Livelits: Filling Typed Holes with Live GUIs (Extended Abstract)

Cyrus Omar
University of Chicago
comar@cs.uchicago.edu

Nick Collins
University of Chicago
nickmc@uchicago.edu

David Moon
University of Colorado Boulder
domo9239@colorado.edu

Ian Voysey
Carnegie Mellon University
iev@cs.cmu.edu

Ravi Chugh
University of Chicago
rchugh@cs.uchicago.edu

Abstract

Text editing is powerful, but entering text is often not the most natural way to construct certain types of expressions, such as those encoding colors, animations, musical sequences, tabular data, plots, widgets, graphs, and various other data structures. We introduce *live literals*, or *livelits*, which allow the programmer to fill holes of such types by directly manipulating a graphical user interface. Livelits are compositional: this GUI can itself contain typed holes that the programmer can fill with expressions that might contain other livelits. Livelits are also live: they can provide immediate feedback about the dynamic implications of the programmer's choices, even when the subexpressions mention bound variables, because the livelit is given access to closures associated with the hole that the livelit is tasked with filling. We are implementing livelits within Hazel, a live functional programming environment that assigns meaning to every edit state.

1 Introduction

In the popular imagination, programmers spend their days interacting with screens full of text. Text editors are indeed flexible and powerful user interfaces. However, textual user interfaces are not the best tool for every job. In particular, there exist many data structures for which a non-textual user interface might be more natural.

For example, consider a record type encoding colors:

```
type color = { r: Int, g: Int, b: Int };
```

It is possible to construct a particular color by writing out its textual representation. The problem is that this sort of textual user interface for color selection provides no feedback and limited affordances. In other words, it is difficult for the programmer, or the reader, to know which color is represented and to interactively explore the space around it if a slightly different color is desired.

Color selectors in graphical end-user applications, in contrast, provide live feedback (by displaying the selected color) and richer affordances of various designs (e.g. they might present swatches or a spectrum of colors arranged in two dimensions, which the user can explore using the mouse).

The trade-off, of course, is that these applications have limited or no support for abstraction and composition. It is difficult to, for example, assign a color to a variable for use in multiple places, or to darken a color by passing it to a function, or to compute the value of, for example, the red component of a color, or to use a slider to manipulate its value when one has not already been provided.

We believe that it possible to resolve this apparent tension between direct and programmatic manipulation, i.e. to design a programming system that provides the rich feedback and affordances of graphical end-user applications in situations where they are useful, while supporting the full array of abstraction and composition mechanisms available in modern general-purpose programming languages.

The basic idea, introduced in the prior work on the Graphite system for Java [Omar et al. 2012], is simple: we allow library providers to associate graphical user interfaces with types. The programming environment provides the programmer with the option, via the code completion menu, to activate the associated GUI wherever an expression of the corresponding type is needed, i.e. wherever there is a *hole* of such a type in the program. It is the GUI's job to fill the hole, i.e. to generate an appropriate expression of the needed type. Figure 1, reproduced from our prior work, demonstrates an example of a simple color chooser.

There are two major limitations with this prior work. First, GUIs in Graphite are **ephemeral**, i.e. they disappear once the initial interaction is complete, leaving behind only the textual representation. This means that feedback and assistance is available only to the programmer that first writes the code.

Second, Graphite does not provide any compositional way to enter subexpressions within the GUI. This implies that Graphite's GUIs can only generate **closed expressions**. For example, in Figure 1, there is no way to specify that the `R`, `G`, or `B` values of the color being entered should be computed by a specified expression—only closed colors are supported.

2 Livelits

To address these limitations, we introduce live literals, or livelits. Figure 2 shows a mockup of the definition and application of a livelit named `$grade_cutoffs` for adjusting grade cutoffs, represented as values of record type `grade_cutoffs`. Like textual literal forms (e.g. list literals), livelits are alternative representations of expressions of the associated type

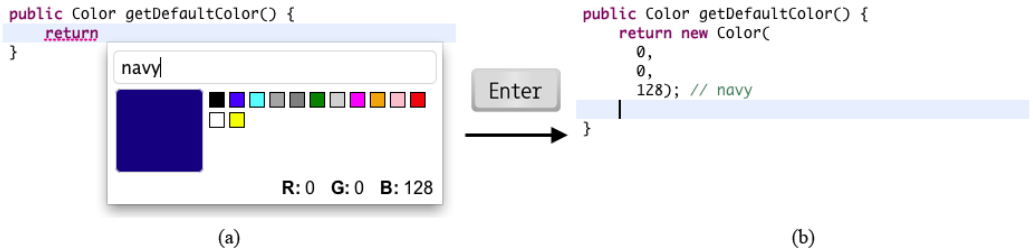

Figure 1. This figure, reproduced from the prior work [Omar et al. 2012], shows (a) a simple example GUI associated with the `Color` type, and (b) the code generated by this GUI once the ephemeral interaction is complete.

```
type grade_cutoffs = { a: float, b: float, c: float, d: float, f: float };

livelit $grade_cutoffs at grade_cutoffs {
  type model = ...; let init_model: model = ...;
  type msg = ...;   let update(msg: msg, model: model) : model = ...;

  let view(ctx: Ctx.t, env: Env.t, model: model) : html(msg) =>
    let hole_spec = { id: "data", ty: TList(TFloat) };
    let data_input = Div [] [Text "data = "; HtmlHole hole_spec];
    let data_val = Env.run env (EHoleRef "data");
    /* ... now construct rest of UI (omitted) ... */
  }
  let to_exp(model: model) : expr_ast = /* construct an AST of grade_cutoffs type */
};

let cutoffs: grade_cutoffs = $grade_cutoffs;
```

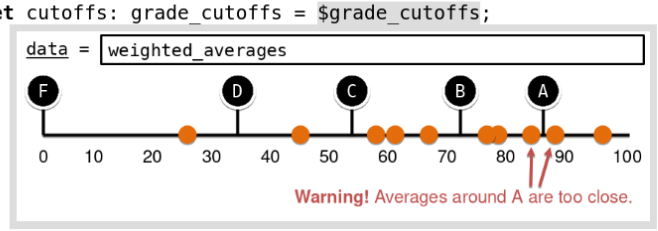

Figure 2. A mockup of a livelit for adjusting grade cutoffs. Livelits are persistent and can access the live environment.

[Omar and Aldrich 2018]. We are implementing livelits in Hazel (hazel.org), a live functional programming environment with support for typed holes [Omar et al. 2017]. We plan to perform a live demo.

The definition of the livelit, outlined in Figure 2, follows the Elm architecture, i.e. there are types representing the abstract model and the messages that the GUI generates. Livelits are **persistent** rather than ephemeral, i.e. the model is recorded in the underlying syntax tree. The `view` function generates the GUI, implemented using HTML, on demand (so the view is not persisted). Another function, `to_exp`, is responsible for generating the underlying expression, called the *expansion*, from the `model`. While other projectional editors, e.g. those generated by Citrus [Ko and Myers 2005], also support persistent GUIs in code, they are not user-extensible.

The GUI can itself contain typed holes, represented by the `HtmlHole` constructor. In this case, there is a single hole for entering an expression of type `list(float)`, i.e. the list of weighted averages. In this example, the user has filled this hole with a variable, `weighted_averages` (the definition of which is not shown). In other words, livelits support **open expressions** and therefore interact cleanly with standard abstraction mechanisms, i.e. they can appear under binders. We follow the reasoning principles for literals with spliced sub-expressions established by Omar and Aldrich [2018].

The main complication when dealing with open expressions relates to how live feedback is to be generated. Given just the symbolic expression in the hole, it would be impossible to plot (as orange dots) the actual data from the list that the variable refers to. To resolve this issue, the system evaluates the program as if it the livelits were empty holes, relying on the support for evaluating incomplete programs described recently by Omar et al. [2019]. The result of evaluation is an expression containing hole closures, i.e. holes equipped with environments. The livelit can then evaluate the expression in the hole against the closure selected by the user (not shown) using `Env.run`.

References

Andrew Jensen 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)*. https://doi.org/10.1145/1095034.1095037

Cyrus Omar and Jonathan Aldrich. 2018. Reasonably Programmable Literal Notation. *PACMPL* 2, ICFP (2018), 106:1–106:32. https://doi.org/10.1145/3236801

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *PACMPL* 3, POPL (2019), 14:1–14:32. https://doi.org/10.1145/3290327

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 2017)*. http://dl.acm.org/citation.cfm?id=3009900

Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. 2012. Active Code Completion. In *34th International Conference on Software Engineering, ICSE 2012*. https://doi.org/10.1109/ICSE.2012.6227133