CS431: Programming Languages Lab

Functional Programming using Haskell

Name: Annanya Pratap Singh Chauhan

Roll Number: 170101008

Q1. Write the algorithm (in pseudocode) that you devised to solve the problem

In the problem, we are given the total space, the number of Bedrooms and the number of Halls. We have to then print the most optimal design that we can print. By optimal design we mean the design that utilises the most total space. In our Algorithm, we have considered all possible room dimensions as our search space for optimal design. The main function where most of the work happens is **design.** I have explained its working below.

(PseudoCode) design Total_space, Number_of_bedrooms, Number_of_halls {

// We first decide upon the number of Kitchens, Bathrooms, Gardens, Balconies that will be constructed from the count of bedrooms and count of halls

Number_of_kitchens = Ceiling(Number_of_bedrooms/3)

Number_of_bathrooms = Number_of_bedrooms + 1

Number_of_gardens = 1

Number_of_balconies = 1

Numbers_of_Rooms = [Number_of_bedrooms, Number_of_halls, Number_of_kitchens, Number_of_bathrooms, Number_of_gardens, Number_of_balconies] *// Store these number in a list*

// Next store all possible Room dimensions in lists for all room types

All_possible_bedrooms = [(10, 10).....(15, 15)]

All_possible_halls = [(15, 10).....(20, 15)]

All_possible_kitchens = [(7, 5).....(15, 13)]

All_possible_bathroom = [(4, 5).....(8, 9)]

All_possible_balcony = [(5, 5).....(10, 10)]

All_possible_garden = [(10, 10).....(20, 20)]

// Now we will generate all valid possibilities of rooms. A possibility will be a tuple of the form (Bedroom Dimension, Hall Dimension, Kitchen Dimension, Bathroom Dimension, Balcony Dimension, Garden Dimension). These possibilities are not constructed in one shot but are constructed iteratively. In the first iteration, we construct all possible Bedroom Dimensions and Hall Dimension pairs. Only valid pairs are considered. This results in a list of tuples of the form [(Bedroom Dimension_1, Hall Dimension_1).....]. In the next iteration, we add the Kitchen dimension to these pairs. The algorithm thus progresses to the point when we add garden and balcony dimensions. Then among these valid possibilities, the best answer is chosen. Constructing possibilities iteratively helps us in optimising our method. After each iteration, we prune our generated list to remove redundant possibilities. This helps us in saving on computation time. Now I will illustrate the steps

*// In the below step we construct all possible dimension pairs of **Bedrooms, Halls**. We remove the cases where the area of the selected rooms are more than the provided total space*

Combination_of_2 = remove_false possibilities (all_combinations (All_possible bedrooms , All_possible_halls), Numbers_of_Rooms)

*// In the below step we construct all possible dimension tuples of **Bedrooms, Halls, Kitchens**. We remove the cases where the area of the selected rooms are more than the provided total space. We also take care of the fact the kitchen dimension should be less than both the bedroom and hall dimension and remove the unwanted tuples*

Combination_of_3 = remove_false possibilities (all_combinations (Combination_of_2 , All_possible_kitchens), Numbers_of_Rooms)

*// In the below step we construct all possible dimension tuples of **Bedrooms, Halls, Kitchens, Bathrooms**. We remove the cases where the area of the selected rooms are more than the provided total space. We also take care of the fact the bathroom dimension should be less than kitchen dimension and remove the unwanted tuples.*

Combination_of_4_temp = remove_false possibilities (all_combinations (Combination_of_3 , All_possible_bathrooms), Numbers_of_Rooms)

// After this step is over we try to optimise the constructed list. We group the elements by their area coverage and just keep one element for each area value. This reduces the dimensionality of the constructed list and doesn't affect our solution search as each possible area value is still present in our list. We didn't use this optimisation in the earlier two iterations as we needed to make our condition checks for our Kitchen and Bathroom dimensions.

Combination_of_4 = remove_duplicate_areas(Combination_of_4_temp)

*// In the below step we construct all possible dimension tuples of **Bedrooms, Halls, Kitchens, Bathrooms, Balcony**. We remove the cases where the area of the selected rooms are more than the provided total space.*

Combination_of_5_temp = remove_false possibilities (all_combinations (Combination_of_4 , All_possible_balcony), Numbers_of_Rooms)

Combination_of_5 = remove_duplicate_areas(Combination_of_5_temp)

*// In the below step we construct all possible dimension tuples of **Bedrooms, Halls, Kitchens, Bathrooms, Balcony, Garden**. We remove the cases where the area of the selected rooms are more than the provided total space.*

Combination_of_6_temp = remove_false possibilities (all_combinations (Combination_of_5 , All_possible_balcony), Numbers_of_Rooms)

Combination_of_6 = remove_duplicate_areas(Combination_of_6_temp)

*// Now we have all the valid possibilities with us. We now take the one with the most area coverage and **print Results***

max_area_among _possibilities = max_area (Combination_of_6, Numbers)

result_dimensions = search(Combination_of_6, max_area_among_possibilities)

print Results

}

Q2. How many functions did you use?

The number of functions I used in my code were **10**:

- design
- make_all_combinations_2
- make_all_combinations_3
- make_all_combinations_4
- make_all_combinations_5
- make_all_combinations_6
- filter_repeating_area
- answer_area
- max_area
- area

Q3. Are all those functions pure?

All functions **except design** are **pure.** The **design function is not pure** because we are performing **I/O** operations inside our design function. **I/O** operations can have side effects that's why such functions are considered impure. All other functions don't have any side effect and for a given input they produce a fixed output every time.

Q4. If not, why? (Means, why the problem can't be solved with pure functions only).

In any problem, we need to take I/O from/to the user and our problem is the same in that sense. Here **design** is an IO function. **I/O** operations can have side effects that's why **design** function is an impure function. So this problem can't be resolved by pure function only.

Short Notes on:

a. 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?

Lazy evaluation is a method to evaluate a Haskell program. It means that expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations. This feature of Haskell allows it to be used in a wide variety of fields. We can declare infinite lists in Haskell. Such structures find a lot of use in Scientific Computing. This concept of deferred evaluation is one of the backbones of processing big data. It allows us to write many operations without generating intermediate results at each step. The queries are run at the final step and the compiler can optimise these operations which would then be executed in very small time.

We also s**aw the benefits of lazy evaluation in our code**. In Task 2 we used permutation function from the List Module of Haskell. This function generates all possible permutations for a given list and returns a list of lists. Accessing any permutation in a conventional programming paradigm would have required us to first calculate all the permutations and then access the elements from the list. This would have taken a lot of time as the permutations grow very fast (12! = 479001600). But Haskell's lazy execution generates the permutations till the required index when required and thus reduces the execution time.

b. We can solve the problems using an 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?

Haskell has a lot of advantage over imperative language. Haskell's design is centred around pure functions and immutable data. Managing global state, mutable data, and side effects are error-prone, and Haskell gives the programmer an opportunity to avoid or minimize these sources of complexity. Haskell leads to a very safe and consistent code. Haskell code is very easy to debug as we are sure of code output. Haskell simplifies parallel programming as fixed values of variables remove the problem of variable assignment in the critical section.

We found Haskell very useful in our assignments especially the lack of side effects. Functional Paradigm made the code very structured and modular. Debugging the code was also very easy in Haskell as pure functions can be tested very easily in isolation.