LVA 185.A05 Advanced Functional Programming (SS 2020)

Self-Assessment Test 2

Monday, 16 March 2020

Topic: Part IV, Chapter 7, Chapter 8; Part II, Chapter 3

Functional Arrays, Abstract Data Types, Algorithm Patterns

(No submission, no grading)

Part IV, Chapter 7 'Functional Arrays'

1. Functional programming languages offers lists for storing data. The name of the functional language 'Lisp' is actually an acronym standing for 'List Processing.' Why should there be something like 'arrays' in functional languages? Why is adding arrays to a functional language a challenge?
2. Can a functional programming language offering *dynamic* arrays be pure?
3. Referential transparency, static arrays, dynamic arrays. Which of these terms go well along with each other in functional programming languages? Which ones do not? Why?
4. What are appealing features of functional lists? What are appealing features of imperative arrays?
5. What are distracting features of functional lists? What are distracting features of imperative arrays?
6. What means 'updating an entry of a static array' from an implementation point of view? What 'updating an entry of a dynamic array?'

— Consider the non-standard library `Array` and the static arrays and supporting functions it provides.

7. `Array` offers various means to create and initialize an array value. Which ones? How do they differ from each other?
8. Give an Haskell expression (together with its type signature) for creating a three-dimensional array value of size $10 \times 5 \times 20$ of type `int`. All array entries shall be set to `Float` value 3.14 by this Haskell expression.
9. The implementation of:

```
fibs n = a where
           a = array (1,n) ([(1,0),(2},1)] ++ [(i,a!(i-1) + a!(i-2)) | i <- [3..n]])
```

   shows very poor performance. Why? How could the implementation be modified to improve performance (while still using arrays)?
10. What is type class `Ix` good for? What are its member functions?

Part IV, Chapter 8 'Abstract Data Types'

1. *Abstract data types* are appealing. Why? Why conceptually? Why pragmatically? Why from a software-engineering point of view?
2. The definition of an abstract data type consists of several parts. Which ones? What is their role?
3. Programming languages supporting abstract data types do not need to additionally support concrete data types. Right or wrong? Why?
4. What are challenges of defining abstract data types in general? What are particular challenges of doing so in Haskell?
5. What are the proof obligations when defining an abstract data type? Who is in charge for accomplishing them?

6. The lecture notes provide some examples of abstract data types. What could be other examples?
7. Who is to be named for pioneering work on abstract data types?
8. Considering stacks and queues as abstract data type examples. What is the conceptual difference between stacks and queues? How is this difference taken care of by the definitions of stacks and queues as abstract data types?
9. Define a data type `set` as abstract data type in Haskell. What operations should be supported? What properties must they enjoy? Give two different implementations of `set` and discuss their relative advantages and disadvantages.
10. If there were need to display `set` values, how could this be done in Haskell? What should be taken care of? Why?

Part II, Chapter 3 'Algorithm Patterns'

1. Algorithm patterns and glue in the sense of John Hughes. What is the link between the two?
2. The search space for backtracking, priority-first or greedy search can be (conceptually) infinite. Why isn't this a problem for the Haskell implementations provided for these algorithm patterns?
3. Conceptually, the algorithm patterns considered in Chapter 3 fall into two groups. Which ones?
4. Memoization is linked to one of the algorithm patterns considered in Chapter 3. Which one? In what respect?
5. Some problems and algorithm patterns seem to be a good fit. But some characteristics of the problem can make the pattern inadequate for solving the problem. What are examples of such problem/pattern pairs? What are the general problems behind these problem/pattern pairs? What are concrete problem/pattern pair examples, where these problems materialize?

— What are the ingredients, i.e., the parameters resp. arguments of the

6. divide-and-conquer
7. backtracking search

algorithm patterns?

– Consider the syntactical signatures of the higher-order functions for

8. priority-first search:
   `search_dfs :: (Eq node) => (node -> [node]) -> (node -> Bool) -> node -> [node]`
9. greedy search:
   `search_greedy :: (Ord node) => (node -> [node]) -> (node -> Bool) -> node -> [node]`

Explain the meaning of all items occurring in the two signatures in detail. Why is the type context in `search_ds` different from the one in `search_greedy` (i.e., `(Eq node)` vs. `(Ord node)`)?

10. What are application fields, problems often amenable to dynamic programming?