Computational Effects:

A Story of Categories and Algebras

Yunkai Zhang

yz9522@ic.ac.uk

Computational Effects:

A tory of Categories and Algebras

What the heck is this?

We want to interact with the world

...in a functional taste!

We want to interact with the world

...in a functional taste!

Why do people ❤ functional programming?

- We compose our programs out of:


Input(s)

Our Lovely Program

result

All the way – from the tiniest bits to the big components.

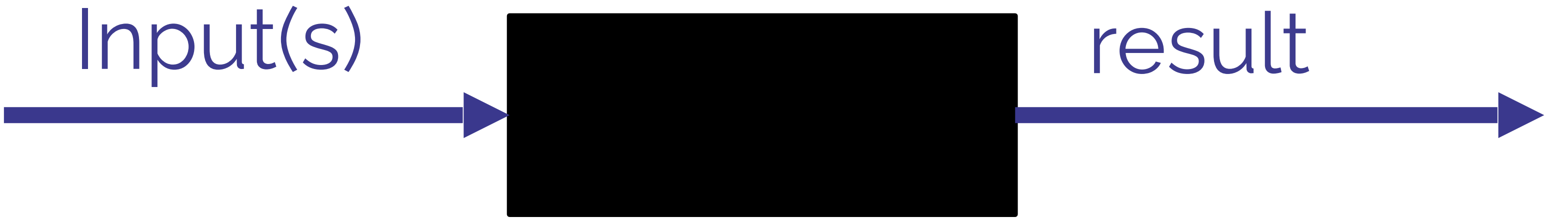
Why do people ❤️ functional programming?

- We compose our programs out of:

All the way – from the tiniest bits to the big components.

Why do people ❤️ functional programming?

- We compose our programs out of:



All the way – from the tiniest bits to the big components.

We love modularity!

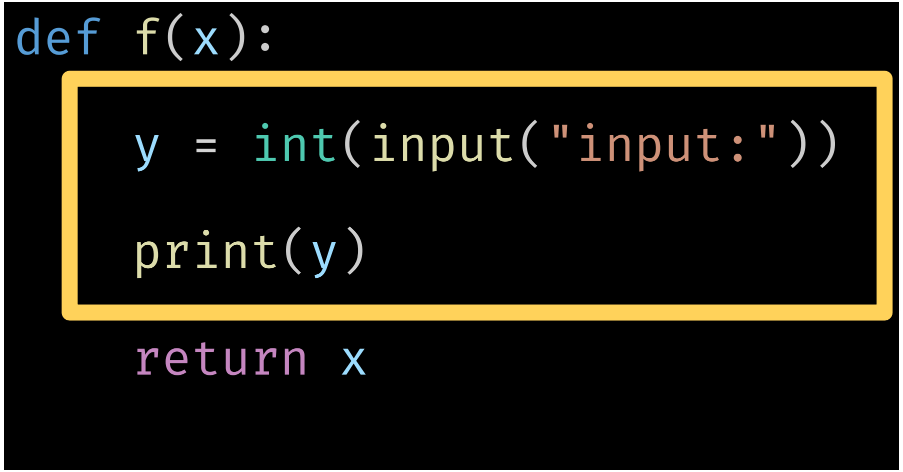
Why do people ❤️ functional programming?

- We compose our programs out of:

All the way – from the tiniest bits to the big components.

Well… Yes… But…

Consider an annoying example...



```
def f(x):
    y = int(input("input:"))
    print(y)
    return x
```

```
def f(x):
    return x
```

They return the same thing, Shouldn't they be the same function?

Why do people ❤ functional programming?

- We compose our programs out of:

Input(s)

Our Lovely Program

result

Effects Happening

Some Common Computational Effects

- Input / Output
- Error Handling and Exceptions
- Mutable State
- Logging
- Concurrency
- Generating Random Values

……

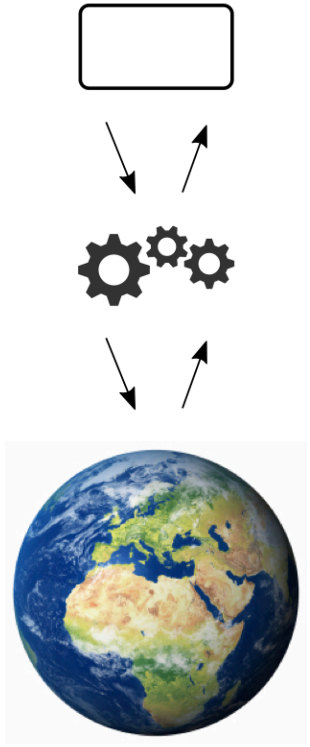
We want to interact with the world

…in a functional taste!

We want to interact with the world

...in a functional taste!

What is an effect?



What is an Effect?

Categories 101

$$\begin{tikzcd}
 & B \arrow[dr, "g"] & \\
A \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & C
\end{tikzcd}$$

What is an Effect?

Categories 101

Monads

What is an Effect?

Categories 101

Monads

What is an Effect?

Categories 101

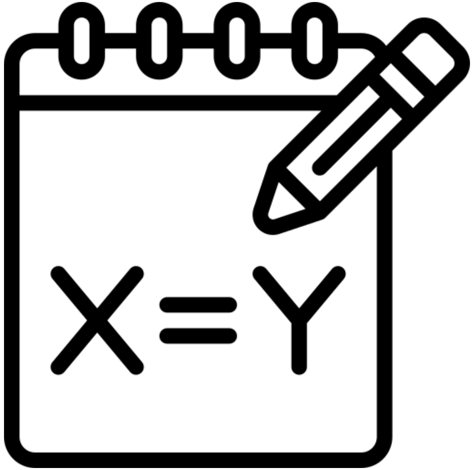
Monads
(Categories for Computational Effects)

What is an Effect?

Categories 101

Monads
(Categories for Computational Effects)

Lawvere Theories

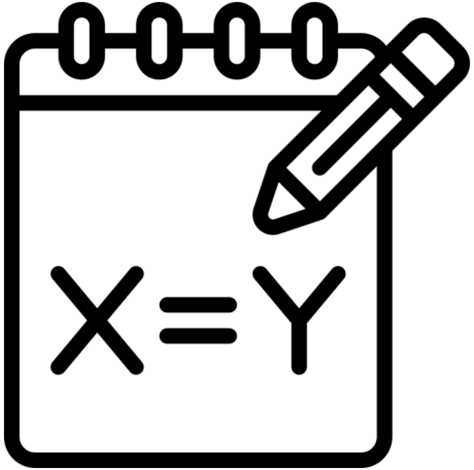


What is an Effect?

Categories 101

Monads
(Categories for Computational Effects)

Lawvere Theories
(Categories of Operations and Equations)



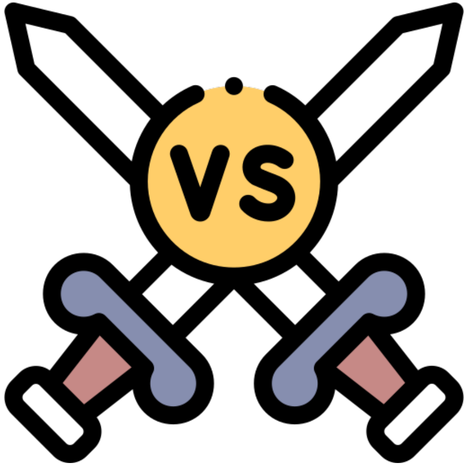
What is an Effect?

Categories 101

Monads
(Categories for Computational Effects)

Lawvere Theories
(Categories of Operations and Equations

A Comparison



Categories 101

Functions

A function, e.g. $f(x) = x^3 - 9x + 7$

comes with specified sets of “possible input values” and “potential output values.” $f : I \to R$

We write

$$I \xrightarrow{f} O$$

to indicate f has source **I** and target **O**.

Functions Compose!

- So why bother with source and targets?
- They indicate when are two functions composable:

$$A \xrightarrow{f} B \qquad \text{and} \qquad B \xrightarrow{g} C$$

- Are composable:
- just when the target of **f** equals the source of **g**.

What is a category?

- A category is a two-sorted structure encoding the algebra of composition.
- It has:
    - Objects: A, B, C, ...and
    - Morphisms: $A \xrightarrow{f} B,\ B \xrightarrow{g} C$, each specified with source and target
- So that,
    - each pair of composable arrows f and g will have a composite arrow $f \circ g$
    - Each object has an identity arrow $A \xrightarrow{\mathrm{id}_A} A$

$$\begin{tikzcd}
 & B \arrow[dr, "g"] & \\
A \arrow[ur, "f"] \arrow[rr, "g \circ f"'] & & C
\end{tikzcd}$$

- for which the composition operation is associative and unital.

Isomorphisms

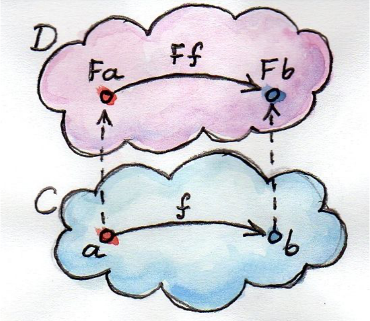
- An **isomorphism** consists of:

$$\begin{tikzcd} A \arrow[r, shift left, "f"] & B \arrow[l, shift left, "g"] \end{tikzcd}$$

- So that:
- $g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$
- If A and B are isomorphic
- then every category theoretic property of A is also true of B.

Functors

- Given categories **C** and **D**, a **functor** $F : C \to D$ is specified by:
  - A mapping obj $C \to$ obj D whose value at X is written FX
  - For all $X, Y \in C$, a function $Hom_C(X, Y) \to Hom_D(FX, FY)$ whose value at $f : X \to Y$ is written $Ff : FX \to FY$
  - …which is required to preserve composition and identity morphisms:
    - $F(g \circ f) = F(g) \circ F(f)$
    - $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$



Exemplar Categories

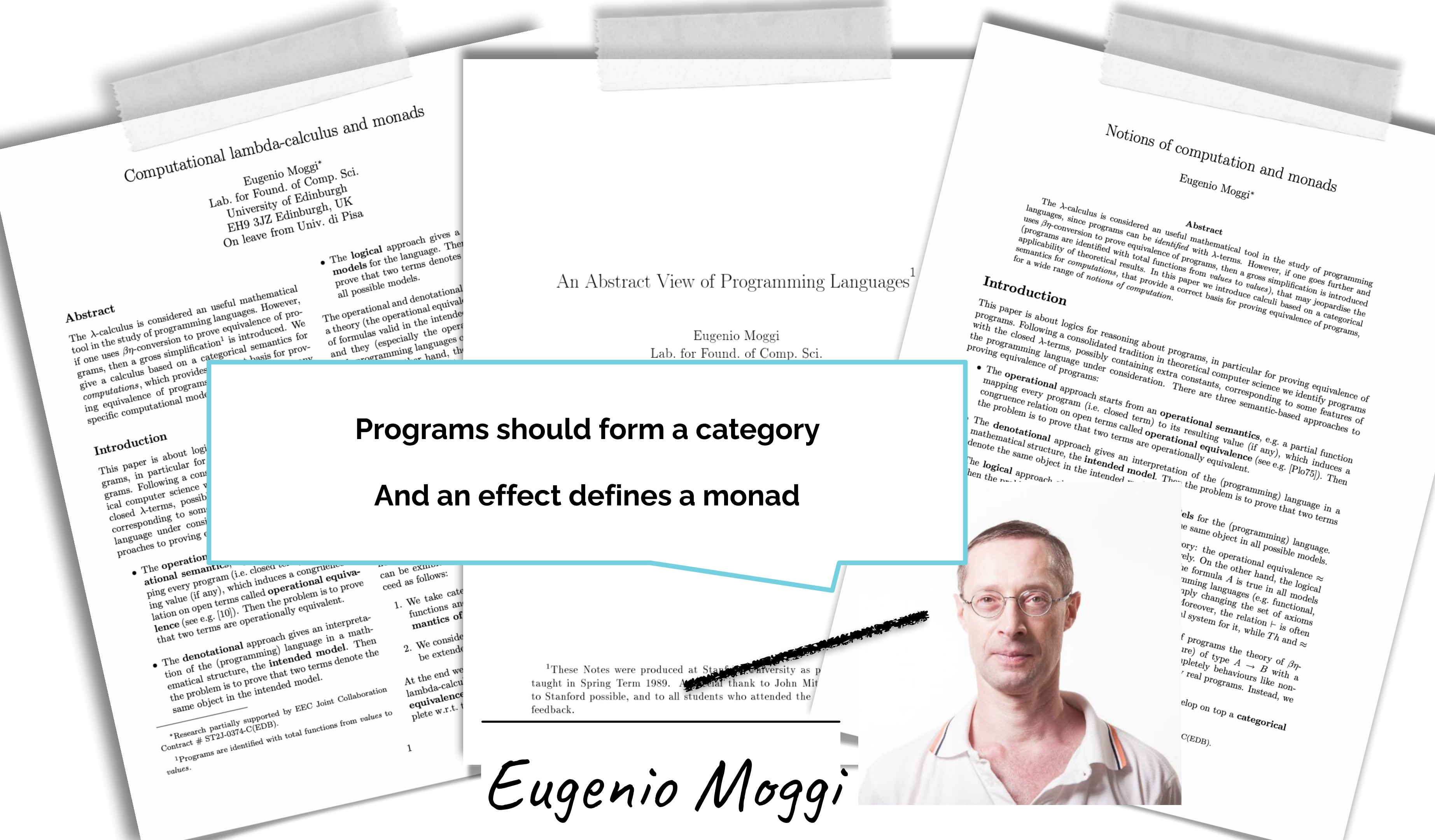
- In the category Set, the
  - Objects are (finite) sets X, Y, …
  - Arrows are functions $X \xrightarrow{f} Y$…

- In the syntactic category for some programming language, the
  - Objects are types X, Y, …
  - Arrows are programs $X \xrightarrow{f} Y$

- The precise ontology of the objects and arrows won't matter much.

Categories for Computational Effects

(Monads!)



Programs should form a category

And an effect defines a monad

Eugenio Moggi

Computational lambda-calculus and monads

Eugenio Moggi[*]
Lab. for Found. of Comp. Sci.
University of Edinburgh
EH9 3JZ Edinburgh, UK
On leave from Univ. di Pisa

Abstract

The λ-calculus is considered an useful mathematical tool in the study of programming languages. However, if one uses $\beta\eta$-conversion to prove equivalence of programs, then a gross simplification[1] is introduced. We give a calculus based on a categorical semantics for *computations*, which provides ... ing equivalence of programs ... specific computational mode...

Introduction

This paper is about logi... grams, in particular for ... grams. Following a cons... ical computer science ... closed λ-terms, possib... corresponding to som... language under consi... proaches to proving e...

- The **operation**... **ational semantics**, ... ping every program (i.e. closed ... ing value (if any), which induces a congru... lation on open terms called **operational equivalence** (see e.g. [10]). Then the problem is to prove that two terms are operationally equivalent.
- The **denotational** approach gives an interpretation of the (programming) language in a mathematical structure, the **intended model**. Then the problem is to prove that two terms denote the same object in the intended model.
- The **logical** approach gives a ... **models** for the language. The... prove that two terms denotes ... all possible models.

The operational and denotational ... a theory (the operational equival... of formulas valid in the intende... and they (especially the opera... programming languages ... other hand, th...

... can be exhib... ceed as follows:

1. We take cate... functions an... **mantics of**
2. We conside... be extende...

At the end we ... lambda-calcu... **equivalence** ... plete w.r.t. ...

---

[*] Research partially supported by EEC Joint Collaboration Contract # ST2J-0374-C(EDB).

[1] Programs are identified with total functions from *values* to *values*.

An Abstract View of Programming Languages[1]

Eugenio Moggi
Lab. for Found. of Comp. Sci.

---

[1] These Notes were produced at Sta[redacted]iversity as p... taught in Spring Term 1989. A...ial thank to John Mit... to Stanford possible, and to all students who attended the ... feedback.

Notions of computation and monads

Eugenio Moggi*

Abstract

The λ-calculus is considered an useful mathematical tool in the study of programming languages, since programs can be *identified* with λ-terms. However, if one goes further and uses $\beta\eta$-conversion to prove equivalence of programs, then a gross simplification is introduced (programs are identified with total functions from *values* to *values*), that may jeopardise the applicability of theoretical results. In this paper we introduce calculi based on a categorical semantics for *computations*, that provide a correct basis for proving equivalence of programs, for a wide range of *notions of computation*.

Introduction

This paper is about logics for reasoning about programs, in particular for proving equivalence of programs. Following a consolidated tradition in theoretical computer science we identify programs with the closed λ-terms, possibly containing extra constants, corresponding to some features of the programming language under consideration. There are three semantic-based approaches to proving equivalence of programs:

- The **operational** approach starts from an **operational semantics**, e.g. a partial function mapping every program (i.e. closed term) to its resulting value (if any), which induces a congruence relation on open terms called **operational equivalence** (see e.g. [Plo75]). Then the problem is to prove that two terms are operationally equivalent.
- The **denotational** approach gives an interpretation of the (programming) language in a mathematical structure, the **intended model**. Then the problem is to prove that two terms denote the same object in the intended m...
- The **logical** approach ... hen the pro... ...**els** for the (programming) language. ...e same object in all possible models.

...ory: the operational equivalence ≈ ...ely. On the other hand, the logical ...e formula A is true in all models ...mming languages (e.g. functional, ...ply changing the set of axioms ...oreover, the relation ⊢ is often ...l system for it, while Th and ≈

...f programs the theory of $\beta\eta$- ...re) of type $A \to B$ with a ...pletely behaviours like non- ...r real programs. Instead, we

...elop on top a **categorical**

...C(EDB).

Monads, Categorically

- A monad over a category **C** is a triple (T, η, μ), where
  - $T : C \to C$ is a functor,
  - Morphisms $\eta_A : A \to TA$ and $\mu_A : T^2(A) \to T(A)$ for every object $A \in C$
  - (For those of you who know: actually natural transformations)
  - Make the following diagrams commute:

$$\begin{tikzcd}
T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"'] & TA
\end{tikzcd}$$

$$\begin{tikzcd}
TA \arrow[r, "\eta_{TA}"] \arrow[dr, "\mathrm{id}_{TA}"] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "\mathrm{id}_{TA}"] \\
& TA &
\end{tikzcd}$$

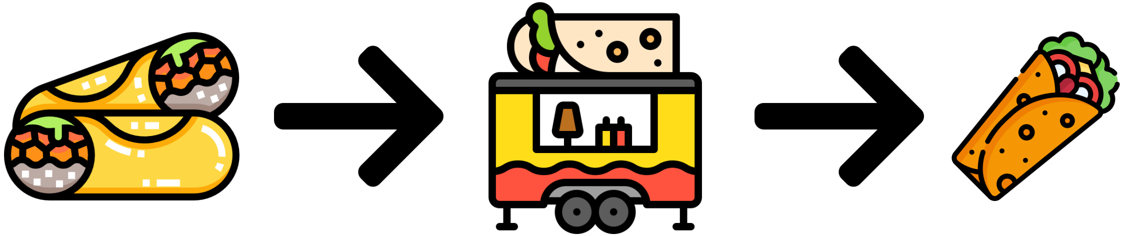
Monads are Burritos

Assume C is a category of foods.

- $T : C \to C$ is a functor, like burritos

- $\eta_A : A \to TA$ takes a regular value and turns it into a burrito

- $\mu_A : T^2(A) \to T(A)$ takes a ridiculous burrito of burritos and turns them into a regular burrito.



Monads are Monoids

$$\begin{tikzcd}
T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"'] & TA
\end{tikzcd}$$

$$\begin{tikzcd}
TA \arrow[r, "\eta_{TA}"] \arrow[dr, "\mathrm{id}_{TA}"] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "\mathrm{id}_{TA}"] \\
& TA &
\end{tikzcd}$$

A monad is a monoid in the category of endofunctors

T-Programs

Let **T** be a notion of computation (encoding an effect)

A T-program from A to B is a function $A \xrightarrow{f} T(B)$,
from the set of values of type A to the set of T-computations of type B.

We also write $A \overset{f}{\rightsquigarrow} B$ to denote $A \xrightarrow{f} T(B)$

T-Programs

```
def greaterThanM (m : ℕ) (n : ℕ)  : Prop
  := n > m

-- setOf (p : α → Prop) : Set α

def TNat (m : ℕ) : Set ℕ :=
  setOf (greaterThanM m)
```

ℕ → LeanSet ℕ

T-Programs

```
def greaterThanM (m : ℕ) (n : ℕ)  : Prop
  := n > m

-- setOf (p : α → Prop) : Set α

def TNat (m : ℕ) : Set ℕ :=
  setOf (greaterThanM m)
```

$$\mathbb{N} \to T(\mathbb{N})$$

Sample "Notions of Computation" / T-Programs in the Category of Sets

- **Exceptions:** $TA = A + E$ where E is the set of exceptions
- **Partiality:** $TA = A + \{ \perp \}$ (i.e. `Option A` in Lean 4)
- **Side-Effect:** $TA = (S \times A)^S$ or $TA = \{S \to S \times A\}$ for fixed set S

  modelling the effect of a single mutable state storing a value of S.
- **Continuation:** $TA = R^{(R^A)}$, or $TA = (A \to R) \to R$ for a fixed set R

  models the effect of ***call with current continuation*** (`call/cc`).
- **Interactive Input:** $TA = \mu X \,.\, A + X^U$, where U is the set of characters
  - i.e set of **U**-branching trees (with finite branches) and **A**-labelled leaves, where $\mu X \,.\, \tau$ is the least solution to domain equation $X = \tau$

Sample "Notions of Computation" / T-Programs in the Category of Sets

- **Exceptions:** $TA = A + E$ where E is the set of exceptions
- **Partiality:** $TA = A + \{ \perp \}$ (i.e. `Option A` in Lean 4)
- **Side-Effect:** $TA = (S \times A)^S$ or $TA = \{S \to S \times A\}$ for fixed set S

  modelling the effect of a single mutable state storing a value of S.
- **Continuation:** $TA = R^{(R^A)}$, or $TA = (A \to R) \to R$ for a fixed set R

  models the effect of ***call with current continuation*** (`call/cc`).
- **Interactive Input:** $TA = \mu X \,.\, A + X^U$, where U is the set of characters
  - i.e set of **U**-branching trees (with finite branches) and **A**-labelled leaves, where $\mu X \,.\, \tau$ is the least solution to domain equation $X = \tau$

Slogan

A computational effect **T** defines a **monad**

just when the T-programs $A \overset{f}{\rightsquigarrow} B$

defines the arrows in a category of **T**-programs Kl_T

Slogan

A computational effect **T** defines a **monad**

just when the T-programs $A \overset{f}{\rightsquigarrow} B$

defines the arrows in a category of **T**-programs Kl_T

The Category of T-Programs

- To define the category of T-programs we need:
  - Identity Arrows: $A \overset{\mathrm{id}_A}{\rightsquigarrow} A$
  - We can use η in monads! (Reminder: $\eta : \mathrm{Id}_C \to T$) so $A \xrightarrow{\eta_A} T(A)$
  - We need to define composition making this diagram commute:

$$\begin{tikzcd}
& B \arrow[dr, rightsquigarrow, "g"] & \\
A \arrow[ur, rightsquigarrow, "f"] \arrow[rr, rightsquigarrow, "g \circ f"'] & & C
\end{tikzcd}$$

Problem:
$A \xrightarrow{f} T(B)$ and $B \xrightarrow{g} T(C)$ are not composable!

Kleisli Triple

- A Kleisli Triple over a category **C** is a triple (T, η, μ), where
  - $T : C \rightarrow C$ is a functor,
  - $\eta_A : A \rightarrow TA$ for every object $A \in C$
  - $f^* : TA \rightarrow TB$ for every object $A, B \in C$ and $f : A \rightarrow TB$

- Then we can have the Kleisli Composite of $A \overset{f}{\rightsquigarrow} B$ and $B \overset{g}{\rightsquigarrow} C$

$$\begin{tikzcd}
& \mathtt{T}(B) \arrow[dr, "g^*"] & \\
A \arrow[ur, "f"] \arrow[rr, "g^* o f"'] & & \mathtt{T}(C)
\end{tikzcd}$$

Kleisli Category over T: Kl_T

- Given an endofunctor **T** on category $\mathscr{C}$, the Kleisli Category Kl_T is defined as:
  - Obj $(Kl_T) =$ Obj$(\mathscr{C})$
  - $\mathrm{id}_A = \eta_A : A \to T(A)$ (i.e. $A \overset{\mathrm{id}_A}{\rightsquigarrow} A$)
  - $Hom_{Kl_T}(A, B) = Hom_{\mathscr{C}}(A, TB)$
  - $g \circ_{Kl} f = g^* \circ f : A \to TC$, given $f : A \to T(B)$ and $g : B \to T(C)$

$$\begin{tikzcd}
 & B \arrow[dr, rightsquigarrow, "g"] & \\
A \arrow[ur, rightsquigarrow, "f"] \arrow[rr, rightsquigarrow, "g\circ f"'] & & C
\end{tikzcd}$$

There is a one-to-one correspondence between

Kleisli Triples and Monads

Kl_{list} for list-computations

- A list-program $A \overset{f}{\rightsquigarrow} B$ is a function from A to lists in B
- The identity A $\overset{\mathrm{id}_A}{\rightsquigarrow}$ list(A) is the function $A \xrightarrow{\text{singleton}} \text{list}(A)$
- Any function $B \xrightarrow{g*} \text{list}(C)$ extends to a function $\text{list}(B) \to \text{list}(C)$
  - by applying g to each term in a list of elements of B
  - and concatenating the result.

The Kleisli composite

$$\begin{tikzcd}
 & B \arrow[dr, rightsquigarrow, "g"] & \\
A \arrow[ur, rightsquigarrow, "f"] \arrow[rr, rightsquigarrow, "g \circ f"'] & & C
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \texttt{list}(B) \arrow[dr, "g^*"] & \\
A \arrow[ur, "f"] \arrow[rr, "g^* \circ f"'] & & \texttt{list}(C)
\end{tikzcd}$$

is defined by application of f and g followed by concatenation.

Monads in Programming

```
class Monad m where
    (>>=)  :: m a → (a → m b) → m b
    return :: a → m a
```

Monads in Programming

```
class Monad m where
    (>>=)  :: m a → (a → m b) → m b
    return :: a → m a
```

$T : C \to C$ is a functor

Monads in Programming

```
class Monad m where
    (>>=)  :: m a → (a → m b) → m b
    return :: a → m a
```

$\eta : \mathrm{Id}_C \to T$ is a natural transformation

So $\eta_A : A \to TA$ is a morphism

Monads in Programming

```
class Monad m where
    (>>=)  :: m a → (a → m b) → m b
    return :: a → m a
```

$f^* : TA \to TB$ for every object $A, B \in C$ and $f : A \to TB$

Monads in Programming

```
class Monad m where
    join   :: m (m a) → m a
    return :: a → m a
```

$\mu : T^2 \to T$ is a natural transformation

So $\mu_A : T^2A \to TA$ is a morphism

Monads in Programming

```
class Monad m where
    (>>=)  :: m a → (a → m b) → m b
    return :: a → m a
```

$f^* : TA \to TB$ for every object $A, B \in C$ and $f : A \to TB$

Exceptions as a Monad

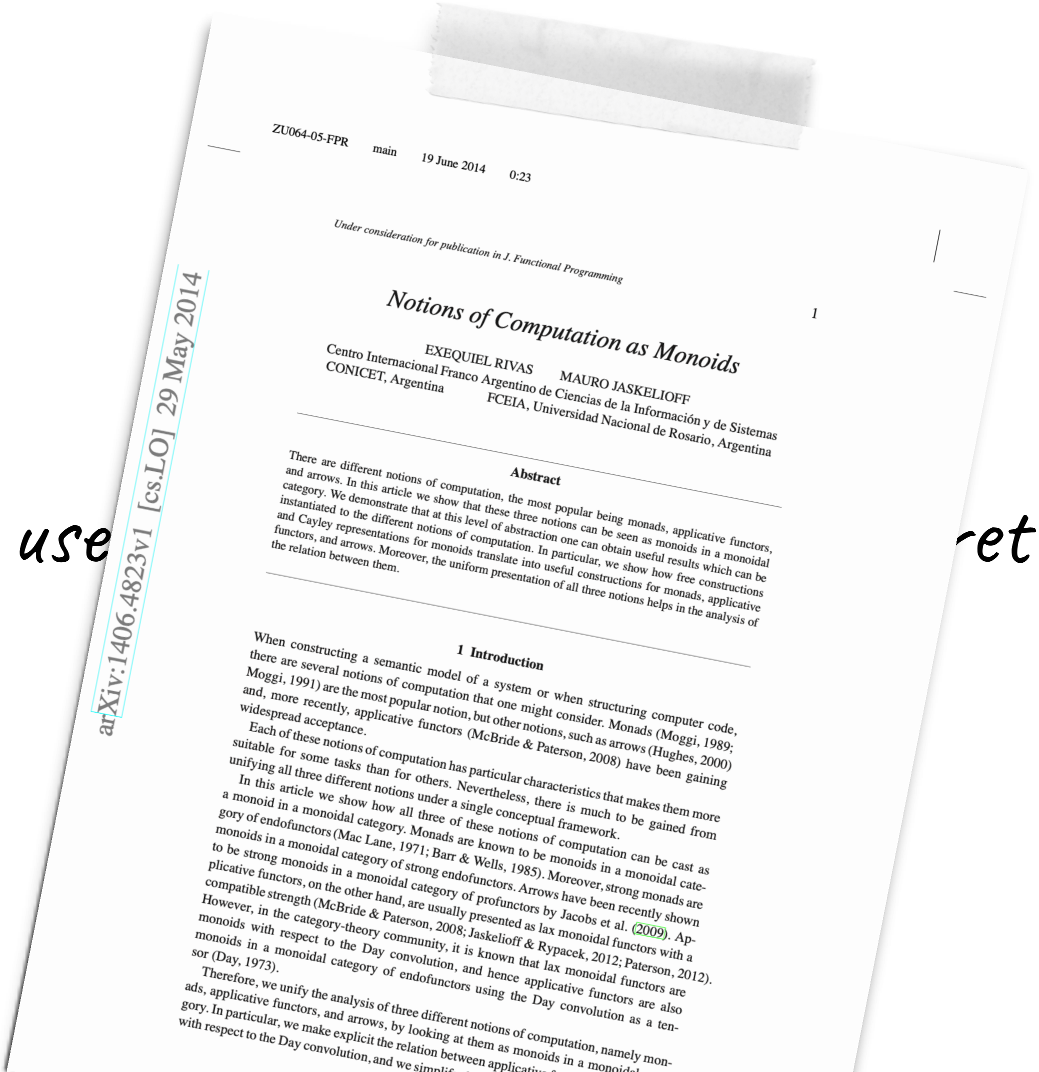
- $TA = \text{val}\,A \,|\, \text{Exn}\,E \quad (\approx A + E)$
- $\eta_v = \text{val}\,V$ (in Haskell Monad, `return v`)
- Given $f : A \to TB$:
  - $(\texttt{val}\ \ V)\ \texttt{>>=}\ f = f\ v$
  - $(\texttt{Exn}\ \ E)\ \texttt{>>=}\ f = \texttt{Exn}\ \ E$
- Operations Specific to Exceptions:
  - $\text{raise}\ e = \text{Exn}\ e$
  - $\text{try}\ a\ \text{with}\ x \to b = \text{match}\ a\ \text{with}\ (\text{val}\ v \to v \,|\, \text{Exn}\ x \to b)$

Mutable States as a Monad

- $TA = S \to A \times S$ (S is the types of states)
- $\eta_v = \forall s\,.\,(v, s)$ (in Haskell Monad, `return v`)
- Given $f : A \to TB$:
  - $a >\!\!>\!\!= f = \forall s_1\,.f\ x\ s_2$ `where` $(x, s_2) = a s_1$ (threading the state)
- Operations Specific to Mutable States:
  - get $loc = \forall s\,.\,(s(loc), s)$
  - set $loc\ v = \forall s\,.\,((), s\{l \leftarrow v\})$

Monads are used as ‘backbones’ for interpreting effects

Monads are use … reting effects



ZU064-05-FPR main 19 June 2014 0:23

Under consideration for publication in J. Functional Programming

1

arXiv:1406.4823v1 [cs.LO] 29 May 2014

Notions of Computation as Monoids

EXEQUIEL RIVAS MAURO JASKELIOFF
Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas
CONICET, Argentina FCEIA, Universidad Nacional de Rosario, Argentina

Abstract

There are different notions of computation, the most popular being monads, applicative functors, and arrows. In this article we show that these three notions can be seen as monoids in a monoidal category. We demonstrate that at this level of abstraction one can obtain useful results which can be instantiated to the different notions of computation. In particular, we show how free constructions and Cayley representations for monoids translate into useful constructions for monads, applicative functors, and arrows. Moreover, the uniform presentation of all three notions helps in the analysis of the relation between them.

1 Introduction

When constructing a semantic model of a system or when structuring computer code, there are several notions of computation that one might consider. Monads (Moggi, 1989; Moggi, 1991) are the most popular notion, but other notions, such as arrows (Hughes, 2000) and, more recently, applicative functors (McBride & Paterson, 2008) have been gaining widespread acceptance.

Each of these notions of computation has particular characteristics that makes them more suitable for some tasks than for others. Nevertheless, there is much to be gained from unifying all three different notions under a single conceptual framework.

In this article we show how all three of these notions of computation can be cast as a monoid in a monoidal category. Monads are known to be monoids in a monoidal category of endofunctors (Mac Lane, 1971; Barr & Wells, 1985). Moreover, strong monads are monoids in a monoidal category of strong endofunctors. Arrows have been recently shown to be strong monoids in a monoidal category of profunctors by Jacobs et al. (2009). Applicative functors, on the other hand, are usually presented as lax monoidal functors with a compatible strength (McBride & Paterson, 2008; Jaskelioff & Rypacek, 2012; Paterson, 2012). However, in the category-theory community, it is known that lax monoidal functors are monoids with respect to the Day convolution, and hence applicative functors are also monoids in a monoidal category of endofunctors using the Day convolution as a tensor (Day, 1973).

Therefore, we unify the analysis of three different notions of computation, namely monads, applicative functors, and arrows, by looking at them as monoids in a monoidal gory. In particular, we make explicit the relation between applicativ with respect to the Day convolution, and we

Monads are used as 'backbones' for interpreting

Some effects

~~Monads~~ are used as ‘backbones’ for interpreting Some effects

Monoids in Monoidal Categories

But where do these monads come from?

Algebraic Theories for Algebraic Effects

(Lawvere Theories!)

Algebraic Theories

Algebraic Theory

A signature $\Sigma = (\Sigma, \mathrm{ar})$ consists of

- a set Σ of operation symbols and
- a function $\mathrm{ar}\colon \Sigma \to \mathbb{N}$, which assigns the arity $\mathrm{ar}(\underline{\mathsf{op}})$ for each $\underline{\mathsf{op}} \in \Sigma$.

For a signature Σ and a set X, the set of Σ-terms $\mathit{Term}_\Sigma(X)$ generated by X is defined as the smallest set such that

- $X \subseteq \mathit{Term}_\Sigma(X)$ and
- for any $\underline{\mathsf{op}} \in \Sigma$ and $t_1, \ldots, t_{\mathrm{ar}(\underline{\mathsf{op}})} \in \mathit{Term}_\Sigma(X)$, $\underline{\mathsf{op}}(t_1, \ldots, t_{\mathrm{ar}(\underline{\mathsf{op}})}) \in \mathit{Term}_\Sigma(X)$.

An equation is a pair (ℓ, r) of Σ-terms $\ell, r \in \mathit{Term}_\Sigma(V)$. We sometime write an equation (ℓ, r) as $V \vdash \ell = r$.

An algebraic theory $\mathfrak{T}$ is a pair $(\Sigma, \mathcal{E})$ of a signature Σ and a set of equations $\mathcal{E} = \{V_i \vdash \ell_i = r_i\}_{i \in I}$.

Algebraic Structures

- An algebraic structure comprises of:
  - a set (or a type), called the **carrier** of the structure;
  - **operations** over this set (with name and arity)
  - **equations** (laws) that these operations satisfy.

Algebraic Structure of Monoids

- A monoid $(T, \epsilon, \circ)$ can be viewed as an algebraic structure, where
  - **Carrier:** T
  - **Operations**:
    - $\epsilon : 0$
    - $\circ : 2$
  - **Equations**:
    - $\epsilon \circ x = x$
    - $x \circ \epsilon = x$
    - $x \circ (y \circ z) = (x \circ y) \circ z$

Algebraic Structure of Monoids

- A monoid $(T, \epsilon, \circ)$ can be viewed as an algebraic Structure, where
  - **Carrier:** T
  - **Operations**:
    - $\epsilon : 0$
    - $m : 2$
  - **Equations**:
    - $m(\epsilon, x) = x$
    - $m(x, \epsilon) = x$
    - $m(x, m(y, z)) = m(m(x, y), z)$

Algebraic Structure of Monoids

- A monoid $(T, \epsilon, \circ)$ can be viewed as an algebraic structure, where
  - **Carrier:** T
  - **Operations**:
    - $\epsilon : 0$
    - $m : 2$
  - **Equations**:
    - $\{x\} \vdash m(\epsilon, x) = x$
    - $\{x\} \vdash m(x, \epsilon) = x$
    - $\{x, y, z\} \vdash m(x, m(y, z)) = m(m(x, y), z)$

Algebraic Theory

- A theory contains
    - the signature of operators (names and types)
    - the equations
    - the carrier is NOT to be specified here (it’s abstract!)

Algebraic Theory (first-order finitary)

- A theory contains
  - the signature of operators (names and types)
  - the equations
  - the carrier is NOT to be specified here (it's abstract!)

Algebraic Theory of Monoid

- **Operations**:
  - $\epsilon : 0$
  - $m : 2$
- **Equations**:
  - $\{x\} \vdash m(\epsilon, x) = x$
  - $\{x\} \vdash m(x, \epsilon) = x$
  - $\{x, y, z\} \vdash m(x, m(y, z)) = m(m(x, y), z)$

Algebraic Theory & Models

- A model of the theory:
  - a definition of the support and of the operations that satisfies the equations.
  - Alternatively, is an interpretation of the signature Σ_T which validates all the equations under some carrier
- We refer to a model of theory **T** as a **T-model** or a **T-algebra**.

- Example models for theory of monoids:
  - $(\mathbb{N}, 0, +)$,
  - $(\mathbb{R}, 1, \times)$,
  - $(T \to T, id, \circ)$

Free Monoid

- Given a set (an “alphabet”) **A**,
  the free monoid over **A** is $(A*, \epsilon, \cdot\ )$, where
  - **support:** $A*$, the set of finite lists of **A** (“words over **A**”) like $a_1a_2\cdots a_n$
  - identity element ϵ: the empty list;
  - composition $\cdot$: list concatenation.

- Example: taking $A = \{1,...,9\}$,
  $1 \cdot (23 \cdot 456) = (1 \cdot 23) \cdot 456 = 123456$

Free Monoid

- Given a set (an "alphabet") **A**,
  the free monoid over **A** is $(A*, \epsilon, \cdot\ )$, where
  - **support:** $A*$, the set of finite lists of **A** ("words over **A**") like $a_1 a_2 \cdots a_n$
  - identity element ϵ: the empty list;
  - composition $\cdot$: list concatenation.

- Example: taking $A = \{1,...,9\}$,
  $1 \cdot (23 \cdot 456) = (1 \cdot 23) \cdot 456 = 123456$

Free Models

- Let **T** be an **algebraic theory** and **X** a set.
- A **free T-model generated by X** is a T-model M and a function $f : X \to$ support(M) such that:

- For every other **T-model** M' and function $f' : X \to$ supp(M'), there exists a unique morphism $\Phi : M \to M'$ such that the following diagram commutes:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[dr, "f'"'] & supp(M) \arrow[d, "\Phi"] \\
 & supp(M')
\end{tikzcd}$$

Free Models of a Theory will Determine a Monad

Lawvere Theories

- **Lawvere's idea**: no matter how you present the theory, the same operations should be derivable and satisfy the same equations

- A Lawvere theory bundles derivable operations and their equations into a category.

Monad-Theory Correspondence and How should we work more

- A monad is:
  - A “computational effect” $Set \xrightarrow{T} Set$
  - So that T-programs $A \xrightarrow{f} T(B)$ define the (squiggly) arrows in the category Kl_T
- The opposite of the category of T-programs between finite sets defines a Lawvere theory L_T^{op}. Conversely, any Lawvere theory L defines a monad T_L on category of set.
- **Theorem:** The category of Lawvere theories is equivalent to the category of finitary monads on Set.
- **Finitary** monads and (Lawvere theories describe equivalent categorical encodings of universal algebra.
-

Moreover if you are interested

Some Readings

Category Theory for Programmers

Sam Lindley's Effect Handler Oriented Programming slides

More to be updated at https://github.com/YunkaiZhang233/effect-reading/blob/main/README.md

Thank You!