Explicitly Comprehensible Functional Reactive Programming

Steven Krouse
steveykrouse@gmail.com

ABSTRACT
Functional Reactive programs written in The Elm Architecture are difficult to comprehend without reading every line of code. A more modular architecture would allow programmers to understand a small piece without reading the entire application. This paper shows how higher-order and cyclic streams, as demonstrated via the Reflex library, can improve comprehensibility.

CCS CONCEPTS
• Software and its engineering → Functional languages; Data types and structures; Patterns; Frameworks;

KEYWORDS
functional reactive programming, elm, reflex

ACM Reference Format:

1 INTRODUCTION
In imperative languages with global mutable state, variables can be modified anywhere and in terms of any other global variables. In functional programming without mutable state, all terms explicitly list what they depend upon. This explicitness makes it easy to determine which parts of the code are dependent and independent of each other.

However, it is still possible to obfuscate the relationships between pieces of state in functional programming. One can simulate global mutable state by passing around an arbitrarily large compound state value as an extra parameter to each function. This is considered an anti-pattern because "ease of reasoning is lost (we still know that each function is dependent only upon its arguments, but one of them has become so large and contains irrelevant values that the benefit of this knowledge as an aid to understanding is almost nothing)." [7]

Yet in client-side Functional Reactive Programming (FRP), a variation on this anti-pattern has become the dominant architecture. Originally conceived for the Elm programming language [2], The Elm Architecture has since inspired ReactJS’s Redux, VueJS’s Vuex, CycleJS’s Onionify, among many other front-end state management libraries. This paper contrasts The Elm Architecture with a state management pattern in the Reflex library that maintains explicit relationships.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
REBLS’18, November 2018, Boston, Massachusetts USA © 2018 Copyright held by the owner/author(s).
The Reflex library was built for Haskell web development via ghcjs, steps the model forward in response to messages.

```haskell
main = mainWidget bodyElement
main :: IO
bodyElement :: MonadWidget t m
= do
  foldDyn :: (a -> b -> b) -> b -> Event a -> m (a -> b)
  leftmost :: [Event a] -> Event a
  ($): (a -> b) -> a -> b
  Show
  display :: el :: Text -> m a -> m a
  button :: Text -> m (Event ())

In the following examples we use Events and Dynamics.

3 Reflex

The Reflex library was built for Haskell web development via ghcjs, a Haskell to JavaScript compiler. Like in traditional FRP[4], Reflex has two main concepts: Events and Behaviors. Events are discrete occurrences in time, while Behaviors are continuously defined values for all points in time. Reflex also has Dynamic values, which have the properties of both: they are defined at all points in time and emit events at the discrete points in time when they change. In the following examples we use Events and Dynamics.

Listing 2: Reflex Counter

```haskell
button :: Text -> m (Event ())
el :: Text -> m a -> m a
display :: Show a => Dynamic a -> m ()
($): a -> Event b -> Event a
leftmost :: [Event a] -> Event a
foldDyn :: (a -> b -> b) -> b -> Event a -> m (Dynamic b)

bodyElement :: MonadWidget t m => m ()
bodyElement = do
  rec evIncr <- button "+1"
  el "div" $ display count
  evDecr <- button "-1"
  count <- foldDyn (+) 0 $ leftmost
  [ 1 <$ evIncr
    , -1 <$ evDecr
  ]
  return ()
main :: IO ()
main = mainWidget bodyElement
```

Reflex uses monadic do syntax to lay out the order of HTML elements [3]. In Listing 2, line 10 and 12, we create buttons with text "+1" and "-1", and bind their click event streams of type Event () to the names evIncr and evDecr, respectively. Notice that in Listing 2, line 11 count is used before it is defined. In Reflex, statements are arranged vertically in the order in which they appear in the HTML DOM tree. The recursive-do syntax [5] allow us to set up an event propagation network at the same time as we lay our HTML elements. To calculate the count from the button click events, we use:

- <$ (which is equivalent to Fran’s $ operator [4]) to to map each click event to either 1 or 1
- leftmost to merge (left-biased for simultaneous events) the two event streams into a single event stream, and
- foldDyn (+) to sum them up.

However this architecture does not properly generalize. For example, say we wanted to be able to set the value of the counter to a specific value, say another Dynamic, dynNum1, in response to a third button press. Instead of summing of Event Int, we can step forward the previous value of state with Event (Int -> Int). In Haskell, the dollar-sign operator, ($) :: (a -> b) -> a -> b, represents functional application, so here it applies each event function to the previous value of count.

```haskell
count <- foldDyn ($) 0 $ leftmost
  [ (+ 1) <$> evIncr
    , (+ (-1)) <$> evDecr
    , (_ <= dynNum1) <$> evSet
  ]
```

This pattern is similar to the Elm Architecture in that it is a reduction over events. However this is a local reduction solely for this piece of state. If there were other independent pieces of state in this application, they would have separate reductions.

Even in this small example we can see how count is defined more explicitly than in Elm. If we wish to understand how count behaves, we have a singular place to look for what it depends upon, evIncr and evDecr, and precisely how, mapping them to (+ 1) and (+ (-1)), respectively, and then merging and applying.

The price we pay for this explicitness is that events are abstracted from the single messages in Elm into streams of values. In Elm we write a global state reducer function that pattern matches on these event messages. In Reflex we use stream combinators to define the model and view as streams of each other.

3.1 Cyclic FRP

Reflex’s higher-order and cyclic streams are necessary to maintain explicitness in cyclical applications. "The DOM tree in an FRP application can change in response to changes in FRP behaviors and events. However, such new elements in the DOM tree may also produce new primitive event streams that represent, for example, button clicks on the event, which the FRP program needs to be able to react to. In other words, there is a cycle of dependencies…” [8].

Imagine a buttons that you can click to create more buttons, and when you click those buttons they also create buttons, and when you click those buttons... In Reflex, you’d accomplish this by first describing a list of buttons of Dynamic length. This would emit a
Explicitly Comprehensible Functional Reactive Programming

Dynamic list of Events, a higher-order stream. If you flattened the higher-order stream, counted all click Event occurrences, and added 1 for the original button, you’d get the desired length of the Dynamic list of buttons. Reflex would let you cyclically feed this Dynamic value back as the length of the list of buttons.

In Elm the cyclic nature of this application wouldn’t be apparent in the code. You’d have a have a list of buttons of length count, a value defined in the model. The value of count would increase in response to Increment messages, so you’d make sure that all your buttons emit Increment messages in response to click events. The issue with the Elm approach is that it’s too general: any view element could also issue Increment messages, and any other messages could affect the count variable.

4 TODOMVC COMPARISON

Let’s compare Elm ToDoMVC¹ with Reflex ToDoMVC². Say we wish to understand the behavior of the list of todo items in both implementations.

4.1 Elm TodoMVC

In Elm, any message can modify any state. For example, in update, the Add message triggers an update of three different pieces of state:

```
Add ->
( model
  | uid = model.uid + 1
  , field = ""
  , entries = if String.isEmpty model.field then
    model.entries else model.entries ++ [ newEntry model.field model.uid ]
)
```

Each piece of state can be modified in terms of any other piece of state. There’s no explicit isolation between independent states, so it’s difficult to reason about the behavior of the application. This hinders modularity: components become difficult to disentangle because the borders between them are blurred into a pooled state.

Technically the Elm Architecture does allow for modular composibility, but it is discouraged. ‘Don’t reach for this initially as a way to organise your application as ’components’. If you do this too soon you will end up with plenty of boilerplate.”³. Thus we can expect Elm applications to grow entangled and stay that way.

Like in a language with global mutable state, there is no single place to look to understand how the todo items list, here called entries, behaves. We must Ctl-F for all occurrences of entries =, as seen in Fig. 2, and then piece together in our head how the sum total of these effects come together to form an integrated behavior. In this way, the Elm Architecture’s reducer simulates the “primitive word-at-a-time style of programming inherited from the von Neumann computer ... instead of encouraging us to think in terms of the larger conceptual units of the task at hand.”¹

Figure 2: Elm TodoMVC entries modifications highlighted in reducer

Additionally, any view element can emit any number of messages. We know from our Ctl-F above that the Add, EditingEntry, UpdateEntry, Delete, DeleteComplete, Check, and CheckAll events can affect entries, so now we Ctl-F for each of those events to see which HTML elements emit those messages in response to which events as seen in Fig. 3.

If we’re looking to understand a single piece of state in Elm, we’re not much better off than with an entirely imperative framework: we still have to read more-or-less the whole application even if we wish only to comprehend only a small piece. Modularity is lost as

¹https://github.com/evancz/elm-todomvc/blob/master/Todo.elm
³https://www.elm-tutorial.org/en-v01/02-elm-arch/08-composing-3.html
surely as if we passed around an arbitrarily-large compound state value to all of our functions, which is in fact what we’ve done.

### 4.2 Reflex TodoMVC

For contrast, if we wish to understand the same piece of state in Reflex’s TodoMVC, there is a single explicit place to look, Listing 3, lines 16-19. This definition uses the more generalized pattern discussed for the counter application above. There we had Event (Int -> Int) and here we have Event (Map Int Task -> Map Int Task). Here tasks is defined as a merging of three event streams:

1. `fmap insertNew_ newTask` is where new tasks are added to the list.
2. `listModifyTasks` handles the vast majority of task mutations, including deletions, completions (and their reversal), and task text editing. This definition depends on `tasks` and `activeFilter`.
3. `fmap (const $ Map.filter $ not . taskCompleted) clearCompleted` filters out the currently-completed tasks all at once when the bottom-right “Clear Completed” button is clicked.

Explicitness allows us to see the shape of this application, how its pieces come together to make an integrated whole. We see where code is independent, such as `taskEntry`, and dependent, such as `tasks` on `activeFilter`. If we only cared about one specific piece of an application, we could rely on these explicit relationships to determine which parts of the code are relevant (dependent) and which we can safely ignore (independent).

## 5 IS THE CURE WORSE THAN THE DISEASE?

This paper argues for higher-order and cyclic streams for the purposes of comprehensibility. Yet the dense Reflex code isn’t easy to understand. The creator of Elm takes this stance, arguing that explicit dependencies lead to a “crazy” graph of dependencies.

The Elm Architecture has many benefits. For one, it simulates global mutable state, which is very familiar to most programmers. The one-message-at-a-time style does simplify the code writing process. It also reduces coupling between the view and the model, making it easier to make changes. Finally, Elm’s model variable is easily serialized, which allows for time-travel debugging, hot reloading, and easy-to-implement undo features.

While it is easier to write in the Elm Architecture, it is harder to navigate an large, unfamiliar codebase. Here Reflex shines by

---

*Listing 3: Reflex TodoMVC*

```haskell
definitions:

- initialTasks :: Map Int Task
  initialTasks = Map.empty

- insertNew_ :: Task -> Map Int Task -> Map Int Task

- todoMVC :: ( DomBuilder t m, DomBuilderSpace m ~ GhcjsDomSpace, MonadFix m, MonadHold t m ) => m ()
todoMVC = do
  el "div" $ do
    elAttr "section" ("class" =: "todoapp") $ do
      mainHeader
      tasks <- foldDyn ($) initialTasks $ mergeWith (.)
      [ fmap insertNew_ newTask, listModifyTasks, fmap (const $ Map.filter $ not . taskCompleted) clearCompleted ]
    newTask <- taskEntry
    listModifyTasks <- taskList activeFilter tasks
    (activeModifyTasks <- taskList activeFilter tasks)
    return ()
    infoFooter
```

4https://youtu.be/DfLvDFxcAIA?t=27m32s
showing how the pieces of state fit together. Reflex does this, in part, by exposing the cyclic nature of the cyclic interfaces, instead of obfuscating them behind a global variable modifiable from anywhere.

While Reflex enables understanding states via a single place in the code, Elm enables understanding messages via a single place in the code. If one wanted to understand the impact of a Reflex stream on downstream states, one would have to play the reverse of the Ctl-F game that we played in Elm. While this duality might seem equivalent, Reflex’s style is more comprehensible because understanding the behaviors of states is much more important than understanding the effects of messages.

To be fair, let’s not understate the difficulty of writing Reflex code. It is hell on earth, grappling with its unwieldy types, double fmaping over streams, and waiting for ghcjs to compile. Reflex is a reasonably sound computational model that is in much need of a usability upgrade.

6 RELATED WORK

Other Haskell FRP implementations, such as Threepenny.gui\(^5\) and Sodium\(^6\) have imperative solutions to the cyclic dependency problem. Slim is a Haskell DSL for FRP cyclic streams “using (safe) recursive definitions” \(^9\).

FlapJax was the first JavaScript-based FRP implementation \(^6\). xstate is modern JavaScript library featuring higher-order streams. It allows for cyclic definitions, defined imperatively \(^7\).

7 CONCLUSION

As the popularity of FRP frameworks continues, it’s increasingly important to have a data model architecture that prioritizes the comprehensibility and modularity of large programs. This paper does not present a direct solution to this problem, but instead attempts to sound the alarm that we’re currently satisfied with. The Elm Architecture, is not good enough. The Reflex library, with its higher-order and cyclic streams, points in the right direction, but we are still far from a complete solution to the problem of comprehensible user interface construction.

8 ACKNOWLEDGEMENTS

Jonathan Edwards provided invaluable encouragement, suggestions, and mentorship.

REFERENCES


\(^5\)https://wiki.haskell.org/Threepenny-gui
\(^6\)https://github.com/SodiumFRP
\(^7\)https://github.com/staltz/xstream#imitatetarget