/800020.808243

[24] 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 *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology*. 140–151. https://doi.org/10.1145/3379337.3415842

[25] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Proceedings of the 12th International Symposium on Functional and Logic Programming (FLOPS) (Lecture Notes in Computer Science, Vol. 8475)*. Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[26] A. J. Ko and Brad A. Myers. 2005. Citrus: a language and toolkit for simplifying the creation of structured editors for code and data. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology (UIST)*. 3–12. https://doi.org/10.1145/1095034.1095037

[27] Rainer Koschke. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice* 15, 2 (2003), 87–109.

[28] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *CHI '20: CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3313831.3376494

[29] Eyal Lotem and Yair Chuchem. 2016. Project Lamdu. http://www.lamdu.org/. Accessed: 2016-04-08.

[30] Simon Marlow et al. 2010. Haskell 2010 language report. (2010). https://www.haskell.org/onlinereport/haskell2010

[31] Philip Miller, John Pane, Glenn Meter, and Scott A. Vorthmann. 1994. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments* 4, 2 (1994), 140–158. https://doi.org/10.1080/1049482940040202

[32] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *ACM Trans. Comput. Log.* 9, 3 (2008). https://doi.org/10.1145/1352582.1352591

[33] Cyrus Omar and Jonathan Aldrich. 2018. Reasonably Programmable Literal Notation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue ICFP* (2018).

[34] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2014. Safely Composable Type-Specific Languages. In *ECOOP*.

[35] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages (PACMPL), Issue POPL* (2019).

[36] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *POPL*.

[37] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *International Conference on Software Engineering (ICSE)*.

[38] Hannah Potter and Cyrus Omar. 2020. Hazel Tutor: Guiding Novices Through Type-Driven Development Strategies. In *Human Aspects of Types and Reasoning Assistants*. https://hazel.org/hazeltutor-hatra2020.pdf

[39] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A Core ML Language for Tierless Web Programming. In *14th Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science, Vol. 10017)*. 377–397. https://doi.org/10.1007/978-3-319-47958-3_20

[40] Michael Read and Chris Marlin. 1996. Generating direct manipulation program editors within the MultiView programming environment. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints' 96) at SIGSOFT'96*. 232–236. https://doi.org/10.1145/243327.243670

[41] Steven P. Reiss. 1984. Graphical Program Development with PECAN Program Development Systems. In *Proceedings of the ACM SIGSOFT-/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM, 30–41. https://doi.org/10.1145/800020.808246

[42] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. 2009. Language Boxes. In *Second International Conference on Software Language Engineering (SLE) (Lecture Notes in Computer Science, Vol. 5969)*. Springer, 274–293. https://doi.org/10.1007/978-3-642-12107-4_20

[43] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*. North-Holland/IFIP, 513–523.

[44] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. http://scheme2006.cs.uchicago.edu/13-siek.pdf

[45] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *International Workshop on Live Programming (LIVE)*.

[46] Jaime Urquiza-Fuentes and J Angel Velázquez-Iturbide. 2004. A survey of program visualizations for the functional paradigm. In *Proc. 3rd Program Visualization Workshop*. 2–9.

[47] Markus Voelter. 2011. Language and IDE Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers (Lecture Notes in Computer Science, Vol. 7680)*. Springer, 383–430. https://doi.org/10.1007/978-3-642-35992-7_11