18.S097 PS2 (in Swift)

Jasdev Singh

January 27, 2020

Acknowledgements.

Tomás Ruiz-López helped me get a better sense of Bow's higher-kinded-type emulation in answering 2(b). Also, while learning how to typeset diagrams, I referenced Brandon Williams' graduate school notes.

Question 1.

Rephrased, Question 1 is asking for the cardinality of the objects in the functor category, $[\mathbf{Set}, \mathbf{3}]$, i.e. $|\mathbf{Obj}([\mathbf{Set}, \mathbf{3}])|$. To start, we have three functors into $\mathbf{3}$ at hand $K_1, K_2, K_3 \in \mathbf{Obj}([\mathbf{Set}, \mathbf{3}])$ (borrowing notation from Question 2).

In asking if there are any functors beyond these, we need to make an observation about **Set**'s hom-sets. $\forall S_1, S_2 \in \mathbf{Obj}(\mathbf{Set}), S_1 \neq \emptyset, S_2 \neq \emptyset : \mathrm{Hom}(S_1, S_2) \neq \emptyset$. Home-sets in **Set** are non-empty for all *non-empty* set pairings. There will always be functions between them. Now, let's turn our attention to the empty set. Its self-hom-set only contains the identity morphism, $\mathrm{id}_\emptyset$. And all of its originating hom-sets—$\mathrm{Hom}(\emptyset, S)$ for $S \in \mathbf{Obj}(\mathbf{Set}), S \neq \emptyset$—are empty since there we can't construct functions from non-empty sets into the empty set.

We can lean on the above to make sure connections (morphisms) aren't broken across our functors.

Let's make this more precise.

The remaining functors need to map $\emptyset$ to an object in $\mathbf{3}$ that *only* has outbound morphisms. Then, all other objects in **Set** must be mapped in the usual way. Listing out the steps for our next candidate functor F_1,

(a) Map $\emptyset$ and $\mathrm{id}_\emptyset$ to 1 and id_1, respectively.

(b) Map all $\emptyset$-originating morphisms to a.

(c) Map all non-empty sets, their identities, and morphisms between non-empty sets to 2 and id_2 (akin to K_2).

We have some degrees of freedom in carrying out (a)–(c). That is, we can form another functor F_2 by, substituting in 2 for all mentions of 1 in (a). b for a in (b). And 3 for all occurrences of 2 in (c).

Similarly, to form an F_3. Leave (a) as is. $b \circ a$ for a in (b). And 3 for all occurrences of 2 in (c).

$K_1, K_2, K_3, F_1, F_2, F_3$ exhaust all possible structure preserving mappings between **Set** and **3**.

Question 2.

(a) K_B preserves

- compositions, since $\forall S_1, S_2, S_3 \in \mathbf{Obj}(\mathbf{Set})$ and $\forall f : S_1 \to S_2, \forall g : S_2 \to S_3, K_B(g \circ f) = \mathrm{id}_B = \mathrm{id}_B \circ \mathrm{id}_B = K_B(g) \circ K_B(f)$.
- identities, since $\forall S \in \mathbf{Obj}(\mathbf{Set}), K_B(\mathrm{id}_S) = \mathrm{id}_B = \mathrm{id}_{K_B(S)}$.

(b) Mirroring Bow's approach to higher-kinded-type emulation and with `boolConst` representing the constant functor on Swift's `Bool` type.

```
class Kind<F, A> {
        init() {}
}

final class ForConst {}
final class ConstPartial<Constant>: Kind<ForConst, Constant> {}
typealias ConstOf<Constant, Tag> = Kind<ConstPartial<Constant>, Tag>

final class Const<Constant, Tag>: ConstOf<Constant, Tag> {
        let constant: Constant

        init(_ constant: Constant) {
                self.constant = constant
        }

        static func fix(
                _ fConstant: ConstOf<Constant, Tag>
        ) -> Const<Constant, Tag> {
                fConstant as! Const<Constant, Tag>
        }
}

postfix operator ^
postfix func ^<Constant, Tag>(
        _ fConstant: ConstOf<Constant, Tag>
) -> Const<Constant, Tag> {
        Const.fix(fConstant)
}

protocol Functor {
        static func map<A, B>(
                _ fA: Kind<Self, A>,
                _ transform: (A) -> B
        ) -> Kind<Self, B>
}

extension ConstPartial: Functor {
        static func map<OldTag, NewTag>(
                _ fA: ConstOf<Constant, OldTag>,
                _ transform: (OldTag) -> NewTag
        ) -> ConstOf<Constant, NewTag> {
                Const(fA^.constant)
        }
}

func boolConst<A>(_ constant: Bool) -> Const<Bool, A> {
        Const(constant)
}
```

Question 3.

(a) For δ to define a natural transformation, the following diagram must commute ($\forall f : X \to Y$ in $\mathrm{Hom}(X, Y)$, with $X, Y \in \mathbf{Obj}(\mathbf{Set})$).

(Note, we can drop the $\mathrm{id}_{\mathbf{Set}}$ prefixes along the top row for $\mathrm{id}_{\mathbf{Set}}(X)$, $\mathrm{id}_{\mathbf{Set}}(f)$, and $\mathrm{id}_{\mathbf{Set}}(Y)$ since the identity functor maps all objects and morphisms to themselves.)

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\delta_X"] & Y \arrow[d, "\delta_Y"] \\
X \times X \arrow[r, "\mathrm{Double}(f)"] & Y \times Y
\end{tikzcd}$$

Following the bottom path for an arbitrary $x \in X$,

$$x \xmapsto{\delta_X} (x, x) \xmapsto{\mathrm{Double}(f)} (f(x), f(x))$$

And the upper,

$$x \xmapsto{f} f(x) \xmapsto{\delta_Y} (f(x), f(x))$$

Both paths are equivalent! δ is indeed a natural transformation between $\mathrm{id}_{\mathbf{Set}}$ and Double.

(b)
```
func diag<A>(_ a: A) -> (A, A) {
        (a, a)
}
```

Question 4.

(a) Since t, t' are both terminal objects, each have unique maps, say u and v, to one another.

$$\begin{tikzcd} t \arrow[r, bend left, dashed, "u"] & t' \arrow[l, bend left, dashed, "v"] \end{tikzcd}$$

Moreover, since the containing category must contain composite morphisms, $\mathrm{Hom}(t, t)$ and $\mathrm{Hom}(t', t')$ must contain $v \circ u$ and $u \circ v$, respectively. But, again leaning on their terminality, there can only be one endomorphism for each—their identities. This forces both compositions to cancel one another out, making $t \cong t'$.

(b) Mirroring the approach in (a), we lean on the product's universal properties.

That is, there exists unique morphisms between p, p' that we'll call u and v.

$$\begin{tikzcd} & p' \arrow[dl, bend right, "\pi'_a"'] \arrow[dr, bend left, "\pi'_b"] \arrow[d, dashed, bend left, "u"] & \\ & p \arrow[u, dashed, bend left, "v"] \arrow[dl, "\pi_a"] \arrow[dr, "\pi_b"'] & \\ a & & b \end{tikzcd}$$

.

Their composites must also be in the containing category, and again leaning on their universal properties, their must only be one endomorphism for each, that're forced to be identities.

(c) It could! Each of these isomprhisms are forced by the *uniqueness* of morphisms originating from or pointing towards the universal constructions at hand.

Question 5.

(a) 3. It has projections towards both 42 and 27 in a way such at all other naturals with similar projects—namely, 1—has a unique morphism through 3. The product defines the greatest common divisor in this setting.

(b) $\{b, c\}$—this operation is set intersection.

(c) `false`—it has projections to itself and `true` in the category and there are no other candidate products for it to factorize. This binary operation is material implication, coinciding with the definition of the morphism in this setting.

Question 6.

(a)

$$\begin{tikzcd} \mathtt{A} \times \mathtt{B} \arrow[r, bend left, "\mathtt{swap}"] & \mathtt{B} \times \mathtt{A} \arrow[l, bend left, "\mathtt{unswap}"] \end{tikzcd}$$

```
func swap<A, B>(_ pair: (A, B)) -> (B, A) {
        (pair.1, pair.0)
}

func unswap<A, B>(_ pair: (B, A)) -> (A, B) {
        (pair.1, pair.0)
}
```

`swap` and `unswap` are inverses because their forward and backwards compositions leave inputs unchanged. That is, e.g. when grounding `A` and `B` with `Int` and `Bool`, `swapUnswap` and `unswapSwap` are equivalent to Swift's identity function: { `$0` }.

```
let swapUnswap: ((Int, Bool)) -> (Int, Bool) = swap >>> unswap
let unswapSwap: ((Int, Bool)) -> (Int, Bool) = unswap >>> swap
```

(b) Replaying the approach in (a) with the following diagram and implementation (note, `Void` is Swift's unit type),

$$\begin{tikzcd} \mathtt{A} \arrow[r, bend left, "\mathtt{unit}"] & \mathtt{Void} \times \mathtt{A} \arrow[l, bend left, "\mathtt{ununit}"] \end{tikzcd}$$

```
func unit<A>(_ value: A) -> ((), A) {
        ((), value)
}

func ununit<A>(_ pair: ((), A)) -> A {
        pair.1
}
```

(c) Once more,

$$\begin{tikzcd}
\texttt{A} \times (\texttt{B} \times \texttt{C}) \arrow[r, bend left, "\texttt{assoc}"] & (\texttt{A} \times \texttt{B}) \times \texttt{C} \arrow[l, bend left, "\texttt{unassoc}"]
\end{tikzcd}$$

```
func assoc<A, B, C>(_ value: (A, (B, C))) -> ((A, B), C) {
        ((value.0, value.1.0), value.1.1)
}

func unassoc<A, B, C>(_ value: ((A, B), C)) -> (A, (B, C)) {
        (value.0.0, (value.0.1, value.1))
}
```

Question 7.

We'll approach this in two steps. First, let's define projection functors down from $\mathcal{C} \times \mathcal{D}$ to $\mathcal{C}$ and $\mathcal{D}$.

$\pi_{\mathcal{C}}$ takes an arbitrary object $(c, d) \in \text{Obj}(\mathcal{C} \times \mathcal{D})$ to c. And similarly, an arbitrary morphism in $\mathcal{C} \times \mathcal{D}$, say (f, g), to f in $\mathcal{C}$.

$\pi_{\mathcal{D}}$ mirrors $\pi_{\mathcal{C}}$'s definition—focusing in on the $\mathcal{D}$ components of the product.

We now need to show that for all other candidate products, $\mathcal{C} \times' \mathcal{D}$, there exists a unique morphism between $\mathcal{C} \times' \mathcal{D}$ and $\mathcal{C} \times \mathcal{D}$. Leaning on 4(b), the tho products must be isomorphic, and since their respective universal properties force a unique morphism between the two, we then shake out a unique morphism between the candidate product and the one posed in this problem.

Question 8.

```
enum Either<A, B> {
        case left(A)
        case right(B)
}

func bimap<A, B, NewA, NewB>(
        _ aTransform: @escaping (A) -> NewA,
        _ bTransform: @escaping (B) -> NewB
)
-> (Either<A, B>)
-> Either<NewA, NewB> {
        { either in
                switch either {
                case let .left(a):
                        return .left(aTransform(a))
                case let .right(b):
                        return .right(bTransform(b))
                }
        }
}
```