A few comments on interfaces for computational effects

Exequiel Rivas
April 2018

$\pi r^2$ à Fontainebleau
The good old simply typed $\lambda$-calculus

Let’s begin with a basic $\lambda$-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 $\lambda$-calculus

Let’s begin with a basic $\lambda$-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.

$$\tau, \sigma ::= B \mid \tau \rightarrow \sigma \mid \tau \times \sigma \mid \star$$
Let’s begin with a basic $\lambda$-calculus with products and unit.

\[ M, N ::= x \mid MN \mid \lambda x . M \mid (M, N) \mid \pi_i(M) \mid * \]

It has a simple type system.

\[ \tau, \sigma ::= B \mid \tau \rightarrow \sigma \mid \tau \times \sigma \mid * \]

Its semantics can be given by cartesian closed categories (CCC).
A neat example of the Curry–Howard–Lambek correspondence.
A neat example of the Curry–Howard–Lambek correspondence.

However...

- No interaction with the environment.
- It means... not very useful for programming *computers*. 
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.
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 proposes a calculus based on semantics through monads.

*Programs should form a category.*
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 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.
Moggi: monads are a good pattern to \( \textit{structure semantics} \).
Moggi: monads are a good pattern to *structure semantics*.

Could they be used to *structure code*?
Moggi: monads are a good pattern to *structure semantics*.

Could they be used to *structure code*? Wadler: Yes!
Moggi: monads are a good pattern to \textit{structure semantics}.

Could they be used to \textit{structure code}? Wadler: Yes!

Abstracted as a type-class:

\begin{verbatim}
class Monad m where
  return :: a \rightarrow m a
  (>>=) :: m a \rightarrow (a \rightarrow m b) \rightarrow m b
\end{verbatim}
Effects as monads: examples

```haskell
data Exc a = Fail | Success a
instance Monad Exc where ...
```

```haskell
data Nondet a = Nondet [a]
instance Monad Nondet where ...
```

```haskell
data Stateₚ a = Stateₚ (s → (a, s))
instance Monad Stateₚ where ...
```
Effects as monads: examples

```hs

data Exc a = Fail | Success a
instance Monad Exc where ...

data Nondet a = Nondet [a]
instance Monad Nondet where ...

data State s a = State s (s → (a, s))
instance Monad State s where ...
```


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

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

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

Equivalently expressed in a Yoneda-style:

$$or :: (m a, m a) \rightarrow m a \quad \text{vs.} \quad \textit{choose} :: () \rightarrow m \text{Bool}$$
In general, a monad \( m \) is endowed with operations of the form

\[
op :: a \rightarrow m b
\]

Computations are written using \textit{return}, \((\gg=)\) and these operations.

Equivalently expressed in a Yoneda-style:

\[
\text{or} :: (m a, m a) \rightarrow m a \quad \text{vs.} \quad \text{choose} :: () \rightarrow m \text{Bool}
\]

\[
\begin{align*}
\text{or} (m_1, m_2) &= \text{choose} () \gg= \lambda b \rightarrow \text{if } b \text{ then } m_1 \text{ else } m_2 \\
\text{choose} () &= \text{or} (\text{return True, return False})
\end{align*}
\]
The constructor State\_s is endowed with two *operations*:

\[
\text{get} :: () \rightarrow \text{State}_s \ s \\
\text{put} :: s \rightarrow \text{State}_s ()
\]
The constructor $\text{State}_s$ is endowed with two operations:

\[
\text{get} :: () \rightarrow \text{State}_s\ s \\
\text{put} :: s \rightarrow \text{State}_s\ ()
\]

A program that generates fresh variables "$x0", "$x1", ... by using a state with an $\text{Int}$:

\[
\text{freshVar} :: \text{State}_{\text{Int}}\ \text{String} \\
\text{freshVar} = \text{get}\ () \Rightarrow \lambda s \rightarrow \\
\hspace{1cm} \text{put} (s + 1) \Rightarrow \lambda () \rightarrow \\
\hspace{2cm} \text{return} \ ('x' \#\ show\ s)
\]
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.
Some effects would benefit from having static information around.

$$(\Rightarrow) :: m a \to (a \to m b) \to m b$$

The combinator $(\Rightarrow)$ is too dynamic.

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

```haskell
data $WR_A a = WR_A \text{Log} (Env \to a)$
```
However...

Some effects would benefit of having static information around.

\[(\gg) \colon m \ a \to (a \to m \ b) \to m \ b\]

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

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

\[
data \; WR_A \ a = WR_A \ Log \; (Env \to a)\]

These limitations lead to new interfaces.
Hughes introduced a new interface for computations.

```haskell
class Arrow a where
    arr :: (x → y) → a x y
    (≫≫) :: a x y → a y z → a x z
    first :: a x y → a (x, z) (y, z)
```
Hughes introduced a new interface for computations.

```haskell
class Arrow a where
    arr :: (x -> y) -> a x y
    (>>>) :: a x y -> a y z -> a x z
    first :: a x y -> a (x, z) (y, z)
```

This interface provides static information.

- Originally applied to parsers.
- Used in reactive functional programming.
- A monad and a comonad give rise to an arrow.
Later, McBride and Patterson introduced yet another type-class to represent effects.

\[
\text{class } \text{Functor } f \Rightarrow \text{Applicative } f \text{ where }
\]
\[
\text{pure :: } a \rightarrow f \ a
\]
\[
(\star) :: f \ (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b
\]
Later, McBride and Patterson introduced yet another type-class to represent effects.

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (⊛) :: f (a → b) → f a → f b
```

- Popular instances: monoids, zip lists, parsers.
- The constructor $WR_A$ can be instantiated as an `Applicative`.
- `Monad m ⇒ Applicative m`. Not the converse.
An unified framework?

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

Applicative  Monad  Arrow
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:

\[
\begin{align*}
\textit{return} :: a & \rightarrow m a \\
\textit{pure} :: a & \rightarrow f a \\
\textit{arr} :: (a \rightarrow b) & \rightarrow (a \rightsquigarrow b)
\end{align*}
\]

And also sequencing of computation:

\[
\begin{align*}
(\gg) :: m a & \rightarrow (a \rightarrow m b) \rightarrow m b \\
(\otimes) :: f (a \rightarrow b) & \rightarrow f a \rightarrow f b \\
(\ggg) :: (a \rightsquigarrow b) & \rightarrow (b \rightsquigarrow c) \rightarrow (a \rightsquigarrow c)
\end{align*}
\]
Looking again...

The three interfaces have a neuter lifting operation:

\[
\text{return} :: a \rightarrow m\, a \\
\text{pure} :: a \rightarrow f\, a \\
\text{arr} :: (a \rightarrow b) \rightarrow (a \leadsto b)
\]

And also sequencing of computation:

\[
(\gg\gg) :: m\, a \rightarrow (a \rightarrow m\, b) \rightarrow m\, b \\
(\otimes) :: f\, (a \rightarrow b) \rightarrow f\, a \rightarrow f\, b \\
(\gg\gg\gg) :: (a \leadsto b) \rightarrow (b \leadsto c) \rightarrow (a \leadsto c)
\]

They look a bit like 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 \(\alpha, \lambda\) and \(\rho\) subject to coherence.
Generalised monoids

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

- a bifunctor \(\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}\),
- an object \(I\),
- natural isomorphisms \(\alpha, \lambda\) and \(\rho\) subject to coherence.

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

- \(M\) is an object,
- \(m : M \otimes M \rightarrow M\) is a morphism,
- \(e : I \rightarrow M\) 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:

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

```haskell
class Functor m \Rightarrow Monad m where
  return :: a \to m a
  (\_\_\_\rightarrow) :: m a \to (a \to m b) \to m b
```
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:

\[
\text{class } Functor \ m \Rightarrow \ Monad \ m \text{ where}
\]

\[
\text{return} :: \forall a. \ a \to m \ a
\]

\[
(\Rightarrow) :: \forall a, b. \ m \ a \to (a \to m \ b) \to m \ b
\]
Endofunctors have a substitution monoidal structure:

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

Its monoids are monads:

```haskell
class Functor m \Rightarrow Monad m where
    return :: \forall a. a \rightarrow m a
    (\_ >>= \_) :: \forall b. (\exists a. (m a, a \rightarrow m b)) \rightarrow m b
```
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:

```haskell
class Functor m \Rightarrow Monad m where
  return :: \forall a. Id a \to m a
  (\_\_\_\_\_\_\_\_) :: \forall b. (m \circ m) b \to m b
```
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:

```haskell
class Functor m ⇒ Monad m where
    return :: Id → m
    (>) :: m ⧵ m → m
```

Endofunctors have another monoidal structure:

- \((F \star G)X = \int^Y FY \times G(X \to Y)\) as tensor.
- The identity functor as unit.
Endofunctors have another monoidal structure:

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

Its monoids are applicative functors:

```haskell
class Functor f => Applicative f where
  pure :: a \rightarrow f a
  (\otimes) :: f a \rightarrow f (a \rightarrow b) \rightarrow f b
```
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:

```haskell
class Functor f \Rightarrow Applicative f where
  pure :: \forall a. a \to f a
  (⊛) :: \forall a, b. f a \to f (a \to b) \to f b
```
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:

```haskell
class Functor f \Rightarrow Applicative f where
    pure :: \forall a. a \to f a
    (⊛) :: \forall b. (\exists a. (f a, f (a \to b))) \to f b
```
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:

```haskell
class Functor f => Applicative f where
  pure :: \forall a. Id a \to f a
  (⊛) :: \forall b. (f \star f) b \to f b
```
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:

```haskell
class Functor f \Rightarrow Applicative f where

  pure :: Id \rightarrow f

  (\star) :: f \star f \rightarrow f
```
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.
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:

```haskell
class Arrow a where
  arr :: (x \rightarrow y) \rightarrow a x y
  (≫≫) :: a x y \rightarrow a y z \rightarrow a x z
  first :: a x y \rightarrow a (x, z) (y, z)
```
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:

```haskell
class Profunctor p ⇒ StrongProfunctor p where
  first :: p x y → p (x, z) (y, z)

class StrongProfunctor a ⇒ Arrow a where
  arr :: (x → y) → a x y
  (≫≫) :: a x y → a y z → a x z
```
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:

```haskell
class Profunctor p ⇒ StrongProfunctor p where
  first :: p x y → p (x, z) (y, z)

class StrongProfunctor a ⇒ Arrow a where
  arr :: ∀x, y. (x → y) → a x y
  (≫≫) :: ∀x, y, z. a x y → a y z → a x z
```
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:

```haskell
class Profunctor p ⇒ StrongProfunctor p where
    first :: p x y → p (x, z) (y, z)

class StrongProfunctor a ⇒ Arrow a where
    arr :: ∀x, y. (x → y) → a x y
    (≫≫) :: ∀x, z. (∃ z. (a x y, a y z)) → a x z
```
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:

```haskell
class Profunctor p ⇒ StrongProfunctor p where
  first :: p x y → p (x, z) (y, z)

class StrongProfunctor a ⇒ Arrow a where
  arr :: ∀x, y. Hom x y → a x y
  (≫≫) :: ∀x, z. (a ⊗ a) x z → a x z
```
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:

```haskell
class Profunctor p ⇒ StrongProfunctor p where
  first :: p × y → p (x, z) (y, z)

class StrongProfunctor a ⇒ Arrow a where
  arr :: Hom •• a
  (⧺⧺) :: a ⊗ a •• a
```
Summing up: computational effects as monoids

Monad

Monoid in \((\text{End}, \circ, \text{Id})\)
Summing up: computational effects as monoids

- **Monad**: Monoid in \((\text{End}, \circ, \text{Id})\)
- **Applicative**: Monoid in \((\text{End}, \ast, \text{Id})\)

Not just a pretty theory...

We can put the unification to work!
Summing up: computational effects as monoids

- **Monad**: Monoid in \((\text{End}, \circ, \text{Id})\)
- **Applicative**: Monoid in \((\text{End}, \star, \text{Id})\)
- **Arrow**: Monoid in \((\text{Prof}, \otimes, \text{Hom})\)
Summing up: computational effects as monoids

- **Monad**
  - Monoid in \((\text{End}, \circ, \text{Id})\)

- **Applicative**
  - Monoid in \((\text{End}, \ast, \text{Id})\)

- **Arrow**
  - Monoid in \((\text{Prof}, \otimes, \text{Hom})\)

Not just a pretty theory...
Summing up: computational effects as monoids

- **Monad**: Monoid in \((\text{End}, \circ, \text{Id})\)
- **Applicative**: Monoid in \((\text{End}, \star, \text{Id})\)
- **Arrow**: Monoid in \((\text{Prof}, \otimes, \text{Hom})\)

Not just a pretty theory...
We can put the unification to work!
Free monoid.
Free monoid.

Free monoid in a monoidal cat.
Free monoid express abstract programs, useful for DSLs.
Free monoid express abstract programs, useful for DSLs.
Given a set of operations

\[ \{ \text{op}_w : I_w \to O_w \}_w \in W \]

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

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

The set of trees.
Free applicative functors from operations

\{\text{get} : () \to \text{Bool}, \text{put} : \text{Bool} \to ()\}\n
The set of trees with static control and data flow.

```
{get : () \to \text{Bool}, put : \text{Bool} \to ()}
```

```
The set of trees with static control and data flow.
```
Free arrows from operations

\{\text{get} : () \rightarrow \text{Bool}, \text{put} : \text{Bool} \rightarrow ()\}

The set of trees with static control flow.
**Exponential.** In the presence of exponentials (closure), it is possible to construct a generalisation of the monoid of endofunctions $X \Rightarrow X$. 
**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

\[ M \xrightarrow{\text{rep}} M \Rightarrow M \]

\[ M \xleftarrow{\text{abs}} \]

It underlies Hughes lists and the codensity monad.
**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

\[
M \xleftarrow{\text{rep}} \xrightarrow{\text{abs}} M \Rightarrow M
\]

It underlies Hughes lists and the codensity monad.
There are monoidal adjunctions (in a broad sense)

\[(\text{End}, \star, \text{Id}) \quad \perp \quad (\text{Prof}, \otimes, \text{Hom}) \quad \perp \quad (\text{End}, \circ, \text{Id})\]
There are monoidal adjunctions (in a broad sense)

\[(\text{End}, \star, \text{Id}) \dashv (\text{Prof}, \otimes, \text{Hom}) \dashv (\text{End}, \circ, \text{Id})\]

Monoidal functors lift to categories of monoids.
There are monoidal adjunctions (in a broad sense)

$$(\text{End}, \star, \text{Id}) \perp (\text{Prof}, \otimes, \text{Hom}) \perp (\text{End}, \circ, \text{Id})$$

Monoidal functors lift to categories of monoids.

The relationship between the interfaces is explained by the lifted functors.
Non-deterministic monads extend monads with failure and choice:

```haskell
class Monad m ⇒ MonadPlus m where
    mzero :: m a
    mplus :: m a → m a → m a
```
Non-deterministic monads extend monads with failure and choice:

```haskell
class Monad m ⇒ MonadPlus m where
    mzero :: m a
    mplus :: m a → m a → m a
```

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.

These can be understood as generalised *near-rigs*. 
Analogously, *Alternative* and *ArrowPlus* extend applicative functors and arrows.

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

Free near-rigs can be constructed, and also a representation. 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

\[
\begin{align*}
\alpha_{A,B,B} & : A \otimes (B \otimes B) \cong (A \otimes B) \otimes C \quad \lambda_A : I \otimes A \cong A \\
\kappa_A & : J \otimes I \cong A \\
\alpha_{A,B,C} & : A \oplus (B \oplus B) \cong (A \oplus B) \oplus C \\
\lambda_A & : I \oplus A \cong A \\
\kappa_A & : J \otimes A \cong A \\
\lambda_A & : I \otimes A \cong A \\
r_A & : A \oplus J \cong A
\end{align*}
\]

- natural morphisms

\[
\begin{align*}
\delta_A : (A \oplus B) \otimes C & \to (A \otimes C) \oplus (B \otimes C) \\
\kappa_A : J \otimes A & \to J
\end{align*}
\]
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

\[
\begin{align*}
\alpha_{A,B,B} & : A \otimes (B \otimes B) \cong (A \otimes B) \otimes C \\
\lambda_A & : I \otimes A \cong A \\
\rho_A & : A \otimes I \cong A \\
\alpha_{A,B,C} & : A \oplus (B \oplus B) \cong (A \oplus B) \oplus C \\
l_A & : J \oplus A \cong A \\
r_A & : A \oplus J \cong A
\end{align*}
\]

- natural morphisms

\[
\begin{align*}
\delta_A & : (A \oplus B) \otimes C \to (A \otimes C) \oplus (B \otimes C) \\
\kappa_A & : J \otimes A \to J
\end{align*}
\]

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.

Further work?

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