ASSIGNMENT 3 – HASKELL

PRIYANSHU SINGH, 170101049

House Planner

Q.1 *Write the algorithm (in pseudo-code) that you devised to solve the problem (**you must not write the code for the algorithm in the report**).*

Brief Explanation : This is a question which we can solve with brute force with some optimization. Normal brute force would work but it would take hours for code to run if we iterate over all possible dimensions over all possible rooms. So, I assume that rooms of same type will have same dimension and I will *remove dimension pairs that correspond to the same area*. This will speed up our code.

Pseudo Code :

retrieveAllDimensions dim1 dim2 = do

1. [(a, b) | a <- [dim1.x to dim2.x], b <- [dim1.y to dim2.y]]
2. return

unique_pos_ list1 list2 = do

// list1 contains all tuples, list2 contains different areas

1. for tuple in list1:
2. if area(tuple) in list2:
3. reject this tuple
4. else keep this tuple
5. return

design <maxArea> <no of bedrooms> <no of halls> = do

1. initialize minimum and maximum dimensions of rooms
2. retrieve all possible dimensions for each room
3. initialize the no of rooms required of various types
4. posBH <- all possible dimensions of bedroom, hall pair in a list of tuples
5. similary find posBHK, posBHKB, posBHKBG, posBHKBHB where BHKBHB means bedroom, hall, kitchen, bathroom, hall and balcony.
6. temp_ <- *unique_ pos_*, this function retrieves only the cases in which the space occupied is unique. Here _ is filled with {BH, BHK, BHKB, BHKBG, BHKBHB}.
7. max area <- *maximumArea tempBHKBHB* (gets maximum area from all possible combinations)
8. answer <- *get6TupleWithMaxArea <all_possible_6tuples>*
9. if answer is null then “No design” else *printAnswer*

This is the high overview of the code, rest functions were quite small and intuitive from the given pseudo-code, so I have not added them here.

Q.2 *How many functions did you use?*

I have used 17 functions in my code. They are as follows:

- isMember
- uniqueBH
- uniqueBHK
- uniqueBHKB
- uniqueBHKBG
- uniqueBHKBGB
- maximumArea
- getFirstFromTuple
- getSecondFromTuple
- retrieveAllDimensions
- bedHall

- bedHallKitch
- bedHallKitchBath
- bedHallKitchBathGar
- bedHallKitchBathGarBal
- printAnswer
- **design** (main function)

Q.3 *Are all those pure?*

A function is called **pure** if it corresponds to a function in the mathematical sense: it associates each possible input value with an output value, and does nothing else. In particular,

- it has no *side effects*, that is to say, invoking it produces no observable effect other than the result it returns; it cannot also *e.g.* write to disk, or print to a screen.
- it does not depend on anything other than its parameters, so when invoked in a different context or at a different time with the same arguments, it will produce the same result.

No, not all functions that I have written are pure. There are two impure functions which are **design** and **printAnswer**. These functions are impure because they are doing IO and IO is impure in Haskell (Haskell is almost pure, not absolutely pure).

Q.4 *If not, why? (Means, why the problem can't be solved with pure functions only).*

This problem can't be solved with only pure functions because we need to have an IO system for the problem (we need to get input from user and show output). Because, IO is impure in Haskell, we can't get around this issue and hence no way to solve this problem only impure functions. However, I can reduce my number of impure functions to only 1 by removing the **printAnswer** function and including it in **design**. I chose not to do it in the wake of good modularity of code.

Short Notes

Q.1 *Do you think the lazy evaluation feature of Haskell can be exploited for better performance in the solutions to the assignments? If so, which solution(s) and how?*

Yes, lazy evaluation is a very useful technique in general which has 2 advantages:

- no evaluation is done until it is needed. Therefore, it saves time and space by not doing un-necessary calculations.
- we can express and use an infinite data structure in some recursive form, because you will only ever evaluate it to the depth/level you need, as opposed to evaluating (eagerly) in its entirety, which would be impossible.

In *fixture generation problem*, we can generate random fixtures in a way that uses *lazy eval*, however I have not used so as it will not increase any performance. We can create a team list and randomly shuffle it using a seed value using permutations. Here, on declaration of permutations nothing would be shuffled unless the team fixtures creation is called. So, this is an example of *lazy eval* feature that could have been used.

Q.2 *We can solve the problems using any imperative language as well. Do you find any advantage of using Haskell for these problems (w.r.t the property of lack of side effect)? If your answer is no, elaborate on why not?*

Imperative programming is a programming paradigm that uses statements that change a program's state, whereas Haskell is a functional programming language that is (almost) pure.

Haskell is good for a lot of things but 2 of its properties stand out which is *No Side Effects* and *lazy eval* and it’s definitely why it is advantageous for us (according to me).

Because it is (almost) pure, it is considered very safe as function states and variables do not change (they are immutable), leading to very safe and consistent code. We can be very sure what our code will be outputting and hence it helps in very fast debugging and removes uncertainty that is present with the languages like C. Removing side-effects from the equation allows expressions to be evaluated in any order. A function will always return the same result if passed the same input - no exceptions. This determinism removes a whole class of bugs found in imperative programs.

Laziness means nothing is evaluated until it is necessary. This makes sure that nothing is needlessly computed, thus saving running time.