Musa Al-hassy https://github.com/alhassy/CatsCheatSheet September 29, 2018

Reference Sheet for Elementary Category Theory

Categories

A **category** $\mathcal{C}$ consists of a collection of "objects" `Obj` $\mathcal{C}$, a collection of "morphisms" `Mor` $\mathcal{C}$, an operation `Id` associating a morphism $\texttt{Id}_a$ `:` `a` $\to$ `a` to each object `a`, a parallel pair of functions `src, tgt :` `Mor` $\mathcal{C} \to$ `Obj` $\mathcal{C}$, and a "composition" operation $\_;\_ : \forall\{A\ B\ C : Obj\} \to (A \to B) \to (B \to C) \to (A \to C)$ where for objects `X` and `Y` we define the *type* `X` $\to$ `Y` as follows

$$f : X \to Y \quad \equiv \quad \mathsf{src}\ f = X \ \wedge\ \mathsf{tgt}\ f = Y \qquad \text{type-Definition}$$

Moreover composition is required to be associative with `Id` as identity.
Instead of `src` and `tgt` we can instead assume primitive a ternary relation $\_:\_\to\_$ and regain the operations precisely when the relation is functional in its last two arguments:

$$f : A \to B \ \wedge\ f : A' \to B' \implies A = A' \ \wedge\ B = B' \qquad \text{type-Unique}$$

When this condition is dropped, we obtain a *pre-category*; e.g., the familiar *Sets* is a pre-category that is usually treated as a category by making morphisms contain the information about their source and target: `(A, f, B) : A` $\to$ `B` rather than just `f`.
This is sometimes easier to give, then src and tgt! C.f. Alg(F).

Here's an equivalence-preserving property that is useful in algebraic calculations,

$$f : A \to B \quad \wedge \quad g : B \to A \quad \equiv \quad f;g : A \to A \quad \wedge \quad g;f : B \to B \qquad \text{Composition}$$

Example Categories.

- Each digraph determines a category: The objects are the nodes and the paths are the morphisms typed with their starting and ending node. Composition is catenation of paths and identity is the empty path.
- Each preorder determines a category: The objects are the elements and there is a morphism `a` $\to$ `b` named, say, `(a, b),` precisely when $a \leq b$.

Even when morphisms are functions, the objects need not be sets: Sometimes the objects are *operations* –with an appropriate definition of typing for the functions. The categories of F-algebras are an example of this.

Functors

A **functor** $F : \mathcal{A} \to \mathcal{B}$ is a pair of mappings, denoted by one name, from the objects, and morphisms, of $\mathcal{A}$ to those of $\mathcal{B}$ such that it respects the categorical structure:

$$F\,f : F\,A \to_{\mathcal{B}} F\,B \quad \Leftarrow \quad f : A \to_{\mathcal{A}} B \qquad \text{functor-Type}$$

$$F\,\mathsf{Id}_A \ = \ \mathsf{Id}_{F\,A} \qquad \text{Functor}$$

$$F\,(f;g) \ = \ F\,f;F\,g \qquad \text{Functor}$$

The two axioms are equivalent to the single statement that *functors distribute over finite compositions, with* Id *being the empty composition*

$$F(f;\cdots;g) = F\,f;\cdots;F\,g$$

Use of Functors.

- In the definition of a category, "objects" are "just things" for which no internal structure is observable by categorical means –composition, identities, morphisms, typing.
  Functors form the tool to deal with "structured" objects
  Indeed in $\mathcal{S}et$ the aspect of a structure is that it has "constituents", and that it is possible to apply a function to all the individual constituents; this is done by $F\,f : F\,A \to F\,B$.
- For example, let $\mathbb{I}\,A = A \times A$ and $\mathbb{I}\,f = (x, y) \mapsto (f\,x, f\,y)$. So $\mathbb{I}$ is or represents the structure of pairs; $\mathbb{I}\,A$ is the set of pairs of A, and $\mathbb{I}\,f$ is the function that applies f to each constituent of a pair.
  - A *binary operation on A* is then just a function $\mathbb{I}\,A \to A$; in the same sense we obtain *F-ary operations.*
- Also, *Seq* is or represents the structure of sequences; *Seq A* is the structure of sequences over A, and *Seq f* is the function that applies f to each constituent of a sequence.
- Even though $F\,A$ is still just an object, a thing with no observable internal structure, the functor properties enable to exploit the "structure" of $F\,A$ by allowing us to "apply" a f to each "constituent" by using $F\,f$.

Category $\mathcal{A}lg(F)$

- For a functor $F : \mathcal{A} \to \mathcal{D}$, this category has *F-algebras*, F-ary operations in $\mathcal{D}$ as, objects – i.e., objects are $\mathcal{D}$-arrows $F\,A \to A$ – and F-homomorphisms as morphisms, and it inherits composition and identities from $\mathcal{D}$.

  $$f : \oplus \to_F \otimes \quad \equiv \quad \oplus;f = F\,f;\otimes \qquad \text{defn-Homomorphism}$$

  $$\mathsf{Id} : \oplus \to_F \oplus \qquad \text{id-Homomorphism}$$

  $$f;g : \oplus \to_F \odot \qquad \Leftarrow \qquad f : \oplus \to_F \otimes \ \wedge\ g : \otimes \to_F \odot \qquad \text{comp-Homomorhism}$$

  Note that category axiom (**??**) is not fulfilled since a function can be a homomorphism between several distinct operations. However, we pretend it is a category in the way discussed earlier, and so the carrier of an algebra is fully determined by the operation itself, so that the operation itself can be considered the algebra.

  comp-Homomorhism renders a semantic property as a syntactic condition!

- A **contravariant functor** $\mathcal{C} \to \mathcal{D}$ is just a functor $\mathcal{C}^{op} \to \mathcal{D}^{op}$.
- A **bifunctor** from $\mathcal{C}$ to $\mathcal{D}$ is just a functor $\mathcal{C}^{\,2} \to \mathcal{D}$.

Naturality

A natural transformation is nothing but a structure preserving map between functors. "Structure preservation" makes sense, here, since we've seen already that a functor is, or represents, a structure that objects might have.

As discussed before for the case $F : \mathcal{C} \to \mathcal{S}et$, each $F\ A$ denotes a structured set and F denotes the structure itself.

Example Structures: $\mathbb{I}$ is the structure of pairs, *Seq* is the structure of sequences, *Seq Seq* the structure of sequences of sequences, $\mathbb{I}\ Seq$ the structure of pairs of sequences –which is naturally isomorphic to $Seq\ \mathbb{I}$ the structure of sequences of pairs!–, and so on.

A "transformation" from structure F to structure G is a family of functions $\eta : \forall\{A\} \to F\ A \to G\ A$; and it is "natural" if each η_A doesn't affect the *constituents* of the structured elements in $F\ A$ but only reshapes the structure of the elements, from an F-structure to a G-structure.

Reshaping the structure by η commutes with subjecting the constituents to an arbitrary morphism.

$$\eta : F \dot{\to} G \quad \equiv \quad \forall f \bullet F\,f \fatsemi \eta_{\mathsf{tgt}\,f} \;=\; \eta_{\mathsf{src}\,f} \fatsemi G\,f \qquad \text{ntrf-Def}$$

This is 'naturally' remembered: Morphism $\eta_{\mathsf{tgt}\,f}$ has type $F(\mathsf{tgt}\,f) \to G(\mathsf{tgt}\,f)$ and therefore appears at the target side of an occurrence of f; similarly $\eta_{\mathsf{src}\,f}$ occurs at the source side of an f. *Moreover* since η is a transformation *from* F to G, functor F occurs at the source side of an η and functor G at the target side.

- ◇ One also says η_a *is natural in* its parameter a.
- ◇ If we take $G = \mathsf{Id}$, then natural transformations $F \to \mathsf{Id}$ are precisely F-homomorphisms.
- ◇ Indeed, a natural transformation is a sort-of homomorphism in that the image of a morphism after reshaping is the same as the reshaping of the image.

Example natural transformations

- ◇ *rev : Seq $\dot{\to}$ Seq : $[a_1, \ldots, a_n] \mapsto [a_n, \ldots, a_1]$* reverses its argument thereby reshaping a sequence structure into a sequence structure without affecting the constituents.
- ◇ *inits : Seq $\dot{\to}$ Seq Seq : $[a_1, \ldots, a_n] \mapsto [[], [a_1], \cdots, [a_1, \ldots, a_n]]$* yields all initial parts of its argument thereby reshaping a sequence structure into a sequence of sequences structure, not affecting the constituents of its argument.

$$J\eta : JF \dot{\to} JG \quad \Leftarrow \quad \eta : F \dot{\to} G \quad \text{where } (J\eta)_A \coloneqq J(\eta_A) \qquad \text{ntr-Ftr}$$

$$\eta K : FK \dot{\to} GK \quad \Leftarrow \quad \eta : F \dot{\to} G \quad \text{where } (\eta K)_A \coloneqq \eta_{(K\,A)} \qquad \text{ntr-Poly}$$

$$\mathsf{Id}_F : F \dot{\to} F \quad \text{where } (\mathsf{Id}_F)_A \coloneqq \mathsf{Id}_{(F\,A)} \qquad \text{ntrf-Id}$$

$$\epsilon \fatsemi \eta : F \dot{\to} H \quad \Leftarrow \quad \epsilon : F \dot{\to} G \quad \wedge \quad \eta : G \dot{\to} H \quad \text{where } (\epsilon \fatsemi \eta)_A = \epsilon_A \fatsemi \eta_A \qquad \text{ntrf-Compose}$$

Category $\mathcal{F}unc(\mathcal{C}, \mathcal{D})$ consists of functors $\mathcal{C} \to \mathcal{D}$ as objects and natrual transformations between them as objects. The identity transformation is indeed an identity for transformation composition, which is associative.

Heuristic To prove $\phi = \phi_1 \fatsemi \cdots \fatsemi \phi_n : F \dot{\to} G$ is a natural transformation, it suffices to show that each ϕ is a natural transformation.

- ◇ Theorem (ntrf-Compose) renders proofs of semantic properties to be trivial type checking!
- ◇ E.g., It's trivial to prove *tails = rev ; inits ; Seq rev* is a natural transformation by type checking, but to prove the naturality equation by using the naturality equations of *rev* and *inits* –no definitions required– necessitates more writing, and worse: Actual thought!

Adjunctions

An adjunction is a particular one-one correspondence between different kinds of morphisms in different categories.

An **adjunction** consists of two functors $L : \mathcal{A} \to \mathcal{B}$ and $R : \mathcal{B} \to \mathcal{A}$, as well as two (not necessarily natural!) transformations $\eta : \mathsf{Id} \to RL$ and $\epsilon : LR \to \mathsf{Id}$ such that

Provided $f : A \to_{\mathcal{A}} RB$ and $g : LA \to_{\mathcal{B}} B$

$$f = \eta_A \fatsemi Rg \qquad \equiv \qquad Lf \fatsemi \epsilon_B = g \qquad \text{Adjunction}$$

Reading right-to-left: In the equation $Lf \fatsemi \epsilon_B = g$ there is a unique solution to the unknown f. Dually for the other direction.

That is, *each L-algebra g is uniquely determined –as an L-map followed by an ϵ-reduce– by its restriction to the adjunction's unit η.*

A famous example is "Free ⊣ Forgetful", e.g. to *define* the list datatype, for which the above becomes: Homomorphisms on lists are uniquely determined, as a map followed by a reduce, by their restriction to the singleton sequences.

We may call f the restriction, or lowering, of g to the "unital case" and write $f = \lfloor g \rfloor = \eta_A \fatsemi Rg$. Also, we may call g the extension, or raising, of f to an L-homomorphism and write $g = \lceil f \rceil = Lf \fatsemi \epsilon_B$. The above equivalence now reads:

$$f = \lfloor g \rfloor \qquad \equiv \qquad \lceil f \rceil = g \qquad \text{Adjunction-Inverse}$$

$$\lfloor g \rfloor_{A,B} = \eta_A \fatsemi Rg \;:\; A \to_{\mathcal{A}} RB \text{ where } g : LA \to_{\mathcal{B}} B \qquad \text{lad-Type}$$

$$\lceil f \rceil_{A,B} = Lf \fatsemi \epsilon_B \;:\; LA \to_{\mathcal{B}} B \text{ where } f : A \to_{\mathcal{A}} RB \qquad \text{rad-Type}$$

Note that $\lceil$ is like 'r' and the argument to $\lceil\rceil$ must involve the R-ight adjoint in its type; **L**ad takes morphisms involving the **L**eft adjoint ;)

This equivalence expresses that 'lad' $\lfloor\rfloor$, from *l*eft *ad*jungate, and 'rad' $\lceil\rceil$, from *r*ight *ad*jungate, are each other's inverses and constitute a correspondence between certain morphisms. *Being a bijective pair, lad and rad are injective, surjective, and undo one another.*

We may think of $\mathcal{B}$ as having all complicated problems so we abstract away some difficulties by *r*aising up to a cleaner, simpler, domain via rad $\lceil\rceil$; we then solve our problem there, then go back *down* to the more complicated concrete issue via $\lfloor\rfloor$, lad.
(E.g., $\mathcal{B}$ is the category of monoids, and $\mathcal{A}$ is the category of sets; L is the list functor.)

The η and ϵ determine each other and they are *natural* transformations. ntrf-Adj

"zig-zag laws" The unit has a post-inverse while the counit has a pre-inverse:

$$\mathsf{Id} = \eta \fatsemi R\epsilon \qquad \text{unit-Inverse}$$

$$\mathsf{Id} = L\eta \fatsemi \epsilon \qquad \text{Inverse-counit}$$

The unit and counit can be regained from the adjunction inverses,

$$\eta = \lfloor \mathsf{Id} \rfloor \qquad \text{unit-Def}$$

$$\epsilon = \lceil \mathsf{Id} \rceil \qquad \text{counit-Def}$$

Lad and rad themselves are solutions to the problems of interest, (Adjunction).

$$L\lfloor g \rfloor ; \epsilon = g \qquad \text{lad-Self}$$

$$\eta ; R\lceil f \rceil = f \qquad \text{rad-Self}$$

The following laws assert a kind of monoic-ness for ϵ and a kind of epic-ness for η. Pragmatically they allow us to prove an equality by shifting to a possibly easier equality obligation.

$$\eta ; Rg = \eta ; Rg' \quad \equiv \quad g = g' \qquad \text{lad-Unique}$$

$$Lf ; \epsilon = Lf' ; \epsilon \quad \equiv \quad f = f' \qquad \text{rad-Unique}$$

Lad and rad are natural transformations in the category $\mathcal{F}unc(\mathcal{A}^{op} \times \mathcal{B}, \mathcal{S}et)$ realising $(LX \to Y) \cong (X \to GY)$ where X, Y are the first and second projection functors and $(- \to -) : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{S}et$ is the hom-functor such that $(f \to g)h = f ; h ; g$.
By extensionality in $\mathcal{S}et$, their naturality amounts to the laws:

$$\lfloor Lx ; g ; y \rfloor \ = \ x ; \lfloor g \rfloor ; Ry \qquad \text{lad-Fusion}$$

$$\lceil x ; f ; Ry \rceil \ = \ Lx ; \lceil f \rceil ; y \qquad \text{rad-Fusion}$$

Also,

- ◇ Left adjoints preserve colimits such as initial objects and sums.
- ◇ Right adjoints preserve limits such as terminal objects and products.

Skolemisation

If a property P holds for precisely one class of isomorphic objects, and for any two objects in the same class there is precisely one isomorphism from one to the other, then we say that *the P-object is unique up to unique isomorphism.* For example, in $\mathcal{S}et$ the one-point set is unique up to a unique isomorphism, but the two-point set is not.

For example, an object A is "initial" iff $\forall B \bullet \exists_1 f \bullet f : A \to B$, and such objects are unique up to unique isomorphism –prove it! The formulation of the definition is clear but it's not very well suited for *algebraic manipulation.*

A convenient formulation is obtained by 'skolemisation': An assertion of the form

$$\forall x \bullet \exists_1 y \bullet R\, x\, y$$

is equivalent to: There's a function $\mathcal{F}$ such that

$$\forall x, y \bullet R\, x\, y \ \equiv \ y = \mathcal{F}\, x$$

In the former formulation it is the existential quantification "$\exists y$" inside the scope of a universal one that hinders effective calculation. In the latter formulation the existence claim is brought to a more global level: A reasoning need no longer be interrupted by the declaration and naming of the existence of a unique y that depends on x; it can be denoted just $\mathcal{F}\, x$. As usual, the final universal quantification can be omitted, thus simplifying the formulation once more.

In view of the important role of the various y's, these y's deserve a particular notation that triggers the reader of their particular properties. We employ bracket notation such as $(\!|x|\!)$ for such $\mathcal{F}\, x$: An advantage of the bracket notation is that no extra parentheses are needed for composite arguments x, which we expect to occur often.

The formula *characterising* $\mathcal{F}$ may be called '$\mathcal{F}$-Char' and it immediately give us some results by truthifying each side, namely 'Self' and 'Id'. A bit more on the naming:

Type	Possibly non-syntactic constraint on notation being well-formed
Self	It, itself, is a solution
Id	How Id can be expressed using it
Uniq	It's problem has a unique solution
Fusion	How it behaves wrt composition
Composition	How two instances, in full subcategories, compose

Note that the last 3 indicate how the concept interacts with the categorical structure: $=, ;, \mathsf{Id}$. Also note that Self says there's at least one solution and Uniq says there is at most one solution, so together they are equivalent to $\mathcal{F}$-Char –however those two proofs are usually not easier nor more elegant than a proof of $\mathcal{F}$-Char directly.

Initiality

An object *0 is initial* if there's a mapping $(\!| - |\!)$, from objects to morphisms, such that initial-Char holds; from which we obtain a host of useful corollaries. Alternative notations for $(\!|B|\!)$ are $¡_B$, or $(\!|0 \to B|\!)$ to make the dependency on 0 explicit.

$$f : 0 \to B \quad \equiv \quad f = (\!|B|\!) \qquad \text{initial-Char}$$

$$(\!|B|\!) : 0 \to B \qquad \text{initial-Self}$$

$$id_0 = (\!|0|\!) \qquad \text{initial-Id}$$

$$f, g : 0 \to B \quad \Rightarrow \quad f = g \qquad \text{initial-Uniq}$$

$$f : B \to C \quad \Rightarrow \quad (\!|B|\!) ; f = (\!|C|\!) \qquad \text{initial-Fusion}$$

Provided objects B, C are both in $\mathcal{A}$ and $\mathcal{B}$, which are full subcategories of some category $\mathcal{C}$:
$(\!|A \to B|\!)_{\mathcal{A}} ; (\!|B \to C|\!)_{\mathcal{B}} = (\!|A \to C|\!)_{\mathcal{A}}$ initial-Compose

Provided $\mathcal{D}$ is built on top of $\mathcal{C}$: $\mathcal{D}$-objects are composite entities in $\mathcal{C}$
B is an object in $\mathcal{D}$ $\quad \Rightarrow \quad$ $(\!|B|\!)$ is a morphism in $\mathcal{C}$ initial-Type

These laws become much more interesting when the category is built upon another one and the typing is expressed as one or more equations in the underlying category. In particular the importance of fusion laws cannot be over-emphasised; it is proven by a strengthening step of the form $(\!|B|\!) ; f : 0 \to C \quad \Leftarrow \quad (\!|B|\!) : 0 \to B \ \wedge \ f : B \to C.$

For example, it can be seen that the datatype of sequences is 'the' initial object in a suitable category, and the mediator $(\!| - |\!)$ captures "definitions by induction on the structure"! Hence induction arguments can be replaced by initiality arguments! Woah!

Proving Initiality One may prove that an object 0 is initial by providing a definition for $(\!| - |\!)$ and establishing initial-Char. Almost every such proof has the following format, or a circular implication thereof: For arbitrary f and B,

$$
\begin{array}{ll}
\equiv & f : A \to B \\
\equiv & \vdots \\
& f = \text{“an expression not involving } f\text{”} \\
\equiv & \quad \{\ define\ (\!|B|\!) \text{ to be the rhs of the previous equation}\ \} \\
& f = (\!|B|\!)
\end{array}
$$

Colimits

Each colimit is a certain initial object, and each initial object is a certain colimit.

- ◇ A *diagram in* $\mathcal{C}$ is a functor $D : \mathcal{D} \to \mathcal{C}$.
- ◇ Define the constant functor $\underline{C}\, x = C$ for objects x and $\underline{C}\, f = \mathsf{Id}_C$ for morphisms f. For functions $g : A \to B$, we define the natural transformation $\underline{g} = x \mapsto g : \underline{A} \dot{\to} \underline{B}$.
- ◇ The category $\bigvee D$, built upon $\mathcal{C}$, has objects $\gamma : D \dot{\to} \underline{C}$ called “co-cones”, for some object $C =: \mathsf{tgt}\,\gamma$, and a morphism from γ to δ is a $\mathcal{C}$-morphism x such that $\gamma \fatsemi \underline{x} = \delta$.
- ◇ A *colimit for* D is an initial object in $\bigvee D$; which may or may not exist.

Writing $\gamma\backslash -$ for $(\!|-|\!)$ and working out the definition of co-cone in terms of equations in $\mathcal{C}$, we obtain: $\gamma : Obj(\bigvee D)$ *is a colimit for* D if there is a mapping $\gamma\backslash -$ such that $\backslash$*-Type* and $\backslash$*-Char* hold.

$$\delta \text{ cocone for } D \qquad \Rightarrow \qquad \gamma\backslash\delta : \mathsf{tgt}\,\gamma \to \mathsf{tgt}\,\delta \qquad \backslash\text{-Type}$$

Well-formedness convention: In each law the variables are quantified in such a way that the premise of \-Type is met. The notation $\cdots\backslash\delta$ is only senseful if δ is a co-cone for D, like in arithmetic where the notation m/n is only sensful if n differs from 0.

$$\gamma \fatsemi \underline{x} = \delta \qquad \equiv \qquad x = \gamma\backslash\delta \qquad \backslash\text{-Char}$$

Notice that for given $x : C \to C'$ the equation $\gamma\backslash\delta = x$ defines δ, since by \-Char that one equation equivales the family of equations $\delta_A = \gamma_A \fatsemi x$. This allows us to define a natural transformation –or ‘eithers’ in the case of sums– using a single function *having* the type of the mediating arrow.

$$\gamma; \underline{\gamma\backslash\delta} = \delta \qquad \backslash\text{-Self}$$

$$\gamma\backslash\gamma = \mathsf{Id} \qquad \backslash\text{-Id}$$

$$\gamma\backslash\delta \fatsemi x = \gamma\backslash(\delta \fatsemi \underline{x}) \qquad \backslash\text{-Fusion}$$

$$\gamma \fatsemi \underline{x} = \gamma \fatsemi \underline{y} \qquad \Rightarrow \qquad x = y \qquad \backslash\text{-Unique}$$

This expresses that colimits γ have an epic-like property:
The component morphisms γ_A are *jointly epic*.

The following law confirms the choice of notation once more.

$$\gamma\backslash\delta \fatsemi \delta\backslash\epsilon = \gamma\backslash\epsilon \qquad \backslash\text{-Compose}$$

The next law tells us that functors distribute over the \-notation provided the implicit well-formedness condition that $F\gamma$ is a colimit holds –clearly this condition is valid when F preserves colimits.

$$F(\gamma\backslash\delta) = F\gamma\backslash F\delta \qquad \backslash\text{-Functor-Dist}$$

$$\gamma F\backslash\delta F = \gamma\backslash\delta \qquad \backslash\text{-Pre-Functor-Elim}$$

Limits

Dually, the category $\bigwedge D$ has objects being “cones” $\gamma : \underline{C} \dot{\to} D$ where $C =: \mathsf{src}\,\gamma$ is a $\mathcal{C}$-object, and a morphism to γ *from* δ is a $\mathcal{C}$-morphism x such that $\underline{x} \fatsemi \gamma = \delta$. In terms of $\mathcal{C}$, $\gamma : Obj(\bigwedge D)$ *is a limit for* D if there is a mapping $\gamma/-$ such that the following /-Type and /-Char hold, from which we obtain a host of corollaries. As usual, there is the implicit well-formedness condition. /-Unique expresses that limits γ have an monic-like property: The component morphisms γ_A are *jointly monic*.

$$\delta \text{ cone for } D \qquad \Rightarrow \qquad \delta/\gamma : \mathsf{src}\,\delta \to \mathsf{src}\,\gamma \qquad /\text{-Type}$$

$$\underline{x} \fatsemi \gamma = \delta \qquad \equiv \qquad x = \delta/\gamma \qquad /\text{-Char}$$

$$\underline{\delta/\gamma} \fatsemi \gamma = \delta \qquad /\text{-Self}$$

$$\gamma/\gamma = \mathsf{Id} \qquad /\text{-Id}$$

$$x \fatsemi \delta/\gamma = (\underline{x} \fatsemi \delta)/\gamma \qquad /\text{-Fusion}$$

$$\underline{x} \fatsemi \gamma = \underline{y} \fatsemi \gamma \qquad \Rightarrow \qquad x = y \qquad /\text{-Unique}$$

$$F(\delta/\gamma) = F\delta/F\gamma \qquad /\text{-Functor-Dist}$$

$$\delta F/\gamma F = \delta/\gamma \qquad /\text{-Pre-Functor-Elim}$$

Coequaliser

Take D and $\mathcal{D}$ as suggested by $D\mathcal{D} = \left(\overset{A}{\bullet} \rightrightarrows^{f}_{g} \overset{B}{\bullet}\right)$; where $f, g : A \to B$ are given. Then a cocone δ for D is a two-member family $\delta = (q', q)$ with $q' : A \to C, q : B \to C, C = \mathsf{tgt}\,\delta$ and $\underline{C}h \fatsemi \delta_A = \delta_B \fatsemi Dh$; in-particular $q' = q \fatsemi f = g \fatsemi q$ whence q' is fully-determined by q alone.
Let $\gamma = (p', p) : Obj(\bigvee D)$ be a colimit for D and write $p\backslash -$ in-place of $\gamma\backslash -$, then the \-laws yield: p *is a coequaliser of* (f, g) if there is a mapping $p\backslash -$ such that *CoEq-Type* and *CoEq-Char* hold.

$$q \fatsemi f = q \fatsemi g \qquad \Rightarrow \qquad p\backslash q : \mathsf{tgt}\,p \to \mathsf{tgt}\,q \qquad \text{CoEq-Type}$$

Well-formedness convention: In each law the variables are quantified in such a way that the premise of CoEq-Type is met. The notation $\cdots\backslash q$ is only senseful if $q \fatsemi f = q \fatsemi g$, like in arithmetic where the notation m/n is only sensful if n differs from 0.

$$p \fatsemi x = q \qquad \equiv \qquad x = p\backslash q \qquad \text{CoEq-Char}$$

$$p; p\backslash q = q \qquad \text{CoEq-Self}$$

$$p\backslash p = \mathsf{Id} \qquad \text{CoEq-Id}$$

$$p\backslash q \fatsemi x = p\backslash(q \fatsemi x) \qquad \text{CoEq-Fusion}$$

$$p \fatsemi x = p \fatsemi y \qquad \Rightarrow \qquad x = y \qquad \text{CoEq-Unique}$$

$$p\backslash q \fatsemi q\backslash r = p\backslash r \qquad \text{CoEq-Compose}$$

Sums

Take D and $\mathcal{D}$ as suggested by $D\mathcal{D} = \binom{A \quad B}{\bullet \quad \bullet}$. Then a cocone δ for D is a two-member family $\delta = (f, g)$ with $f : A \to C$ and $g : B \to C$, where $C = \mathsf{tgt}\,\delta$.

Let $\gamma = (\mathsf{inl}, \mathsf{inr})$ be a colimit for D, let $A + B = \mathsf{tgt}\,\gamma$, and write $[f, g]$ in-place of $\gamma\backslash(f, g)$, then the $\backslash$-laws yield: $(\mathsf{inl}, \mathsf{inr}, A + B)$ *form a sum of* A *and* B if there is a mapping $[-, -]$ such that []-Type and []-Char hold.

$$f : A \to C \quad \wedge \quad g : B \to C \qquad \Rightarrow \qquad [f, g] : A + B \to C \qquad \text{[]-Type}$$

$$\mathsf{inl} ; x = f \quad \wedge \quad \mathsf{inr} ; x = g \qquad \equiv \qquad x = [f, g] \qquad \text{[]-Char}$$

$$\mathsf{inl} ; [f, g] = f \qquad \wedge \qquad \mathsf{inr} ; [f, g] = g \qquad \text{[]-Cancellation; []-Self}$$

$$[\mathsf{inl}, \mathsf{inr}] = \mathsf{Id} \qquad \text{[]-Id}$$

$$\mathsf{inl} ; x = \mathsf{inl} ; y \quad \wedge \quad \mathsf{inr} ; x = \mathsf{inr} ; y \qquad \Rightarrow \qquad x = y \qquad \text{[]-Unique}$$

$$[f, g] ; x = [f ; x, g ; x] \qquad \text{[]-Fusion}$$

The implicit well-formedness condition in the next law is that $(F\,\mathsf{inl}, F\,\mathsf{inr}, F(A + B))$ form a sum of $F\,A$ and $F\,B$.

$$F\,[f, g]_{\mathcal{C}} = [F\,f, F\,g]_{\mathcal{D}} \qquad \text{where } F : \mathcal{C} \to \mathcal{D} \qquad \text{[]-Functor-Dist}$$

Duality: Sums & Products

In category theory there are two popular notations for composition, $f ; g = g \circ f$, and there are two arrow notations $A \to B \;=\; B \leftarrow A$, known as the "forwards" and "backwards" notations.

Some people prefer one notation and stick with it; however having both in-hand allows us to say: The *dual* of a categorical statement formed with $;, \to$ is obtained by syntactically replacing these two with $\circ, \leftarrow$ respectively while leaving variables and Id's alone.

For example, applying this process to sums yields *products*:

$$\begin{array}{ll} & h = \langle f, g \rangle \\ = & \quad \{ \text{ We define products as dual to sums } \} \\ & h = \mathsf{dual}[f, g] \\ = & \quad \{ \text{ dual operation definition } \} \\ & \mathsf{dual}\,(h = [f, g]) \\ = & \quad \{ \text{ []-Char and Leibniz } \} \\ & \mathsf{dual}\,(\mathsf{inl} ; h = f \quad \wedge \quad \mathsf{inr} ; h = g) \\ = & \quad \{ \text{ dual operation definition } \} \\ & \mathsf{dual}\,(\mathsf{inl} ; h) = f \quad \wedge \quad \mathsf{dual}\,(\mathsf{inr} ; h) = g) \\ = & \quad \{ \text{ dual operation definition } \} \\ & \mathsf{dual}\;\mathsf{inl} \circ h = f \quad \wedge \quad \mathsf{dual}\;\mathsf{inr} \circ h = g \\ = & \quad \{ \text{ Define: } \mathsf{fst} = \mathsf{dual}\,\mathsf{inl},\ \mathsf{snd} = \mathsf{dual}\,\mathsf{inr} \ \} \\ & \mathsf{fst} \circ h = f \quad \wedge \quad \mathsf{snd} \circ h = g \\ = & \quad \{ \text{ Switch back to } ;\text{-notation } \} \\ & h ; \mathsf{fst} = f \quad \wedge \quad h ; \mathsf{snd} = g \end{array}$$

$(\mathsf{fst}, \mathsf{snd}, A \times B)$ *form a product of* A *and* B if there is an operation $\langle -, - \rangle$ satisfying the Char and Type laws below; from which we obtain a host of corollaries.

$$f : C \to A \quad \wedge \quad g : C \to B \qquad \Rightarrow \qquad \langle f, g \rangle : C \to A \times B \qquad \langle\rangle\text{-Type}$$

$$x ; \mathsf{fst} = f \quad \wedge \quad x ; \mathsf{snd} = g \qquad \equiv \qquad x = \langle f, g \rangle \qquad \langle\rangle\text{-Char}$$

$$\langle f, g \rangle ; \mathsf{fst} = f \qquad \wedge \qquad \langle f, g \rangle ; \mathsf{snd} = g \qquad \langle\rangle\text{-Cancellation; } \langle\rangle\text{-Self}$$

$$\langle \mathsf{fst}, \mathsf{snd} \rangle = \mathsf{Id} \qquad \langle\rangle\text{-Id}$$

$$x ; \mathsf{fst} = y ; \mathsf{fst} \quad \wedge \quad x ; \mathsf{snd} = y ; \mathsf{snd} \qquad \Rightarrow \qquad x = y \qquad \langle\rangle\text{-Unique}$$

$$x ; \langle f, g \rangle = \langle x ; f, x ; g \rangle \qquad \langle\rangle\text{-Fusion}$$

$$F\,\langle f, g \rangle_{\mathcal{C}} = \langle F\,f, F\,g \rangle_{\mathcal{D}} \qquad \text{where } F : \mathcal{C} \to \mathcal{D} \qquad \langle\rangle\text{-Functor-Dist}$$

These are essentially a re-write of the sum laws; let's write the next set of laws only once.

Let the tuple "$(\!|\ |\!)$, $\star$, l, r, $\circledcirc$" be either "$\langle\ \rangle$, $\times$, fst, snd, $\circ$" or "$[\ ]$, $+$, inl, inr, $;$".

For categories in which sums and products exist, we define for $f : A \to B$ and $g : C \to D$,

$$f \star g = (\!| f \circledcirc l, g \circledcirc r |\!) : A \star C \to B \star D \qquad \star\text{-Definition}$$

$$l \circledcirc (f \star g) = f \circledcirc l \qquad \wedge \qquad r \circledcirc (f \star g) = g \circledcirc r \qquad l, r\text{-Naturality}$$

$$(\!| l \circledcirc h, r \circledcirc h |\!) = h \qquad \text{Extensionality}$$

$$(f \star g) \circledcirc (\!| h, j |\!) = (\!| f \circledcirc h, g \circledcirc j |\!) \qquad \text{Absorption}$$

$$\mathsf{Id} \star \mathsf{Id} = \mathsf{Id} \qquad \wedge \qquad (f \star g) \circledcirc (h \star j) = (f \circledcirc h) \star (g \circledcirc j) \qquad \star\text{-Bi-Functoriality}$$

$$(\!| f, g |\!) = (\!| h, j |\!) \qquad \equiv \qquad f = h \quad \wedge \quad g = j \qquad \text{Structural Equality}$$

$$\langle [f, g], [h, j] \rangle = [\langle f, h \rangle, \langle g, j \rangle] \qquad \text{Interchange Rule}$$

Notice that the last law above is self-dual.

Reference

A Gentle Introduction to Category Theory — the calculational approach
by Maarten Fokkinga

An excellent introduction to category theory with examples motivated from programming, in-particular working with sequences. All steps are shown in a calculational style –which Fokkinga has made available for use with LaTeX– thereby making it suitable for self-study.

Clear, concise, and an illuminating read.

Now notes from *Program Design by Calculation* of J. Oliveira.

§2, upto §2.3, An Introduction to Pointfree Programming

The main emphasis is on *compositionality*, one of the main advantages of "thinking functionally", explaining how to construct new functions out of other functions using a minimal set of predefined functional combinators. This leads to a style which is *pointfree* in the sense that function descriptions dispense with variables –also known as *points*.
Why do people look for compact notations? A compact notation leads to shorter documents, less lines of code in programming, in which patterns are easier to identify and reason about. Properties can be stated in clear-cut, one-line long equations which are easy to memorise.
The notation "$f : A \to B$" focuses on what is relevant about f and can it be regard as a kind of contract:

> f *commits itself* to producing a B-value provided it is supplied with an A-value.

Types provide the "glue", or interface, for putting functions together to obtain more complex functions.
What do we want functions for? One wants functions for modelling and reasoning about the behaviour of real things.
In order to switch between the pointwise and pointfree settings we need two "bridges": One lifting equality to the function level and the other lifting function application.
Two functions $f, g : A \to B$ are the same function iff they agree at the pointwise level:

$$f = g \quad \equiv \quad (\forall a : A \bullet f\, a = g\, a) \qquad \text{Extensionality}$$

Function application is replaced by the more generic concept of functional *composition* suggested by function-arrow chaining: Whenever we have two functions such that the target type of one of them, say $g : B \leftarrow A$ is the same as the source type of the other, say $f : C \leftarrow B$ then "f after g", their *composite function*, $f \circ g : C \leftarrow A$ can be defined. It "glues" f and g together, "sequentially":

$$C \xleftarrow{f} B \xleftarrow{g} A \quad \Rightarrow \quad C \xleftarrow{f \circ g} A \qquad \text{composition-Type}$$

§2.5 Constant functions

Opposite to the identity functions which do not lose any information, we find functions which lose all, or almost all, information. Regardless of their input, the output of these functions is always the same value.
Let C be a nonempty data domain and let $c : C$. Then we define the *everywhere* c function as follows, for arbitrary A:

$$\underline{c}\, a = c \qquad \text{constant-Defn}$$

The following property defines constant functions at the pointfree level:
Constant functions force any difference in behaviour for any two functions to disappear:

$$\underline{c} \circ f = \underline{c} \circ g \qquad \text{constant-Equality}$$

In $\mathcal{S}et$, we have that composition and application are bridged explicitly by the constant functions:

$$f \circ \underline{c} = \underline{f\, c} \qquad \mathcal{S}et\text{-Bridge}$$

§2.6 Monics and Epics

Identity functions and constant functions are limit points of the functional spectrum with respect to information preservation. All the other functions are in-between: They "lose" some information, which is regarded as uninteresting for some reason.
How do functions lose information? Basically in two ways: They may be "blind" enough to confuse different inputs, by mapping them to the same output, or they may ignore values of their target. For instance, $\underline{c}$ confuses all inputs by mapping them all onto c. Moreover, it ignores all values of its target apart from c.
Functions which do not confuse their inputs are called *monics*: They are "post-cancellable":

$$f \ \mathsf{monic} \quad \equiv \quad (\forall h, k \bullet f \circ h = f \circ k \quad \equiv \quad h = k) \qquad \text{monic-Defn}$$

Functions which do not ignore values of their target are called *epics*: They are "pre-cancellable":

$$f \ \mathsf{epic} \quad \equiv \quad (\forall h, k \bullet h \circ f = k \circ f \quad \equiv \quad h = k) \qquad \text{epic-Defn}$$

Intuitively, $h = k$ on all points of their source precisely when they are equal on all image points of f, since f being epic means it outputs all values of their source.
It is easy to check that "the" identity function is monic and epic, while any constant function $\underline{c}$ is not monic and is only epic when its target consists only of c.

§2.7 Isos

A function is an *iso* iff it is invertible: There is a function $f^\smile$ called its "inverse" or "converse" such that

$$f \circ f^\smile = \mathsf{Id} \quad \wedge \quad f^\smile \circ f = \mathsf{Id} \qquad \text{inverse-Char}$$

To *construct* $f^\smile$, we begin by identifying its type which may give insight into its necessary 'shape' –e.g., as a sum or a product– then we pick one of these equations and try to reduce it as much as possible until we arrive at a definition of $f^\smile$, or its 'components'. –E.g., the inverse of `assoc`.

- ◇ E.g., $coassocr = [\mathsf{Id} + \mathsf{inl}, \mathsf{inr} \circ \mathsf{inr}] : (A+B)+C \cong A+(B+C)$, its inverse *coassocl* must be of the shape $[x,[y,z]]$ for unknowns x, y, z which can be calculated by solving the equation $[x,[y,z]] \circ coassocr = \mathsf{Id}$ –Do it!

If, for some reason, $f^\smile$ is found handier than isomorphism f in reasoning, then the following rules can be of help.

$$f \circ x = y \quad \equiv \quad x = f^\smile \circ y \qquad \text{inverse-Shunting}_1$$

$$x \circ f = y \quad \equiv \quad x = y \circ f^\smile \qquad \text{inverse-Shunting}_2$$

Consequently, Isos are necessarily monic and epic, but in general the other way around is not true.

Isomorphisms are very important because they convert data from one "format", say A, to another format, say B, without losing information. So f and $f^\smile$ are faithful protocols between the two formats A and B. **Of course, these formats contain the same "amount" of information although the same data adopts a "different" shape in each of them.** —c.f. Example Structures.

Isomorphic data domains are regarded as "abstractly" the same; then one write $A \cong B$.
Finally, note that all classes of functions referred to so far —identities, constants, epics, monics, and isos— are closed under composition.

§2.8 Gluing functions which do not compose – products

Function composition has been presented above as a basis for gluing functions together in order to build more complex functions. However, not every two functions can be glued together by composition.

For instance, functions $A^f C^g B$ do not compose with each other since the source of one is not the target of the other.
Since f and g share the same source, their outputs can be paired: $c \mapsto (f\,c, g\,c)$. We may think of the operation which pairs the outputs of f and g as a new function combinator: $\langle f, g\rangle = c \mapsto (f\,c, g\,c)$ –read "f *split* g".

$\langle f, g\rangle\, c$ duplicates c so that f and g can be executed in "parallel" on it.

Function $\langle f, g\rangle$ keeps the information of both f and g in the same way Cartesian product $A \times B$ keeps the information of A and B.
So, in the same way that A data or B data can be retrieved from $A \times B$ data via the projections $A^{\mathsf{fst}} A \times B^{\mathsf{snd}} B$, f and g can be retrieved via the same projections:

$$\mathsf{fst} \circ \langle f, g\rangle = f \quad \wedge \quad \mathsf{snd} \circ \langle f, g\rangle = g \qquad \text{Cancellation}$$

A *split* arises wherever two functions do not compose but share the same source.

How do we glue functions that fail such a requisite, say $f : A \leftarrow C$ and $g : B \leftarrow D$? We regard their sources as projections of a product: $A^{f \circ \mathsf{fst}} C \times D^{g \circ \mathsf{snd}} B$. Now they have the same source and so the split combinator can be used: $f \times g = (c, d) \mapsto (f\,c, g\,d)$.
This corresponds to the "parallel" application of f and g, each with its *own* input.

What is the *interplay* among the functional combinators: Composition, split, product? The first two relate to each other via the fusion law

$$\langle f, g\rangle \circ c = \langle f \circ c, g \circ c\rangle \qquad \text{Fusion}$$

Notice how it looks like the *definition* of the split operator but all applications have been lifted to compositions! Woah!

- ◇ Moreover, the absorption property is just the lifting of the pointwise definition! Woah!

All three combinators interact via the $\times$-absorption property.
The following is self-inverse,

$$swap = \langle \mathsf{snd}, \mathsf{fst}\rangle : A \times B \cong B \times A \qquad \text{swap-Def}$$

§2.9 Gluing functions which do not compose – coproducts

In the scenario $A^f C^g B$, it is clear that the kind of glue we need should make it possible to apply f if the input is from the "A side" or apply g if it is from the "B side".
We denote this new combinator "either f or g", $[f, g] : A + B \to C$, where the values of $A + B$ can be thought of as "copies" of A or B values which are "stamped" with different tags in order to guarantee that values which are simultaneously in A and B do not get mixed up.
Duality is of great *conceptual economy* since everything that can be said of concept X can be rephrased for *co-X*. **That $ ⚲ $-listing provides eloquent evidence of duality.**
Notice that the fusion law,

$$f \circ [g, h] = [f \circ g, f \circ h] \qquad \text{Fusion}$$

Is essentially the definition: If we're in the left 'g' then the result is 'f' applied to it; otherwise if we're in the right 'h' then the result is 'f' applied to it! Woah: From pointwise to pointfree.

§2.10 Mixing products and coproducts

Any $f : A + B \to C \times D$ can be expressed alternatively as an *either* or as a *split*. It turns out that both formats are identical: The exchange rule.
For example, $undistr = \langle [\mathsf{fst}, \mathsf{fst}], \mathsf{snd} + \mathsf{snd}\rangle = [\mathsf{Id} \times \mathsf{inl}, \mathsf{Id} \times \mathsf{inr}] : (A \times B) + (A \times C) \to A \times (B + C)$.
Also, by the exchange rule,

$$[f \times g, h \times k] \;=\; \langle [f, h] \circ (\mathsf{fst} + \mathsf{fst}), [g, k] \circ (\mathsf{snd} + \mathsf{snd})\rangle \qquad \text{Cool-Property}$$

$$\langle f + g, h + k\rangle \;=\; [\langle f, h\rangle (\mathsf{inl} \times \mathsf{inl}), \langle g, k\rangle (\mathsf{inr} \times \mathsf{inr})] \qquad \text{Co-cool-Property}$$

Also, since constants ignore their inputs,

$$\{ \qquad \text{Exchange-with-constant}$$

$$[\langle\, \mathrm{f},\ \underline{\mathrm{k}}\ \rangle, \langle\, \mathrm{g}\ ,\ \underline{\mathrm{k}}\ \rangle] = \langle\, [\mathrm{f}, \mathrm{g}],\ \underline{\mathrm{k}}\ \rangle\ \}$$

§2.13 Universal Properties

The compositional combinators put forward so far have been equipped with a concise *set of properties* which enable programmers to transform programs, reason about them, and perform useful calculations. This raises a *programming methodology* which is scientific and stable.
The relevance of universal properties, such as $\mathcal{F}$-Char, is that it offers a way of *solving equations* of the form $y = \mathcal{F}\,x$. For example, can the identity be expressed, or 'reflected', using this combinator? We just solve the equation $\mathsf{Id} = \mathcal{F}\,x$ for unknown(s) x by appealing to the universal property.

§2.14 Guards and McCarthy's conditional

Define

{ ?-Defn

(p?)\, a = If\, p\, a \,Then\, inl\, a \,Else\, inr\, a Fi}
We call $p? : A \to A + A$ the *guard* associated to predicated $p : A \to$. The guard is more informative than p alone: It provides information about the outcome of testing p on some input a, encoded in terms of the sum injections, inl for *true* and inr for *false*, without losing the input a itself.

$$p \to g, h \quad = \quad [g, h] \circ p? \qquad \text{McCarthy-Conditional-Defn}$$

Hence to reason about conditionals one may seek help in the algebra of sums.

$$f \circ (p \to g, h) \quad = \quad p \to f \circ h, f \circ h \qquad \text{conditional-Fusion}_1$$

$$(p \to g, h) \circ k \quad = \quad p \circ k \to g \circ k, h \circ k \qquad \text{conditional-Fusion}_2$$

$$k \circ \langle (p \to f, h), (p \to g, i) \rangle \quad = \quad p \to k \circ \langle f, g \rangle, k \circ \langle h, i \rangle \qquad \text{conditional-product-Fusion}$$

$$\langle (p \to f, h), (p \to g, i) \rangle \quad = \quad p \to \langle f, g \rangle, \langle h, i \rangle \qquad \text{conditional-Abides-Distributivity}$$

$$p \to f, f \quad = \quad f \qquad \text{conditional-Idempotency}$$

$$\langle f, (p \to g, h) \rangle \quad = \quad p \to \langle f, g \rangle, \langle f, h \rangle \qquad \text{conditional-}\langle\rangle\text{-Distributivity}$$

$$(p \to g, h) \times f \quad = \quad p \circ \mathsf{fst} \to g \times f, h \times f \qquad \text{conditional-}\times\text{-Distributivity}$$

As well as their duals!

Wow –another example– involving conditionals

Just as in the case of proving *tails* is a natural transformation **without** using any implementing definitions, here's a similar example.

Here's an illustration of how *smart* pointfree algebra can be in reasoning about functions that *one does not actually defined explicitly.*
It also shows how relevant the *natural properties* are.
The issue is that our definition of a guard, ?-Defn, is pointwise and most likely unsuitable to prove facts such as, for instance,

$$p? \circ f = (f + f) \circ (p \circ f)?$$

Thinking better, instead of 'inventing' ?-Defn, we might –and perhaps should!– have defined

$$2 \;\cong\; 1 + 1 \qquad \text{two-Defn}$$

$$2 \times A \;\cong\; A + A \qquad \text{double-Defn}$$

$$p? = \mathit{double} \circ \langle p, \mathsf{Id} \rangle \qquad \text{?-Defn-Pointfree}$$

which actually express rather closely our strategy of switching from products to coproducts in the definition of $(p?)$.

In particular, **we do not need to define *double* explicitly**. From its type we immediately infer its natural, or free, property:

$$\mathit{double} \circ (\mathsf{Id} \times f) \quad = \quad (f + f) \circ \mathit{double}$$

It turns out that this is the *knowledge* we need about *double* in order to prove the above mentioned property.

The less one has to write to solve a problem, the better. One saves time and one's brain, adding to productivity. This is often called *elegance* when applying a scientific method.

§2.15 Gluing functions which do not compose –exponentials

Functional application is the bridge between the pointfree and pointwise worlds.
Suppose we are given the task to combine two functions, one binary $B^f C \times A$ and the other unary $D^g A$. Clearly none of the combinations $f \circ g, \langle f, g \rangle$, or $[f, g]$ is well-typed. Hence f and g cannot be put together and require some extra interfacing.

Note that $\langle f, g \rangle$ would be well-defined in case the C components of f's source could be somehow "ignored". Suppose, in fact, that in some particular context the first argument of f happens to be "irrelevant", or to be frozen to some $c : C$. It is easy to derive a new function $f_c : A \to B : a \mapsto f(c, a)$ from f, combines nicely with g via the split combinator: $\langle f_c, g \rangle$ is well-typed and bears the type $B \times D \leftarrow A$.

Since f_c ... paragraph after (2.80). **working here**

FIX

1. §2.4, last line: Should be *'natural'* instead of *'natural'*.

2. §2.5, exercise 2.1: Maybe place this at the end of §2.4 since it's a problem on identity functions, rather than at the end of §2.5 which is on constant functions.

3. §2.6, end: It's easy to see how monics generalise injective functions but it's not as easy for the surjective-epic case. I've conjured up the following explanation, which might be helpful to include.

   $$f \;\mathsf{epic} \qquad \equiv \qquad (\forall h, k \bullet h \circ f = k \circ f \quad \equiv \quad h = k) \qquad \text{epic-Defn}$$

   Intuitively, $h = k$ on all points of their source precisely when they are equal on all image points of f, since f being epic means it outputs all values of their source.

4. §2.7, before equation 2.18: I think you meant *pointfree* not *pointwise* as clearly 2.18 involves no points.

5. §2.13, $\times$-fusion calculation: In the final hint, perhaps add *after (2.63)* so that it is clear which *derived above* law is being referenced –since its name is in the surrounding text and may not be easy to locate.

6. §2.14, page 39, second paragraph: Typo *market* should be *marked*.

7. §2.14, page 39, diagram: Typos, left arrow should be labelled g not f, and the right arrow should be labelled h not g.

8. §2.14, exercise 2.21: It seems you have the accidental typos λap appearing twice.

9. §2.15, second paragraph: For $f : B \leftarrow C \times A$, the phrase $f\ c \in B$ is ill-typed and it may be incorrect to say it, *$f\ c$, denotes a value of type B.*