18.S097 PS1 (in Swift)

Jasdev Singh

January 19, 2020

Question 1.

(Assume Bow is imported for all snippets.)

(a)
```
func f(_ x: Int) -> Int {
        x * x
}

func g(_ x: Int) -> Int {
        x + 1
}
```

(b)
```
let h = g >>> f
h(2) // 9
```

(c)
```
let i = f >>> g
i(2) // 5
```

Question 2.

(a) $\text{Ob}(\mathbf{2}) = \{1, 2\}$

$\mathbf{2}(1,1) = \{\text{id}_1\}$

$\mathbf{2}(1,2) = \{f\}$

$\mathbf{2}(2,1) = \emptyset$

$\mathbf{2}(2,2) = \{\text{id}_2\}$

Compositions: $\{f \circ \text{id}_1, \text{id}_2 \circ f, \text{id}_2 \circ f \circ \text{id}_1\}$

Identity morphisms: $\{\text{id}_1, \text{id}_2\}$

(b) The unital law is held by way of $f \circ \text{id}_1 = f$ and $\text{id}_2 \circ f = f$. That is, taking either the identity morphism on 1 or on 2 before or after f, respectively, is the same as only following f.

The associative law holds for the only triple composition—$\text{id}_2 \circ f \circ \text{id}_1$—in that both $\text{id}_2 \circ (f \circ \text{id}_1)$ and $(\text{id}_2 \circ f) \circ \text{id}_1$ result in the same path.

Question 3.

It is! The "but no other morphisms" bit is key—it implies there isn't a non-identity morphisms with the same domain and codomain in the category that $f \circ g$ or $g \circ f$ could be equal to.

Question 4.

(a) Not satisfying the unit law implies that the identity morphisms in the category, when composed with a non-identity morphism (say, f), isn't simply f. Which, implies that the category doesn't have identity morphisms in the first place (and by extension, is not a category).

(b) A single object category with morphisms corresponding to division and the object being the set of rational numbers. It's unital in that we can divide by one, but the operation isn't associative.

Question 5.

(a) $(\mathbb{N}, +, 0)$ forms a monoid by way of 0 acting as the unital element and associativity extending from the associativity of addition on the naturals.

(b) Denoting string concatenation as $+$, $(\text{List}_{\{0,1\}}, +, [])$ forms a monoid. The unital law is held since $\forall a \in \text{List}_{\{0,1\}}, a + [] = a = [] + a$. Moreover, concatenation of three strings $a, b, c \in \text{List}_{\{0,1\}}, a + (b + c) = a + bc = abc = ab + c = (a + b) + c$.

(c) Every monoid gives rise to a single object category by the following procedure:

- First, treat the underlying set of the monoid, say M, as the object.
- Map each member $m \in M$ to a morphism in $\text{Mor}(M, M)$. In doing so, there will be a morphism corresponding to the unital element of the monoid acting as the identity arrow. And, for each nonzero member of the monoid, n, following its corresponding arrow in the category represents $+n$ in the underlying monoid.
- This construction allows for compositions to be represented by repeated applications of the monoid's operator, from which associativity also follows.

Question 6.

(a) 12's identity morphism is the only member of $\mathcal{P}(12, 12) = \{x \in \mathbb{N}_{\geq 1} \mid x * 12 = 12\} = \{1\}$.

(b) Given morphisms x and y such that $x : a \to b$ and $y : b \to c$, there exists positive integers x_1 and x_2 for which $b = x_1 * a$ and $c = x_2 * b$. Combining the equalities, we get $c = x_2 * (x_1 * a)$—and shuffling parentheses, $c = (x_2 * x_1) * a$. $x_2 * x_1 \in \mathbb{N}$, so by the preorder's construction, there exists a morphism $y \circ x$ serving as the composite.

(c) Unfortunately not. To see why, let's allow 0 into the category. Its self-hom-set (lets pretend that's a term) is given by $\mathcal{P}(0,0) = \{x \in \mathbb{N} \mid x * 0 = 0\} = \mathbb{N}$. This is already suspicious. 0's hom-set on itself isn't a singleton *or* the empty set.

$0 \leq 0$ in the underlying preorder then corresponds to infinitely many identity arrows. Choosing two candidate identities, id_0 and id_0' where $\mathrm{id}_0 \neq \mathrm{id}_0'$:

We then simultaneously have $\mathrm{id}_0' \circ \mathrm{id}_0 = \mathrm{id}_0'$ and $\mathrm{id}_0' \circ \mathrm{id}_0 = \mathrm{id}_0$ by way of both candidates being identities. Further, leaning on associativity from the category,

$$\begin{aligned}
(\mathrm{id}_0' \circ \mathrm{id}_0) \circ \mathrm{id}_0 &= \mathrm{id}_0' \circ (\mathrm{id}_0 \circ \mathrm{id}_0) && \text{By associativity.} \\
\mathrm{id}_0 \circ \mathrm{id}_0 &= \mathrm{id}_0' \circ \mathrm{id}_0 && (1). \\
\mathrm{id}_0 &= \mathrm{id}_0', \perp && (2).
\end{aligned}$$

(1) Choosing id_0 for the left parentheses and reducing the set on the right.

(2) Reducing further on the left and leaning on id_0 being an identity on the right.

A contradiction. Allowing 0 into the monoid generates infinitely many unique identity morphisms for it and we showed that they all must coincide.

Question 7.

$$\begin{aligned}
(\text{AND True})\ \text{False} &= \text{True False True} \\
&= (\lambda y.\text{False})\ \text{True} \\
&= \text{False}
\end{aligned}$$

$$\begin{aligned}
(\text{OR False})\ \text{True} &= \text{False False True} \\
&= (\lambda y.y)\ \text{True} \\
&= \text{True}
\end{aligned}$$

Question 8.

$$
\begin{aligned}
Yg &= (\lambda x.g(xx))(\lambda x.g(xx)) \\
&= g(\lambda x.g(xx))(\lambda x.g(xx)) \\
&= g(Yg) \\
&= g(\ldots g(Yg)\ldots) && \text{Recursing infinitely.}
\end{aligned}
$$

Question 9.

Taking the protocol witness-based approach covered in episodes 33–36 of Point-Free:

```
struct Category<Object, Morphism> {
        let domain: (Morphism) -> Object
        let codomain: (Morphism) -> Object
        let identity: (Object) -> Morphism
        let compose: (Morphism) -> (Morphism) -> Morphism?
}

enum TwoObject {
        case one, two
}

enum TwoMorphism {
        case id1, f, id2
}

let twoCategoryWitness = Category<TwoObject, TwoMorphism>(
        domain: { morphism in
                switch morphism {
                case .f, .id1:
                        return .one
                case .id2:
                        return .two
                }
        },
        codomain: { morphism in
                switch morphism {
                case .id1:
                        return .one
                case .id2, .f:
                        return .two
                }
        },
```

```
        identity: { object in
                switch object {
                case .one:
                        return .id1
                case .two:
                        return .id2
                }
        }
) { second in
        { first in
                // Assuming `second ∘ first` order.
                switch (second, first) {
                case (.id1, .id1):
                        return .id1
                case (.id1, .id2), (.id1, .f), (.f, .f), (.f, .id2), (.id2, .id1):
                        return nil
                case (.f, .id1), (.id2, .f):
                        return .f
                case (.id2, .id2):
                        return .id2
                }
        }
}
```