Dynamic Hover Detection: Creating GUIs in Elm

Max Goldstein & Eliot Alter

Abstract

The language Elm was designed to be a practical application of Functional Reactive Programming (FRP) to create Graphical User Interfaces (GUIs). We attempted to create a menu bar as commonly seen on desktop operating systems, and quickly discovered an apparent incompatibility between Elm's mouse-hover detection primitive and the constraints of the language. This concern only surfaces when dealing with highly dynamic data. We present a non-obvious technique to address the issue without modifying the Elm compiler. It generalizes to other functions in Elm's `Graphics.Input` library, which includes (besides hover detection) GUI mainstays such as buttons, checkboxes, and text fields. We also contribute other techniques for the representation and display of menus in Elm, and contrast our work with an existing Elm web application.

1 Introduction

Elm was introduced in March 2012 in Evan Czaplicki's senior thesis. [Cza12] He and his adviser, Stephen Chong, published a formal description of Elm's semantics at PLDI 2013. [CC13] Both papers are comprehensive overviews of Elm, and additionally provide excellent literature reviews of previous FRP GUI endeavors, of which there are several. Elm compiles down to JavaScript to run in the browser, making it remarkably portable.

Elm was designed to make implementing GUIs easier, so we decided to try to implement menus as an example GUI. The particular design places the top-level menu at the top of the screen, similar to Mac OS X and many Linux distributions, with selections coming down from the top. Menus extend when the top-level item is hovered upon, and remain extended while the mouse hovers over any item in the menu. Therefore it is necessary to know hover information about each menu item. This time-varying information is also used to detect selections upon click and highlight the moused-over item. It is simple to do this when the hover-detecting area is constant. This paper describes the much more

difficult task of managing time-varying hover information about time-varying areas.

There are two features of Elm we are deliberately avoiding. First is the extensive raster drawing library, `Graphics.Collage`. Dynamic hover detection is not problematic when using this library because it can be done purely using geometric collision detection. However, if we used this library, our GUI would be a single raster animation and not a DOM tree. Secondly, the `Graphics.Input` library contains wrappers around HTML checkboxes and dropdowns. We rejected creating GUIs with these and wanted to pursue something more general. This led to the choice of an interface typically found in the operating system rather than the browser. We do refer to the native GUI constructs to show how the technique we develop for hovering generalizes to them, as their API is similar to the one for hover detection.

It is difficult for the authors to assess what level of knowledge should be assumed on the part of the reader. Firstly, readers will range from Elm's creators and experienced users, who are already familiar with its restrictions and standard libraries, to those with no FRP background, who need an introduction to signals. Secondly, while we have found no prior discussion on either the problem we identify nor its solution, we cannot know for certain that either are novel. We continue in the hopes of presenting new and non-trivial techniques to the Elm community.

We contribute:

- The identification of an apparent limitation in Elm's hover detection library, solved by a non-trivial usage pattern that does not require modifications to the Elm compiler or runtime, and that generalizes to other functions in `Graphics.Input`. Concretely, this is the implementation of `hoverableJoin`.
- An implementation of desktop-style menus in Elm, which incorporates several noteworthy "tricks" as well as general style best practices.
- An analysis of TodoFRP, the current state-of-the-art in dynamic Elm GUIs. We demonstrate how it operates in the absence of our technique, and how it could operate in its presence.

Section 2 introduces Elm, signals, state, and the prohibition on signals of signals. It is targeted to readers familiar with functional programming but not FRP, and may be skipped by those already comfortable with Elm. Section 3 presents Elm's hover detection API for DOM elements, why it initially appears inadequate, and how it can be extended to meet our needs. Section 4 explains our menu implementation in detail. Section 5 analyzes TodoFRP. Section 6 concludes with a notice to the Elm community.

2 Elm for Functional Programmers

A typical functional program is *transformative*: all input is available at the start of execution, and after a hopefully finite amount of time the program terminates with some output. In contrast, Elm programs are *reactive*: not all input is available immediately and the program may indefinitely adjust output with each input. In simple programs, the inputs at a given time fully determine the output. More complex programs will take advantage of Elm's ability to remember state.

2.1 Signals: Time-varying values

A time-varying value of a polymorphic type `a` is represented by `Signal a`. For example, the term `constant 150` has type `Signal Int`. The combinator `constant` creates a signal whose value never changes. A more interesting signal is the primitive `Window.dimensions : Signal (Int, Int)`. This signal represents the browser window size and updates whenever it is resized. Signals are asynchronous in that they update at no set time, just as the window may remain the same size indefinitely. Signals update in discrete events, but are continuous in the sense that they are always defined.

The function `lift` allows us to execute a pure function on a signal of inputs, producing a signal of outputs. (Although lifting is a general functional concept, in Elm it has only this meaning.)

```
lift : (a -> b) -> Signal a -> Signal b
```

For example, we can multiply the width and height of the window together to find its area.

```
area : Signal Int
area = lift (uncurry (*)) Window.dimensions
```

In this case, the lifted function is multiplication, uncurried as to operate on pairs. As the Window library exposes width and height as both a pair and individually, we can also write `area2 = lift2 (*) Window.width Window.height`.

We can print the current area to the screen with `main = lift asText area`. The primitive `asText : a -> Element` renders almost anything into an `Element`, which represents a DOM element.[1]

[1] Those following along in an Elm compiler, such as the one available at `elm-lang.org/try`, should add `import Window` to the top of the file.

The signals in an Elm program can be thought of as a Directed Acyclic Graph (DAG). Many signals depend on other signals for their output. For example, the `area` signal depends on `Window.dimensions` signal. Similarly, the `area2` signal depends on both the values of the `Window.width` and the `Window.height` signals. When an event is fired on a signal, it propagates down the DAG to all signals who depend on that signal and the outputs of those signals are reevaluated, recursively.

2.2 Remembering State

Signals can remember state by using the `foldp` combinator. Familiar list folds apply a binary operation of an element and an accumulator to produce a new accumulator, over each list element in sequence. Folding from the *past* operates on all of the values of a signal as they occur and produces a signal of the accumulator.

```
foldr : (a -> b -> b) -> b -> [a]      -> b
foldp : (a -> b -> b) -> b -> Signal a -> Signal b
```

When the event signal updates, a pure function is called with the new event and the old accumulator (a default is supplied), producing a new accumulator that is the new value of the output signal. For example, `foldp max 0 Window.width` is a signal of the maximum width ever obtained by the window. With `foldp`, it is possible to create signals that depend on every event to ever occur on a signal. However, most folded functions do not store every value explicitly (cons is an exception) and space can be saved by remembering only the accumulator.

2.3 No Signals of Signals

The ability to create a signal dependent on past state has a potentially disastrous implication for space performance. Czaplicki and Chong explain:

> Intuitively, if we have signals of signals, then after a program has executed for, say 10 minutes, we might create a signal that (through the use of `foldp`) depends on the history of an input signal, say `Window.width`. [CC13]

That is, we can define a clock that measures the amount of time since program execution began:

```
clock : Signal Time
clock = foldp (+) 0 (fps 10)
```

The `fps n` combinator produces a signal of the time elapsed since its last event, updated `n` times a second. We sum these deltas starting with zero. Then we create a function from a time to one of two signals, one of which is trivial and one of which depends on the history of `Window.width`:

```
switcher : Time -> Signal Int
switcher t = if t < 10*minute
             then constant 0
             else foldp max 0 Window.width
```

We could, in theory, create

```
switched : Signal (Signal Int)
switched = lift switched clock
```

Why is this problematic? Czaplicki and Chong continue,

> To compute the current value of this signal, should we use the entire history of `Window.width`? But that would require saving all history of `Window.width` from the beginning of execution, even though we do not know whether the history will be needed later. Alternatively, we could compute the current value of the signal just using the current and new values of `Window.width` (i.e., ignoring the history). But this would allow the possibility of having two identically defined signals that have different values, based on when they were created. We avoid these issues by ruling out signals of signals. [CC13]

Indeed, Elm's type system is principled on disallowing signals of signals. Programs may not create in the future signals that depend on the past. Were they allowed to, the system must either remember indefinitely every event to ever occur on any signal, so that a newly created signal can use them, or tolerate signals that vary based only on when they were created, losing referential transparency.

To a reader familiar with Haskell, this means signals are functors (and in fact applicative functors) but not monads, as monads support the following operation:

```
join :: Monad m => m (m a) -> m a
```

Such an operation for signals would condense a `Signal (Signal a)` into a mere `Signal a`, but it cannot exist in general.

3 Detecting Hover Information

A cursory inspection of `Graphics.Input` leads to the conclusion that menus cannot be implemented as we had hoped. The remarkable solution generalizes from detecting hover information to other GUI functions in the library.

3.1 A Naïve Menu

Elm's `Graphics.Input` library provides a function to obtain the hover information from an element.

```
hoverable :  Element -> (Element, Signal Bool)
```

The returned element is visually identical to the argument, but now detects hover information. The signal of booleans reflects the hover status of the this element, not the original. This function works well when the element is pure (not a signal). For example, if the top-level menu items are known statically and never change, they can be rendered as pure elements with this function. We can also create a function, perhaps of type `Bool -> Element`, which we can lift onto the hover data to display the submenu only on mouseover. The result is a value of type `Signal Element`. If we try to lift `hoverable` to accept this value, we get

```
lift hoverable :  Signal Element -> Signal (Element, Signal Bool)
```

Although it may not appear to be so at first, the result type is a signal of signals. It is possible to transform the result into `Signal Signal (Element, Bool)`. A more mentally convenient type, though impossible to obtain,[2] is `(Signal Element, Signal Signal Bool)`. There does not exist a general join function to operate on the `Signal Signal Bool`.

This is rather unfortunate, as we need to detect hover information on a dynamic element for a number of reasons. First, after a menu is extended by hovering over its parent, it must remain so as long as the mouse hovers over the menu itself. Without this information, the menu will disappear as soon as we try to mouse over it. Secondly, if we wish the indicate the currently hovered-over item to the user by visually highlighting it, the element must be dynamic. Thirdly, we do not wish to require that menu labels be string literals in the source code, which would be necessary if we could not use signals of elements. Rather, they may be retrieved or generated in some other manner, or they may change as the user switches among applications. Finally, in order to determine when a user has clicked on an element so that the system may respond, we must know which one is being hovered over.

[2] We explain why in Section 4.1.

3.2 Dynamic Hovering

However, it is possible to implement the following function:

```
hoverableJoin:  Signal Element -> (Signal Element, Signal Bool)
```

Comparing its type to the mentally convenient type of `lift hoverable`, we see it is identical except that the boolean value is a signal, not a signal of signals. That is, it has been joined.[3] It is implemented using the more general primitive `hoverables` (note the plural), of the following type:

```
hoverables : a -> { events : Signal a,
                    hoverable : (Bool -> a) -> Element -> Element }
```

The first parameter to `hoverables` denotes the initial value of the `events` signal (signals in Elm must always be defined). The record's `hoverable` function is applied to an element and attaches hover detection to it. Whenever the mouse hovers (or stops hovering) over the element, an event is fired on the `events` signal. The value of that event is determined by the `(Bool -> a)` parameter of the `hoverable` function. In many cases, `a` will be `Bool` and the function will be `id`. In general, the record's `hoverable` may be applied multiple times, with all elements it produces reporting their hover information on `events`. In this case, `a` can serve as an identifier, and is provided by the function argument.

The more widely used `hoverable` function is implemented using `hoverables`.

```
hoverable : Element -> (Element, Signal Bool)
hoverable elem =
    let pool = hoverables False
    in  (pool.hoverable id elem, pool.events)
```

Using only `hoverable`, we are restricted to one element per signal. By contrast, `hoverables` permits multiple elements on a single signal. The hover information of an element can be added to an existing signal, and can even be added dynamically, that is, when the element is a signal. As the value of a signal of elements changes, we attach the hover information to the each new element without having to create a new output signal. We do this by partially applying `pool.hoverable` to `id` purely, and then lifting it on to the argument.

```
hoverableJoin : Signal Element -> (Signal Element, Signal Bool)
hoverableJoin elem =
    let pool = hoverables False
    in (lift (pool.hoverable id) elem, pool.events)
```

[3] While the name `hoverableJoin` reinforces this fact for our paper, we do not recommend it for a user-facing API.

Just how novel is this small, but hugely significant change? The documentation for `hoverables` states that it allows users to "create and destroy elements dynamically and still detect hover information," but gives no further indicators on how to do so. Though the Elm website is full of examples, there are none for either `hoverable` or `hoverables`. Moreover, in the mailing list post that introduced these functions, Czaplicki said that "`hoverables` is very low level, but the idea is that you can build any kind of nicer abstraction on top of it." [Cza13a] We have done just that.

When a element changes, `hoverableJoin` attaches the new element to the same hovering signal. If we were to display both the old and the new elements on the screen, hovering over either would trigger the signal.

More dangerously, it becomes easy to create an infinite loop. Suppose an element shrinks on hover and returns to its original size without hover. Now suppose the cursor hovers on the element, which is then replaced by a smaller element, so the cursor is no longer hovering on the element. Then the original element is put back, but is now being hovered on because the mouse has not moved, so the cycle repeats. This condition often manifests itself as flickering between the two elements. As this description is implementation-independent, it is unavoidable in any language or system that allows hover targets to change size in response to hover events on that target.

3.3 Generalized Joins on `Graphics.Input`

It is not just hover detection that follows the paradigm of two primitives with singular and plural names. Most of Elm's wrappers around HTML GUI components do so as well, and have analogous types. They are therefore receptive of the same join technique. For text-labeled buttons, press events are represented by unit or a provided identifier:

```
button : String -> (Element, Signal ())
buttons : a -> { events : Signal a,
                 button : a -> String -> Element }
```

If we wish to create a button whose string label varies, the types allow the same exact same technique to be applied. (We have not tested whether this approach works, but the types allow it.) The same goes for text fields, which can be given text based on what else is known in the program, and checkboxes. As currently implemented, checkboxes have boolean state so dynamic behavior has limited use. However, we can imagine the ability to dynamically disable ("gray-out") a checkbox in response to other choices in a form.

Drop-down menus do not use the same paradigm as the rest; there is `dropDown` but no `dropDowns`. This is unfortunate not only because there seems to be no

way to supply a `Signal [String]`, but also because drop-downs are commonly used in groups. Multiple menus reporting on one signal would be very convenient. If the menu options were dynamic, it would be useful for the API to indicate when an event occurred because the user made a selection vs. when the menu updated and the previously selected string was no longer available.

4 Implementing Menus

Having established the need for `hoverableJoin` and implementing it, we now turn to stated task of creating menus.

4.1 Menu Representations

A menu specification, the abstract model of a menu, can be thought of as an instance of a tree.

```
data Tree a = Tree a [a]
```

The obvious type for a menu structure that may change is `Signal (Tree String)`. It is easy enough to convert a value of this type to one of type `Signal (Tree Element)`, but we cannot add hover detection to this tree. We are able to map from `Signal Element -> (Signal Element, Signal Bool)`, but not (in any useful way) from `Element -> (Element, Bool)`. We obviously cannot create a `Signal (Tree (Signal Element, Signal Bool))` because that would constitute a signal of signals. Therefore, we would need to create a function of type `Tree String -> Tree (Element, Bool)` and then `lift` it onto the `Signal (Tree String)`. However, since we cannot write the hovering function of type `Element -> (Element, Bool)`, this is impossible. Therefore, we cannot detect hoverable information on the `Signal (Tree String)`. Hover information requires a (distinct) signal at each item in the tree.

Instead of a signal of trees, we therefore use a tree of signals. Each individual menu element can contain dynamic information, but the menu structure must be static. We can, in practice, get around this restriction by creating a tree that is larger than necessary and filling the unused nodes with empty string, which our implementation handles appropriately.

Note that we can (and will) convert a tree of signals into a signal of trees, but it is impossible to go the other way. When converting many signals into one, the information on which signal changed is lost. The combinator `combine : [Signal a] -> Signal [a]` does this, and will be used below. It has no inverse because such a function would need to create information: which signal in the list updated? What if the list changes length? What if there is a repeat

event, where the list does not change, which signals should propagate repeats? Similar arguments show why the mentally convenient type of `lift hoverable`, which turned a signal of pairs into a pair of signals, cannot actually be obtained.

4.2 Menu Rendering

Therefore, a menu is specified by a value of type `Tree (Signal String)`. We can lift and map the combinator `plainText : String -> Element` to obtain from a menu specification a value of type `Tree (Signal Element)`. From there, we can use `hoverableJoin` to get the hover information of each element and create a value of type `Tree (Signal Element, Signal Bool)` to represent our menus. *Now* we convert (by structural recursion) our tree of signals to a signal of trees, a value of type `Signal (Tree (Element, Bool))`. We do the rest of our work with pure functions lifted to operate on the tree, for example, our rendering function `renderMenu : Tree (Element, Bool) -> Element`. Similarly to limiting the use of the IO monad in Haskell code, it is advantageous to limit the use of signals in Elm code.

Because `hoverableJoin` was applied throughout the entire tree, it contains every element in every menu, and booleans that indicate which ones to display. Rather than omitting elements associated with false, we replace these hidden submenus with a spacer. A spacer is simply a blank rectangle with a specified width and height (`spacer : Int -> Int -> Element`). For our purposes, the spacer has the same width as the element, but a height of 1.[4] We then switch between menu and the spacer based on hover information. The spacer helps to align the submenu with its parent element in a way a `Maybe Element` would not have done. By replacing hidden elements with spacers, we are able to use the `flow` primitive to position our elements instead of manually doing "absolute" positioning.

```
elemOrSpacer : (Element, Bool) -> Element
elemOrSpacer (elem, hover) =
    if hover then elem
    else spacer (widthOf elem) 1
```

There was another complication involving the hover information. A submenu is rendered to the screen if one of three conditions is met: the mouse is hovering upon its parent, the mouse is hovering upon it, or the mouse is hovering upon any of its descendants. While this definition allows recursive submenus to be correctly rendered to the screen, it creates a race condition. As the mouse moves from the parent to the its submenu, the child is often replaced with a spacer before it can detect that the mouse is hovering over it. The results in menus

[4] We were concerned a height of zero would cause browser rendering issues.

disappearing as the mouse changes which element it is hovering upon. To fix this problem, we created the following function:

```
delayFalse : Signal Bool -> Signal Bool
delayFalse b = lift2 (||) b (delay millisecond b)
```

Applying `delayFalse` makes a `Signal Bool` wait a millisecond to transition from true to false. Unlike a typical delay, false-to-true transitions still occur immediately. By applying this function to every hovering boolean in the menu structure, we give the mouse time to move from a menu element to its submenu before that submenu disappears.

A menu that does not detect mouse clicks is a fairly useless menu. There is no primitive in Elm that can simply attach click detection to an element. To get around this issue, we used a combination of `Mouse.clicks : Signal ()` (which fires an event whenever the mouse clicks anywhere on the screen) and the hoverable information to determine which element (if any) the mouse was hovering over. (This implementation uses Elm's infix function application operator `<|`, which is the same as the Haskell operator `$`).

```
clicksFromHover : Signal Bool -> Signal ()
clicksFromHover hover =
    lift (\_ -> ())
    <| keepIf id False
    <| sampleOn Mouse.clicks hover
```

For each menu element in the tree, we found its "path" (with type `[Int]`) which denotes the list indexes needed to traverse to that element. Combining this with the click information gave each element a signal of its path that fired whenever the user clicked on that element. We then combined all of those signals into one signal using the `merges` combinator (`merges : [Signal a] -> Signal a`). The user of the menu could then use this `Signal [Int]` to respond to mouse clicks. This is significantly more flexible than returning a `Signal String` because a path is guaranteed to be unique, whereas a menu string might have a duplicate.

Interestingly, we were able to implement menus without explicitly storing the state of the menu, i.e. without using `foldp` (or Elm's implementation of Arrowized FRP known as `Automaton`). Rather we use the current browser DOM state to remember state. Instead of storing the menu currently being displayed on the screen, we can derive it based on which element (if any) the mouse is currently hovering over. The elimination of state from our code allows it to be much cleaner.

5 Related Work: TodoFRP

We examine TodoFRP, [Cza13b] a simple Elm web app created by Czaplicki, to provide another example of the problem solved by `hoverableJoin`. TodoFRP is the current state-of-the-art in highly reactive Elm GUIs. It provides examples of different levels of reactivity, a familiar context for Elm veterans, and a few dirty tricks of its own.

TodoFRP presents the user with a text field asking, "what needs to be done?". Entered TODO items become DOM elements, which can be deleted with a "x" button, also a DOM element. The button is implemented using the `Graphics.Input` function

```
customButtons : a -> { events : Signal a,
                       customButton : a -> Element -> Element -> Element -> Element }
```

Notice the similarity with `hoverables`. Each call of `customButton` provides the identifier event when the button is clicked, and three (pure) elements to display: one normally, one on hover, and one on click. The result is an `Element`, not a `Signal Element`, that nevertheless changes among those three in response to the mouse. This is possible because the result element's dimensions are taken to be the maximum of the three inputs' dimensions. Even if the elements have different sizes, the resulting element and therefore the hover surface remains fixed in size. Although the same join technique can be applied, we find it likely that it would not work as intended.

In the case of TodoFRP, these three elements are different colors of the "x" and the same for each TODO entry. The polymorphic `a`s are unique identifiers (ascending integers) for each entry.

The TODO label elements are dynamic and do not detect hover information. The button elements that do detect hover information are known statically. With `hoverableJoin`, it becomes possible to dismiss a TODO by clicking on its text label.

6 Conclusion: To the Elm Community

It's true that we've used the `hoverables` function in a way that it was (probably) never meant to be used, and there are some caveats involved in doing so. Many small GUIs do not require dynamic, hover-detecting elements. However, most large mouse-based GUIs do, and creating them in Elm will necessarily encounter the obstacles we have described.

We have implemented all of this without language modifications. However, it is hoped that as the community becomes more familiar with functional GUIs, new libraries are added that incorporate some of our tricks, or even make them unnecessary. Elm's upcoming third-party library sharing system looks to be an excellent opportunity to refine abstractions and idioms for GUIs.

We present these techniques and analysis in the hopes they are of service to the Elm and FRP communities. There are a number of obvious paths that dead-end, and non-obvious paths that are fruitful. We hope this paper becomes useful in Elm's goal of making GUIs simpler to implement and more robust to use.

Our code is available at `https://github.com/ealter/Elm-GUIs`.

Acknowledgments

We would like to thank Evan Czaplicki and Stephen Chong for creating Elm, and the Elm community for growing it. We thank Norman Ramsey for his guidance through functional programming, and our paper reviewer, Caroline Marcks.

References

[CC13] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422, New York, NY, USA, June 2013. ACM Press.

[Cza12] Evan Czaplicki. Elm: Concurrent functional reactive programming for GUIs. Harvard University Senior Thesis, 2012.

[Cza13a] Evan Czaplicki. First reply to *Question on road-map.* `https://groups.google.com/forum/#!msg/elm-discuss/QgowLy5jdhA/CZQfjkbjMsEJ`, June 2013.

[Cza13b] Evan Czaplicki. TodoFRP. `https://github.com/evancz/TodoFRP`, 2013.