LVA 185.A05 Advanced Functional Programming (SS 2020)

Self-Assessment Test 6

Monday, 11 May 2020

Topics: Part IV, Chapters 15, 16

Parsing, Logical Programming Functionally

(No submission, no grading)

Part V, Chapter 15 'Parsing'

1. What is parsing concerned with?
2. What is the type of a parser?
3. Explain all elements occurring in the parser type definition.
4. When do we speak of a universal parser basis?
5. What parsers constitute the combinator parser basis of Chapter 15.2?
6. Do the monadic parsers of Chapter 15.3 a parser basis, too?
7. What could be arguments in favour of combinator parsing, what in favour of monadic parsing?
8. Are all parsers used by monadic parsing truely monadic operations?
9. Which parsers are truely monadic operations? Which ones are not?
10. What is recursive descent parsing?
11. Illustrate the idea of recursive descent parsing by means of an example.
12. Can you define the `spot` parser right now, on the spot so to speak?
13. What is the purpose of the always succeeding parser?
14. Why is it good to have the build parser?
15. The monad-plus operations of the parser instance represent two parsers. Which ones?
16. What is the rôle of the list of successes technique in parsing?
17. The type classes `Show` and `Read` are related to the parsing problem. Why? In what respect?
18. What is the 'white space' problem of parsing?
19. Illustrate the typical structure of a monadic parser.
20. The general parser type is defined as a two-ary type constructor. What is the rôle, meaning of the argument types of this type constructor?
21. If the general parser type is a two-ary type constructor and monads are on-ary type constructor, how does this go along with monadic parsing?
22. Can you pairwise oppose the parsers of the combinator and the monadic parser approach?
23. Parsing is sometimes (or even often) considered a show-case for the elegance of functional programming. Can you explain why?
24. John Hughes named higher-order functions and lazy evaluation as two features of functional programming languages, the strength and elegance of functional programming builds upon without referring to parsing to underpin his thesis (cf. Chapter 1). Could he have done so? Why? How?
25. What is a another feature most typical for functional programming languages that is important for parsing?

Part V, Chapter 16 'Logical Programming Functionally'

1. Functional and logic programming are representatives of the same programming paradigm. Which one?
2. What are characteristics of this paradigm? What is its essence?
3. What is the rôle, the relevance of abstraction in the evolution of programming languages?
4. What are important milestones in the history and evolution of programming languages?
5. Functional and logic programming (languages) belong to the same programming paradigm but are built upon different concepts and foundations. Can you explain them in more detail?
6. What are key problems to be solved when extending a functional programming language towards enabling (to some extent) logic programming?
7. Diagonalization and diagonalization based reasoning is an important proof technique but is also important for coping with infinite search spaces as often occur in logic programming problems. Explain the idea of diagonalization in the context of search space exploration in more detail.
8. Can you name a few examples of proofs where diagonalization arguments are key for completing the proof?
9. The approach for logic programming functionally of Chapter 16 makes intensive use of monads. To which purpose?
10. If solutions occur rarely in huge search spaces, it can be difficult for a user to distinguish an active program just not finding a(nother) solution for some time from a hanging one. How can this problem be dealt with?

Parts I – V, Various Chapters

1. Are there any links between `do` notation and list comprehension?
2. What is a *partial order*, what is a *domain*?
3. Let $S =_{df} \{s \mid s \text{ partial list or stream}\}$ be the set of partial lists and streams, and $\sqsubseteq \subseteq S \times S$ defined by:

   $$\begin{array}{lll} \bot & \sqsubseteq & \texttt{xs} \\ \texttt{x : xs} & \sqsubseteq & \texttt{y : ys} \quad \Longleftrightarrow_{df} \quad \texttt{x} \equiv \texttt{y} \wedge \texttt{xs} \sqsubseteq \texttt{ys} \end{array}$$

   where $\equiv$ denotes equality on list resp. stream entries.

   Prove that $(S, \sqsubseteq)$ is a

   (a) partial order.

   (b) domain.
4. What have functors and monads have in common? What have functors, monads, and arrows have in common?
5. What considered Leibniz a monad?
6. What does soundness of a monad mean? What does it require?
7. Rewrite `do a <- as; if (p a) then return a else failure` equivalenly using

   (a) list comprehension

   (b) monadic operations
8. Complete the instance declaration: `instance Monad ((->) d) where ...`
9. Prove that your `((->) d)` instance of `Monad` satisfies the monad laws.

10. What is a sound monad-plus instance?
11. How can stream programming help improving the performance of a functional program? Give also an example to illustrate your answer.
12. What are the German terms for natural and strong induction?
13. What is another Englisch term for natural induction?
14. Give a typical example of a property that can most naturally be proved using strong induction.
15. Properties that can be proved using strong induction need not be provable using mathematical induction. Right or wrong?
16. What is a partial list?
17. Every Haskell data type contains a special value, called undefined. How can we conceptually or in terms of the result of a computation think about a value 'undefined?'
18. Streams are also called lazy lists. Why?
19. The monadic sequencing operation `(>>=)` can be commutative but need not. Right or wrong?
20. Monads inherit the `fmap` operation of functors. Right or wrong?
21. What are higher-order functions? What are higher-order type classes?
22. Functional arrays of module Data.Array are strict in their elements. Right or wrong?
23. What are design goals when implementing functional arrays?
24. How do functional arrays and lists differ from each other?
25. How do concrete and abstract data types differ from each other?

– What can be (computational/conceptual) risks when applying the

26. divide-and-conquer
27. greedy

algorithm pattern too carelessly? Can you examples yielding evidence for your answer?

– When should you think of making a type a

28. monoid?
29. functor?
30. monad?

– What is the meaning of the `fmap` operation when read

31. uncurried?
32. curried?

33. Give a sketch of the Haskell type class hierarchy, e.g., as far it is dealt with in this course and the introductory course on functional programming.
34. Why have type classes (possibly) been introduced into Haskell? What are they good for?
35. Recursion and co-recursion are conceptually essentially the same. Do you agree? Why? Or why not?
36. Define the stream of natural numbers in at least two different ways. Can you define it in even more than two ways?
37. What are typical means or ways for proving properties of streams, functions on streams?
38. What are typical means or ways for proving properties of lists, functions on lists?
39. Functional programming is better suited for equational reasoning than imperative one. Why?
40. Programming in a functional language is powerful, highly productive and not being error-prone because of the absence of go to statements and loops. Do you agree? Why? Or why not?