Live and Direct Functional Programming with Palette Expressions

Anonymous Author(s)

Abstract

We present *nested palettes*, an approach to programming custom graphical user interfaces (GUIs) that are used to build program expressions—to "fill holes" within an incomplete program. Nested palettes serve as an alternative and complementary code editing mechanism to traditional text-editing. Compared to other "hybrid" text-and-GUI code editors, nested palettes provide several unique capabilities. First, a palette persists even after user interactions produce a complete expression that fills the hole, so that the programmer can return to the GUI to make subsequent changes. Second, palettes for smaller expression types can appear nested within those for larger expressions, allowing palettes to be composed in complex ways. Third, palettes have access to the execution environment—even when the overall program is not yet complete—and can, thus, provide live feedback to the programmer while working to fill an incomplete expression.

This paper investigates the design and implementation of nested palettes in several steps. First, we propose a formal framework for nested palettes within Hazelnut, a foundational semantics for live programming that provides continuous code editor services through a combination of structured edits and dynamic evaluation of incomplete programs. Second, to demonstrate the expressiveness of our framework, we describe several example palettes, many of which are informed by a prior study that identified desirable use cases for hybrid editors. Lastly, we provide a prototype implementation of nested palettes within Hazel, a programming environment based on the Hazelnut theory extended with palettes. Based on our experience, we believe nested palettes are an extensible and expressive approach for mixing text- and GUI-based code editing.

Keywords Live Programming, Palettes, Hazel, Code Editors

1 Introduction

Palettes...

2 Examples

2.1 Boolean

Checkbox

2.2 Matrix

Simple example

2.3 Grade Cutoffs

Uses liveness more obviously

2.4 Table

Uses type reflection

2.5 Pixel Art

Cool example

2.6 Regex

Similar to Graphite paper, can cite the empirical study we did there

2.7 Forms

Show off composition + mention full-screening stuff

2.8 Equation Editor, Judgement Editor and Category Diagrams

Things that PL people like

2.9 TikZ diagrams

...

PL'18, 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

```
module type IPalette = {
  type args
  type output
  type model
  type msg
  val init    : args -> HRG.t(model)
  val update  : (model, msg) -> HRG.t(model)
  val view    : model -> HRE.t(Html.t(msg))
  val compute : model -> HRE.t(output)
}
```

Figure 1. `IPalette` module type

3 Defining Palettes

Palette definitions take the following general form:

```
palette $name
  (arg1 : t1)
  ...
  (argn : tn)
  at t
  implementation P in package pkg;
```

The static semantics requires the following:

1. `arg1` ... `argn` for $n \geq 0$ are distinct labels
2. `t1` ...`t1` are valid types
3. `t` is a valid type
4. and `P` identifies a module that can be loaded from package `pkg` such that

   ```
   P : IPalette
       with type args = {
         arg1 : HoleRef.t<t1>,
         ...,
         argn : HoleRef.t<tn>
       }
       with type output = t
   ```

   where `IPalette` is defined in Fig. 1 and `HoleRef` in Fig. 2.

3.1 Background: Elm architecture

3.2 GUIs with Typed Holes

3.3 Palette-Specific Actions?

3.4 Reasoning Principles

3.5 Deriving Palettes from Type Definitions?

4 Formal System

5 Related Work

Graphite, Relit, projectional editors, notebooks / Mathematica,

6 Discussion

References

```
(* Abstract type of hole refs *)
module HoleRef : {
  type t('a)
}

(* HRG is the hole ref generation monad *)
module HRG : {
  type t('a)
  val fresh('a) : t(HoleRef.t('a))
  val bind : t('a) ->
             ('a -> t('b)) ->
             t('b)
  val return : 'a -> t('a)
}

(* HRE is the hole ref evaluation monad *)
module HRE : {
  type t('a)
  type result('a) = Value('a * Exp)
                  | Indet(Exp)
  val eval('a) : HoleRef.t('a) ->
                 t(result('a))
  val bind : t('a) ->
             ('a -> t('b)) ->
             t('b)
  val return : 'a -> t('a)
}
```

Figure 2. Modules for working with hole refs

Expressions	$e ::= \texttt{let palette}\ x = p\ \texttt{in}\ e$	Palette definition p as x in e
	$\mid p\ e$	Palette "instantiation" (?)
Typing contexts	$\Gamma ::= \cdot \mid \Gamma, x : \tau$	Empty context; Variable binding for value type τ
Types	$\tau ::= \texttt{Href}\ \langle \tau \rangle$	Hole reference type; parameterized by type within hole τ
	$\mid \texttt{Num} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2$	Numbers (basetype); Sum types; Product types
Palette definitions	$D ::= \left\{ \begin{array}{l} \tau_{\text{expand}}, \tau_{\text{expand}}, \tau_{\text{msg}}, \\ e_{\text{init}}, e_{\text{update}}, e_{\text{expand}}, e_{\text{view}} \end{array} \right\}$	Palette definition
Palette expressions	$p ::= D$	Body of palette definition: Types and expressions
	$\mid \lambda x.p$	Palette abstraction
	$\mid p\ e$	Palette application
Pallete typing contexts	$\Delta ::= \cdot \mid \Delta, x : T$	Empty context; Variable binding for palette type T
Palette types	$T ::= \texttt{Palette}\ \tau_{\text{expand}}\ \tau_{\text{model}}$	Palette type
	$\mid \tau \rightarrow T$	Palette arrow type

	Graphite	Projectional Editors	TLMs	Live Palette Expressions
interactive	Y	Y	N	Y
extensible	Y	N	Y	Y
persistent	N	Y	Y	Y
compositional	N	N	Y	Y
live	N	N	N	Y
formalized	N	N	Y	Y

Figure 3. Related Work Table