CSCI 239—Discrete Structures of Computer Science

Lab 5—Haskell Data Structures

This lab will explore data structures in Haskell. It will introduce modules as a way to encapsulate code in files. Students will implement a Set data structure along with associated functions

Objectives:

- to gain experience writing Haskell modules and data structures
- to explore the implementation of a computational model of a mathematical structure

Preliminaries

Copy the folder `/usr/people/classes/CS239/labs/Lab05_DataStructures` and its contents into your *CS239* directory. Open the files *Set.lhs* and *TestSet.lhs* from that directory in *gedit* or another text editor and follow the directions below. Put your Haskell code into the appropriate file.

Part 1: Haskell Modules

Like most modern languages, Haskell has facilities for encapsulating code into coherent packages to facilitate reuse in other contexts. The Haskell *module* is similar to a Java class, but also has features of a Java package. Look at the *Set.lhs* file. After the initial heading material, its Haskell code begins with a *module* declaration that consists of the keyword `module`, the module name (*Set*), a list (in parentheses) of the items exported from the module, and the keyword `where`. Everything after this in the file defines the exported items in Haskell code.

Now compare this to the file *TestSet.lhs*. This file is also a module definition, as you can see from the first Haskell code. The TestSet module, however, doesn't export anything, because it doesn't implement anything that's likely to be of use anywhere else. It does import the Set module, and so everything exported by the Set module is visible to the TestSet module, which makes sense, since the purpose of the TestSet module is to test the Set module.

In this lab, you will be developing code in the Set module, and using the TestSet module to test your code (so you don't have to keep typing those tests over and over). To use the TestSet module, you just load it in *ghci* as you would any other Haskell file, and since the Set module is imported, you now have access to the code in both files. In fact, importing modules into other files is the only way to have access to more than one Haskell file at a time in *ghci*.

Part 2: Testing a module with stub functions and a test suite

Load the file *TestSet.lhs* into ghci. Oh! No! A bunch of error messages! The first one is a “Not in scope” error for the symbol “subset” in line 30 of *Set.lhs*. If you look at line 30, you won't see an error, because there isn't one there. Instead, scan forward until you find another reference to “subset” in line 127; it contains an incomplete function definition, because it's going to be your job to complete this function and several others. In order to begin testing code as you implement it, rather than waiting to test everything until the end, we need to have everything implemented. Wait! This sounds like a contradiction! We want to test before we implement, but we can't test until after we implement. The solution is to use stub implementations.

A *stub implementation* is an implementation that does not seek to compute the correct result, but does behave as if it were doing so. For a function (which is all we have in Haskell), a stub implementation is just to set up some dummy parameters and then return a value (any value) of the correct type. For example, we need a stub implementation of the *subset* function; the definition `subset set1 set2 = False` fills the bill; it doesn't actually tell whether *set1* is a subset of *set2* or not; it just says “No.” Try putting this definition in, and verify that you no longer get the error in line 30.

Now put stub implementations in all the other functions that have fill in the blank comments. (You may want to keep the comments to identify the places you need to go back and finish.) Each stub implementation is a one-line definition with dummy parameters (variables rather than patterns) and a dummy return value such as *False* for *Bool*, 0 for a number, [] where a list is returned, and *emptySet* where a set is returned. If you do this correctly, you should be able to load the file without syntax errors. Keep at it until you can.

Once the file loads without errors, you can begin testing. At this point, the only things you can really test are the parts you've been given, which have already been tested. Try the *showSets* function. It shows the result of using the set constructor and factory functions to make a few set objects. Verify that the sets it shows are the ones that should result from the operations that create them. The size values will be wrong, but that's OK for now since you haven't implemented the *size* function yet.

Note: There are several tests in the *TestSets.lhs* file, and they are intended to be fairly thorough, but they are not exhaustive. If you feel there are important test cases that have not been included, feel free to add more of your own, but please leave the ones that are there to help verify your implementations. Please bring any problems with the test file to the instructor or a TA's attention so the file can be improved.

Part 3: Completing the Set module

If you have not already done so, put your name and today's date in place of line three of the *Set.lhs* file. Since you will be implementing some of the code, you can claim some of the authorship credit.

the *size* function

Start with the *size* function around line 130 in the file; this is the easiest function to implement. Use the type signature to determine the parameters and return type: the function takes one parameter of type *Set a*, and it returns an int. The size of the set is the same as the length of the internal list, so you need to take advantage of that equivalence with a pattern for the parameter consisting of the set constructor and its associated list. (See, for example, the parameters of the *union* function around line 84.) The function is defined to be the length of the list part of the parameter pattern. Save your work, reload the *TestSet.lhs* file, and try the *showSets* function; it should now display the correct set sizes. When this function is working, go on to the next function.

the *setMember* function

The next function you need to implement is the *setMember* function on about line 75 of the file. Again, use the type signature to work out the function parameters. The function takes two parameters: the element to search for and the set of elements to search. Since the return value is true when the search element is in the internal list of the set, you need a pattern again for the set parameter. (Hint: remember that the *elem* function determines whether a value is an element of a list.) Once you get the syntax right for this function, you can use the *showMembers* function to determine if it is working right.

the *union*, *intersection*, and *difference* functions

Now we turn to the primary set functions: *union*, *intersection*, and *difference*. Note first, that all three of these functions have the same type signature. Thus, there parameter patterns can probably be very similar or even the same. Why don't you just copy the function definition for *union* into the appropriate spots for *intersection* and *difference*, changing the function names where appropriate? (Note that the definitions of *union* and *listUnion* are given.)

Study the *union* function carefully. It really doesn't do much. Instead, it calls the *listUnion* function to do the heavy lifting. What the definition says is that we compute the union of a set of *xs* and a set of *ys* by computing the union of the two lists (without any duplicates) and then making a set of the resulting list. Why do we do it this way? Because to write *union* as a stand-alone function would require a lot of constructing and deconstructing lists that would make the code hard to understand and hard to debug. Our approach is to write an *auxiliary* function that does the recursive part of the function in a simpler way, and then to use the auxiliary result to construct the final value.

To better understand how this works, study the *listUnion* function next. This function treats two lists as if they were sets and computes a list with all the elements in both lists, but without duplicating the elements in their intersection. (We're not concerned about the two parameters containing duplicates because this function is not exported, and we never call it with lists that contain duplicates.) The type signature of *listUnion* says that it takes two lists of the same type of elements and returns a third list of the same type, where the element type is in the *Eq* type class. There are three top-level cases: if either of the two parameter lists is empty (two cases), then the union of the two lists is whatever is in the other list (i.e., the list itself); if both parameter lists contain elements, there are two sub-cases. The first sub-case is when the first element of the first parameter list is also present in the second list; in this case, we can discard that element and return the union of the rest of the first list with the second list. The second sub-case is when the first element of the first parameter list is not present in the second list; in this case, we return that element followed by the union of the rest of the first list with the second list. This recursive method proceeds by testing all the elements of the first list, one-by-one for membership in the second list, putting those that are not in the second list at the front of the result, until the first list is reduced to the empty list; at that point, the function switches to the first top-level case and returns the second list (and, as the recursion rolls back, all those from the first list it identified as not in the second on the way down). If the recursion seems ineffable, don't sweat it. Instead, concentrate on understanding the structural pattern of the *union* and *listUnion* functions.

the *intersection* function

With some understanding of how the union process works, let's turn to the *intersection* function and its auxiliary function. The definition that you now have for *intersection*, the same as for *union*, should work fine as long as you change the call to *listUnion* to a call to *listIntersection*. The *listIntersection* function requires a little more work, but it *does* have the same recursive structure, so you can start by copying the definition parts of the *listUnion* function into the definition part of the *listIntersection* function (that is, everything except the type signature line). If you test the *intersection* function now, it should return exactly the same thing as the *union* function, since it's defined the same way. Consider the first two cases where one of the two parameter lists is empty: in an intersection, what should the result be? Change the definitions for the first two cases accordingly. The two sub-cases of the third case also need to change. We still want to process the elements of the first parameter list one-by-one until the first list is empty, but now we want to keep the elements that are also in the second list and discard those that are not. How does that change the definition of the two sub-cases? Debug the syntax of your functions and then test them with the *showCombinations* function. (Note that any recursive calls should be to *listIntersection* in this function.) You should see both the union and the intersection of the specified sets correctly displayed. If your results are not correct, you've probably made a logic error in defining *listIntersection*; try to work out what's wrong on your own before asking for help.

the *difference* function

The *difference* function and its auxiliary function again have the same structure as the other two pairs, so you can proceed as you did for the *intersection* function. Recall that set difference is not a symmetric function like union and intersection. What is the difference between an empty list and another list? What is the difference between a non-empty list and the empty list? Implement your definition of the first two cases accordingly. For the third case and its two sub-cases, consider the question: which elements of the first parameter list are part of the result, those in the second list or those not in the second list? Again, implement according to your answer. Now rinse and repeat until you get correct results.

the *subset* function

We'll finish with an easier one; it's straightforward enough that we don't need an auxiliary function. There are two cases: the empty set is a subset of any other set (including itself); a set with at least one element is a subset of a second set if its first element is a member of the second set and the set consisting of the rest of its elements is a subset of the second set. One comment about the second parameter: you never explicitly look at the members of the second set, so you don't need a complicated pattern for the second parameter, just a simple variable will do. When you have finished testing this function, all the test functions in the test set module should give you correct results.