A few comments on interfaces for computational effects

Exequiel Rivas

April 2018

$\pi.r^2$ à Fontainebleau

The good old simply typed λ-calculus

Let's begin with a basic λ-calculus with products and unit.

$$M, N ::= x \mid MN \mid \lambda x.M \mid (M, N) \mid \pi_i(M) \mid \star$$

The good old simply typed λ-calculus

Let's begin with a basic λ-calculus with products and unit.

$$M, N ::= x \mid MN \mid \lambda x.M \mid (M, N) \mid \pi_i(M) \mid \star$$

It has a simple type system.

$$\tau, \sigma ::= B \mid \tau \to \sigma \mid \tau \times \sigma \mid \star$$

The good old simply typed λ-calculus

Let's begin with a basic λ-calculus with products and unit.

$$M, N ::= x \mid MN \mid \lambda x.M \mid (M, N) \mid \pi_i(M) \mid \star$$

It has a simple type system.

$$\tau, \sigma ::= B \mid \tau \to \sigma \mid \tau \times \sigma \mid \star$$

Its semantics can be given by cartesian closed categories (CCC).

$$\boxed{\lambda^{\to}} \xrightarrow[{[\![\cdot]\!]}]{} \boxed{\mathrm{CCC}}$$

λ-calculus and effects

A neat example of the Curry–Howard–Lambek correspondence.

λ-calculus and effects

A neat example of the Curry–Howard–Lambek correspondence.

However...

- No interaction with the environment.
- It means... not very useful for programming *computers*.

λ-calculus and effects

A neat example of the Curry–Howard–Lambek correspondence.

However...

- No interaction with the environment.
- It means... not very useful for programming *computers*.

We want *computational effects*: input/output, exceptions, non–determinism, continuations, etc.

λ-calculus and effects

A neat example of the Curry–Howard–Lambek correspondence.

However...

- No interaction with the environment.
- It means... not very useful for programming *computers*.

We want *computational effects*: input/output, exceptions, non–determinism, continuations, etc.

If we just throw effects to it, difficulties arise:

- Evaluation order must be handed carefully.
- Equational reasoning stops working so gracefully.

Moggi: λ_C-calculus

Moggi proposes a calculus based on semantics through monads.

Programs should form a category.

Moggi: λ_C-calculus

Moggi proposes a calculus based on semantics through monads.

Programs should form a category.

Two new constructions on terms:

$M, N ::= \ldots \mid \text{let } x \Leftarrow M \text{ in } N \mid [M]$

Moggi: λ_C-calculus

Moggi proposes a calculus based on semantics through monads.

> *Programs should form a category.*

Two new constructions on terms:

$$M, N ::= \ldots \mid \text{let } x \Leftarrow M \text{ in } N \mid [M]$$

Based on the monad semantics.

$$\boxed{\lambda_C} \xrightarrow{[\![\cdot]\!]} \boxed{\text{CCC} + \text{monad}}$$

Wadler: internalising monads

Moggi: monads are a good pattern to *structure semantics*.

Wadler: internalising monads

Moggi: monads are a good pattern to *structure semantics*.

Could they be used to *structure code*?

Wadler: internalising monads

Moggi: monads are a good pattern to *structure semantics*.

Could they be used to *structure code*? Wadler: Yes!

Wadler: internalising monads

Moggi: monads are a good pattern to *structure semantics*.

Could they be used to *structure code*? Wadler: Yes!

Abstracted as a type-class:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \to m\ a \\
\quad (\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

Effects as monads: examples

data $Exc\ a = Fail \mid Success\ a$
instance $Monad\ Exc$ **where** ...

data $Nondet\ a = Nondet\ [a]$
instance $Monad\ Nondet$ **where** ...

data $\mathrm{State}_s\ a = \mathrm{State}_s\ (s \to (a, s))$
instance $Monad\ \mathrm{State}_s$ **where** ...

Effects as monads: examples

$\textbf{data}\ \mathit{Exc}\ a = \mathit{Fail} \mid \mathit{Success}\ a$

$\textbf{instance}\ \mathit{Monad}\ \mathit{Exc}\ \textbf{where}\ \ldots$

$\textbf{data}\ \mathit{Nondet}\ a = \mathit{Nondet}\ [a]$

$\textbf{instance}\ \mathit{Monad}\ \mathit{Nondet}\ \textbf{where}\ \ldots$

$\textbf{data}\ \mathrm{State}_s\ a = \mathrm{State}_s\ (s \to (a, s))$

$\textbf{instance}\ \mathit{Monad}\ \mathrm{State}_s\ \textbf{where}\ \ldots$

Operations

In general, a monad m is endowed with operations of the form

$$op :: a \to m\ b$$

Computations are written using *return*, $(\gg\!=)$ and these operations.

Operations

In general, a monad m is endowed with operations of the form

$$op :: a \rightarrow m\ b$$

Computations are written using *return*, $(\gg=)$ and these operations.

Equivalently expressed in a Yoneda-style:

$$or :: (m\ a, m\ a) \rightarrow m\ a \qquad \text{vs.} \qquad choose :: () \rightarrow m\ Bool$$

Operations

In general, a monad m is endowed with operations of the form

$$op :: a \to m\ b$$

Computations are written using *return*, $(\gg\!=)$ and these operations.

Equivalently expressed in a Yoneda-style:

$$or :: (m\ a, m\ a) \to m\ a \qquad \text{vs.} \qquad choose :: () \to m\ Bool$$

$$\left(\begin{array}{lcl} or\ (m_1, m_2) & = & choose\ () \gg\!= \lambda b \to \mathbf{if}\ b\ \mathbf{then}\ m_1\ \mathbf{else}\ m_2 \\ choose\ () & = & or\ (return\ True, return\ False) \end{array}\right)$$

Usage example

The constructor State_s is endowed with two *operations*:

$$\begin{aligned} &get :: () \rightarrow \mathsf{State}_s\ s \\ &put :: s \ \ \rightarrow \mathsf{State}_s\ () \end{aligned}$$

Usage example

The constructor State_s is endowed with two *operations*:

$$
\begin{aligned}
&get :: () \to \mathsf{State}_s\ s \\
&put :: s \ \ \to \mathsf{State}_s\ ()
\end{aligned}
$$

A program that generates fresh variables `"x0"`, `"x1"`, ... by using a state with an *Int*:

$$
\begin{aligned}
&\mathit{freshVar} :: \mathsf{State}_{Int}\ \mathit{String} \\
&\mathit{freshVar} = \mathit{get}\ () \gg\!= \lambda s \to \\
&\qquad\qquad\quad \mathit{put}\ (s+1) \gg\!= \lambda() \to \\
&\qquad\qquad\quad \mathit{return}\ (\texttt{"x"} +\!\!+ \mathit{show}\ s)
\end{aligned}
$$

However...

Some effects would benefit of having static information around.

$$(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$$

The combinator $(\gg\!=)$ is too dynamic.

However...

Some effects would benefit of having static information around.

$$(\gg=) :: m\ a \to (a \to m\ b) \to m\ b$$

The combinator $(\gg=)$ is too dynamic.

There are computational effects that cannot be captured by the monad interface.

$$\mathbf{data}\ WR_A\ a = WR_A\ Log\ (Env \to a)$$

However...

Some effects would benefit of having static information around.

$$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

The combinator $(\gg=)$ is too dynamic.

There are computational effects that cannot be captured by the monad interface.

$$\mathbf{data}\ WR_A\ a = WR_A\ Log\ (Env \rightarrow a)$$

These limitations lead to new interfaces.

Arrows

Hughes introduced a new interface for computations.

$$
\begin{array}{l}
\textbf{class } \mathit{Arrow}\ a\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{arr} & :: (x \to y) \to a\ x\ y \\
(\ggg) & :: a\ x\ y \to a\ y\ z \to a\ x\ z \\
\mathit{first} & :: a\ x\ y \to a\ (x, z)\ (y, z)
\end{array}
\end{array}
$$

Arrows

Hughes introduced a new interface for computations.

$$
\begin{aligned}
&\textbf{class}\ \mathit{Arrow}\ a\ \textbf{where}\\
&\quad \mathit{arr} \;\; :: (x \to y) \to a\ x\ y\\
&\quad (\ggg) :: a\ x\ y \to a\ y\ z \to a\ x\ z\\
&\quad \mathit{first} :: a\ x\ y \to a\ (x, z)\ (y, z)
\end{aligned}
$$

This interface provides static information.

- Originally applied to parsers.
- Used in reactive functional programming.
- A monad and a comonad give rise to an arrow.

Applicative functors

Later, McBride and Patterson introduced yet another type-class to represent effects.

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \mathbf{where} \\
\quad \mathit{pure} :: a \to f\ a \\
\quad (\circledast)\ :: f\ (a \to b) \to f\ a \to f\ b
\end{array}
$$

Applicative functors

Later, McBride and Patterson introduced yet another type-class to represent effects.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where} \\
\quad \mathit{pure} :: a \rightarrow f\ a \\
\quad (\circledast)\ :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b
\end{array}
$$

- Popular instances: monoids, zip lists, parsers.
- The constructor WR_A can be instantiated as an *Applicative*.
- $\mathit{Monad}\ m \Rightarrow \mathit{Applicative}\ m$. Not the converse.

An unified framework?

So far, we have seen three different interfaces to represent computational effects.

Applicative *Monad* *Arrow*

An unified framework?

So far, we have seen three different interfaces to represent computational effects.

Applicative *Monad* *Arrow*

It would be nice if it was possible to give an *unified* framework for these interfaces.

Looking again...

The three interfaces have a neuter lifting operation:

$$\mathit{return} :: a \to m\ a$$
$$\mathit{pure} :: a \to f\ a$$
$$\mathit{arr} :: (a \to b) \to (a \leadsto b)$$

And also sequencing of computation:

$$(\gg=) :: m\ a \to (a \to m\ b) \to m\ b$$
$$(\circledast) :: f\ (a \to b) \to f\ a \to f\ b$$
$$(\ggg) :: (a \leadsto b) \to (b \leadsto c) \to (a \leadsto c)$$

Looking again...

The three interfaces have a neuter lifting operation:

$$
\begin{aligned}
&\mathit{return} :: a \to m\ a \\
&\mathit{pure} :: a \to f\ a \\
&\mathit{arr} :: (a \to b) \to (a \leadsto b)
\end{aligned}
$$

And also sequencing of computation:

$$
\begin{aligned}
&(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b \\
&(\circledast) :: f\ (a \to b) \to f\ a \to f\ b \\
&(\ggg) :: (a \leadsto b) \to (b \leadsto c) \to (a \leadsto c)
\end{aligned}
$$

They look a bit like monoids...

Generalised monoids

A monoidal category $(\mathcal{C}, \otimes, I)$ is a category $\mathcal{C}$ together with:

- a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$,
- an object I,
- natural isomorphisms α, λ and ρ subject to coherence.

Generalised monoids

A monoidal category $(\mathcal{C}, \otimes, I)$ is a category $\mathcal{C}$ together with:

- a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$,
- an object I,
- natural isomorphisms α, λ and ρ subject to coherence.

A monoid in $(\mathcal{C}, \otimes, I)$ is a triple (M, m, e) where:

- M is an object,
- $m : M \otimes M \to M$ is a morphism,
- $e : I \to M$ is a morphism.

Monads as monoids

Endofunctors have a substitution monoidal structure:

- $(F \circ G)X = \int^Y FY \times (Y \to GX)$ as tensor.
- The identity functor as unit.

Monads as monoids

Endofunctors have a substitution monoidal structure:

- $(F \circ G)X = \int^{Y} FY \times (Y \to GX)$ as tensor.
- The identity functor as unit.

Its monoids are monads:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
&\quad \mathit{return} :: a \to m\ a \\
&\quad (\ggg\!=) :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}
$$

Monads as monoids

Endofunctors have a substitution monoidal structure:

- $(F \circ G)X = \int^{Y} FY \times (Y \to GX)$ as tensor.
- The identity functor as unit.

Its monoids are monads:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m \textbf{ where} \\
\quad \mathit{return} :: a \to m\ a \\
\quad (\ggg\!=) :: m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

Monads as monoids

Endofunctors have a substitution monoidal structure:

- $(F \circ G)X = \int^{Y} FY \times (Y \to GX)$ as tensor.
- The identity functor as unit.

Its monoids are monads:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m \textbf{ where} \\
\quad \mathit{return} :: \forall a.\ a \to m\ a \\
\quad (\gg\!=)\ :: \forall a, b.\ m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

Monads as monoids

Endofunctors have a substitution monoidal structure:

- $(F \circ G)X = \int^{Y} FY \times (Y \to GX)$ as tensor.
- The identity functor as unit.

Its monoids are monads:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m \textbf{ where} \\
\quad \mathit{return} :: \forall a.\ a \to m\ a \\
\quad (\ggg\!=) \ :: \forall b.\ (\exists\ a.\ (m\ a, a \to m\ b)) \to m\ b
\end{array}
$$

Monads as monoids

Endofunctors have a substitution monoidal structure:

- $(F \circ G)X = \int^{Y} FY \times (Y \to GX)$ as tensor.
- The identity functor as unit.

Its monoids are monads:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: \forall a.\ \mathit{Id}\ a \to m\ a \\
\quad (\gg\!=) \ :: \forall b.\ (m \circ m)\ b \to m\ b
\end{array}
$$

Monads as monoids

Endofunctors have a substitution monoidal structure:

- $(F \circ G)X = \int^{Y} FY \times (Y \to GX)$ as tensor.
- The identity functor as unit.

Its monoids are monads:

$$
\begin{array}{l}
\textbf{class } \mathit{Functor}\ m \Rightarrow \mathit{Monad}\ m \textbf{ where} \\
\quad \mathit{return} :: \mathit{Id} \stackrel{\bullet}{\to} m \\
\quad (\ggg\!=) :: m \circ m \stackrel{\bullet}{\to} m
\end{array}
$$

Applicative functors as monoids

Endofunctors have another monoidal structure:

- $(F \star G)X = \int^{Y} FY \times G(X \to Y)$ as tensor.
- The identity functor as unit.

Applicative functors as monoids

Endofunctors have another monoidal structure:

- $(F \star G)X = \int^{Y} FY \times G(X \to Y)$ as tensor.
- The identity functor as unit.

Its monoids are applicative functors:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where} \\
\quad \mathit{pure} :: a \to f\ a \\
\quad (\circledast)\ :: f\ a \to f\ (a \to b) \to f\ b
\end{array}
$$

Applicative functors as monoids

Endofunctors have another monoidal structure:

- $(F \star G)X = \int^{Y} FY \times G(X \to Y)$ as tensor.
- The identity functor as unit.

Its monoids are applicative functors:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where} \\
&\quad \mathit{pure} :: \forall a.\ a \to f\ a \\
&\quad (\circledast)\ :: \forall a, b.\ f\ a \to f\ (a \to b) \to f\ b
\end{aligned}
$$

Applicative functors as monoids

Endofunctors have another monoidal structure:

- $(F \star G)X = \int^{Y} FY \times G(X \to Y)$ as tensor.
- The identity functor as unit.

Its monoids are applicative functors:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f\ \textbf{where} \\
\quad \mathit{pure} :: \forall a.\ a \to f\ a \\
\quad (\circledast)\ :: \forall b.\ (\exists\ a.\ (f\ a, f\ (a \to b))) \to f\ b
\end{array}
$$

Applicative functors as monoids

Endofunctors have another monoidal structure:

- $(F \star G)X = \int^{Y} FY \times G(X \to Y)$ as tensor.
- The identity functor as unit.

Its monoids are applicative functors:

$$\begin{aligned}
&\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f \textbf{ where}\\
&\quad \mathit{pure} :: \forall a.\ \mathit{Id}\ a \to f\ a\\
&\quad (\circledast)\ :: \forall b.\ (f \star f)\ b \to f\ b
\end{aligned}$$

Applicative functors as monoids

Endofunctors have another monoidal structure:

- $(F \star G)X = \int^{Y} FY \times G(X \to Y)$ as tensor.
- The identity functor as unit.

Its monoids are applicative functors:

$$
\begin{aligned}
&\textbf{class } \mathit{Functor}\ f \Rightarrow \mathit{Applicative}\ f \textbf{ where}\\
&\quad \mathit{pure} :: \mathit{Id} \xrightarrow{\bullet} f\\
&\quad (\circledast) :: f \star f \xrightarrow{\bullet} f
\end{aligned}
$$

Arrows as monoids

Strong profunctors inherit Bénabou's monoidal structure:

- $(P \otimes Q)(X, Y) = \int^{W} P(X, W) \times Q(W, Y)$ as tensor.
- The hom-set as unit.

Arrows as monoids

Strong profunctors inherit Bénabou's monoidal structure:

- $(P \otimes Q)(X, Y) = \int^{W} P(X, W) \times Q(W, Y)$ as tensor.
- The hom-set as unit.

Its monoids are arrows:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Arrow}\ a\ \textbf{where} \\
\quad \mathit{arr} \quad :: (x \to y) \to a\ x\ y \\
\quad (\ggg) :: a\ x\ y \to a\ y\ z \to a\ x\ z \\
\quad \mathit{first} \ \ :: a\ x\ y \to a\ (x, z)\ (y, z)
\end{array}
$$

Arrows as monoids

Strong profunctors inherit Bénabou's monoidal structure:

- $(P \otimes Q)(X, Y) = \int^{W} P(X, W) \times Q(W, Y)$ as tensor.
- The hom-set as unit.

Its monoids are arrows:

$$
\begin{array}{l}
\textbf{class } \mathit{Profunctor}\ p \Rightarrow \mathit{StrongProfunctor}\ p \textbf{ where} \\
\quad \mathit{first} :: p\ x\ y \rightarrow p\ (x, z)\ (y, z) \\
\textbf{class } \mathit{StrongProfunctor}\ a \Rightarrow \mathit{Arrow}\ a \textbf{ where} \\
\quad \mathit{arr} \quad :: (x \rightarrow y) \rightarrow a\ x\ y \\
\quad (\ggg) :: a\ x\ y \rightarrow a\ y\ z \rightarrow a\ x\ z
\end{array}
$$

Arrows as monoids

Strong profunctors inherit Bénabou's monoidal structure:

- $(P \otimes Q)(X, Y) = \int^{W} P(X, W) \times Q(W, Y)$ as tensor.
- The hom-set as unit.

Its monoids are arrows:

$$
\begin{aligned}
&\textbf{class } \mathit{Profunctor}\ p \Rightarrow \mathit{StrongProfunctor}\ p \textbf{ where}\\
&\quad \mathit{first} :: p\ x\ y \to p\ (x, z)\ (y, z)\\
&\textbf{class } \mathit{StrongProfunctor}\ a \Rightarrow \mathit{Arrow}\ a \textbf{ where}\\
&\quad \mathit{arr} \quad :: \forall x, y.\ (x \to y) \to a\ x\ y\\
&\quad (\ggg) :: \forall x, y, z.\ a\ x\ y \to a\ y\ z \to a\ x\ z
\end{aligned}
$$

Arrows as monoids

Strong profunctors inherit Bénabou's monoidal structure:

- $(P \otimes Q)(X, Y) = \int^{W} P(X, W) \times Q(W, Y)$ as tensor.
- The hom-set as unit.

Its monoids are arrows:

$$
\begin{array}{l}
\textbf{class } \mathit{Profunctor}\ p \Rightarrow \mathit{StrongProfunctor}\ p \textbf{ where} \\
\quad \mathit{first} :: p\ x\ y \to p\ (x, z)\ (y, z) \\
\textbf{class } \mathit{StrongProfunctor}\ a \Rightarrow \mathit{Arrow}\ a \textbf{ where} \\
\quad \mathit{arr} \quad :: \forall x, y.\ (x \to y) \to a\ x\ y \\
\quad (\ggg) :: \forall x, z.\ (\exists\ z.\ (a\ x\ y, a\ y\ z)) \to a\ x\ z
\end{array}
$$

Arrows as monoids

Strong profunctors inherit Bénabou's monoidal structure:

- $(P \otimes Q)(X, Y) = \int^{W} P(X, W) \times Q(W, Y)$ as tensor.
- The hom-set as unit.

Its monoids are arrows:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Profunctor}\ p \Rightarrow \mathit{StrongProfunctor}\ p\ \textbf{where} \\
&\quad \mathit{first} :: p\ x\ y \to p\ (x, z)\ (y, z) \\
&\textbf{class}\ \mathit{StrongProfunctor}\ a \Rightarrow \mathit{Arrow}\ a\ \textbf{where} \\
&\quad \mathit{arr} \quad :: \forall x, y.\ \mathit{Hom}\ x\ y \to a\ x\ y \\
&\quad (\ggg) :: \forall x, z.\ (a \otimes a)\ x\ z \to a\ x\ z
\end{aligned}
$$

Arrows as monoids

Strong profunctors inherit Bénabou's monoidal structure:

- $(P \otimes Q)(X, Y) = \int^{W} P(X, W) \times Q(W, Y)$ as tensor.
- The hom-set as unit.

Its monoids are arrows:

$$
\begin{aligned}
&\textbf{class}\ \mathit{Profunctor}\ p \Rightarrow \mathit{StrongProfunctor}\ p\ \textbf{where} \\
&\quad \mathit{first} :: p\ x\ y \to p\ (x, z)\ (y, z) \\
&\textbf{class}\ \mathit{StrongProfunctor}\ a \Rightarrow \mathit{Arrow}\ a\ \textbf{where} \\
&\quad \mathit{arr} \quad :: \mathit{Hom} \stackrel{..}{\to} a \\
&\quad (\ggg) :: a \otimes a \stackrel{..}{\to} a
\end{aligned}
$$

Summing up: computational effects as monoids

$\boxed{\textit{Monad}}$ Monoid in $(\mathrm{End}, \circ, \mathrm{Id})$

Summing up: computational effects as monoids

Monad Monoid in $(\mathrm{End}, \circ, \mathrm{Id})$

Applicative Monoid in $(\mathrm{End}, \star, \mathrm{Id})$

Summing up: computational effects as monoids

Monad	Monoid in $(\mathsf{End}, \circ, \mathsf{Id})$
Applicative	Monoid in $(\mathsf{End}, \star, \mathsf{Id})$
Arrow	Monoid in $(\mathsf{Prof}, \otimes, \mathsf{Hom})$

Summing up: computational effects as monoids

$\boxed{\textit{Monad}}$	Monoid in $(\mathrm{End}, \circ, \mathrm{Id})$
$\boxed{\textit{Applicative}}$	Monoid in $(\mathrm{End}, \star, \mathrm{Id})$
$\boxed{\textit{Arrow}}$	Monoid in $(\mathrm{Prof}, \otimes, \mathrm{Hom})$

Not just a pretty theory...

Summing up: computational effects as monoids

Monad	Monoid in $(\mathrm{End}, \circ, \mathrm{Id})$
Applicative	Monoid in $(\mathrm{End}, \star, \mathrm{Id})$
Arrow	Monoid in $(\mathrm{Prof}, \otimes, \mathrm{Hom})$

Not just a pretty theory...

We can put the unification to work!

Exploiting the unification for fun and profit (1)

Free monoid.

$\boxed{\text{Free monoid}}$

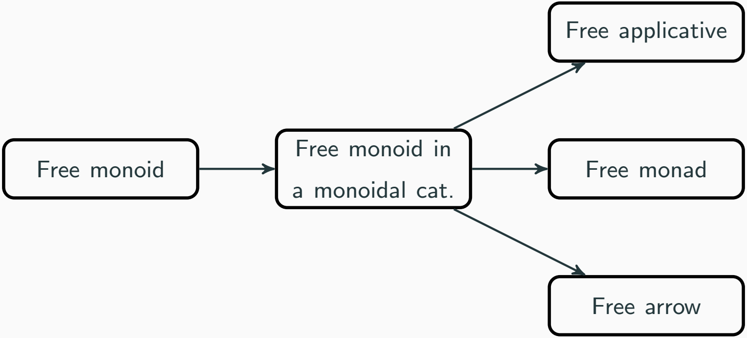
Exploiting the unification for fun and profit (1)

Free monoid.

$$\begin{tikzcd} \boxed{\text{Free monoid}} \arrow[r] & \boxed{\begin{array}{c}\text{Free monoid in}\\ \text{a monoidal cat.}\end{array}} \end{tikzcd}$$

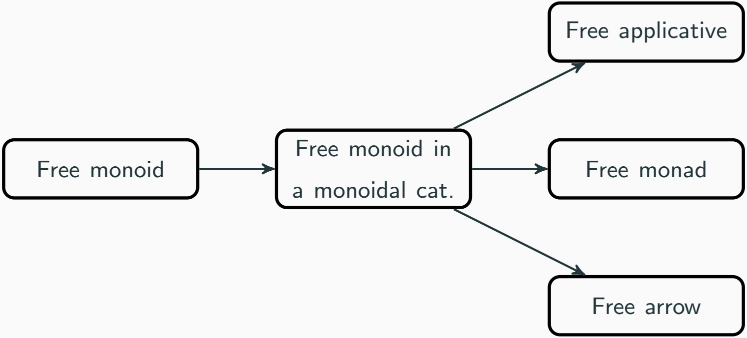
Exploiting the unification for fun and profit (1)

Free monoid.



Exploiting the unification for fun and profit (1)

Free monoid.



Free monoid express abstract programs, useful for DSLs.

Free notions of computation from operations

Given a set of operations

$$\{op_w : I_w \to O_w\}_{w \in W}$$

We can construct a functor:

$$F(X) = \sum_{w \in W} I_w \times X^{O_w}$$

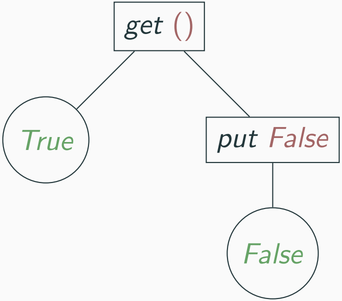
Or a strong profunctor:

$$P(X, Y) = \sum_{w \in W} {I_w}^X \times Y^{O_W \times X}$$

Free monads from operations

$$\{get : () \to Bool, put : Bool \to ()\}$$

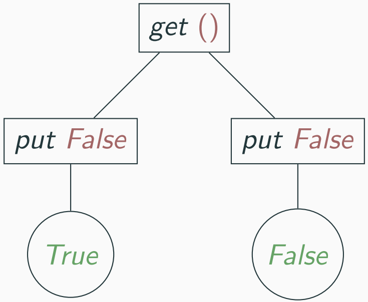
The set of trees.



Free applicative functors from operations

$$\{get : () \to Bool, put : Bool \to ()\}$$

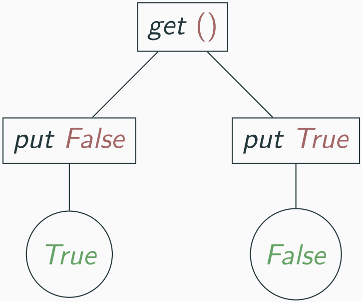
The set of trees with static control and data flow.



Free arrows from operations

$$\{get : () \to Bool, put : Bool \to ()\}$$

The set of trees with static control flow.



Exploiting the unification for fun and profit (2)

Exponential. In the presence of exponentials (closure), it is possible to construct a generalisation of the monoid of endofunctions $X \Rightarrow X$.

Exploiting the unification for fun and profit (2)

Exponential. In the presence of exponentials (closure), it is possible to construct a generalisation of the monoid of endofunctions $X \Rightarrow X$.

Cayley representation. If (M, m, e) is a monoid, then we have a representation

$$\begin{tikzcd} M \arrow[r, hook, "\mathsf{rep}"'] & M \Rightarrow M \arrow[l, bend right, dotted, "\mathsf{abs}"'] \end{tikzcd}$$

Exponential. In the presence of exponentials (closure), it is possible to construct a generalisation of the monoid of endofunctions $X \Rightarrow X$.

Cayley representation. If (M, m, e) is a monoid, then we have a representation

$$\begin{tikzcd} M \arrow[rr, hook, "\mathsf{rep}"'] & & M \Rightarrow M \arrow[ll, dotted, bend right, "\mathsf{abs}"'] \end{tikzcd}$$

It underlies Hughes lists and the codensity monad.

Exploiting the unification for fun and profit (3)

There are monoidal adjunctions (in a broad sense)

$$\begin{tikzcd}
(\mathrm{End}, \star, \mathrm{Id}) \arrow[r, bend left, "_{!}"] \arrow[r, phantom, "\perp"] & (\mathrm{Prof}, \otimes, \mathrm{Hom}) \arrow[l, bend left, "_^{*}"] \arrow[r, bend left, "_^{*}"] \arrow[r, phantom, "\perp"] & (\mathrm{End}, \circ, \mathrm{Id}) \arrow[l, bend left, "_{*}"]
\end{tikzcd}$$

Exploiting the unification for fun and profit (3)

There are monoidal adjunctions (in a broad sense)

$$\begin{tikzcd}
(\mathsf{End}, \star, \mathsf{Id}) \arrow[r, bend left, "-_{!}"] \arrow[r, phantom, "\perp"] & (\mathsf{Prof}, \otimes, \mathsf{Hom}) \arrow[l, bend left, "-^{*}"] \arrow[r, bend left, "-^{*}"] \arrow[r, phantom, "\perp"] & (\mathsf{End}, \circ, \mathsf{Id}) \arrow[l, bend left, "-_{*}"]
\end{tikzcd}$$

Monoidal functors lift to categories of monoids.

Exploiting the unification for fun and profit (3)

There are monoidal adjunctions (in a broad sense)

$$\begin{tikzcd}
(\mathsf{End}, \star, \mathsf{Id}) \arrow[rr, bend left, "-_!"] \arrow[rr, phantom, "\perp"] & & (\mathsf{Prof}, \otimes, \mathsf{Hom}) \arrow[ll, bend left, "-^*"] \arrow[rr, bend left, "-^*"] \arrow[rr, phantom, "\perp"] & & (\mathsf{End}, \circ, \mathsf{Id}) \arrow[ll, bend left, "-_*"]
\end{tikzcd}$$

Monoidal functors lift to categories of monoids.

The relationship between the interfaces is explained by the lifted functors.

Handling non-determinism

Non-deterministic monads extend monads with failure and choice:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where} \\
\quad \mathit{mzero} :: m\ a \\
\quad \mathit{mplus} :: m\ a \to m\ a \to m\ a
\end{array}
$$

Handling non-determinism

Non-deterministic monads extend monads with failure and choice:

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ m\ \textbf{where}\\ &\quad \mathit{mzero} :: m\ a\\ &\quad \mathit{mplus} :: m\ a \to m\ a \to m\ a\end{aligned}$$

An instance provides two extra operations:

- *mzero* represents a failing computation.
- *mplus* expresses a computation that behaves non-deterministically.

Near-rigs

Analogously, *Alternative* and *ArrowPlus* extend applicative functors and arrows.

Alternative *MonadPlus* *ArrowPlus*

Near-rigs

Analogously, *Alternative* and *ArrowPlus* extend applicative functors and arrows.

$\boxed{\textit{Alternative}}$ $\boxed{\textit{MonadPlus}}$ $\boxed{\textit{ArrowPlus}}$

These can be understood as generalised *near-rigs*.

Near-rigs

Analogously, *Alternative* and *ArrowPlus* extend applicative functors and arrows.

$\boxed{\textit{Alternative}}$ $\boxed{\textit{MonadPlus}}$ $\boxed{\textit{ArrowPlus}}$

These can be understood as generalised *near-rigs*.

Free near-rigs can be constructed, and also a represetation. However, the theory of near-rig categories is not well developed.

Near-rig categories

A near-rig category $(\mathcal{C}, \otimes, I, \oplus, J)$ is a category $\mathcal{C}$ together with:

- bifunctors $\otimes, \oplus : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$
- objects I,
- natural isomorphisms

$$\alpha_{A,B,B} : A \otimes (B \otimes B) \cong (A \otimes B) \otimes C \qquad \lambda_A : I \otimes A \cong A \qquad \rho_A : A \otimes I \cong A$$
$$a_{A,B,C} : A \oplus (B \oplus B) \cong (A \oplus B) \oplus C \qquad l_A : J \oplus A \cong A \qquad r_A : A \oplus J \cong A$$

- natural *morphisms*

$$\delta_A : (A \oplus B) \otimes C \to (A \otimes C) \oplus (B \otimes C) \qquad \kappa_A : J \otimes A \to J$$

Near-rig categories

A near-rig category $(\mathcal{C}, \otimes, I, \oplus, J)$ is a category $\mathcal{C}$ together with:

- bifunctors $\otimes, \oplus : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$
- objects I,
- natural isomorphisms

$$\alpha_{A,B,B} : A \otimes (B \otimes B) \cong (A \otimes B) \otimes C \qquad \lambda_A : I \otimes A \cong A \qquad \rho_A : A \otimes I \cong A$$
$$a_{A,B,C} : A \oplus (B \oplus B) \cong (A \oplus B) \oplus C \qquad l_A : J \oplus A \cong A \qquad r_A : A \oplus J \cong A$$

- natural *morphisms*

$$\delta_A : (A \oplus B) \otimes C \to (A \otimes C) \oplus (B \otimes C) \qquad \kappa_A : J \otimes A \to J$$

Coherence with skew structures?

Conclusions and further work

Conclusions.

- Monoidal categories work as an unification of interfaces for computational effects.
- This unification comes with: free structures, representations, connection between the interfaces.
- In principle, it can be extended to richer interfaces.

Conclusions and further work

Conclusions.

- Monoidal categories work as an unification of interfaces for computational effects.
- This unification comes with: free structures, representations, connection between the interfaces.
- In principle, it can be extended to richer interfaces.

Further work?

- Handle non-algebraic operations? Concurrency?
- Develop a calculus for computational effects as monoids.
- Formalize part of it in a proof assistant.