#### Imperial College London

#### September 5th, 2023

Modular Models Monoids with Operations

Zhixuan Yang and Nicolas Wu

ICFP 2023 @ Seattle

# **This Paper**

1/20

A new perspective on *algebraic effects*:

- 1. allows more *expressive forms of operations* 
  - catch, parallel composition, lock etc

- 1. This talk is about mathematical models of computational effects in programming languages.
- 2. The goal is to develop a new perspective on *algebraic effects*, such that more general forms of operations can be expressed, including exception catching, parallel composition of concurrent programs, acquiring a lock.

# **This Paper**

1/20

A new perspective on *algebraic effects*:

- 1. allows more expressive forms of operations
  - catch, parallel composition, lock etc
- 2. allows non-monadic effects
  - applicative functors, graded monads, etc

- 1. And the new perspective also encompasses effects that are not monads, but are applicative functors, graded monads, arrows, and so on.
- 2. Before I explain what all these things mean, I will spend the first half of the talk on a brief recap on the history of modelling effects, so hopefully all these things will make sense later if they don't make sense to you now.

2/20

Following Moggi (1991), a unary type theory  $(\mathbb{O}, \mathbb{H}, \mathbb{E})$  has

$$\frac{\tau \in \mathbb{O}}{\vdash \tau \ type} \qquad \frac{x : \tau \vdash x : \tau}{x : \tau \vdash x : \tau} \qquad \frac{x : \tau \vdash e : \tau_1 \qquad f \in \mathbb{H}(\tau_1, \tau_2)}{x : \tau \vdash f(e) : \tau_2}$$

 $\mathbb E$  is a set of judgemental equalities of terms.

- 1. 35 years ago, Eugenio Moggi pioneered using the concept of *monads* from category theory to give semantics to non-pure features of programming languages, which are usually called computational effects nowadays. This is our starting point today.
- 2. Let's first review what a monad is. Some of you may instinctively say, "a monad is a monoid in the monoidal category of endofunctors and composition". That's exactly right, but today I'm gonna use internal languages to present category theory as much as possible, as I figured most of the audience here are more familiar with type theory.
- 3. The internal language of categories is what may be called unary type theory. Moggi called it 'monadic multi-sorted algebraic theory', but I like the name unary type theory better. A category, sorry, a unary type theory has three things, a set of base types <sup>(D)</sup>, a set of primitive terms <sup>III</sup> and a set of equations <sup>E</sup>.
- 4. The type theory is extremely simple: the only type former is using a base type, and the only term formers are using variables and primitive terms.
- 5. Additionally a unary type theory is parameterised by a set of judgemental equalities of terms.
- 6. Some unary type theories may have some additional structures, such as product types or function types, but in general we only these three simple rules.

2/20

Following Moggi (1991), a unary type theory  $(\mathbb{O}, \mathbb{H}, \mathbb{E})$  has

$$\frac{\tau \in \mathbb{O}}{\vdash \tau \ type} \qquad \frac{x : \tau \vdash x : \tau}{x : \tau \vdash x : \tau} \qquad \frac{x : \tau \vdash e : \tau_1 \qquad f \in \mathbb{H}(\tau_1, \tau_2)}{x : \tau \vdash f(e) : \tau_2}$$

 $\mathbb{E}$  is a set of judgemental equalities of terms.

For example the category  $id \longrightarrow * \oiint f$  is presented by  $\mathbb{O} = \{*\}$   $\mathbb{H}(*, *) = \{f\}$   $\mathbb{E} = \{x : * \vdash f(f(x)) \equiv x\}$ 

- 1. For a small example, consider a small category with just one object \* and two morphisms, the identity morphism and a morphism called f, where the composition of f after f is the identity.
- 2. This little category is presented by a unary type theory with exactly one base type \*, one primitive term f, and the equation saying that f of f of x is judgementally equal to x.

A monad  $(T, ret, let \cdot in \cdot)$  is

 $\frac{\vdash \tau \ type}{\vdash T\tau \ type} \qquad \qquad \frac{x:\tau \vdash e:\tau'}{x:\tau \vdash ret \ e:T\tau'}$ 

$$\frac{x:\tau \vdash e_1: T\tau_1 \qquad y:\tau_1 \vdash e_2: T\tau_2}{x:\tau \vdash let \, y = e_1 \, in \, e_2: T\tau_2}$$

satisfying certain judgemental equalities.

3/20

- 1. In this setting, a monad is a type constructor T. The intuition is that for every type  $\tau$ , the type  $T\tau$  is the type of *computations* returning values of type  $\tau$ .
- 2. The monad T also comes with other two things: for every term e of type  $\tau'$ , a term ret e of type  $T\tau'$  that is supposed to be the pure computation that just returns e.
- 3. Additionally, for a computation  $e_1$  that computes  $\tau_1$  values, and another computation  $e_2$  in the context of a  $\tau_1$ -value, there's a sequential composition of them, which we write as the let-binding on the slide
- 4. These two term constructors are also required to satisfy some equational laws, which I will omit here.
- 5. The connection between the type theory here and the standard definition of monads in category theory is that, if we interpret this type theory in category, by interpreting types as objects, terms as morphisms, then the interpretation of *T*, *ret*, *let* is precisely a monad over that category.

Some effects and monads:

- Mutable state
- ► Exception
- Nondeterminism

 $T\tau \equiv S \Rightarrow (\tau \times S)$  $T\tau \equiv \tau + 1$  $T\tau \equiv List \tau$ 

4/20

- 1. Here are some well known examples: the effect of mutable state is modelled by the state monad  $S \Rightarrow (\tau \times S)$ , where S is the type of state. Of course, this monad only makes sense when the type theory has function types and product types.
- 2. Exceptions are modeled by the maybe monad. Nondeterminism is modelled by the list monad. On the slide, only the type constructor part is shown, but the returns and bindings are also a part of the monads.
- 3. The monads shown are the simplest ones, but there are more sophisticated ones that model more complex effects, such as higher-order store and parallel computation.

Some effects and monads:

- Mutable state
- ► Exception
- Nondeterminism

 $T\tau \equiv S \Rightarrow (\tau \times S)$  $T\tau \equiv \tau + 1$  $T\tau \equiv List \tau$ 

4/20

#### Question

Where do these monads come from?

1. One question is, where do these monads come from? We shouldn't expect ourselves to just spell out a monad whenever we have a new computational effect, right? They should logically come from somewhere.

# **Plotkin and Power's Answer**

Algebraic theories of effectful operations.

Theory of Nondeterminism

Operations: or : 2 and fail : 0Equations: or(x, or(y, z)) = or(or(x, y), z), etc 5/20

- 1. One answer, given by Plotkin and Power, is that these monads come from the algebraic theories of effectful operations.
- 2. An algebraic theory is just a bunch of operation symbols and a set of equational laws, for example you might know the theory of monoids, which consists of a binary operation and a nullary operation, and the equations are associativity and left and right identities.
- 3. Here, Plotkin and Power's perspective is that we should understand an effect by thinking about the algebraic theory of some primitive effectful operations.
- 4. For example, the computational effect of nondeterminism has a binary operation *or* for nondeterministic choice and *fail* for failure. The equations kind of depend on what we mean by nondeterminism, but we usually have at least associativity of *or* and left-and-right identity of fail.

# **Plotkin and Power's Answer**

Algebraic theories of effectful operations.

#### Theory of Nondeterminism

Operations: or : 2 and fail : 0Equations: or(x, or(y, z)) = or(or(x, y), z), etc

#### Theory of State

Operations: get : S and  $put_s : 1$  for each s : SEquations:  $put_{s_1}(put_{s_2}(x)) = put_{s_2}(x)$ , etc 5/20

- 1. Another standard example is the theory of state, where the operations are *get* and *put*. Get is an *S*-ary operation, where *S* is the type of state. Each of the *S*-arguments to *get* is a computation, the result of *get* is intuitively a computation that first reads the state, and the continues as one of the argument to *get* based on the state.
- 2. Correspondingly, there is a unary operation  $put_s$  for each s : S, which updates the state to s and continues as its only argument.
- 3. The equations of the theory of state characterise the interaction between *put* and *get*. I'm just gonna show one of them here, saying one put directly following another put overwrites the first one.

### **Effects Determine Monads**

6/20

An algebraic theory have *models*.

#### Example

A model of state (A, p, g) is a type A with

$$s: S, k: A \vdash p: A$$
  $k: S \Rightarrow A \vdash g: A$ 

where p and g implement put and get and satisfy the equations.

- 1. Algebraic theories are for talking about models. A model of an algebraic theory is a type equipped with the operations of the theory, satisfying the equations.
- 2. For example, a model of the theory of state is a type A together two terms p and g that implement *put* and *get* on the type A.

### **Effects Determine Monads**

7/20

*Free models* of a theory  $\Sigma$  determine a monad  $\Sigma^*$ .

#### Example

The *free model* of state over  $\tau$  is

$$(\mu X. \tau + S \times X + (S \Rightarrow X)) / \approx_{state}$$

which is isomorphic to  $S \Rightarrow (\tau \times S)$ .

- Moreover, when the unary type theory satisfies some conditions, every algebraic theory has *free models*, which can be concretely constructed by first inductively building syntactic terms of operations from the theory and variables from a type τ, and then quotienting these terms by the equations of the theory.
- 2. For example, the free model of the theory of state, over a type  $\tau$  is the inductive type here:  $\mu X$  is the binder for recursion,  $\tau$  is the base case,  $S \times$  is a syntactic put,  $S \Rightarrow$  is a syntactic get, and we quotient this inductive type with the equations from the theory of state.
- 3. It turns out the resulting type is isomorphic to the state monad. And indeed many monads that we use to model computational effects can be obtained in this way. Thus Plotkin and Power's paper was titled 'notions of computations *determine* monads'.

#### **Freeness**



#### Freeness means there is a term

$$(A, \mathit{ops}) \text{ models } \Sigma \qquad x: \tau \vdash r: A$$

 $c: \Sigma^*\tau \vdash \textit{handle } c \textit{ with } \{r; ops\}: A$ 

satisfying certain judgemental equalities.

- 1. Since the free models are basically syntax trees of operations, given any other model A of the theory  $\Sigma$ , and a mapping r that sends the variables  $\tau$  to the model A, we can fold the syntax tree with the algebra A.
- 2. Here I'm calling this fold *handle*, because the computational intuition is that an element c of the free model is a computation that calls  $\Sigma$ -operations, and the model A handles the these operation calls like an operation system handles system calls.

#### **Freeness**



#### Freeness means there is a term

$$(A, \mathit{ops}) \operatorname{\mathsf{models}} \Sigma \qquad x: \tau \vdash r: A$$

 $c: \Sigma^*\tau \vdash \textit{handle } c \textit{ with } \{r; \textit{ops}\}: A$ 

satisfying certain judgemental equalities.

Effect Handlers (Plotkin and Pretnar 2013)

Internalising the above as a language feature.

1. This construction is turned into a programming language feature by Gordon Plotkin and Matija Pretnar, called effect handlers, allowing the programmer to define elements of free models and handle them with customised models easily.

### **Two Modularities**



Effect handlers have two nice properties:

- Syntactic modularity: effects can be combined
  - Two theories  $\Sigma_1$  and  $\Sigma_2$  can be combined by coproduct  $\Sigma_1 + \Sigma_2$

- 1. Effect handlers are a very nice language feature.
- 2. It not only allows the programmer to interpret a syntactic computation with different semantic models. It has some extraordinarily nice properties.
- 3. One thing is what I call *syntactic modularity* here, which means that we can combine two effects easily by taking the disjoint union of their operations and equations, and possibly adding more equation to characterise the interaction of these two effects.

### **Two Modularities**



Effect handlers have two nice properties:

- Syntactic modularity: effects can be combined
  - Two theories  $\Sigma_1$  and  $\Sigma_2$  can be combined by coproduct  $\Sigma_1 + \Sigma_2$
- Semantic modularity: effects can be handled one by one
  - Every model of  $\Sigma_1$  on  $\Sigma_2^*A$  can be extended to a model of  $\Sigma_1 + \Sigma_2$

- 1. Another thing is what I call *semantic modularity*, which means that if a computation uses two kinds of effects  $\Sigma_1$  and  $\Sigma_2$ , we can just handle  $\Sigma_1$ , leaving  $\Sigma_2$  unhandled, so effects can be modularly handled.
- 2. This is possible because a model of  $\Sigma_1$  on  $\Sigma_2$ -computations can be extended to a model of  $\Sigma_1 + \Sigma_2$  in a unique and an almost trivial way.
- 3. These nice features make effect handlers a very powerful language feature.

#### **Some Issues**



# 1. Sequential composition >>= is only a *meta-level operation* on free models:

• Impossible to state equations about >>=, e.g.  $x \gg fail = fail *$ 

- 1. So far I've been talking about things many of you already now. But what's new?
- 2. The new things in our papers are motived by two issues in algebraic effects.
- 3. The first thing is that sequential composition isn't really an operation in the algebraic theory of an effect. Instead, it is an emergent operation that happen to exist on the free models. But for other models in general, there isn't a notion of sequential composition for them.
- 4. Consequently, we cannot use equations to state the interaction of some effectful operations and sequential compositions.
- 5. For example, sometimes we may want to state the property that every nondetministic computation x followed by a failure is equal to just a failure. But in algebraic effects, we cannot state this as an equation meaningfully because the the models don't have sequential composition in general.

#### **Some Issues**



- 1. Sequential composition >>= is only a *meta-level operation* on free models:
  - Impossible to state equations about >>=, e.g.  $x \gg fail = fail *$
  - Algebraicity  $op(x) \gg k = op(x \gg k)$  is forced to be true

1. Moreover, the sequential composition on the free model necessarily satisfy a property called *algebraicity*, which says that every operation commutes with sequential composition. But some effectful operations in practice just don't satisfy this property, so they cannot be modelled by algebraic effects.
### **Some Issues**



- 1. Sequential composition >>= is only a *meta-level operation* on free models:
  - Impossible to state equations about  $\gg$ , e.g.  $x \gg fail = fail *$
  - Algebraicity  $op(x) \gg k = op(x \gg k)$  is forced to be true
- 2. Not all computational effects are monads: graded monads, applicatives, arrows, ...

1. Another problem is that the framework of algebraic effects is only about monads, but there are quite a few other concepts that are also used for modelling computational effects, such graded monads, applicative functors, etc.

### **Our Work**



We should consider

1. algebraic theories of *monads with operations* rather than *types/objects with operations*,

1. These issues are the motivation of our work. Firstly, to make sequential composition a real operation in the algebraic theory, we shift our perspective from algebraic theories on types or objects to algebraic theories on monads.

## **Our Work**



We should consider

- 1. algebraic theories of *monads with operations* rather than *types/objects with operations*,
- 2. monoids in monoidal categories rather than monads.

 Secondly, we generalise from monads to the more general concept of monoids in monoidal categories, so that other notions of computational effects such as applicatives can be modelled.

## **Our Work**



We should consider

1. algebraic theories of *monads with operations* rather than *types/objects with operations*,

2. monoids in monoidal categories rather than monads.

We'd like to preserve both modularities.

1. And when doing so, we'd like to preserve the two kinds of modularities that effect handlers have, since these nice properties are what make effect handlers so convenient.

## **Monoidal Categories**



A new type former  $\Box$  with rules:

 $\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1, \Gamma_2 \vdash (t_1, t_2) : A \square B}$ 

 $\frac{\Gamma \vdash t_1 : A_1 \Box A_2 \qquad \Gamma_l, x_1 : A_1, x_2 : A_2, \Gamma_r \vdash t_2 : B}{\Gamma_l, \Gamma, \Gamma_r \vdash match (x_1, x_2) = t_1 in t_2 : B}$ 

and a similar monoidal unit type *I*.

- 1. Let me first recap what a monoidal category is in a type theoretic setting.
- 2. Compared to unary type theory, the internal language for monoidal category has a new type former  $\Box$ . It is a linear product type, and it has a term former that allows two terms to be paired together. Note that this rule also introduce contexts that have more than one variables.
- 3. The contexts in this type theory is linear: we cannot duplicate, throw away, or even swap the variables in the context.
- 4. The elimination rule for this monoidal product allows us to unpack a monoidal product to two variables in the context. This is the only way to use the monoidal product; we don't have projections for this monoidal product.
- 5. The two term formers satisfy beta and eta equalities, which are not shown on the slide.
- 6. Moreover, there is a similar nullary monoidal unit type *I*, which has similar rules except that it has zero components instead of two components.

## Monoids



### The theory **Mon** of *monoids* has operations:

$$x:\tau,y:\tau\vdash\mu:\tau\qquad\qquad \vdash\eta:\tau$$

#### and equations:

$$x: \tau \vdash \mu(\eta, x) = x$$
  $x: \tau \vdash x = \mu(x, \eta)$ 

$$x:\tau,y:\tau,z:\tau\vdash\mu(x,\mu(y,z))=\mu(\mu(x,y),z)$$

Monads, applicatives, graded monads, arrows are all monoids.

- 1. Using this type theory, the theory of monoids is exactly what you may expect.
- 2. There are two operations  $\mu$  and  $\eta.~\eta$  is the left and right identities of  $\mu$ , and  $\mu$  is associative.
- 3. So syntactically, it looks exactly the same as the usual theory of monoids, but this type theory can be interpreted in any monoidal category. For example, if this type theory is interpreted in the category of endofunctors where the monoidal product is interpreted as composition, then monoids become monads.

14/20

A theory  $(\Sigma, E)$  of monoids with operations is

- the theory Mon of monoids plus
- an operation  $x : \Sigma \tau \vdash op : \tau$  for a functor  $\Sigma$
- ► some more equations *E*

- What we are interested in is really monoids equipped with operations. A theory of monoids with operations is just the theory Mon of monoids equipped with a new operation *op* of signature Σ, where Σ is an arbitrary functor.
- 2. Additionally, the theory can have some equations that state the properties of the operation and the monoid structure.



A theory  $(\Sigma, E)$  of monoids with operations is

- the theory Mon of monoids plus
- an operation  $x : \Sigma \tau \vdash op : \tau$  for a functor  $\Sigma$
- **some more equations** *E*

#### **Old Things Recovered**

Theory of throw:  $\Sigma \tau \equiv 1$ . Theory of state:  $\Sigma \tau \equiv (\coprod_S \tau) + (\prod_S \tau)$  and algebraicity equation  $\mu(op(x), k) = op(\mu(x, k))$ , etc.

- We can easily recover the old things. The theory of exception throwing is the theory of monoids with an operation whose signature functor is the unit type. The theory of state is the theory of monoids with an operation whose signature is the coproduct of a put operation and a get operation. And the equations include algebraicity, saying get and put commute with monoidal multiplication, and some other equations on get and put.
- 2. This is all very similar to the algebraic theories of exception and state that we saw earlier, except that now the operations are on a monoid rather than on a type.



#### New Things

1. Algebraicity is optional now: *catch, parallel composition* can be modelled as operations

 OK, so old things can be recovered, what are the new things. The first thing is that now we can talk about operations of more complex forms. In particular, we can talk about operations that do not satisfy algebraicity, such as exception catching and parallel composition. Now they can be real operations in the theory, so can be given different semantic models like algebraic operations.

### New Things

- 1. Algebraicity is optional now: *catch, parallel composition* can be modelled as operations
- 2. Equations can mention the monadic bind, e.g.

 $x \gg fail = fail$ 

 Another benefit is that now equations can talk about both the operations and the monoid structure. For example, now we can have the equation saying that a computation x followed by fail is equal to fail. This equation wasn't possible before because the variable x didn't have a monad structure, but it has one now, so this equation can be stated.

### New Things

- 1. Algebraicity is optional now: *catch, parallel composition* can be modelled as operations
- 2. Equations can mention the monadic bind, e.g.

 $x \gg fail = fail$ 

3. Theories of operations on applicatives, graded monads, substitution monoids etc.

1. Lastly, we have generalised from monads to monoids, so now we can talk about theories of operations of other kinds of monoids, such as applicative functors and graded monads.

# **Initial Algebras**



### Such a theory $(\Sigma, E)$ determines a monoid, its initial algebra: $(\mu X. \ I + (\Sigma X) \Box X) / \approx_E$

#### A Special Case for Monads in Haskell

- 1. What I'd like to say in the remaining time is that this new way of doing algebraic effects does preserve the nice properties of algebraic effects.
- 2. First of all, each algebraic theory of monoids with operations has an initial algebra when the underlying category is nice enough. Concretely, this initial algebra is a tree structure, and quotiented by the equations from the theory.
- 3. Since the theory extends the theory of monoids, the initial algebra is of course a monoid. So it is still computational effects determine monoids as before, but in a slightly different and more general way.
- 4. If we do not have equations on operation, the initial algebra for the special case of monads can be readily implemented in Haskell, which is quite like the free monads, except that the signature functor can now be a higher-order functor. My coauthor Nick will have a talk in the Haskell Symposium going into more details of this monad, so if you are interested in hearing more about concrete implementations of what I've been talking about, don't miss his talk.

## **Families of Theories**

Theories are sorted into *theory families* based on the shape of operations  $\Sigma$ :

- Algebraic operations:
- Scoped operations:
- Variable-binding operations:

 $\Sigma \tau \equiv A \Box \tau$  $\Sigma \tau \equiv A \Box \tau \Box \tau$  $\Sigma \tau \equiv \tau^{V}$ 

- 1. There are also a few theoretically interesting things that we can say. First of all, we can organize all theories of monoids with operation into subcategories based on the shape of their operations. We call these subcategories theory families here.
- 2. For example we have the theory family of algebraic operations, which contains all theories whose signature functor is of the shape  $A \Box \tau$ .
- 3. And we also have the family of scoped operations, which are operations that delimit scopes, such as exception catching, and there's also the family of variable-binding operations, such as lambda abstraction.

### **Some Results**



Theorem (Theory-Monoid Correspondence)

The theory family  $ALG(\mathcal{E})$  of algebraic operations is *equivalent to the category of monoids* in  $\mathcal{E}$ .

 It turns out the theory family of algebraic operations plays a special role among others. First of all, it can be shown to be equivalent to the category of monoids, resulting in a generalisation of the classical monad-theory correspondence at the generality of monoids.

### **Some Results**



Theorem (Theory-Monoid Correspondence)

The theory family  $ALG(\mathcal{E})$  of algebraic operations is *equivalent to the category of monoids* in  $\mathcal{E}$ .

#### Theorem

 $ALG(\mathcal{E})$  is a *coreflective subcategory* of a wide class of theories, and the coreflector preserves initial algebras.

 And it can be further shown that the category of theories of algebraic operations is a coreflective subcategory of a wide class of theories, and the coreflector preserves initial algebras. This mean that the syntax of more complex forms of operations can be simulated by algebraic operations.

## **Modular Models**



### Semantic modularity is recovered using

Modular Models

A strict *modular model* M of  $\Gamma$  among a theory family  $\mathcal{F}$  is a collection of functors natural in  $\Sigma \in \mathcal{F}$ :

 $M_{\Sigma}: \Sigma \operatorname{-Alg} \to (\Sigma + \Gamma) \operatorname{-Alg}$ 

### Examples: monad transformers, free modular models, etc

- 1. The last thing I want to say is that semantic modularity is not automatic in our setting. Therefore we formulate a notion of modular models of a theory. A modular model is pretty much like a monad transformer, it basically takes in a monoid equipped with some operations  $\Sigma$ , and transforms it into a new monoid with both  $\Sigma$  and  $\Gamma$  operations. And this transformation should be natural in  $\Sigma$  in a certain sense.
- 2. So modular models are a glorified version of monad transformers. A monad transformer sends a monad to a new monad, while a modular model sends a monoid with some operation to a new monoid with more operation.
- 3. Such modular models can be obtained by a nubmer of ways, such as from monad transformers, and there are a few other constructions in the paper.





#### A shift of perspective

theories on objects to theories on monads, or better, monoids

and we gain a lot more generality!