The thrill of algebraic effects, the agony of monad transformers

Reasoning about programs is hard, and our current techniques are not very composable. In most programming languages any *computational effect* – like input/output (I/O), non-determinism or optional values – can happen at any time. Explicitly enumerating which computational effects happen when in a program allows you to more easily reason about what a program actually does. We can use *monads* (defined later) to describe any computational effects. The thrill of. But monads do not compose without losing expressive power, so recently work has been done to develop the theory of *algebraic effects* (also defined later) which compose while keeping their expressive power.

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*, *pure*, or *unit –* and an operation to chain computations together – generally named *bind*, *flatMap*, or *chain*. Some well known monads are: the optional value monad (called *Maybe* or *Option*) depicting a computation that has exactly 0 or 1 value; the non-determinism monad (can be represented as *List*) for computations which have 0 or more values; and the *State* monad, which describes computations with some globally accessible state that can be read from or written to. If we want to specify a computation with more than one monad, we may attempt to compose them together. Unfortunately, the composition of two monads (like the Maybe monad and the List monad) directly does not necessarily give a new monad. The composition may become some weaker structure that cannot describe all computational effects. However, we can still formalize the composition of certain monads by implementing a *monad transformer*. A monad transformer requires an additional operation – usually named *lift* or *prod –* 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*. In fact, algebraic effects are so recent that very few programming languages have an implementation available, and most languages lack the expressive power to properly encode the concept. This formalization arose specifically as a way of generically describing the composition of effects. So by construction, composing two algebraic effects is always still an algebraic effect.

**Monad transformers: Computational effects in disguise**

Since monad transformers are just compositions of monads, and monads can describe any computational effect, monad transformers can be used to describe any computational effect as well. We could use monads to fully describe every computational effect that any program had. This level of granularity is not done in practice, for reasons to be shown later. The story is different for algebraic effects. It is not immediately obvious which computational effects can be described by algebraic effects. Continuations, for example, cannot be encoded by algebraic effects. It can be shown that the continuation monad can actually describe all other monads (and thus all computational effects), but since we cannot represent continuations with algebraic effects, they must necessarily be a weaker construct than monads and their transformer compositions.

**With our effects combined... we are algebraic effects!**

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. The distinctions arise from the way each composes. Monad transformers create a linear stack based composition, while algebraic effects create a flatter graph based composition.

Let’s say that we have a computation that has the following effects: I/O, State, and Randomness. 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. We do not have the option to perform I/O with a random state of values or use random state to perform I/O. If some different computational effect is required, then the entire monad transformer stack must be changed to reflect this. The unwieldiness of monad transformers also shows up when we want an additional computational effect.

If we wanted to introduce non-determinism into our program, we have some options as to where that effect will take place. On the one hand, if we put 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. These options are only two of many, but with each they lock down the computational effects that the program can now perform.

With algebraic effects, each effect is at the same level as the others as shown in Figure 1. This flatness means that if you wanted to introduce non-determinism to the effects we already have, it does not matter where you introduce the non-determinism. 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, we can perform I/O with a state of randomly generated values, or we can just generate random values. Algebraic effects allow us to perform any of these computations (and many more) without changing the type signature.

If we again wanted to introduce non-determinism we end up having an easier time with it. After we add this new effect, the computation can now perform I/O, read/write to state, generate random values, and do all of this non-deterministically. This method of composition means we can now perform I/O with a non-deterministic state of random values, non-deterministically perform I/O, or simply generate random values still. It does not matter which additional effects are added, the previous behavior is still valid and new combinations are just as valid.

**Like sands through the hourglass, so are the algebraic effects of our program**

Since monad transformers compose in a linear stack fashion, each effect we want to describe must be inserted in a specific order. If the stack is created as the first part of Figure 1, then we need to *lift* an I/O operation twice (once to bring it up to state and again to bring it up to Random) when we want to use it. And when we introduce a new effect, we have to go through our program and update all of the I/O *lift*s to be called three times instead of two. This same transformation has to be done for each effect we have in our stack depending on where the new effect is placed. Forcing changes like this dictates designs where different computational effects are lumped together if they share some similarities. I/O is a perfect example of this lumpiness. It is much easier to state that all things which interact with the real world happen in the I/O effect. But I/O can be much more granular than that. Reading from a database is not the same computational effect as writing to a database. Writing to a file handle is not the same computational effect as getting the current time. However, each of these effects is usually lumped into I/O in practice. So, when you see that a program is using an I/O effect, you do not necessarily know what the program is actually doing. It could be something as mundane as sending a file to the printer, or it could be as catastrophic as launching missles!

Algebraic effects also allow for a more fine grained approach to describing computations. As there is no order to how effects are accessed it makes no difference when you add a new effect. In practice, the flatness of effects means that there is less burden to divide kinds of effects up. We can more easily state that a program only reads from a database and writes to a file. We do not need to carry around implicit information about possible computations a program can have. We instead state exactly what a program can compute. We can use this when we compose smaller programs into larger ones.

If we have the option of choosing between two programs A and B, to use within program C (all using fine grained algebraic effects), and we see that A and B share the same effects, but B also introduces a missle launching effect, then we can make a better decision about which of A or B to use. Whereas if we were using the lumpier monad transformer style, then we would just assume A and B were equivalent programs and choose one to use arbitrarily. Our choice could end up with a more dramatic effect than we intended. The granularity of algebraic effects allows us to reason more effectively about the programs we create.

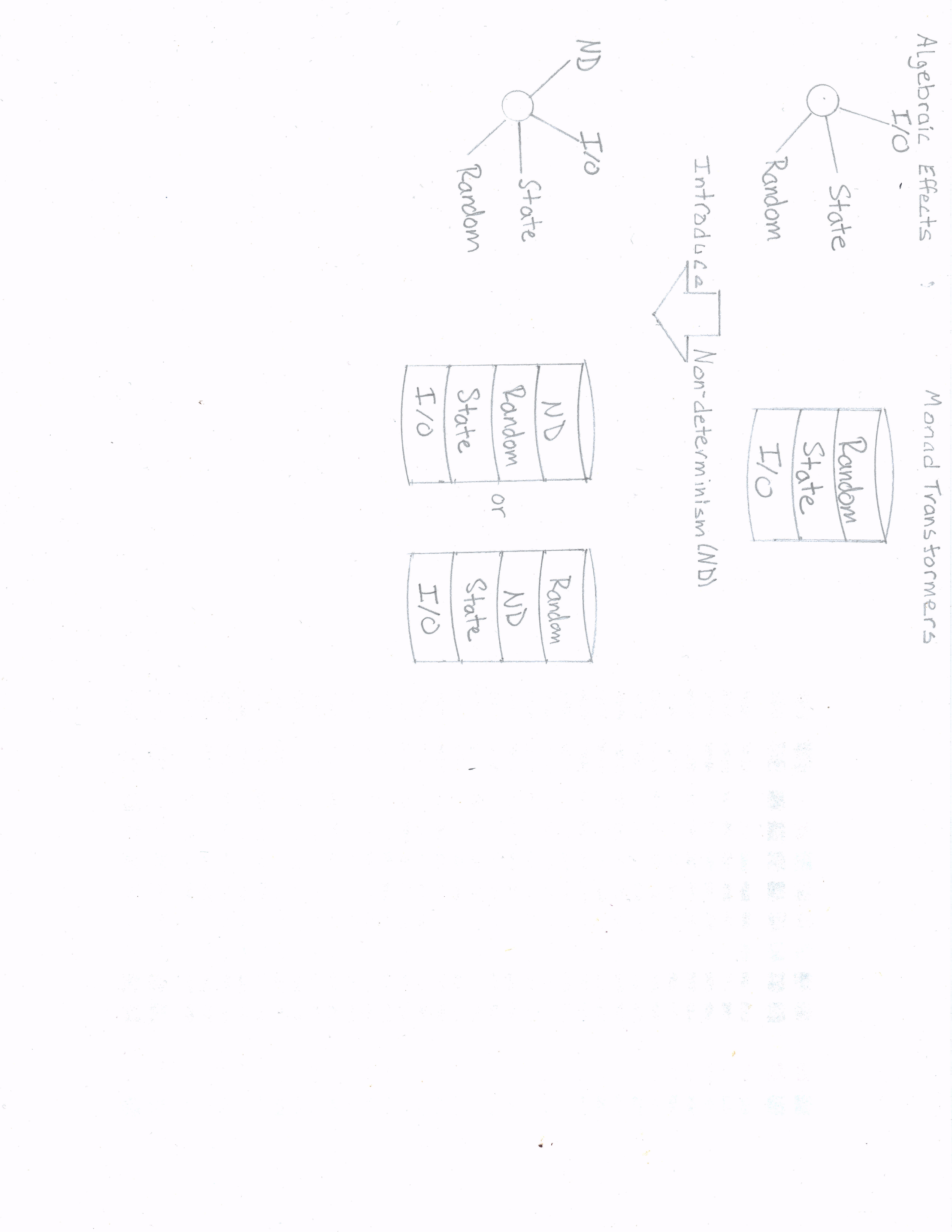


Figure 1