Towards Operadic Programming

a preliminary report

Dmitry Vagner

09 November 2019

String Diagrams

we represent an arrow

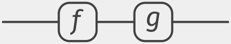
$$a \xrightarrow{f} b$$

by a box

and a composite of arrows

$$a \xrightarrow{f} b \xrightarrow{g} c$$

by a **string diagram**

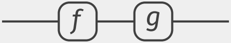


Wiring Diagrams

What about the composition process itself?

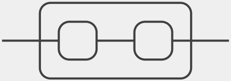
$$\fatsemi : \hom(x, y) \times \hom(y, z) \to \hom(x, z)$$

visually, this maps the string diagram expression



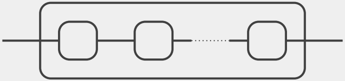
to a single box expression

we visualise this transformation with a **wiring diagram**



Generic Compositions

for any $n : \mathbb{N}$, there is an n-ary composition $\mathtt{chain}_\mathbf{n}$



of type

$$\hom(x_0, x_1) \times \cdots \times \hom(x_{n-1}, x_n) \to \hom(x_0, x_n)$$

special case when $n = 0, 1$:

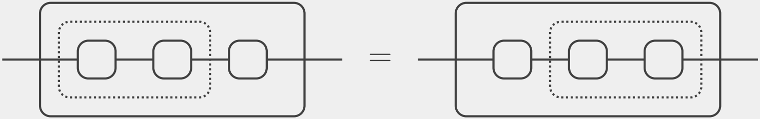
$$: \star \to \hom(x, x)$$
$$* \mapsto \mathbf{1}_x$$

$$: \hom(x, y) \to \hom(x, y)$$
$$f \mapsto f$$

We can encapsulate all composition laws by the condition

ignore intermediary boxes

- **associativity**:



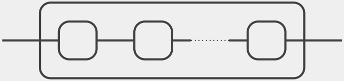
- **unitality**:

Given a type T of objects, define a typed operad $\mathbf{Chain}_T$

- objects are given by abstract **boxes**, i.e. pairs $\langle\langle x, y\rangle\rangle : T^2$

- for each $\mathbf{t} = [t_0, \ldots, t_n]$: `NList` T, precisely one arrow

$$\texttt{chain}_{\mathbf{t}} : \langle\langle t_0, t_1\rangle\rangle, \langle\langle t_1, t_2\rangle\rangle, \ldots, \langle\langle t_{n-1}, t_n\rangle\rangle \to \langle\langle t_0, t_n\rangle\rangle$$



$\texttt{chain}_{\mathbf{n}}$ is the polymorphic version of $\texttt{chain}_{\mathbf{t}}$

We say $\mathbf{Chain}_T$ is **thin**, which has the consequence:

all arrow diagrams commute

Given an operad algebra (think functor, homomorphism, …)

$$A : \mathbf{Chain}_T \to \mathbf{Set}$$

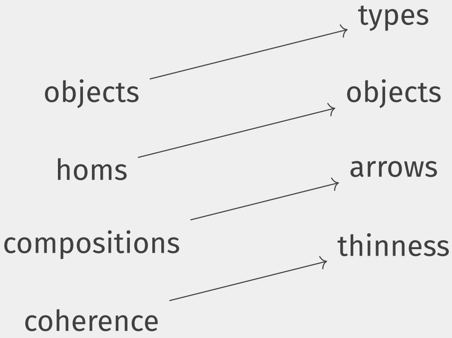
can canonically define a category $\overline{A}$

$$\begin{aligned} \operatorname{ob}\overline{A} &\triangleq T \\ \overline{A}(x,y) &\triangleq A(\langle\langle x,y\rangle\rangle) \\ ;_{x,y,z} &\triangleq A(\mathtt{chain}_{[x,y,z]}) \\ \mathbf{1}_x &\triangleq A(\mathtt{chain}_{[x]}) \end{aligned}$$

The thinness of $\mathbf{Chain}_T$ implies the coherence condition
…which in turn implies associativity and unitality

Level Shift

This invokes a level-shift in perspective:



Given a compositional gadget, rather than asking

> *what (typically, known) categorical structure do instances of this gadget assemble themselves into?*

one can instead ask

> *what are the ways I am allowed stitch instances of these gadgets to form composite gadgets?*

then, if such stitchings form an operad, we can conclude

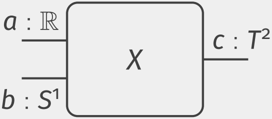
> *the categorical structure for these gadgets is given by algebras over this stitching operad*

from this perspective, we can retcon the following view

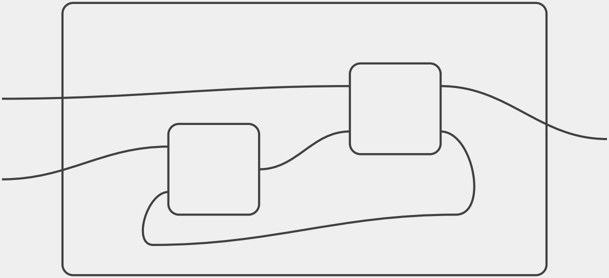
> *a category is the natural structure for housing the compositional theory of univariate maps*

Case Study: Open Dynamical Systems

Given (multi-port) boxes for dynamical systems, e.g.



We want to form compositions like



Wiring Diagrams for Open Systems

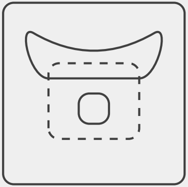
- boxes X are pairs (X^-, X^+) where $X^\pm$ are typed finite sets
- wiring diagrams $X \to Y$ are typed bijections

$$\varphi : X^- + Y^+ \to X^+ + Y^-$$

satisfying **no passing wires**:

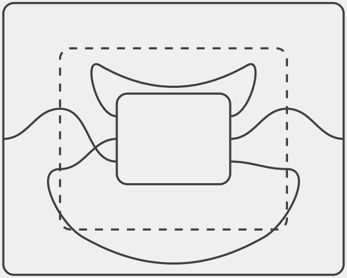
$$\varphi(Y^+) \cap Y^- = \varnothing$$

this avoids closed loops:



Nesting Wiring Diagrams

Nesting is visually simple…just erase intermediary boxes

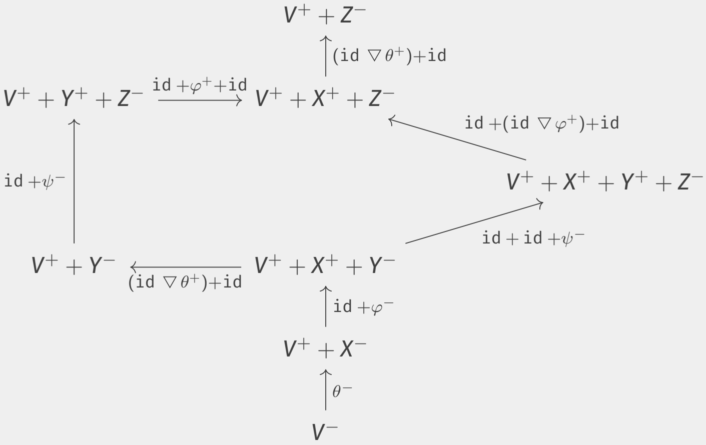


but is slightly trickier to formalise

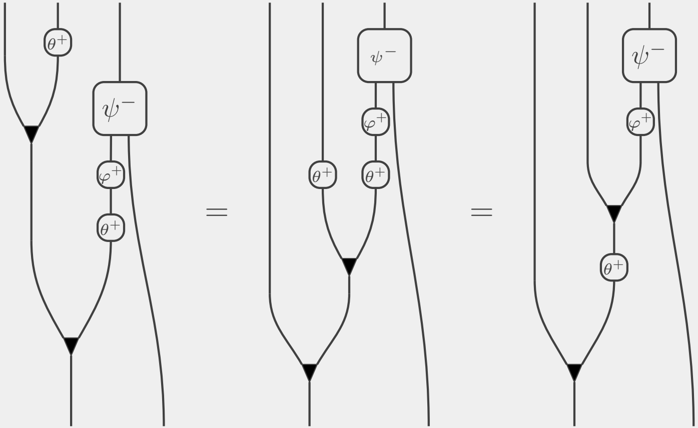
$$\begin{array}{ccc} X^- & \overset{\omega^-}{\dashrightarrow} & X^+ + Z^- \\ {\scriptstyle\varphi^-}\downarrow & & \uparrow{\scriptstyle(\mathtt{id}\,\triangledown\,\varphi^+)+\mathtt{id}} \\ X^+ + Y^- & \xrightarrow[\mathtt{id}+\psi^-]{} & X^+ + Y^+ + Z^- \end{array}$$

$$\begin{array}{ccc} Z^+ & \overset{\omega^+}{\dashrightarrow} & X^+ \\ & {\scriptstyle\psi^+}\searrow \quad \nearrow{\scriptstyle\varphi^+} & \\ & Y^+ & \end{array}$$

Defining Associativity



Proving Associativity



broad goal:

define operads whose algebras are categorical structures

milestone:

Spivak, Schultz, Rupel: String diagrams for traced and compact categories are oriented 1-cobordisms

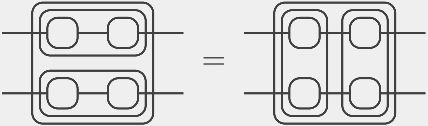
specific goal:

define an operad whose algebras are SMC's

can automatically produce all kinds of relations

e.g. the **interchange law**

$$(f \mathbin{⨟} f') \otimes (g \mathbin{⨟} g') = (f \otimes g) \mathbin{⨟} (f' \otimes g')$$



Flows: Wiring Diagrams for SMCs

- Boxes β are still pairs (X^-, X^+) of typed finite sets
- A **flow** Φ is given by
  - *slots*—a poset $(\mathbf{s}, \preceq)$ of boxes
  - *screen*—a box $(\mathbf{t}_-, \mathbf{t}_+)$
  - *wires*—a typed finite set ω

  and, letting

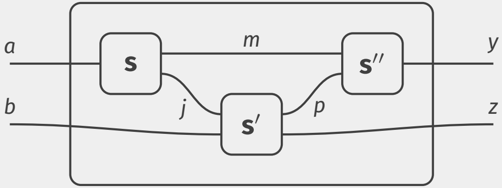
$$\mathbf{s}_\pm \triangleq \sum_{s:\mathbf{s}} s_\pm \quad \text{and} \quad \mathbf{t}_- \prec \mathbf{s}_\pm \prec \mathbf{t}_+$$

  a span of typed bijections

$$\begin{array}{ccc} & \omega & \\ {}^{\Phi_-}\swarrow & & \searrow^{\Phi_+} \\ \mathbf{t}_- + \mathbf{s}_+ & & \mathbf{s}_- + \mathbf{t}_+ \end{array}$$

  satisfying the **progress condition**

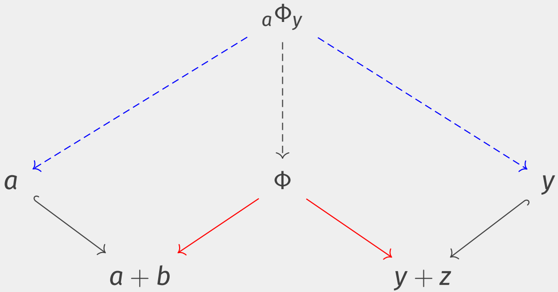
$$\Phi_- \prec \Phi_+$$



- the slot poset $\{\mathbf{s} \prec \mathbf{s}' \prec \mathbf{s}''\}$
- the wires $\{a, b, j, m, p, y, z\}$
- enumerating ports from top to bottom, the wiring is

	a	b	j	m	p	y	z
Φ_-	$\mathbf{t}_{0-}$	$\mathbf{t}_{1-}$	$\mathbf{s}_{0+}$	$\mathbf{s}_{1+}$	$\mathbf{s}'_{0+}$	$\mathbf{s}''_{0+}$	$\mathbf{s}'_{1+}$
Φ_+	$\mathbf{s}_{0-}$	$\mathbf{s}'_{1-}$	$\mathbf{s}'_{0-}$	$\mathbf{s}''_{0-}$	$\mathbf{s}''_{1-}$	$\mathbf{t}_{0+}$	$\mathbf{t}_{1+}$

Given span Φ, can define sub-span ${}_a\Phi_y$ via pullback.



can conceive of the total span Φ as a matrix of subspans

$$\begin{bmatrix} {}_a\Phi_y & {}_a\Phi_z \\ {}_b\Phi_y & {}_b\Phi_z \end{bmatrix}$$

- composition behaves like matrix multiplication
- spans with roof $\varnothing$ behave like zero maps $\mathbf{0} : x \to y$
- sums are biproducts; in particular, given two maps

$$f : S \to T$$
$$g : S \to T$$

we can form a flattened sum (which we'll still denote as $+$)

$$S \xrightarrow{\nabla} S + S \xrightarrow{f+g} T + T \xrightarrow{\Delta} T$$

Composing Flows—formalising Nesting

Given flows defined by the spans

$$\Phi\colon \mathbf{t}_- + \mathbf{s}_+ \to \mathbf{t}_+ + \mathbf{s}_-$$
$$\Psi\colon \mathbf{v}_- + \mathbf{t}_+ \to \mathbf{v}_+ + \mathbf{t}_-$$

We want a composite flow given by a span

$$\omega\colon \mathbf{v}_- + \mathbf{s}_+ \to \mathbf{v}_+ + \mathbf{s}_-$$

do this component-wise, e.g. ${}_{\mathbf{s}_+}\omega_{\mathbf{s}_-}$ is given by the flattened sum

$$\begin{aligned} {}_{\mathbf{s}_+}\Phi_{\mathbf{s}_-} &+ [{}_{\mathbf{s}_+}\Phi_{\mathbf{t}_+}][{}_{\mathbf{t}_+}\Psi_{\mathbf{t}_-}][{}_{\mathbf{t}_-}\Phi_{\mathbf{s}_-}] \\ &+ [{}_{\mathbf{s}_+}\Phi_{\mathbf{t}_+}][{}_{\mathbf{t}_+}\Psi_{\mathbf{t}_-}][{}_{\mathbf{t}_-}\Phi_{\mathbf{t}_+}][{}_{\mathbf{t}_+}\Phi_{\mathbf{t}_-}][{}_{\mathbf{t}_-}\Psi_{\mathbf{t}_+}][{}_{\mathbf{t}_+}\Phi_{\mathbf{s}_-}] \\ &+ \ \cdots \end{aligned}$$

the progress condition forces this to converge!

$$a_0 \prec a_1 \prec a_2 \cdots$$

must terminate in a finite poset (Noetherian condition)

Idris is a **Haskell**-family language with **dependent types**

- programs consist of mathematical functions

```
-- first a type signature
-- and then the program specification
function : domain   -> codomain
function   argument =  value
```

- where types are **first class citizens**

```
-- function returning a type
AsInt : Bool  -> Type
AsInt   True  =  Int
AsInt   False =  String

-- function whose type depends on its argument
getStrOrInt : (isInt : Bool) -> AsInt isInt
getStrOrInt    True           =  7
getStrOrInt    False          =  "seven"
```

Recursively Defined Type Families

```
-- first, recall the inductive definition of naturals
data Nat : Type where
    Z : Nat
    S : Nat -> Nat

-- finite sets
data Fin : Nat -> Type where
    FZ : Fin (S k)
    FS : Fin k -> Fin (S k)

-- fixed length vectors
data Vect : Nat -> Type -> Type where
    Nil  : Vect 0 a
    (::) : a -> Vect k a -> Vect (S k) a

-- heterogeneous vectors
-- these model strictified Cartesian products
data HVect : Vect k Type -> Type where
    Nil  : HVect []
    (::) : t -> HVect ts -> HVect (t::ts)
```

- parametric polymorphism: defined for *all* types

```
-- find the length of a list
length : List a -> Nat
length = foldr (const S) Z
```

- ad-hoc polymorphism: defined for *featureful* types

```
-- multiply a list of monoid elements
mconcat : Monoid m => List m -> m
mconcat = foldr (<>) mempty
```

Haskell/Idris equip types with such features via instantiating them as typeclasses/interfaces

Flows in Idris: initial attempts

```
-- abstract box
record Box where
    constructor BoxIt
    imports : Vect k Type
    exports : Vect j Type

-- filled in box with semantics
fill : Box -> Type
fill box = (HVect $ imports box) -> (HVect $ exports box)

-- flow
record Flow where
    constructor FlowIt
    screen : Box
    slots  : Vect k Box
    wires  : Type
    leftWire : wires -> im screen :+: exs slots
    rghtWire : wires -> ex screen :+: ims slots
```

The Desired Function

we want a function of type

```
animate : (phi : Flow)
        -> HVect (fill <$> slots phi)
        ->       (fill  $ screen phi)
```

Even better: polymorphic filling and animation

```
fill : {V : SMC} -> Box -> Obj V

animate : {V : SMC}
        -> (phi : Flow)
        -> HVect (fill <$> slots phi)
        ->       (fill  $ screen phi)
```

…but getting stuff to compile is hard

```
interval : (i, j : Nat) -> Vect (i + (j + k)) a -> Vect j a
interval i j xs = take j (drop i xs)
```

must hard-code associativity!

Roadmap

- formally prove that flows form an operad
- ascertain (and if so prove) if flow algebras really do correspond to strict symmetric monoidal categories (or some adjacent truth)
- define flows in the cartesian and cocartesian cases
- implement generic compositions in the category `Idris`
- implement generic compositions *polymorphically* for any instance of the symmetric monoidal category interface (importing the lovely CT library developed by StateBox)
- implement the above proofs themselves
- create a front-end GUI for specifying flows, and allow users to "fill in" slots to specify programs