# CSCI 239—Discrete Structures of Computer Science

# Lab 4—Haskell Functions 1

This lab will explore functions in Haskell, from the point of view of mathematical functions.

### Objectives:

- to gain experience writing Haskell functions
- to learn to use the ideas of domain and co-domain to improve the correctness and reliability of code
- to continue developing the ability to use pattern matching and recursion in Haskell functions

### Preliminaries

Copy the folder /usr/people/classes/CS239/labs/Lab04_Functions1 and its contents into your CS239 directory. Open the file CSCI239Lab04.Functions1.lhs in that directory in *gedit* or another text editor and follow the directions in that file. Put your Haskell code into that file.

Run *ghci* in a terminal window in the same directory as this file, which should be in your *CS239/Lab04* directory.

### Part 1: Domains and targets in Haskell functions

Haskell has this wonderful feature that is can (usually) figure out the types of parameters and return values of functions without the programmer's help. It is often best to write your function without a type specification and then use the `:type` command to see what Haskell thinks the type is.

Consider the following one-line function that returns *True* if one of the two parameters divides the other. What do you thing the type of this function should be?

```
oneDividesTheOther n m = n `mod` m == 0 || m `mod` n == 0
```

Load this file into *ghci* and use the `:type` command to see what Haskell infers the type to be. You might have reasonably predicted that Haskell would infer the type `Int -> Int -> Bool` or `Integer -> Integer -> Bool`. (Recall that *Int* is a fixed precision integer type and *Integer* is an unlimited precision integer type.) Haskell recognizes that the function matches both of these types; in fact, the function matches any type where the two parameters have the same type and that type supports the `mod` operator. The type class *Integral* includes all such types, so Haskell infers the most general possible type: `Integral a => a -> a -> Bool`, meaning that the two parameters may be any type in the type class *Integral*, but the return value must be *Bool*.

Here is the function with its type:

```
oneDividesTheOther :: Integral a => a -> a -> Bool
oneDividesTheOther n m = n `mod` m == 0 || m `mod` n == 0
```

Now test the function with several different values for *n* and *m*; include cases where neither divides the other, *n* divides *m*, *m* divides *n*, both are equal, and either or both are 0. Notice that the function fails due to division by zero whenever *m* is zero, but it succeeds, returning *True*, when *n* is zero but *m* is not zero. (Why is that?)

Haskell functions are very much like mathematical functions, but there are a few differences. Mathematically, we say that the *domain* of a function is the set of all parameter values that the function is defined for. So the function as implemented has the domain { $n \in Z, m \in Z : m \neq 0$ }. The Haskell type system, on the other hand, says that the domain is all integers *n* and *m*. It cannot detect that there is a value of *m* that is not in the domain.

So, we have discovered a bunch of parameter values for which the function doesn't work correctly. We need to decide whether to live with that or if we need to fix it. `oneDividesTheOther 0 10` returns *True* because `0 `mod` 10 == 0`; `oneDividesTheOther 10 0` ought to return *True* for the same reason, but it fails because trying to compute `10 `mod` 0` causes division by zero, and that raises an exception. The function is behaving inconsistently, and that should be fixed. The immediate fix is to add a special case before the general case for when *m* is zero:

```
oneDividesTheOther n 0 = True
```

This causes the function to behave consistently, and it is now defined for all values in its domain. We're not quite done, however. We need to go back and retest the function for all values we've already tested. We find now that everything works as we want it to, except possibly when `oneDividesTheOther 0 0` returns *True*. Is this what we want? Division by zero is usually considered undefined, so shouldn't `oneDividesTheOther 0 0` return *False*? We can fix this by adding one more case before the other two:

```
oneDividesTheOther 0 0 = False
```

Haskell pattern matching to the rescue again! Of course, you could write a Java method to do the same thing, and you could fix it by doing parameter checking inside the method body, since Java doesn't have pattern matching. In either case, however, the thinking that we do about the domain of the function can help us to write reliable code.

Let's explore this further by developing a function that computes the roots of the quadratic equation: $ax^2 + bx + c = 0$. Be warned that parts of this will get messy.

We know that the general solution is $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, but that we can have two real roots, one real root, or two complex roots, depending on the values of *a*, *b*, and *c*. Let's start by ignoring the possibility of complex roots, knowing that doing so will limit the domain of the function.

We still have to deal with the fact that in most cases, there are two roots. (When $b^2 - 4ac$ is zero, we have only one root, but a mathematician will tell you that's still one root "of multiplicity two.") A function in Haskell (and most languages) can only return one value, but the value returned can be a structure, so we can use a tuple to return the two roots.

### *Part 1.a*

> Write a Haskell function *quadRoots1* that takes three parameters *a*, *b*, and *c*, and returns a tuple containing the values $\frac{-b+\sqrt{b^2-4ac}}{2a}$ and $\frac{-b-\sqrt{b^2-4ac}}{2a}$ as its two elements. This should be a function with no cases, for now.

Test your function on the following values for *a*, *b*, and *c*: 1 5 4, 1 4 4, 1 6 10. You should get two distinct roots in the first set, two equal roots in the second, and an error in the third.

Use the `:type` command to see what Haskell infers as the type of this function, and verify that it makes sense.

Now let's think about the domain of this function. It appears to be { *a b c* : *a*, *b*, and *c* are real numbers }. We have found, as we expected, that there are some values of the parameters that don't give the expected values. It's time to handle the cases where the roots are complex, that is, numbers of the form *x* + *yi* where *i* is the square root of minus one, a so-called imaginary number. Now, however, we need to consider the *target* (or *co-domain*) of the function. The first version has returned a pair of real (floating-point) values; now we want to be able to return a pair of complex values when the equation has complex roots. Unfortunately, a function cannot have two different targets. In this case, that's not a problem since the real numbers are a subset of the complex numbers. (Every real number *x* can be expressed as the complex number *x* + 0i.) Therefore, we need to modify the function to return a pair of pairs.

### *Part 1.b*

> Write a new Haskell function *quadRoots2* that modifies *quadRoots1* so that instead of returning a tuple in the form ($r_1$, $r_2$), it returns a pair of pairs in the form (($r_1$, 0), ($r_2$, 0)).

Test it using the same values as before, and verify that its behavior is unchanged except for the form of the return value. Check the type again to see how it has changed; the domain should be the same, but the target should be a more complicated type.

We still aren't actually computing the complex roots. To do that, we need to break the function into the two cases: $b^2 - 4ac < 0$ and $b^2 - 4ac \geq 0$. The function currently computes the second case. For the first case, the two roots are the pairs $(\frac{-b}{2a}, \frac{\sqrt{4ac-b^2}}{2a})$, $(\frac{-b}{2a}, \frac{-\sqrt{4ac-b^2}}{2a})$.

### *Part 1.c*

> Modify your *quadRoots2* function to handle the two cases. Since the cases are based on a test that must be computed, you can't use pattern matching on the parameters to separate the cases. Instead, use the | case structure:

```
foo x y
  | <computed condition1> = ...
  | <computed condition2> = ...
 ...
  | otherwise             = ...
```

For this function you have just one computed condition and the otherwise definition. Be careful to maintain consistent indenting, since Haskell, like Python, uses indenting to infer the meaning

of the code. Test your function again using the same test cases. You should now get correct answers for all three cases. Check the type again; if it has changed since you did 1.b, try to explain why (or why not).

Testing a function like this one with only three test cases is not usually sufficient. Try a few more test cases. What happens if *a* is zero? If *a* is non-zero but *b* is zero? Etc.?

Your function should succeed whenever *a* is non-zero, but fail when *a* is zero. That's because your computation always divides by 2*a*, so when *a* is zero, there is a divide by zero exception. The equation we're trying to solve is no longer quadratic; it has become $bx + c = 0$, which has one real root: $-c/b$. However, if both *a* and *b* are zero, the equation becomes $c = 0$, which is a proposition that is either *true* or *false* for all values of *x*, depending on whether *c* is zero or not.

We could add another case to our function for the situation where a = 0, but there's no easy way to handle the cases when both *a* and *b* are zero. In this situation, we'll leave this as it is and accept that the domain for parameter *a* is all floating-point values except zero, while the domains for *b* and *c* are all floating-point values.

## Part 2: The domain of functions on lists

Let's consider another example where the function domain needs to be restricted:

```
minList :: Ord a => [a] -> a
minList (x:[])  = x
minList (x:xs)
  | x < minRest = x
  | otherwise   = minRest
  where minRest = minList xs
```

Before we discuss this function, there's one bit of new Haskell syntax to introduce. We could have defined the function as follows:

```
minList :: Ord a => [a] -> a
minList (x:[])  = x
minList (x:xs)
  | x < (minList xs) = x
  | otherwise        = minList xs
```

This version is almost the same, but it includes the call "`minList xs`" twice, which means we're doing a lot of computation multiple times, not just twice, but twice for every recursive call, so the overall performance is exponential, and therefore *s l o w*.

The *where* clause in the first version allows us to assign names to a particular value and use those names in the code immediately above at the same indentation level. The *where* clause is evaluated before the choice is selected, so names defined in the clause are available throughout the preceding code *at that indentation level*. In this case, we now perform the computation `minList xs` just once for each level of recursion, and our performance time is directly proportional to the length of the list, which is the best we can hope for.

Now try testing the function a few times with different lists to verify that it works. Well, it works except when the list is empty; in this case, there is an exception for "Non-exhaustive patterns in function *minList*." Basically, the error message means that there is a parameter pattern that our function can't match—the empty list. Stop to think what it means to find the minimum element in an empty list. There is no such element. For our purposes here, we don't want to return a meaningless dummy value, so it's OK to have this behavior. The domain of our function is all lists of ordinal values, except the empty list. (The Haskell type class *Ord* includes all types that support the inequality operators like "<".)

*Part 2.a*

> Using *minList* as a model, write and test a function *maxList* that returns the maximum element in a list.

*Part 2.b*

> Write a Haskell function *median1st3* that returns the median of the first three or fewer elements in a list as follows. If the list has only one element, it returns that element; if the list has two elements, it returns the first; if the lists has at least three elements, it returns the middle of the first three. Note that you can use the pattern `x:[]` for a list with one

element, and `x:y:[]` for a list with exactly two elements and `x:y:z:xs` for a list with three or more elements. In the case with three or more elements, you'll need to compare them to determine which is in the middle. Test your function, and note that again, it doesn't work for empty lists.

*Part 2.c*

Now write a function *medianAll3s* that returns a list of the medians of every three elements of a list. If the list is empty, it returns the empty list; if the list contains one or two elements, it returns a list containing first element of the list; if the list has at least three elements, it returns the *median1st3* of the list followed by the *medianAll3s* of the rest of the list. Test this with several lists of varying lengths and orders of elements. In this case, the function should work for all lists of ordinal values.

*Part 2.d*

Write a function *medianOfMedian3s* that repeatedly calls *medianAll3s* until it has reduced its list parameter to a single element. If the list has one or two elements, it returns the first element; if the list has at least three elements, it calls itself recursively, passing the *medianAll3s* of the list as its parameter. Test this function, and note again that it does not work on empty lists. Does this function actually find the median of the original list?

This technique of finding the median of medians is a well-known algorithm, and it can be used to improve the performance of certain algorithms (such as *QuickSort*) that depend on choosing a value close to the median of a list.

## Part 3: Sorting a list

In this part you will write a function that implements the *MergeSort* sorting algorithm. The algorithm is simplicity itself: split the list into two approximately equal length lists; sort the two sub-lists (using *MergeSort*); merge the two sorted lists into one sorted list.

*Part 3.a*

To split a list of *n* elements, we could try to take the first *n*/2 elements for the first list and then use the remaining elements for the second list. It's easier in Haskell, however, to take alternate elements from the list and put them in the two lists. That way, we only have to traverse the list once, since we don't need to know how long it is without counting.

Write a Haskell function *splitList* that takes a list parameter and returns a pair of lists that together contain exactly the elements of the original list, evenly divided between the two new lists.

- If the original list is empty, it returns the pair ([], [])
- If the original list has one element *x*, it returns the pair ([*x*], [])
- If the original has two or more elements, it returns the pair (x:xs, y:ys) where *x* is the first element of the original list, *y* is the second element, and (*xs*, *ys*) is the pair of lists returned by splitting the rest of the original list.

Test your function and check its type.

Note that the *splitList* function should work on any types of lists, and the function type will confirm this. That is, there will be no context restrictions on the type of the elements of the lists.

### *Part 3.b*

Next, we need a function that will merge two sorted lists into a single sorted list. This function is almost the inverse of the *splitList* function, except that it always chooses to put the smaller of the first two elements at the head of the list.

> Write a function *mergeLists* that takes two *sorted* lists as its parameters and returns a single list containing exactly the elements of the two lists in sorted order.
>
> - If the first list is empty, it returns the second list
> - If the second list is empty, it returns the first list
> - If both lists contain at least one element, it returns the smaller of the two first elements followed by the merge of the remaining two lists. Note that this case has two sub-cases:
>   - If the smallest element is the head of the first list, it merges the remaining elements of the first list and the whole second list
>   - Otherwise, it merges the remaining elements of the second list with the whole first list.

Again, test this function and check its type. Note that if the input lists are not already sorted, the result will not be sorted either, but that's not your problem. You should see that the *mergeLists* function type specifies that the list elements must be in the *Ord* class, since it performs comparison of elements. We don't have a way to specify that the lists must already be sorted and enforce that on the function, however, so we just say that the domain of the function is any two sorted lists of the same type.

Finally, we put it all together with a function to sort the original list. Since you've worked so hard already, I'll just give it to you. Note that the > signs are omitted in the Haskell file, since it won't compile until you've done the last two functions. All you need to do at this point is put the > signs in (and make sure the indenting is consistent).

```
mergeSort :: Ord a => [a] -> [a]
mergeSort []     = []
mergeSort (x:[]) = x:[]
mergeSort (x:xs) =
  mergeLists sl1 sl2
  where
    (l1, l2) = splitList (x:xs)
    sl1      = mergeSort l1
    sl2      = mergeSort l2
```

Test the *mergeSort* function on a few lists to verify that is sorts correctly. If it doesn't, there must be a problem with one or both of the two functions you supplied.