A Structural Analysis of Semagrams

by Owen Lynch

1 Introduction

The purpose of this document is to develop doctrine and vocabulary for continued development of Semagrams. As Semagrams becomes more integrated with other components of a holistic system, it is necessary to formalize how the other parts of the system will interact with Semagrams. Additionally, development of doctrinal discipline around different parts of the system is necessarily in order to develop test suites for those parts in isolation.

To this end, we present an analysis in three parts: state, communication, and purity.

2 State

In this section, we analyse where state lives in Semagrams. All of the rest of Semagrams is controlled, in one way or another, by the state, so it is important to first analyze where the state lives so that we can manage it.

State in Semagrams can be organized into three categories.

1. Model state. Semagrams operates as an editor of some model, typically an `ACSet`, and this state must live somewhere. Currently, this lives in a Laminar `Var`.

2. Control flow state. The control flow of complex editing operations acts as an implicit "state machine" that can be suspended and resumed via the `IO` monad from `cats-effect`.

3. Application state. There is a certain amount of book-keeping that must be done in order to make sense of the actions of the user. For instance, we keep track of:

   - The current hovered entity
   - The state of a drag action
   - The current mouse position

   The main location for application state is inside `Controller`s, but it also shows up in other places. The idea behind `EditorState` is to collect all of the application state into one structure.

All three of these categories of state might plausibly be exposed to a larger context that Semagrams is embedded in. The model state is most straightforwardly exposed; we can simply share access to the relevant `Var`. The control flow state is less straightforward. We might interact with it externally by passing events to it, and also it might interact with external things by simply running code. Finally, the application state might be queried externally.

3 Communication

In this section, we analyze the communcation between different parts of Semagrams, between Semagrams and the browser, and ultimately between the user.

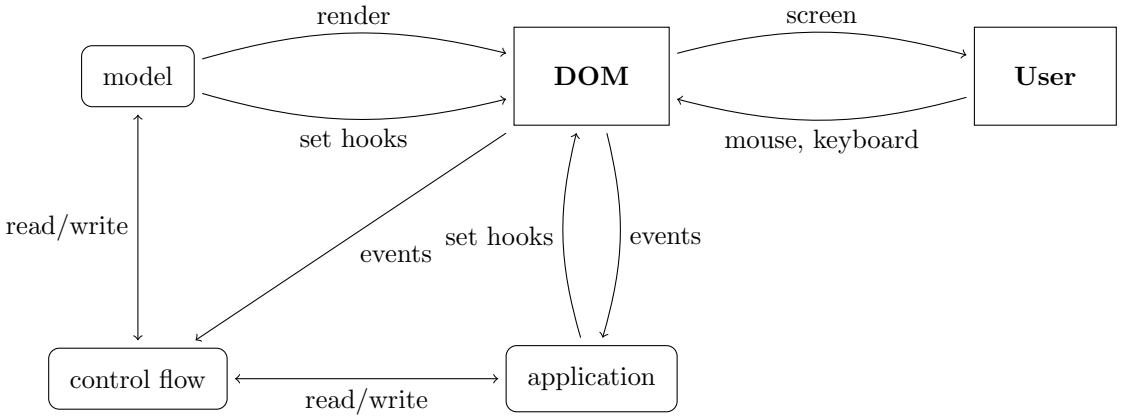

Figure 1. Communication structure of Semagrams

The "throne" of Semagrams is in the control flow, as encapsulated by the `IO` monad. This has read/write access to both application state and model state, which are contained in `Var`s. It also processes events which come from the DOM through a laminar `EventBus` that feeds into a cats-effect `Queue`. This is where any interesting custom logic runs.

The model, unlike the control flow, is "dumb" in that it is simply a data structure. The interesting part of the model is how it is rendered to the DOM, and how it sets up hooks on the DOM to fire events. The specific data structure used, and how the data structure is rendered is custom to the specific Semagrams instance.

Finally, the application state, like the model, is simply a data structure, or rather a collection of data structures. Unlike the model, this is not custom to a specific Semagrams instance, the purpose is merely to enhance the native capabilities of the DOM to book-keep some things we care about.

4 Purity

In this section, we analyze which components of Semagrams can be productively isolated from the browser, and thus tested in continuous integration.

The easiest part of Semagrams to test, and indeed the only part that we currently test, is the data model, i.e. acsets. Testing this simply involves running various methods on an acset (which is a persistent data structure, so these return new acsets) and then checking to see if the desired result is produced.

What is slightly harder to test, but should be more possible, is the control flow. This is because we can synthesize events at a higher level than the DOM, so we can simulate clicks and keyboard actions, etc. in theory without having to actually run inside a browser. Developing support for this is an important step towards getting a more reliable Semagrams.

The most impure thing is the actual browser rendering. It is not really possible to test for, say, rendering glitches or misfiring browser events in continuous integration, so we need to have manual testing processes.