CS431 – Programming Languages Lab

Assignment III Lab Report

Ravi Shankar (170101053)

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).?

The algorithm used to solve problem II is the *"meet-in-the-middle"* approach of competitive programming. Since, the search space of the problem is too large, thus we need to optimal approach that is time efficient and considers all the cases possible for the optimal solution. Here, we write a recursive function that searches for each room i.e., bedroom, hall, kitchen and bathroom in linear fashion and another recursive approach to search for garden and balcony on the same given area. We then sort these two lists containing the possible scenarios of design of those set of 4 room and set of 2 others. We then can simply merge both these lists together minimizing the un-used area left after the design. This method reduces the search space less range and the time limit for even the worst cases comes out to be not more than 30 second.

Pseudo Code:

1. Consider only the four rooms bedroom, hall, kitchen and bathroom, since they have a relation between them.
2. Start by generating bedrooms, and recursively generate the hall, then kitchen and then the bathroom.
3. The recursive nature of the function should be such that it accounts for increase in bedroom dimensions and then recursively the dimensions of the other three rooms based on their mutual constraints like kitchen dimension should always be less than hall and their own dimensions constraint, i.e., the dimension of room cannot exceed (X * Y).

4. Store the area corresponding to each of the plot plan, and sort them in ascending order on the basis of area. (**SET 1**)
5. Similarly repeat the steps from 2 to 3, for the next set of rooms i.e., garden and balcony, and sort this list on the basis of area in descending order. (**SET 2**)
6. Now, we have two sorted list of plans for each set of rooms which are completely independent from each other, i.e., there is no constraint on bedroom, halls etc on garden and balcony.
7. Now, apply greedy selection of the plot plan one from **SET 1** and other from **SET 2**, and merge both of them to see if its combined area is less than the given area, if it is, this can be answer, but a better answer can be choosing the next element in **SET 1**, or **SET 2** depending on the area constraint.
8. Finally, after running the greedy selection from left to right in the list, we will reach a final solution for the problem, which will obviously be optimal.
9. This algorithm may take at most 30 seconds to produce result in case of worst test case.

How many functions did you use?

I used a total of 25 functions. Some of them are – buildBedroom, buildHall, buildKitchen, buildBathroom, buildBalcony, buildGarden, check_bedroom, check_hall, check_kitchen, check_bathroom, check_balcony, check_garden, etc.

Are all those pure?

No, all of them are not pure Haskell functions.

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

Here in my case, there are recursive functions which are impure Haskell functions. Since the sample search size is too large for one to write all possible cases by hand and thus would need a recursive kind to function to cover all possible cases, which will output different values depending on the different cases. So, this cannot be solved by only pure functions.

However, if the sample search space would have been less, one could have written simple functions to calculate the un-used area for those cases (dimensions of different rooms) and thus, can be solved only by pure functions, but in our case the sample size was huge, and thus, only possible way to go was by recursive algorithm, which isn't pure in this case.

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, the lazy evaluation feature of Haskell is exploited for better performance in the solution to the assignments.

One of its very important features is to deal with infinite data structures. In practice they would require an infinite amount of time to evaluate, and it not even possible to have it in languages like C, C++ or python. However, lazy evaluating allows us to compute only portions of the structure rather than the whole. This performance improvement is helpful in all assignments where we have used lists.

The lazy feature of Haskell also saves the computation time from computing un-necessary conditions which aren't required for the execution of the particular function. For example, I may pass three values into a function, but depending on the sequence of conditional expressions, only a subset may actually be used. In a language like C, all three values would be computed anyway, but in Haskell, only the necessary values are computed.

In the assignment Problem 2, when we shuffle the given array, and say we need to find the first fixture, then Haskell will compute only till the first element and will not care for the rest. Similarly, if we sort a list and need the first element, Haskell lazy feature will calculate only that and thus, will save computation time.

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?

Haskell is a high-level programming language which provides basically a birds' view or top view to the other programming languages. In computer science, an operation, function or expression is said to have a side effect, if it modifies some state variables outside its local environment, that is to say has an observable effect besides returning a value to the invoker function. For example, modifying a non-local variable, modifying a static local variable, performing IO or calling other side effect functions.

Haskell is a functional programming language which unlike any other imperative languages, rarely uses side effects. The lack of these side effects makes it easier for the programmer to do formal verifications of the program. For example, the programmer can now easily debug the program knowing the exact changes a procedure does instead in an imperative language where a simple function can have varying side effects making it harder to keep track of by the programmer.

So, yes Haskell's lack of side effects is a property which I as a programmer find very advantageous.