Filling Typed Holes with Live GUIs

Anonymous Author(s)

Abstract

Although text editing is powerful, some types of expressions are more naturally represented and manipulated graphically. This paper introduces *live literals*, or *livelits*, which allow clients to fill holes of these types by directly manipulating a provider-defined GUI embedded persistently into otherwise symbolic code. Uniquely, livelits are compositional: the GUI can itself contain spliced expressions, which have full editor support and can in turn contain other livelits. Splices are typed and the system ensures that livelits treat splices hygienically. Livelits are also uniquely live: they can offer immediate 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 tasked with filling. This paper introduces livelits with case studies that exercise these novel capabilities. We implement livelits in Hazel, a live programming environment able to typecheck and run programs with holes. We then define a typed lambda calculus that captures the essence of livelits as live graphical macros. The macro expansion process has been mechanized in Agda.

1 Introduction

Text-based program editors are flexible and expressive 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.

As a simple example, consider 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: 255, g: 178, b: 45, a: 100 }`. The problem with this textual user interface for color selection is that it offers no live feedback about which color has been selected and limited editing affordances for tweaking the selected color. Analagous critiques apply to strictly textual user interfaces for countless other data structures, such as vector graphics, animation parameters, musical sequences, audio filters, board game states, GUI widgets and layouts, tabular data, plots, geospatial data, neural network diagrams, biological neuron models, mathematical diagrams, and so on.

Practitioners in domains where manipulating data of types like these is a central activity have largely eschewed general-purpose programming environments in favor of more specialized graphical end-user applications, like image and video editors, music composition software, level design tools, and bespoke GUIs written by students or lab technicians, in large part because these applications take seriously the need for domain-specific forms of live feedback, graphical data representations, and direct manipulation affordances, e.g. 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 a function that computes portions of an otherwise directly constructed vector graphic, or to transform a directly constructed musical sequence by passing it through a series of symbolically defined functions. Moreover, it is difficult to add new affordances or to compose affordances in ways that the application developer did not anticipate. Users cannot easily make even simple changes like replacing a numeric text box in a dialog with a slider, much less more ambitious changes like installing an alternative visual interface for expressing geospatial data queries into a database frontend.

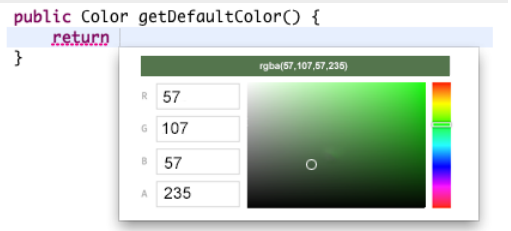
This paper aims to resolve this tension between programmatic and direct manipulation user interfaces by designing a programming environment that is able to surface GUIs when working with types for which they are useful, while retaining full support for symbolic program manipulation and the abstraction and composition mechanisms available in modern general-purpose programming languages.

1.1 Background

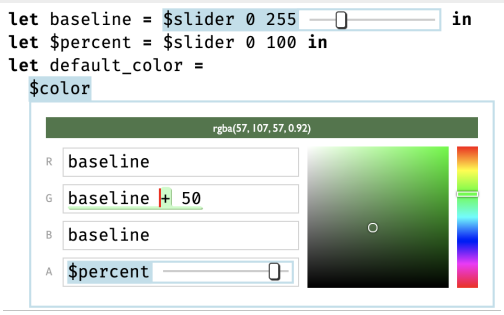
Of course, we are not the first to integrate direct manipulation interfaces into symbolic programming environments. The prior work most relevant to this paper is the Graphite system for Eclipse for Java, demonstrated in Fig. 1a [32]. Graphite allows a library provider to associate a GUI, called a *palette*, with a type (via a Java class annotation). Wherever an expression of this type is needed, i.e. wherever there is a *hole* of that type in the program (as determined by Eclipse's online parser and typechecker), the environment offers the client the option, via the code completion menu, to interact with the palette. Once the interaction is finished, the palette generates a Java expression to fill the hole. Figure 1a, adapted from this prior work, demonstrates a color palette invoked using Graphite. When the user presses the `Enter` key, the Java expression `new Color(173, 173, 173, 85)` is inserted at the cursor and the palette disappears. Several related systems, such as the **mage** system for the Jupyter notebook environment [19] behave fundamentally similarly. Projectional editing, the visual macro system in Racket [3], and a number of other designs also confront this general problem of integrating GUIs and code.

PL'21, January 01–03, 2018, New York, NY, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(a) Prior Work: Graphite [32]



(b) This Paper: Livelits are live and compositional



(c) Case Study: Grading with Livelits

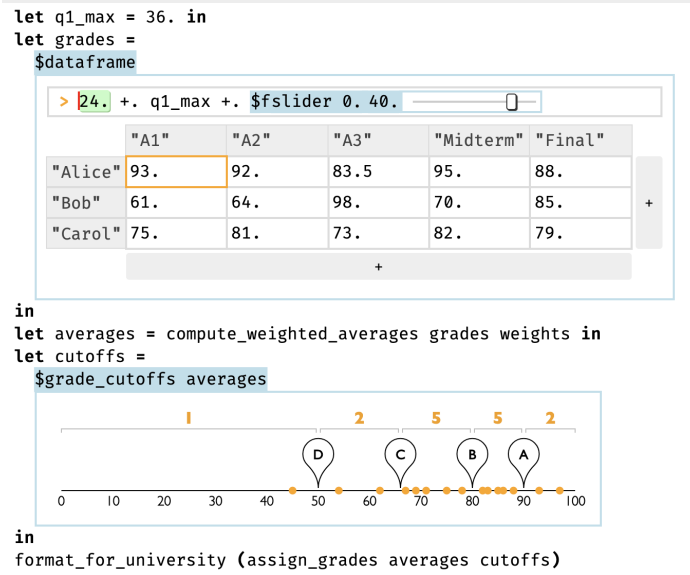


Figure 1. Introductory Examples

Omar et al. [32] evaluated Graphite by surveying 473 developers and Kery et al. [19] 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, 19, 32], and the Graphite survey solicited dozens of additional use cases from participants, which the authors systematically taxonomize [32]. We take these extensive empirical findings as evidence for, and a showcase of, the value of this class of mechanisms for integrating GUIs into code.

1.2 Contributions

We turn our attention in this paper to several fundamental technical deficiencies that limit GUI providers and clients using these prior systems. 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. (We describe which subset of these properties are achieved by prior systems, including those just mentioned, in Sec. 6.)

Decentralized Extensibility. Providers define livelits in libraries, and clients invoke livelits by name. Livelit names, e.g. `$color`, are prefixed by `$`, to distinguish them from variables.

Persistence. Livelits 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 [9]) where only the model needs to be persisted. The dynamic meaning of a livelit is determined by macro expansion.

Hygienic Composition. Livelits support sub-expressions directly in the GUI, which we call *splices* (after Omar and Aldrich [28]). Fig. 1b demonstrates splicing: the RGBA components are splice editors, so the client can define a variable, `baseline`, to relate the color components and use a slider livelit inline to specify the alpha component.

Crucially, composition is strictly governed by a hygiene discipline that ensures (1) **capture avoidance**, i.e. that variables that the client uses in splices will not capture expansion-internal bindings; and (2) **context independence**, i.e. that the livelit can be invoked in any program context.

Parameterization. Livelits can form 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 specifies the type of expansions it generates, and parameters and splices also specify types, so livelits are compatible with type-driven methodologies and tools. Together with the hygiene discipline, this allow clients to reason abstractly about expansions, i.e. without inspecting the expansions or livelit implementations directly.

Liveness. Uniquely, livelits can evaluate splices throughout the editing process (i.e. in a *live* manner [40]) 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 live programming with typed holes developed by Omar et al. [30]. We support live evaluation even for livelits that appear inside a function. Multiple function calls lead to multiple closures that the client can select between.

Outline. We begin in Sec. 2 by introducing livelits from the perspective of client programmers. Our examples are organized into case studies and are chosen to demonstrate the novel contributions of this paper. In Sec. 3, we consider the livelit provider's perspective by introducing livelit definitions with a detailed example. In Sec. 4, we define 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. In Sec. 5, we provide a more detailed account of our two implementations of livelits. Our primary implementation, used in the screenshots in the paper, is integrated into Hazel, a live programming environment designed around hole-driven development. We have also prototyped livelits within a standard text editor. Additionally, we discuss factors that must be considered when integrating livelits into languages with side effects. In Sec. 6, we compare livelits to related work using the design properties outlined above as a rubric. Finally, we conclude in Sec. 7 after a discussion of present limitations and 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. 2.4.3. These case studies have been implemented in Hazel, a browser-based live programming environment for a dialect of Elm. 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 the instructor to record grades using a natural tabular interface, visualize this data in one of a finite number of plot styles, and perform basic computations, with results updated live. However, these affordances are limited. It is difficult to package up 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's grading system).

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

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, `$dataframe`, that implements a tabular user interface. The formula bar allows the selected cell to be filled with an arbitrary Hazel expression. For the sake of demonstration, we show a cell that has been filled using another livelit, `$slider`, in combination with symbolic manipulation. The table itself displays not the expression itself but rather its value, just as in a spreadsheet.

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

Next, the instructor wants to "eyeball" reasonable `cutoffs` between letter grades 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.

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 expand to expressions. For example, the expansion of Fig. 1c is:

```
let grades = Dataframe (
  ["A1", "A2", "A3", "Midterm", "Final"],
  [("Alice", [24. +. 36. +. 33.,
              92., 83.5, 95., 88.]),
   ("Bob", [61., 64., 98., 70., 85.]),
   ("Ciri", [75., 81., 73., 82., 79.]),
   (* ... *) ]) in
let averages = compute_averages grades weights in
let cutoffs = (.A 90., .B 80., .C 70., .D 60.) 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 (which specifies the expansion logic as we will describe in Sec. 3). After all, function clients do not need to look inside function bodies to reason about types and binding. Instead, in the words of Reynolds [38], "type structure is a syntactic discipline for maintaining levels of abstraction". Livelits maintain this discipline by several means, described next in Sec. 2.2.1-2.3.3.

2.2.1 Expansion Typing.

To support abstract reasoning about the type of the expansion, 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) [33].

2.3 Compositionality

Livelits are compositional: they can work with sub-expressions in the form of parameters and splices.

2.3.1 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 [30].)

2.3.2 Splices.

Spliced 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 `$dataframe` GUI in Fig. 1c has a corresponding splice. The formula bar at the top allows the user to edit the splice corresponding to the selected cell, and all of Hazel's editing affordances are available when the client does so. Unlike parameters, the number of splices can change as the user interacts with the livelit, e.g. when changing the number of rows or columns in a `$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 surfaces and uses the expected type when the cursor is in the splice [33].

2.3.3 Hygienic Composition.

Ensuring that clients can reason about binding while leaving expansions invisible requires a hygiene discipline that enforces *capture avoidance* and *context independence* [2, 28].

Capture Avoidance. Splices and parameters can appear anywhere in the expansion. This becomes problematic when they appear under a binder, e.g. in the body of a function or `let` binding. Naïvely, this could cause inadvertent capture of the bound variable 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 len = strlen <splice1> in
Some (<splice2> + len)
```

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 are 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. We discuss how this is implemented in Sec. 3.1.5.

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. 3.1.5).

2.4 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.

2.4.1 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. [30]. Evaluation proceeds around these holes, producing a result containing corresponding hole closures, i.e. holes with environments.

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

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 the corresponding 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 [30]. 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.

2.4.2 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. 3.1.3.

2.4.3 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 now take the perspective of a livelit provider. Fig. 3 defines `$color` from Fig. 1b. We omit certain incidental details and use unimplemented syntactic sugar, including Haskell-style **`do`** notation and quasiquotation, for presentation.

Livelit definitions are scoped and packaged like any other definition. They consist of a declaration and an implementation. Line 2 of Fig. 3 is `$color`'s declaration, which defines its name and its expansion type. Livelit parameters would also appear here. The declaration is part of the client interface, as discussed in Sec. 2.2.1-2.3.1.

3.1 Livelit Implementations

The curly braces delimit the livelit's implementation. Livelits are implemented using a variation on the functional model-view-update architecture popularized by the Elm language, upon which Hazel is based [9]. We add a fourth component, expansion generation. In addition, we add a simple monadic framework (*a la* Haskell [25]) to provide the necessary interface between the livelit and the programming environment, while retaining a pure functional programming model. (We discuss imperative languages in Sec. 5.4.) Livelit implementations can themselves invoke other livelits (see Sec. 4.2.1).

3.1.1 Model. The state of a livelit invocation is encoded in its model value. Line 3 of Fig. 3 specifies the livelit's *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 livelit state is persisted in the syntax tree, so the system requires that the model type supports automatic serialization (so, no functions 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 **`UpdateMonad`**, discussed below, that returns the initial model value after generating four new splices using the `new_splice` command:

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

This command creates a splice of the given type and, optionally, specifies its initial contents. It returns a splice reference, which uniquely identifies that splice. Bolded types are defined in the standard library. The **`Typ`** and **`Exp`** types encode the syntax of Hazel's types and expressions. These two arguments are expressed here using quasiquotes, e.g. `` `0` `` [5].

The system checks that the splice type and initial content are valid assuming only the parameters and explicitly specified set of captured bindings on Line 6. Here, the capture set is empty because **`Int`** is a built in type and the initial content is closed. This serves to ensure *context independence*: the livelit needs to make no assumptions about the typing context at invocation sites. We use an explicit capture set, rather than implicitly capturing all bindings at the definition site, to ensure that private bindings are not unintentionally leaked to clients as detailed by Omar and Aldrich [28].

```
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    captures { }
7
8    let init : UpdateMonad(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 -> ViewMonad(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           (* indetermine 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 -> UpdateMonad(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. Livelit Implementation

3.1.2 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 side of Fig. 1b. Actions are emitted from event handlers (e.g. click handlers) defined in the computed `view`, and actions are consumed by the `update` function, causing a change to the model. Let us discuss each of these functions in turn in Sec. 3.1.3-3.1.4.

3.1.3 View. The `view` function computes the view given a model and access to the commands in the **`ViewMonad`**. The computed view is a value of type **`Html`**`(Action)`. The type family **`Html`**`(a)` provides a simple immutable encoding of an HTML tree, where the type parameter `a` is the type of actions that are emitted by event handlers that can be attached to elements of the tree, e.g. `on_click` and so on. We elide the details of the particular user interface in Fig. 1b to focus on three key editor mechanisms exposed by **`ViewMonad`**: live evaluation, splice editors, and result rendering.

Live Evaluation. As discussed in Sec. 2.4, the view can depend on the result of evaluating a splice under the closure the client has selected. The interface between the view and the live evaluator is via the following command:

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

The `None` case arises when evaluation is not possible, e.g. because no closures are collected. If a result is available, the **`Result`** type 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 [30].

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 indeterminacy with an "X" over the 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 simply playing silence or some default sound when it encounters them. 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) -> ViewMonad(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 shows 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 `eval` command:

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

3.1.4 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) -> UpdateMonad ()
```

The system checks that the expression must satisfy the splice type and onlys make use of the capture set, as in Sec. 3.1.1.

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 `update` function does not itself have the ability to request evaluation, because the model should not depend directly on which closure the user has selected.

3.1.5 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 this expansion based on the model.

The expansion can include splices, but the system does not make the contents of a splice available directly as a value of type **`Exp`**. Instead, `expand` returns a *parameterized expansion* paired with a list of **`SpliceRef`**s. The parameterized expansion is an encoding (of type **`Exp`**) of a function that takes an argument for each listed **`SpliceRef`**s. 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. Here, the parameterized expansion for `$color` encodes a function of type **`Int`** `->` **`Int`** `->` **`Int`** `->` **`Int`** `->` `Color`.

This parameterization strategy makes it simple to enforce hygiene: the parameterized expansion can depend only on the livelit parameters and capture set, whereas the splices are entered by the client and so they can depend only on the client site typing context. The use of function application ensures that splices are capture avoiding. We consider this more formally in the next section.

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

Figure 4. Syntax of types, τ, unexpanded expressions, $\hat{e}$, expanded 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, $[d_1/x_1, \cdots, d_n/x_n]$. We elide standard forms related to product, sum, and recursive types.

4 A Simply Typed Livelit Calculus

In order to communicate 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 expressions*, e, before being *elaborated* to *internal expressions*, d, for evaluation. All three sorts are classified by the same types, τ. We include partial functions, products, sums, and recursive types, all in their standard form [14], but this specific type structure is not critical. Any language expressive enough to encode its own abstract syntax would be a suitable basis for a livelit calculus.

We begin in Sec. 4.1 with a terse overview of the external and internal languages, which are adapted from Hazelnut Live [30]. We then detail livelit expansion (Sec. 4.2) and liveness via closure collection (Sec. 4.3).

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 $(\!|\,|\!)^u$ where u is a *hole name* [30]. We omit non-empty holes (which internalize type inconsistencies [31]) and type holes (which operate like the unknown type from gradual type theory [31, 39]). 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 $x : \tau$ [14].

The internal language is a contextual type theory [27], i.e. the typing judgement is of the form $\Delta; \Gamma \vdash d : \tau$ where Δ is a finite set of hole typing assumptions of the form $u :: \tau[\Gamma]$. We need this 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. The hole context ensure that each closure associated with hole u, written $(\!|\,|\!)^u_\sigma$, shares a type and can be filled with expressions valid under a shared context.

External expressions are given dynamic meaning by typed elaboration to internal expressions, d, according to the judgement $\Gamma \vdash e \rightsquigarrow d : \tau \dashv \Delta$. The main purpose of elaboration is to initialize the substitution σ on each hole closure, which operates to capture the substitutions that have occurred around that hole during evaluation. The key rule is:

$$\frac{}{\Gamma \vdash (\!|\,|\!)^u \rightsquigarrow (\!|\,|\!)^u_{\text{id}(\Gamma)} : \tau \dashv u :: \tau[\Gamma]}\ \texttt{Elab-Hole}$$

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

$$\vdash (\lambda x.(\!|\,|\!)^u)\ \underline{5} \rightsquigarrow (\lambda x.(\!|\,|\!)^u_{[x/x]})\ \underline{5} : \text{nat} \dashv u :: \text{num}[x : \text{nat}]$$

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

$$(\lambda x.(\!|\,|\!)^u_{[x/x]})\ \underline{5} \Downarrow (\!|\,|\!)^u_{[\underline{5}/x]}$$

All of these judgements are adapted directly from Hazelnut Live, differing only in that we use a declarative rather than an algorithmic (bidirectional) formulation for simplicity. Rather than restating the rules, we simply state the key governing metatheorems and refer the reader to the prior work and our Agda mechanization (Sec. 4.2.3) for the full details [30].

First, we have that 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* $\Delta; \Gamma \vdash d : \tau$.

Next, we have that evaluation of a closed expression with holes produces a final (i.e. irreducible) preserves typing.

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 variables and functions, both shown in Fig. 5, are simple.

4.2.1 Livelit Contexts.

Livelit definitions are collected in the livelit context, Φ, which maps livelit names to livelit definitions of the following form:

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

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.

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

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

Figure 5. Expansion

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

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. [29]).

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 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 [11], by requiring that these definitions appear in separately compiled packages [28], or by using live programming mechanisms such as those in Hazel to evaluate "up to" each definition before proceeding [30].

4.2.2 Hygienic Livelit Expansion.

Unexpanded expressions are unique in that they include livelit invocations:

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

Here, $\$a$ names the livelit being applied. 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 (discussed in Sec. 3.1.1).

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

1. **Lookup.** The first premise looks up the livelit name in the livelit context.
2. **Model Validation.** The second premise serves to ensure 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 given above, is of type Exp.
4. **Decoding.** The fourth premise decodes d_{encoded}, producing the *parameterized expansion*, $e_{\text{pexpansion}}$.
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 explicitly recorded in the livelit. We must take care to 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. In this simple formulation of the system, we maintain context independence by requiring that the parameterized expansion be closed. Consequently, any necessary helper functions used in the expansion must be provided explicitly by the client via a splice. In Sec. 3, we discussed how the use of explicit capture sets eliminates this client burden. For simplicity, we omit capture sets from the calculus, but see [28] for a formal 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.

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 properties established for 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 mechanized the external and internal language and typed expansion and proven the aforementioned theorems using the Agda proof assistant, building on the Agda mechanization of Hazelnut Live [30]. This mechanization is available in the supplement.

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. 2.4.1. 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, the hole is in place of the parameterized expansion, which we instead record in a *cc-context*, Ω, on the side. The key rule for the cc-expansion judgement, $\Phi; \Gamma \vdash_{\text{cc}} \hat{e} \rightsquigarrow e : \tau \dashv \Omega$, is:

CCApLivelit

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

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. 2.4.1. 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 [27, 30]). 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 EApLivelit, 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 [30], we can delay hole filling until d_{cc} has first been evaluated to d'_{cc}. □

In a language with side effects, one could evaluate cc-expansions in an alternative mode where the full expansion of each livelit invocation is evaluated in the order that corresponding livelit hole, and any expressions blocked on it are resumed immediately. The results would need to be recorded in memory for use in the filling phase. 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 logging *a la* Lamdu [23]. This is more limited in situations where a livelit is never evaluated, as sometimes occurs during development of a livelit in a function which has yet to be applied, but does not require cc-expansion. 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 [34].

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 [31], rather than a text editor, although the developers are aiming to maintain a text-like experience (this effort is orthogonal to our own).

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 livelits in a textual program editor, Sketch-n-Sketch [8, 15], which recently added support for the necessary mechanisms [24]. Livelit interaction causes the serialized model in the text buffer to be changed, which updates the view. This proof-of-concept is not at feature parity with our main implementation, but it demonstrates that a syntax-recognizing text editor [3, 4, 18] is sufficient to support livelits, albeit with some gaps in availability when there are syntax errors.

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 considering a number of other layout options, e.g. inline-block literals *a la* Wyvern [29], 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, then the state exists at edit-time. The most viable approach would be to retain the model-view-update architecture and specify that any transient mutable state will be reset between updates. If side effects occur in livelit expansions, then the closure collection mechanism requires more care to avoid unsoundness, as described in Sec. 4.3.

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 [32]. Subsequent work on **mage** further explores this design space [19]. 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) [29] and typed literal macros (TLMs) [28] 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 hygiene we take is 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.

Recent work on a interactive visual macro system in 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 hygienic, nor is there 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.

Our structural delimitation of splices is reminiscent of work on *language boxes* [37], which focused on combining different notations, primarily textual, using structural delimiters inserted explicitly using a special-purpose editor.

There has been a long line of research on *projectional editing*, where the user edits graphical representations (projections) of code constructs [20, 26, 35, 36]. 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 [20] and MPS [42] have made it easier to define new projectional editors. However, these systems generate entire editors, whereas livelit definitions are defined in libraries. Furthermore, livelit definitions are governed by a type and binding discipline. Finally, livelits are uniquely live, building on Hazelnut Live [30].

A number of notebook systems, including Mathematica, Jupyter [12], 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 [21, 22, 41]. Livelits differ from these systems in directionality: visualization systems are given code and generate a visualization, whereas livelits are generate code where there would otherwise be a hole.

Conal Elliott's work on tangible functional programming [10] 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 visual representation is customizable.

The Vital programming environment for Haskell [13] 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 [17]

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. [32] 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 [16], 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 [28].

The strict hygiene 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, 15].

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.

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, Tartu, Estonia, August 14-21, 2004, Revised Lectures (Lecture Notes in Computer Science, Vol. 3622)*, Varmo Vene and Tarmo Uustalu (Eds.). Springer, 210–244. https://doi.org/10.1007/11546382_5

[2] Michael D. Adams. 2015. Towards the Essence of Hygiene. *SIGPLAN Not.* 50, 1 (Jan. 2015), 457–469. https://doi.org/10.1145/2775051.2677013

[3] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *PACMPL* 4, OOPSLA (2020).

[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 et al. 1999. Quasiquotation in Lisp.. In *PEPM*. Citeseer, 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 *Conference on Programming Language Design and Implementation (PLDI)*.

[9] Czaplicki, Evan. [n.d.]. Elm Architecture. ([n. d.]). https://guide.elm-lang.org/architecture/.

[10] Conal Elliott. 2007. Tangible Functional Programming. In *ICFP*.

[11] Matthew Flatt. 2002. Composable and compilable macros: : you want it when?. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 72–83. https://doi.org/10.1145/581478.581486

[12] 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 '13, Denver, CO, USA, March 6-9, 2013*. 579–584.

[13] Keith Hanna. 2002. Interactive visual functional programming. *ACM SIGPLAN Notices* 37, 9 (2002), 145–156.

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

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

[16] 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 2020, Dunedin, New Zealand, August 10-14, 2020*, Michael Homer, Felienne Hermans, Steven L. Tanimoto, and Craig Anslow (Eds.). IEEE, 1–5. https://doi.org/10.1109/VL/HCC50065.2020.9127256

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

[18] 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, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, William E. Riddle and Peter B. Henderson (Eds.). ACM, 7–14. https://doi.org/10.1145/800020.808243

[19] 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, Virtual Event, USA, October 20-23, 2020*, Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller (Eds.). ACM, 140–151. https://doi.org/10.1145/3379337.3415842

[20] 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, Seattle, WA, USA, October 23-26, 2005*, Patrick Baudisch, Mary Czerwinski, and Dan R. Olsen (Eds.). ACM, 3–12. https://doi.org/10.1145/1095034.1095037

[21] 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.

[22] Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *CHI '20: CHI Conference on Human Factors in Computing Systems, Honolulu, HI, USA, April 25-30, 2020*, Regina Bernhaupt, Florian 'Floyd' Mueller, David Verweij, Josh Andres, Joanna McGrenere, Andy Cockburn, Ignacio Avellino, Alix Goguey, Pernille Bjøn, Shengdong Zhao, Briane Paul Samson, and Rafal Kocielnik (Eds.). ACM, 1–7. https://doi.org/10.1145/3313831.3376494

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

[24] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 109:1–109:29. https://doi.org/10.1145/3408991

[25] Simon Marlow et al. [n.d.]. Haskell 2010 language report. ([n. d.]).

[26] 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

[27] 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

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

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

[30] 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).

[31] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Principles of Programming Languages (POPL)*.

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

[33] Hannah Potter and Cyrus Omar. 2020. Hazel Tutor: Guiding Novices Through Type-Driven Development Strategies. In *HATRA*.

[34] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A Core ML Language for Tierless Web Programming. In *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10017)*, Atsushi Igarashi (Ed.). 377–397. https://doi.org/10.1007/978-3-319-47958-3_20

[35] 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) on SIGSOFT'96 workshops*. 232–236.

[36] 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, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, William E. Riddle and Peter B. Henderson (Eds.). ACM, 30–41. https://doi.org/10.1145/800020.808246

[37] Lukas Renggli, Marcus Denker, and Oscar Nierstrasz. 2009. Language Boxes. In *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5969)*, Mark van den Brand, Dragan Gasevic, and Jeff Gray (Eds.). Springer, 274–293. https://doi.org/10.1007/978-3-642-12107-4_20

[38] John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

[39] 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

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

[41] 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.

[42] Markus Voelter. 2011. Language and IDE Modularization and Composition with MPS. In *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer.