Describing *computational effects*, like non-determinism or optional values, allows you to reason about what a program does. A popular method of describing computations in a functional language is through the use of *Monads*. Monads are just data types that have an operation that embeds a value within it-usually named *return* or *unit*-and an operation to chain computations together-generally named *bind* or *chain*. Attempting to compose two monads together at the type level (like the Maybe monad and the List monad) does not necessarily give a new monad. However, we can still formalize the composition of certain monads by implementing a *monad transformer*. A monad transformer provides an operation-usually named *lift*-that gives the implementation for embedding a computation from one monad to the other. The monad transformer construction allows us to describe the composition of different computational effects. A relatively new method of describing computational effects requires a sufficiently complex type system that can describe the introduction, composition, and removal of effects has been termed *algebraic effects*. This formalism was built from the ground up as a way of generically describing the composition of effects. Though, since it is not intended to encode all computational effects it is necessarily weaker than monads, so some things cannot be represented entirely by algebraic effects. Monad transformers create a linear stack based composition, while algebraic effects create a flatter graph based composition.

If we already have monads-and monad transformers for composing them-you might be inclined to ask what the motivation is behind creating a weaker abstraction like algebraic effects. There are two important distinctions: creating a new composition from smaller pieces, and accessing the data type that describes the effect. Let’s say that we have a computation that has the following effects: Input/Output (I/O), State, and Randomness (as shown in figure 1). With algebraic effects, each effect is at the same level as the others. This flatness means that if you wanted to introduce non-determinism to the effects we already have, it does not matter where the non-determinism is introduced. The computation started out as being able to perform I/O, read/write to state or generate random values. For example, we can perform I/O with randomly generated state values, or we can perform I/O with a state of randomly generated values. Algebraic effects allow us to perform either computation with the same type signature. After the introduction of non-determinism, the computation can now perform I/O, read/write to state, generate random values, and do all of this non-deterministically. This means we can now perform I/O with a non-deterministic state of random values, for instance.

With monad transformers, however, the semantics of the effects change depending on where the non-determinism is introduced. If our monad transformer stack is as shown in figure 1, then we can perform I/O with a state of random values, and that is the only computational effect we can perform. On the one hand, if we introduce the non-determinism at the top of the monad transformer stack, then we have a computation that can perform I/O with a state of random non-deterministic values. On the other hand, if we introduce non-determinism between the randomness and the state, then we have a computation that can perform I/O with a state of non-deterministic random values. This difference may seem trivial, but it entirely changes the semantics of the computation.

Constructing the effects is not the only point where algebraic effects and monad transformers differ. Accessing the data type that performs the effects is also different between the two. Let’s say that you wanted to get the current state at some point during a computation (still using the description provided in figure 1). Since algebraic effects keep all of the effects at the same level, when you need to get the state, you can do this at the top level. If the effects that are being described change, you do not have to go through and change any of the state calls (assuming your computation still describes the same state). For monad transformers, when you want to get the state, you have to know the layout of the monad transformer stack. State is below randomness in the first part of figure 1, so you have to lift the call to get the state up to the top one level. When non-determinism is introduced, we have to go back and modify all of the state calls we had previously. Meaning that we now need to lift the call to get the state up two levels.

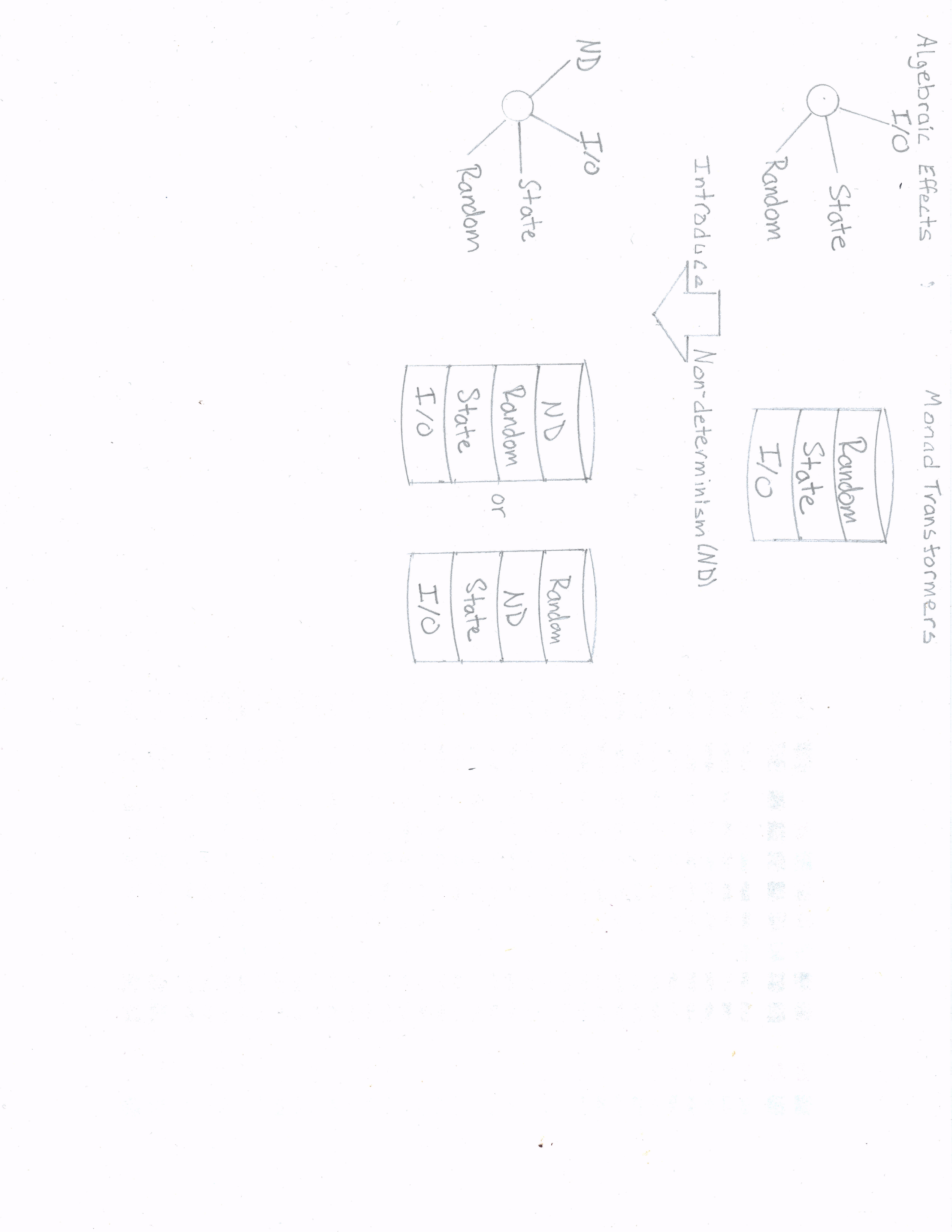


Figure 1

Hello, Hardy.

Please take a few minutes to read the comments I’ve added to your paper. In the margins above, you’ll find comments that apply to specific areas of the text, while in the comments below I try to summarize the most important issues that came up for me as I read and evaluated your work.

* As far as the transitions between sentences and/or between paragraphs goes, I encourage you to continue incorporating more transitions into your writing not only to make it flow more smoothly but to help explain to your reader what you see as the logical connections between one thought and the next.
* Watch out for the invisible elbow passives, which are still popping up fairly often. They are the hardest to recognize, but sometimes easy to fix. Add the agents (heroes, doers) to the beginnings of sentences.
* I noticed, too, that to-be verbs are also quite prevalent in your writing and might be replaced with more action-oriented verbs fairly easily.
* The pace of many of the explanations was too fast. I felt that I could understand your explanations, but they were delivered at a very high pace which required re-reading several times. Slowing down the delivery of new information with more elaboration, more transitions, pauses for examples and metaphors and so on, would probably help.
* In order to better convey to your target audience some of the more complex ideas here, you should really consider including a few metaphors or analogies along the way, either quick simple similes or overarching metaphors.
* As for the integration of text and visual, I could not easily find a connection between the two at times, and I felt that I was on my own as a lay reader when I looked over a visual. Often, therefore, I couldn't figure out what I was looking at.
* In fact, you should try to simplify visual complexity. Try not to overwhelm your everyday reader with a visual that’s too difficult to navigate or figure out. Perhaps you should consider conveying less information per visual, cutting back on how much you’re trying to pack into the visual, and making sure all the parts of the visual are defined and explained in the text.
* Consider splitting up complicated visuals like Figure 1. Perhaps you can divide a visual according to its steps. Alternatively, one visual might serve as a starting point, giving us an overview or general idea of the concept, but now you should zoom in with a narrower focus on a particular aspect of the overview with a second or third visual. Being narrower in scope will allow you to go into more depth and teach more difficult ideas.

As always, if you need elaboration on my comments, please get in touch (during class, office hours, or via email) and I'm more than happy to explain what I mean so you can continue to improve the effectiveness of your writing.

**Grade: B-**

Don