Explicitly Comprehensible Functional Reactive Programming

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 with 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:

. 2018. Explicitly Comprehensible Functional Reactive Programming. In *Proceedings of SPLASH Boston (REBLS'18).* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Today software projects are so large that programmers cannot read every line. This is particularly relevant to open-source development where contributors have limited time. How are we to know which lines of read and which to ignore? In other words, we wish to know the relationships between various pieces: are they dependent or independent?

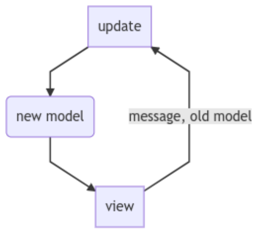
This question is difficult to answer in languages with mutable state, where variables can be modified anywhere they are in scope. In functional programming without mutable state, all terms explicitly list what they depend upon. This explicitness makes it easy to mechanistically determine which parts of the code are dependent and independent of each other, and guide us towards what we do and do not need to read for our present purposes.

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)." [3]

Yet in Functional Reactive Programming (FRP), a variation on this anti-pattern has become the dominant architecture. Originally conceived for the Elm programming language, 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.

2 THE ELM ARCHITECTURE

Elm is a pure functional language in the spirit of ML that compiles to JavaScript. Its streams are first-order and non-cyclic. User interfaces are inherently a cycle of alternating input and output, which is reflected in the Elm architecture. Let's explore the architecture with a simple counter application.



(a) The Elm Architecture

(b) Simple Counter App

Listing 1: Elm Counter

```
type alias Model =
  { count : Int }

initialModel : Model
initialModel =
  { count = 0 }

type Msg
  = Increment
  | Decrement

update : Msg -> Model -> Model
update msg model =
  case msg of
    Increment ->
      { model | count = model.count + 1 }
    Decrement ->
      { model | count = model.count - 1 }

view : Model -> Html Msg
view model =
  div []
    [ button
        [ onClick Increment ]
        [ text "+1" ]
    , div
```

```
      []                                                27
      [ text <| toString model.count ]                  28
   , button                                             29
      [ onClick Decrement ]                             30
      [ text "-1" ]                                     31
   ]                                                    32
                                                        33
main : Program Never Model Msg                          34
main =                                                  35
  Html.beginnerProgram                                  36
   { model = initialModel                               37
   , view = view                                        38
   , update = update                                    39
   }                                                    40
```

The core of the architecture is its a compound state value, `model`, **Listing 1, lines 1-6**. It represents the entirety of an applications state at any given time. Just like in imperative programming, the Elm Architecture is explicit only about the *initial* values of the model, here defined in **Listing 1, line 6**. The reducer, **Listing 1, lines 12-18**, steps the model forward in response to messages, simulating a global bag of mutable variables that change over time. Messages are generated from events in the view **Listing 1, lines 20-32**, such as the `Increment` and `Decrement` messages, both from `onClick` events.

3 REFLEX

The Reflex library was built for Haskell web development via ghcjs, a Haskell to JavaScript compiler. It features higher-order and cyclic streams, which means that streams can contain streams, and streams can reference streams that reference themselves. It is these features that are necessary to maintain explicitness in FRP.

Like in traditional FRP[2], 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 Events and Behaviors, in they are defined in all points in time as well as emitting events at the discrete points in time when they change. In the following examples we use Events and Dynamics.

Listing 2: Reflex Counter

```
button   :: Text -> m (Event ())                           1
el       :: Text -> m a -> m a                             2
display  :: Show a => Dynamic a -> m ()                    3
(<$)     :: a -> Event b -> Event a                        4
leftmost :: [Event a] -> Event a                           5
foldDyn  :: (a -> b -> b) -> b -> Event a -> m (           6
     Dynamic b)
                                                           7
bodyElement :: MonadWidget t m => m ()                     8
bodyElement = do                                           9
  rec evIncr <- button "+1"                               10
    el "div" $ display count                              11
    evDecr <- button "-1"                                 12
    count <- foldDyn (+) 0 $ leftmost                     13
      [  1 <$ evIncr                                      14
      , -1 <$ evDecr                                      15
      ]                                                   16
  return ()                                               17
                                                          18
main :: IO ()                                             19
main = mainWidget bodyElement                             20
```

Reflex uses **`do`** syntax to lay out the order of HTML elements. 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 `evIncr`, 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, not in the order they are "evaluated." There is no explicit evaluation order here. Instead this **`do`** `rec` syntax allow us to set up an event propagation network *at the same time* as we lay our our HTML elements. To calculate the `count` from the button click events, we use

- <$ to map each click event to either `1` or `-1`,
- `leftmost` (which is thusly named because in the case when events occur simultaneously – which is impossible in this example because only one button can be clicked at a time – it chooses only the event leftmost in the list) to merge the two event streams into a single event stream, and
- `foldDyn (+)` to sum them up.

However this architecture does not properly scale. 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 `($) :: (a -> b)-> a -> b` represents functional application, so here it applies each event function to the previous value of count.

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

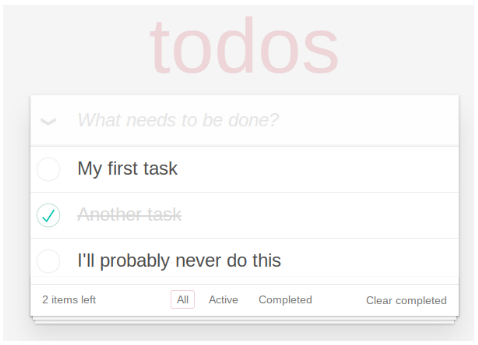
This pattern is similar to the Elm Architecture in that it is a reduction over events. However this is a local reduction solely for this piece of state. If there were other independent pieces of state in this application, they would have separate reductions.

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. We will see the benefits more clearly in a larger example.

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. The single global I/O cycle of Elm becomes a number of cyclic definitions between model and view streams in Reflex.

4 TODOMVC COMPARISON

TodoMVC has become a standard application to compare front-end frameworks. It runs 300 lines in both Elm and Reflex. We compare Elm ToDoMVC [1] with Reflex ToDoMVC [2]. 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 ]
  }
```

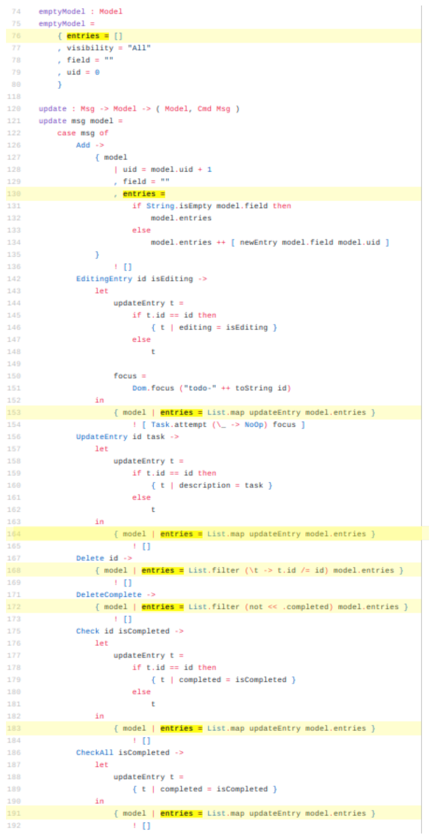
Thus there is no single place to look to understand how the todo items list, here called `entries`, behaves. We must Ctl-F for all occurrences of `entries =` as seen in **Fig. 2**.

In other words, we cannot gain a complete understanding of any piece of state by looking in one place. We have to look through *all* messages to see if it affects the state in question, 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]

There's also a subtler way the Elm Architecture undermines explicitness: each piece of state can be modified in terms of *any other piece of state*. There's no explicit isolation between independent states.

Next, 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**.

[1]https://github.com/evancz/elm-todomvc/blob/master/Todo.elm
[2]https://github.com/reflex-frp/reflex-todomvc/blob/develop/src/Reflex/TodoMVC.hs



```
emptyModel : Model
emptyModel =
    { entries = []
    , visibility = "All"
    , field = ""
    , uid = 0
    }

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        Add ->
            { model
                | uid = model.uid + 1
                , field = ""
                , entries =
                    if String.isEmpty model.field then
                        model.entries
                    else
                        model.entries ++ [ newEntry model.field model.uid ]
            }
                ! []
        EditingEntry id isEditing ->
            let
                updateEntry t =
                    if t.id == id then
                        { t | editing = isEditing }
                    else
                        t

                focus =
                    Dom.focus ("todo-" ++ toString id)
            in
                { model | entries = List.map updateEntry model.entries }
                    ! [ Task.attempt (\_ -> NoOp) focus ]
        UpdateEntry id task ->
            let
                updateEntry t =
                    if t.id == id then
                        { t | description = task }
                    else
                        t
            in
                { model | entries = List.map updateEntry model.entries }
                    ! []
        Delete id ->
            { model | entries = List.filter (\t -> t.id /= id) model.entries }
                ! []
        DeleteComplete ->
            { model | entries = List.filter (not << .completed) model.entries }
                ! []
        Check id isCompleted ->
            let
                updateEntry t =
                    if t.id == id then
                        { t | completed = isCompleted }
                    else
                        t
            in
                { model | entries = List.map updateEntry model.entries }
                    ! []
        CheckAll isCompleted ->
            let
                updateEntry t =
                    { t | completed = isCompleted }
            in
                { model | entries = List.map updateEntry model.entries }
                    ! []
```

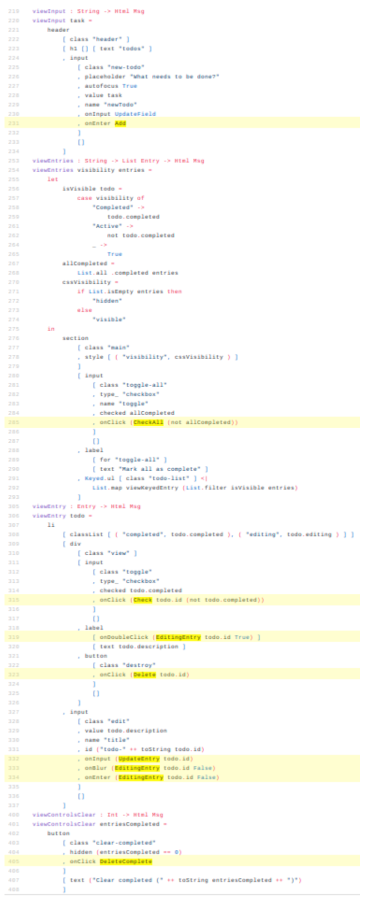
Figure 2: Elm TodoMVC entries modifications highlighted in reducer

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. Explicitness is lost as surely as if we passed around a compound state value to all of our functions, which is in fact what we've done.

4.2 Reflex TodoMVC

By 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 scalable pattern we 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 a merging of three events:

(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 "Clear Completed" button is clicked.



```
viewInput : String -> Html Msg
viewInput task =
    header
        [ class "header" ]
        [ h1 [] [ text "todos" ]
        , input
            [ class "new-todo"
            , placeholder "What needs to be done?"
            , autofocus True
            , value task
            , name "newTodo"
            , onInput UpdateField
            , onEnter Add
            ]
            []
        ]
viewEntries : String -> List Entry -> Html Msg
viewEntries visibility entries =
    let
        isVisible todo =
            case visibility of
                "Completed" ->
                    todo.completed
                "Active" ->
                    not todo.completed
                _ ->
                    True
        allCompleted =
            List.all .completed entries
        cssVisibility =
            if List.isEmpty entries then
                "hidden"
            else
                "visible"
    in
        section
            [ class "main"
            , style [ ( "visibility", cssVisibility ) ]
            ]
            [ input
                [ class "toggle-all"
                , type_ "checkbox"
                , name "toggle"
                , checked allCompleted
                , onClick (CheckAll (not allCompleted))
                ]
                []
            , label
                [ for "toggle-all" ]
                [ text "Mark all as complete" ]
            , Keyed.ul [ class "todo-list" ] <|
                List.map viewKeyedEntry (List.filter isVisible entries)
            ]
viewEntry : Entry -> Html Msg
viewEntry todo =
    li
        [ classList [ ( "completed", todo.completed ), ( "editing", todo.editing ) ] ]
        [ div
            [ class "view" ]
            [ input
                [ class "toggle"
                , type_ "checkbox"
                , checked todo.completed
                , onClick (Check todo.id (not todo.completed))
                ]
                []
            , label
                [ onDoubleClick (EditingEntry todo.id True) ]
                [ text todo.description ]
            , button
                [ class "destroy"
                , onClick (Delete todo.id)
                ]
                []
            ]
        , input
            [ class "edit"
            , value todo.description
            , name "title"
            , id ("todo-" ++ toString todo.id)
            , onInput (UpdateEntry todo.id)
            , onBlur (EditingEntry todo.id False)
            , onEnter (EditingEntry todo.id False)
            ]
            []
        ]
viewControlsClear : Int -> Html Msg
viewControlsClear entriesCompleted =
    button
        [ class "clear-completed"
        , hidden (entriesCompleted == 0)
        , onClick DeleteComplete
        ]
        [ text ("Clear completed (" ++ toString entriesCompleted ++ ")")
        ]
```

Figure 3: Elm TodoMVC relevant actions highlighted in view

Listing 3: Elm TodoMVC

```
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
      rec tasks <- foldDyn ($) initialTasks $
           mergeWith (.)
            [ fmap insertNew_ newTask
            , listModifyTasks
            , fmap (const $ Map.filter $ not .
                 taskCompleted) clearCompleted
            ]
          newTask <- taskEntry
          listModifyTasks <- taskList activeFilter
               tasks
          (activeFilter, clearCompleted) <- controls
               tasks
      return ()
    infoFooter
```

Explicitness allows us to see the shape of this application and 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 piece of behavior and its associated interface, we could rely on this explicit independence to safely ignore the others. In **Fig. 7**, we can see the explicit relationships between Reflex TodoMVC's Dynamics and HTML elements that are obfuscated in Elm TodoMVC **Fig. 6**.

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 and diagram don't seem easy to understand. The creator of Elm takes this stance, arguing that explicit dependencies lead to a "crazy" graph of dependencies[3].

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. 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 in a small application, we are concerned with the more common case of navigating a large application one does not know well. Here the Reflex diagram in **Fig. 7** shines by showing how the application fits together, and what we need to read.

The Elm Architecture reduces coupling, but there can be too little coupling. The amount of coupling in code should reflect the essential coupling in the underlying idea. If the nature of an interface is cyclic, the code shouldn't obfuscate that fact, but expose it clearly. More importantly it should make explicit the independence of independent entities.

To be fair, let's not understate the difficulty of writing Reflex code. It is hell on earth, grappling with its unwieldy types, double `fmap`ing over streams, and waiting for `ghcjs` to compile. Reflex is a reasonably sound computational model that is in much need of a interface upgrade, discussed further in **Section 7**.

6 RELATED WORK

The field of program slicing[4] takes a different approach to the problem of code dependencies. It leaves the underlying imperative programming model the same, and uses both static analysis and execution traces to determine which sections of code are relevant.

[3]https://youtu.be/DfLvDFxcAIA?t=27m32s

It may be possible to keep the simpler semantics of the Elm Architecture, and generate interdependencies via static analysis. In Elm, accessing and modifying records can only be done with static keys, so we can determine exactly which messages modify which pieces of state, which view elements depend on which pieces of state, and which view elements emit which messages. Potentially we'd be able to generate a graph as pleasing as **Fig. 7** with Elm. However this static analysis would be much less powerful in Elm-Architecture-inspired JavaScript frameworks (Redux or VueX), because in JavaScript one can access objects with dynamic string keys, so it'd be difficult to be sure what's accessed and modified where.

7 FUTURE WORK

7.1 Visualizations

While the Reflex library's semantics are ideal for modular comprehensibility, the library itself is difficult to use. For one, the syntax is difficult to parse. Given that our goal is comprehensibility, this is unacceptable. I am inspired by the visual stream combinators of RxMarbles[4] (**Fig. 4**).

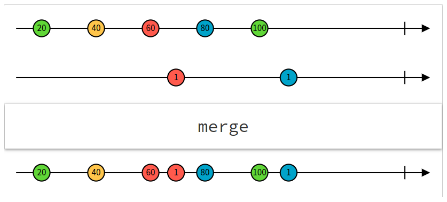
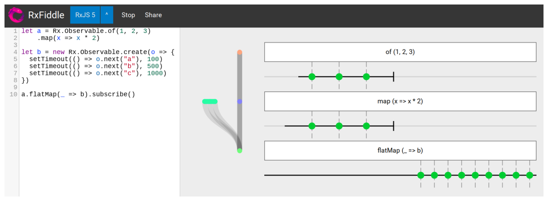


Figure 4: RxMarbles

RxViz[5] and RxFiddle[6] parse streams from code and visualize them dynamically. RxViz does a particularly good job of visualizing the evolution of higher-order streams in realtime. However, RxViz only shows the final output stream, not how the various input streams combine. RxFiddle makes an effort to show the high-level view of stream dependencies (in **Fig. 5** between the code and marble diagrams).



```
let a = Rx.Observable.of(1, 2, 3)
    .map(x => x * 2)

let b = new Rx.Observable.create(o => {
  setTimeout(() => o.next("a"), 100)
  setTimeout(() => o.next("b"), 500)
  setTimeout(() => o.next("c"), 1000)
})

a.flatMap(_ => b).subscribe()
```

Figure 5: RxFiddle

One challenge in particular will be visualizing cyclic dependencies. This may require learning from M. C. Escher.

Eventually, it would be ideal if one could edit the streams from the visual representation directly.

[4]http://rxmarbles.com/
[5]https://rxviz.com/
[6]https://rxfiddle.net

7.2 JavaScript-based

It order to be a true alternative to the Elm Architecture in the broader JavaScript community, a competing model must be performant in JavaScript. In theory, it is possible to create a cyclic event propagation network (to copy Reflex's implementation) in JavaScript, but JavaScript's semantics are quite different from pure functional programming so it may not have a natural feel. The CycleJS library[7] allows for higher-order, and even semi-cyclic streams[8], so maybe it wouldn't be difficult.

8 CONCLUSION

As the popularity of FRP frameworks continues, it's increasingly important to have a data model architecture that prioritizes the comprehensibility of large programs. This paper does not present a direct solution to this problem, but instead attempts to sound the alarm that what 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.

REFERENCES

[1] John Backus. 1978. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM* 21, 8 (Aug. 1978), 613–641. https://doi.org/10.1145/359576.359579
[2] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 263–273.
[3] Ben Moseley and Peter Marks. 2006. Out of the tar pit. *Software Practice Advancement (SPA)* 2006 (2006).
[4] Wikipedia contributors. 2018. Program slicing — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Program_slicing&oldid=849621023 [Online; accessed 16-August-2018].

[7]https://cycle.js.org/
[8]https://github.com/staltz/xstream#-imitatetarget

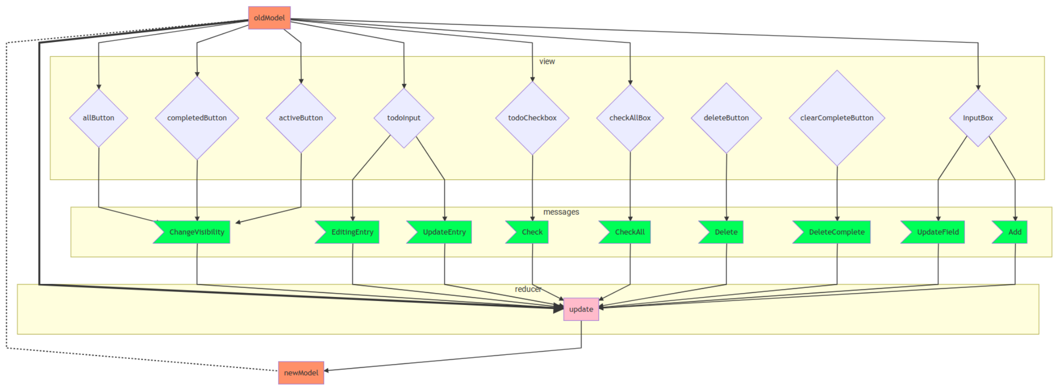


Figure 6: Elm TodomMVC Diagram. HTML elements are rotated blue squares. Messages are green and rectangular. All messages point to the pink reducer, which point to the newModel, which is linked to the oldModel at the top via a dotted line.

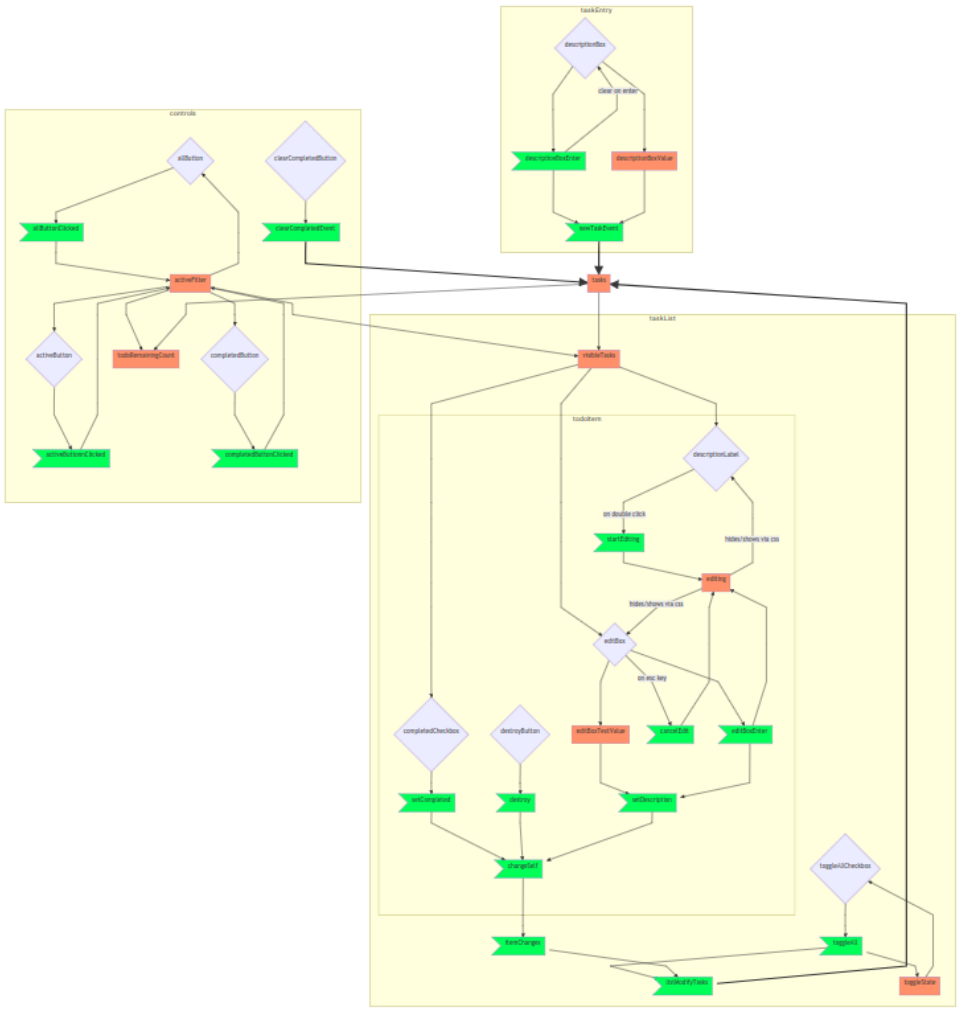


Figure 7: Reflex TodomMVC Diagram. The top-level tan categories represent widget definitions. HTML elements are rotated blue squares, Events are green rectangles, missing a triangle, and Dynamics are orange rectangles. The three arrows for the three dependencies of tasks are bold.