TOWARDS OPERADIC PROGRAMMING

A PRELIMINARY REPORT

Dmitry Vagner

09 November 2019
String Diagrams

we represent an arrow

\[ a \xrightarrow{f} b \]

by a box

\[ a \xrightarrow{\boxed{f}} b \]

and a composite of arrows

\[ a \xrightarrow{f} b \xrightarrow{g} c \]

by a string diagram

\[ \boxed{f} \quad \boxed{g} \]
What about the composition process itself?

\[ \circ : \text{hom}(x, y) \times \text{hom}(y, z) \to \text{hom}(x, z) \]

visually, this maps the string diagram expression

\[ \begin{array}{c}
\text{f} \\
\circ \\
\text{g}
\end{array} \]

to a single box expression

\[ \begin{array}{c}
\text{f} \\
\circ \\
\text{g}
\end{array} \]

we visualise this transformation with a **wiring diagram**
for any $n : \mathbb{N}$, there is an $n$-ary composition chain $\operatorname{chain}_n$

of type

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

special case when $n = 0, 1$:

$$\boxed{} : \ast \to \operatorname{hom}(x, x)$$

$$\ast \mapsto \mathbf{1}_x$$

$$\boxed{} : \operatorname{hom}(x, y) \to \operatorname{hom}(x, y)$$

$$f \mapsto f$$
We can encapsulate all composition laws by the condition

**ignore intermediary boxes**

- **associativity:**

- **unitality:**
Given a type $T$ of objects, define a typed operad $\text{Chain}_T$

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

$$
\begin{array}{c}
x \\
\bigcirc \\
y
\end{array}
$$

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

$$
\text{chain}_t : \langle\langle t_0, t_1 \rangle\rangle, \langle\langle t_1, t_2 \rangle\rangle, \ldots, \langle\langle t_{n-1}, t_n \rangle\rangle \rightarrow \langle\langle t_0, t_n \rangle\rangle
$$

$\text{chain}_n$ is the polymorphic version of $\text{chain}_t$

We say $\text{Chain}_T$ is thin, which has the consequence: all arrow diagrams commute
Given an operad algebra (think functor, homomorphism, …)

\[ A : \text{Chain}_T \rightarrow \text{Set} \]

can canonically define a category \( \overline{A} \)

\[
\begin{align*}
\text{ob} \overline{A} & \triangleq T \\
\overline{A}(x, y) & \triangleq A(\langle\langle x, y \rangle\rangle) \\
\circ_{x,y,z} & \triangleq A(\text{chain}_{[x,y,z]}) \\
\mathbf{1}_x & \triangleq A(\text{chain}_{[x]})
\end{align*}
\]

The thinness of \( \text{Chain}_T \) implies the coherence condition …which in turn implies associativity and unitality
This invokes a level-shift in perspective:

- types
- objects
- objects
- arrows
- compositions
- thinness
- coherence
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
Given (multi-port) boxes for dynamical systems, e.g.

\[
\begin{align*}
    a : \mathbb{R} & \quad X \quad c : T^2 \\
    b : S^1 &
\end{align*}
\]

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 $X \rightarrow Y$ are typed bijections

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

satisfying **no passing wires:**

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

this avoids closed loops:
**Nesting Wiring Diagrams**

**Nesting** is visually simple... just erase intermediary boxes

but is slightly trickier to formalise

\[
X^- \xrightarrow{\omega^-} X^+ + Z^- \\
\varphi^- \\
X^+ + Y^- \xrightarrow{(\text{id} \lor \varphi^+)} (\text{id} \lor \varphi^+) \text{id} \\
X^+ + Y^+ + Z^- \\
\phi^- \\
X^+ \xrightarrow{\omega^+} X^+ \\
\psi^+ \\
Y^+ \xrightarrow{\varphi^+}
\]
Defining Associativity

\[ V^+ + Z^- \]

\[ V^+ + Y^+ + Z^- \xrightarrow{(\text{id} \lor \theta^+)+\text{id}} V^+ + X^+ + Z^- \]

\[ \text{id} + \psi^- \]

\[ V^+ + Y^- \xleftarrow{(\text{id} \lor \theta^+)+\text{id}} V^+ + X^+ + Y^- \xrightarrow{\text{id} + \varphi^-} V^+ + X^- \xrightarrow{\theta^-} V^- \]

\[ \text{id} + (\text{id} \lor \varphi^+)+\text{id} \]

\[ V^+ + X^+ + Y^+ + Z^- \xrightarrow{\text{id} + \text{id} + \psi^-} \]

\[ \text{id} + \varphi^- \]
Proving Associativity

\[ \theta^+ + \psi - \phi^+ + \theta^+ + \theta^+ = \psi - \phi^+ + \theta^+ \]
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

\[(f ; f') \otimes (g ; g') = (f \otimes g) ; (f' \otimes g')\]
Boxes $\beta$ are still pairs $(X^-, X^+)$ of typed finite sets.

A **flow** $\Phi$ is given by:

- **slots**—a poset $(s, \preceq)$ of boxes
- **screen**—a box $(t_-, t_+)$
- **wires**—a typed finite set $\omega$

and, letting

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

a span of typed bijections

![Diagram](attachment:flow_diagram.png)

satisfying the **progress condition**

$$\Phi_- \preceq \Phi_+$$
the slot poset \( \{s \lessdot s' \lessdot s''\} \)

the wires \( \{a, b, j, m, p, y, z\} \)

enumerating ports from top to bottom, the wiring is

<table>
<thead>
<tr>
<th>( \Phi_- )</th>
<th>( \Phi_+ )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Phi_- )</td>
<td>( \Phi_+ )</td>
</tr>
<tr>
<td>( a )</td>
<td>( b )</td>
</tr>
<tr>
<td>( t_0^- )</td>
<td>( t_1^- )</td>
</tr>
</tbody>
</table>
Given span $\Phi$, can define sub-span $a\Phi_y$ via pullback.

can conceive of the total span $\Phi$ as a matrix of subspans

$$
\begin{bmatrix}
a\Phi_y & a\Phi_z \\
b\Phi_y & b\Phi_z
\end{bmatrix}
$$
Span Algebra

- composition behaves like matrix multiplication
- spans with roof $∅$ 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 $+$)

$$S \xrightarrow{\nabla} S + S \xrightarrow{f+g} T + T \xrightarrow{\Delta} T$$
Given flows defined by the spans

\[ \Phi: t^- + s_+ \rightarrow t^- + s_- \]
\[ \Psi: v^- + t_+ \rightarrow v^- + t_- \]

We want a composite flow given by a span

\[ \omega: v^- + s_+ \rightarrow v^- + s_- \]

do this component-wise, e.g. \( s_+ \omega s_- \) is given by the flattened sum

\[
\begin{align*}
    s_+ \Phi s_- &+ [s_+ \Phi t_+][t_+ \Psi t_-][t_- \Phi s_-] \\
    &+ [s_+ \Phi t_+][t_+ \Psi t_-][t_- \Phi t_-][t_+ \Phi t_-][t_- \Psi t_-][t_+ \Phi s_-] \\
    &+ \ldots
\end{align*}
\]

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

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

- where types are **first class citizens**

```haskell
-- 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
{code}
data Nat : Type where
  Z : Nat
  S : Nat -> Nat
{code}

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

-- fixed length vectors
{code}
data Vect : Nat -> Type -> Type where
  Nil : Vect ⊥ a
  (::) : a -> Vect k a -> Vect (S k) a
{code}

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

\[
\text{animate} : (\phi : \text{Flow}) \\
\rightarrow \text{HVect} (\text{fill} \langle\$\rangle \text{ slots} \phi) \\
\rightarrow (\text{fill} \$ \text{ screen} \phi)
\]

Even better: polymorphic filling and animation

\[
\text{fill} : \{V : \text{SMC}\} \rightarrow \text{Box} \rightarrow \text{Obj} V
\]

\[
\text{animate} : \{V : \text{SMC}\} \\
\rightarrow (\phi : \text{Flow}) \\
\rightarrow \text{HVect} (\text{fill} \langle\$\rangle \text{ slots} \phi) \\
\rightarrow (\text{fill} \$ \text{ screen} \phi)
\]

...but getting stuff to compile is hard

\[
\text{interval} : (i, j : \text{Nat}) \rightarrow \text{Vect} (i + (j + k)) \text{a} \rightarrow \text{Vect} j \text{a}
\]

\[
\text{interval} \ i \ j \ \text{xs} = \text{take} \ j \ \text{(drop} \ i \ \text{xs)}
\]

must hard-code associativity!
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