LVA 185.A05 Advanced Functional Programming (SS 2020)

Self-Assessment Test 3

Monday, 30 March 2020

Topics: Part II, Chapter 4; Part IV, Chapters 9, 10, 11, and 14

Functional Pearls, Equational Reasoning, Monoids, Functors, Applicatives, Kinds

(No submission, no grading)

Part II, Chapter 4 'Equational Reasoning for Functional Pearls'

1. The symbol = is used by both functional programming languages like Haskell and imperative/object-orientied programming languages like C or Java. How do the 'functional' and the 'imperative/object-oriented' = differ from each other?
2. What does *equational reasoning* refer to. Explain and illustrate it by a striking example.
3. Where are the roots of equational reasoning?
4. Referential transparency supports equational reasoning. Why?
5. Why is equational reasoning important for functional pearls?
6. What constitutes a *functional pearl*? Can you illustrate your answer by means of an example?
7. What is meant if someone speaks of *'wholemeal'* programming? What are its benefits?
8. What 'tricks' does Richard Bird implement in his Soduko solver for gaining efficiency over the initial solver?
9. What is *'lawful'* programming? Illustrate your answer by an example.
10. What is the link of Jon Bentley to functional pearls programming?
11. *Gofer*, an overloaded term. Why?
12. What is the *SFN* problem?
13. Is solving the *SFN* problem efficiently of practical relevance? Why? Or, why not?
14. The *MNSS* problem and the *MSS* problem are closely related. Why? In what respect?
15. Comparing the initial algorithms for the *MNSS* and *MSS* problem, which one is computationally more complex? Why?

Part IV, Chapter 9 'Monoids'

1. What is a monoid? What can be made an instance of `Monoid`?
2. `Monoid` foresees an operation called `mappend` for its instances. Is the name `mappend` well chosen? Why? Or, why not?
3. Are their other names related to `Monoid` which could be considered weak choices? Why?
4. What are the monoid laws? What do they require?
5. Who is in charge that `Monoid` instances satisfy the monoid laws?
6. Why is it not just a matter of taste, an issue of a 'good' or 'bad' programming style but dangerous to implement a monoid instance failing the monoid laws? Illustrate your reasoning by an example.
7. The implementation of `mappend` of proper monoids must be:

   (a) commutative.

   (b) associative.

   (c) distributive.

   Right or wrong?
8. The below can be made monoids:

   (a) `Int`

   (b) `Bool`

   (c) `Char`

   (d) `Ordering`

   (e) `IO`

   (f) `[Int]`

   (g) `[a]`

   (h) `[]`

   (i) `Maybe Int`

   (j) `Maybe a`

   (k) `Maybe`

   (l) `Either Int`

   (m) `Either a`

   (n) `Either`

   (o) `(Int -> Int)`

   (p) `(a -> Int)`

   (q) `(Int -> b)`

   (r) `(a -> b)`

   (s) `(a ->)`

   (t) `(-> b)`

   (u) `(->)`

   Right or wrong? Meaningful or not? Why?
9. Implement a monoid instance of your choice. Prove that it is a proper monoid.
10. Monoids are made for folding values. Why?

Part IV, Chapter 10 ‘Functors’

1. What is a functor?
2. What (in principle) can ben made an instance of `Functor`?
3. Give the default implementation of the list functor.
4. The below can be made functors:

   (a) `Int`

   (b) `Bool`

   (c) `Char`

   (d) `Ordering`

   (e) `IO`

   (f) `[Int]`

   (g) `[a]`

   (h) `[]`

   (i) `Maybe Int`

   (j) `Maybe a`

   (k) `Maybe`

   (l) `Either Int`

   (m) `Either a`

   (n) `Either`

   (o) `(Int -> Int)`

   (p) `(a -> Int)`

   (q) `(Int -> b)`

   (r) `(a -> b)`

   (s) `(a ->)`

   (t) `(-> b)`

   (u) `(->)`

   Right or wrong? Meaningful or not? Why?

5. `fmap :: (a -> b) -> f a -> f b` can be read in a *curried* and *uncurried* fashion. Explain the two views of `fmap`.
6. Show that

   ```
   fmap (h . g) = fmap h . fmap g
   ```

   is type-correct.
7. What is the meaning of `fmap` of the map functor?
8. Give the standard implementation of the `Maybe` functor together with the laws it has to satisfy.
9. Prove that the standard implementation of the `Maybe` functor satisfies the functor laws.
10. Consider:

    ```
    data Either a b = Left a | Right b

    instance Functor (Either a) where
     fmap g (Right x) = Right (g x)
     fmap g (Left x)  = Left (g x)
    ```

    Does the instance implementation satisfy the functor laws? If yes, is it meaningful, desirable? Explain your reasoning.

Part IV, Chapter 11 ‘Applicatives’

1. What is an applicative?

2. What functions have to be implemented for an applicative?

3. The below can be made applicatives:

   (a) `Int`
   (b) `Bool`
   (c) `Char`
   (d) `Ordering`
   (e) `IO`
   (f) `[Int]`
   (g) `[a]`
   (h) `[]`
   (i) `Maybe Int`
   (j) `Maybe a`
   (k) `Maybe`
   (l) `Either Int`
   (m) `Either a`
   (n) `Either`
   (o) `(Int -> Int)`
   (p) `(a -> Int)`
   (q) `(Int -> b)`
   (r) `(a -> b)`
   (s) `(a ->)`
   (t) `(-> b)`
   (u) `(->)`

   Right or wrong? Meaningful or not? Why?

4. The sets of monoids and applicatives defined in Haskell programs must be disjoint. Why?

5. The list applicative and list comprehension are closely linked to each other. Why? In what respect?

6. Make `(Either a)` an applicative.

7. Show that the defining equations of the applicative operations `pure` and `(<*>)` of your `(Either a)` instance are type correct. Annotate the laws with the (most general) type information applying.

8. Prove that your `(Either a)` instance of `Applicative` satisfies the applicative laws.

9. Complete step by step the below computation sequence assuming that the applicative oerations are the ones of the map applicative:

```
(\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
 ->> (fmap (\\x y z -> [x,y,z]) (+3)) <*> (*2) <*> (/2) $ 5
 ->> ((\x y z -> [x,y,z]) . (+3)) <*> (*2) <*> (/2) $ 5
 ->> ...
 ->> [8.0,10.0,2.5]
```

10. The list and the ziplist applicative are different `Applicative` instances of lists. How do they differ? Can you ilustrate your answer by some example?

Part IV, Chapter 14 ‘Kinds’

1. What are *kinds*? What are they good for?
2. What is the kind of
   (a) `Maybe Int`
   (b) `Maybe a`
   (c) `Maybe`
   (d) `Either Int`
   (e) `Either a`
   (f) `Either`
   (g) `(Float -> Int)`
   (h) `(a -> b)`
   (i) `(a->)`
   (j) `(->)`
   (k) `[]`
   (l) `[a]`
   (m) `[Int]`
3. Give types of the below kinds, where possible:
   (a) `∗`
   (b) `∗ -> ∗`
   (c) `∗ -> ∗ -> ∗`
   (d) `∗ -> ∗ -> ∗ -> ∗`
   (e) `(∗,∗)`
   (f) `(∗,∗,∗)`
   (g) `(∗,∗,∗,∗)`
   (h) `(∗,∗) -> ∗`
   (i) `(∗ -> ∗) -> ∗`
   (j) `[∗]`
4. Let t_1, t_2, t_3 be of kind `*`, `* -> *` and `* -> * -> *`, respectively. Are t_1, t_2, t_3 eligible (in principle) as instances of
   (a) `Eq`?
   (b) `Monoid`?
   (c) `Functor`?
   (d) `Applicative`?

   Why? Or, why not?
5. There can be
   (a) applicatives
   (b) functors and applicatives
   (c) monoids and functors

   differing in their kind. Right or wrong? Why?