15-150 Fall 2015
Homework 05

Out: Wednesday, 11 February 2015
Due: Tuesday, 17 February 2015 at 23:59 EST

1 Introduction

This homework will focus on applications of higher order functions, polymorphism, and user-defined datatypes.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository, as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at `https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/05` directory should contain a file named exactly `hw05.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/05` directory (that contains a `code` folder and a file `hw05.pdf`). This should produce a file `hw05.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw05.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number corresponding to the "check" section of your latest handin on the "Handin History" page. **If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.**

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

All the code that you want to have graded for this assignment should be contained in `hw05.sml`, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 17 February 2015 at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, specify the type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments to be passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns when applied to an argument that satisfies the assumptions in `REQUIRES`).
4. Implement the function.
5. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n! *)
fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)
val 1 = fact 0
val 6 = fact 3
val 720 = fact 6
```

1.5 Style

For this assignment, we will be grading your submissions based on your coding style. There are several ways that you can learn what is good style and what isn't:

- Your returned and graded homework submissions have been graded for style so use the markups for a reference.
- We have published solution code for the previous assignments, labs, and lectures.
- We have published a style guide at

  `https://www.cs.cmu.edu/~15150/resources/handouts/style.pdf`.

  There is also a copy in the `docs` subdirectory of your git clone.
- You can ask your TAs about specific examples, or post on Piazza asking general questions.

2 Types and Polymorphism

In class we discussed typing rules. In particular:

- A function expression `fn x => e` has type `t -> t'` if and only if, by assuming that `x` has type `t`, we can show that `e` has type `t'`.
- An application $\texttt{e}_1$ $\texttt{e}_2$ has type `t'` if and only if there is a type `t` such that $\texttt{e}_1$ has type `t -> t'` and $\texttt{e}_2$ has type `t`.
- An expression can be used at any instance of its most general type.

Task 2.1 (4 pts). Consider the following ML function declaration:

```
fun Woof (play, doggy) =
    case play of
      go::FETCH => go::Woof(FETCH, doggy)
    | _ => [doggy]
```

What is the most general type of `Woof`?

Task 2.2 (3 pts). Now consider the following function expression:

```
fn x => fn y => x y
```

What is its most general type?

Task 2.3 (3 pts). Now look at this slightly different expression:

```
fn x => fn y => x (y + 2)
```

What is the most general type of *this* expression?

3 Rationals

For the next few problems, we will introduce a user-defined type named `rat`, a representation of rational numbers with the following type definition:

```
type rat = int * int
```

A rational number (or "fraction") can be represented by a pair of integers (a, b), where $b \neq 0$ and a and b have no common factors. For example, $(1, 2)$ is a valid representation for "one half", commonly written in math notation as $1/2$ or $\frac{1}{2}$. However, $(2, 4)$ is not a valid representation, because 2 and 4 have a common factor of 2. We define the rational number "zero" to be represented as $(0, 1)$. This is the *only* pair whose "numerator" is 0. [1]

A pair of integers (a, b) is *valid* if and only if $b \neq 0$ and the greatest common divisor of $|a|$ and $|b|$ is 1. A pair (a, b) represents the fraction a/b.

We've provided an implementation of rationals, using pairs behind the scenes, but you may use only the functions provided to work with them. In order to enforce that all rationals are valid, the library code will not allow you to call just *any* pair of integers a rational number. Instead, we have provided an infix function `//` that takes two ints and returns a value of type rat. All rationals in your homework file should be defined this way. So in your homework file you should write:

```
val zero:rat = 0 // 1
val one:rat = 1 // 1
val half:rat = 1 // 2
```

The returned value will *not* look like a pair (and you won't be able to pattern-match against it like you would other pairs.)

To help with the next few problems, we have given you a few helper functions:

3.0.1 Plus

```
++ : rat * rat -> rat
```

For all valid pairs of integers (a, b) and (c, d), `(a//b) ++ (c//d)` returns a valid pair (p, q) such that $a/b + c/d = p/q$. This pair is called the (rational) sum of (a, b) and (c, d). Note: `++` is infix.

Examples:

```
half ++ half = 1 // 1
one ++ one = 2 // 1
```

[1] For a fraction written as $\frac{a}{b}$ we usually call a the *numerator* and b the *denominator*.

3.0.2 Times

```
** : rat * rat -> rat
```

For all valid pairs of integers (a, b) and (c, d), `(a//b) ** (c//d)` returns a valid rational (p, q) such that $(a/b)(c/d) = p/q$. This pair is called the (rational) product of (a, b) and (c, d). Note: `**` is infix.

Examples:

```
one ** one = 1 // 1
half ** half = 1 // 4
```

3.1 More Rational Operations

You can subtract rationals with `--` (infix), and divide with `divide` (*not* infix). You can negate rationals with ~~ (prefix, like a normal SML function).

3.2 Rateq

Since you won't be able to pattern match against rationals, you should use `rateq` to test rational values.

```
rateq : rat * rat -> bool
```

For all valid rats $r1$ and $r2$, `rateq`$(r1, r2)$ returns true if $r1 = r2$.

Examples:

```
rateq(one, one) = true
rateq(half, one) = false
rateq(1//2, 2//4) = true
```

4 Integration and Differentiation

We can represent a polynomial $c_0+c_1x+c_2x^2+\ldots$ as a function that maps a natural number, i, to the coefficient c_i of x^i. For these tasks we will take the coefficients to be rational numbers (of type `rat`). Therefore, we have the following type definition for polynomials:

```
type poly = int -> rat
```

Where `rat` is a rational as defined in the previous section. For example, examine:

```
fun p x =
    case x of
      0 => 1 // 1
    | 1 => 1 // 2
    | 2 => 7 // 1
    | _ => 0 // 1
```

The function `p` would represent the polynomial $1 + \frac{1}{2}x + 7x^2$.

As an example of how to define operations on polynomials defined in this manner, see the following definition for the addition of polynomials:

```
fun plus (p1 : poly, p2 : poly) : poly = fn e => p1 e ++ p2 e
```

Also note the functions such as

```
polynomialEqual : poly * poly * int -> bool
```

in `lib.sml`. Just like the rationals, you will not be able to pattern match against polynomials. Therefore, some of the lib functions can help you test your code.

4.1 Differentiation

Recall from calculus that differentiation of polynomials is defined as follows:

$$\frac{\mathrm{d}}{\mathrm{d}x}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}$$

Task 4.1 (5 pts). Define the function

```
differentiate : poly -> poly
```

that computes the derivative of a polynomial using the definition above. Note that `differentiate` should *not* be recursive.

4.2 Integration

Recall from calculus that integration of polynomials is defined as follows:

$$\int \sum_{i=0}^{n} c_i x^i \,\mathrm{d}x = C + \sum_{i=0}^{n} \frac{c_i}{i+1} x^{i+1}$$

where C is an arbitrary constant known as the constant of integration. Because C can be any number, the result of integration is a family of polynomials, one for each choice of C. Therefore, we will represent the result of integration as a function of type `rat -> poly`.

For example,

$$\int 2x \,\mathrm{d}x = C + x^2$$

so if `p` is the `poly` representation of the polynomial $2x$, `integrate p (150 // 1)` should evaluate to the `poly` representation of the polynomial $150 + x^2$.

Task 4.2 (5 pts). Define the function

```
integrate : poly -> (rat -> poly)
```

that, given a polynomial, computes the family of polynomials corresponding to its integral. Note that `integrate` should *not* be recursive.

5 Specifications and Proofs

In this question you will be using the specifications of helper functions to prove the correctness of the following implementation of merge sort.

```
(* sort : int list -> int list
 * REQUIRES true
 * ENSURES sort L = a sorted permutation of L
 *)
fun sort L = merge_all (map (fn x => [x]) L)
```

Task 5.1 (8 pts). Recall the function `merge` which combines two sorted lists into a single sorted list:

```
(* merge : int list * int list -> int list
 * REQUIRES: L1, L2 are sorted lists
 * ENSURES: merge(L1, L2) = a sorted permutation of L1@L2
 *)
fun merge ([ ], L2) = L2
  | merge (L1, [ ]) = L1
  | merge (L1 as x1::R1, L2 as x2::R2) =
      if x1 <= x2 then x1 :: merge (R1, L2) else x2 :: merge (L1, R2)
```

The following `merge2` function has a different specification based around k-uniformity. A list of lists is k-uniform if all of its members have length k except for the last member, which may be shorter. It follows that the empty list and the list containing a singleton are both vacuously n-uniform, where n is any natural number.

```
(* merge2 : int list list -> int list list
 * REQUIRES: Ls is a k-uniform list of sorted lists
 * ENSURES: merge2 Ls = a 2k-uniform list of sorted lists
 *          consisting of the integers from Ls
 *)
fun merge2 [ ] = [ ]
  | merge2 [L] = [L]
  | merge2 (L1 :: L2 :: R) = merge (L1, L2) :: merge2 R
```

Using the specification of `merge`, prove that `merge2` satisfies its specification.

You may use the following Lemma in your proof.

Lemma 1. *For all lists* `A` *and* `B`, `length (A @ B) = length A + length B`, *where* `length` *is defined in the usual sense.*

Hint: Note that we aren't asking you to prove anything about `merge`. You can assume it's correct and that the postcondition follows from the precondition.

Task 5.2 (10 pts). Prove that `merge_all` shown below satisfies its specification, assuming that `merge2` satisfies its own specification.

```
(* merge_all : int list list -> int list
 * REQUIRES Ls is a k-uniform list of sorted lists
 * ENSURES merge_all Ls = a sorted list consisting of the integers from Ls
 *)
fun merge_all [ ] = [ ]
  | merge_all [L] = L
  | merge_all Ls = merge_all (merge2 Ls)
```

You may use the following Lemmas in your proof.

Lemma 2. *For all lists* `L` *such that* `length L = n`*, where* `n >= 2` *and* `length` *and* `div` *are defined in the usual sense, then*

`length (merge2 L) = n div 2` *if* `n` *is even and*
`length (merge2 L) = (n div 2) + 1` *if* `n` *is odd*

Lemma 3. *For all values* `n : int` *such that* `n >= 2`*, we have that* `(n div 2) + 1 < n`*.*

Task 5.3 (2 pts). Is the function `sort` a correct sorting function for integer lists? Explain briefly why or why not.

6 Trees

Recall the definition of polymorphic trees from lecture:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

Task 6.1 (4 pts). Define a "leaf" as a node with two empty subtrees.

Write a function

```
leaves :  'a tree -> 'a list
```

such that `leaves T` evaluates to a list of all the elements in `T` that are leaves.

Task 6.2 (4 pts). Now define the "interior" of a tree to be the tree resulting from deleting all its leaf nodes.

Write a function

```
interior :  'a tree -> 'a tree
```

such that `interior T` evaluates to `T'`, where `T'` is the interior of `T`.

Task 6.3 (4 pts). We say a tree is "full" if each node has either 0 or 2 non-empty children.

Write a function

```
is_full :  'a tree -> bool
```

such that `is_full T` evaluates to true if `T` is full, and false otherwise.

Task 6.4 (3 pts). Write recurrences for the work and span of `is_full` and and state the big-O for each. You may assume that we are working with balanced trees (i.e. the left and right subtrees differ in depth by at most one).

7 Higher-Order Functions

Recently we introduced two new language features: polymorphism and higher-order functions. In this problem, you will use these new tools to write a simple ML function.

Task 7.1 (15 pts). Write the function

```
rotate : 'a list list -> 'a list list
```

that rotates the rows and columns of a list of lists counter-clockwise. For example,

```
rotate [[1,2]] ==> [[2],
                    [1]]


rotate [[1,2], ==> [[2,4],
        [3,4]]      [1,3]]


rotate [[1,2],
        [3,4], ==> [[2,4,6],
        [5,6]]      [1,3,5]]
```

Also, if the inner lists are all empty, the function should return the empty list. For example,

```
rotate [[], [], []] ==> []


rotate [[]] ==> []


rotate [] ==> []
```

Your function only needs to work if all the inner lists have the same length, so for example

```
rotate [[1,2],[3]]
```

can have whatever behavior you find most convenient. Your implementation of `rotate` may use recursion, but should use higher-order functions where possible.

8 Blocks World

In artificial intelligence, *planning* is the task of figuring what an agent (a robot, Siri, your roommate, etc.) should do. One way to solve planning problems is to simulate the state of the agent, and what moves it can do, so that you can simulate plans, and then search through potential plans for good ones.

A simple planning problem, which is often used to illustrate this idea, is *Blocks World.* The idea is that there are a bunch of blocks on a table and a move consists of transferring a block from the top of a pile either onto the top of another pile or onto the table itself. Of course, you cannot put a block on one that already has something on it.

We will represent Blocks World using some types and datatype definitions in ML.

- We represent each block as a positive integer, and we use 0 for the table.
- We will represent the state of the world as a list of facts. There are two kinds of facts:
  - Clear x, meaning that block x has no other block on top of it and so is available to be picked up and moved
  - On(x,y), meaning that block x is on top of block y. When y is 0 this means that block x is on the table.
- In a given state, the only allowed moves are to transfer a clear block onto another clear block (or onto the table). We can represent such a move as a tuple (x, y, z), in which x is the block that gets transferred, y is the block (or the table) on which x was sitting, and z is the block onto which x gets put. Obviously when we make a move from a state we end up in a different state, as some of the facts that used to be in the original state are no longer accurate and some new facts need to be added to describe the resulting state.

We use the following datatypes to represent blocks, facts, states, and moves.

- `type block = int`
- `datatype fact = Clear of block | On of block * block`
- `type state = fact list`
- `type move = block * block * block`

A value S of type state is legal if it completely describes a possible Blocks World configuration for some collection of blocks. It must say, for each block, if the block is clear and if the block is on top of another block or on the table; there must be no redundant facts in the state (such as a fact that occurs more than once); there must be no contradictory facts (a block cannot be both clear and having another block sitting on it); and there must be no silly facts in the state that say things like the table is on some block or the table is clear.

Task 8.1 (5 pts). We will first define a function which allows us to convert towers of blocks into a valid Blocks World state. Write a function

```
mk_tower_state : block list -> state
```

such that when `L` is a list of unique non-zero integers, `mk_tower_state L` evaluates to a legal state that describes the Blocks World configuration in which the blocks in list L form a tower.

For example:

```
mk_tower_state [1,2,3] = [Clear 1, On(1,2), On(2,3), On(3,0)]
```

Any permutation of valid facts is acceptable. Therefore, defining `mk_tower_state` in such a way that

```
mk_tower_state [1,2,3] = [On(1,2), On(2,3), On(3,0), Clear 1]
```

is also an acceptable implementation.

Task 8.2 (3 pts). Using `mk_tower_state`, implement a function

```
mk_state : block list list -> state
```

which takes a valid list of towers `LS` and evaluates to a legal state that describes the Blocks World configuration. Note that no two lists in `LS` may contain the same block.

For example:

```
mk_state [[1,2,3],[4,5]] = [Clear 1, On(1,2), On(2,3), On(3,0),
                            Clear 4, On(4,5), On(5,0)]
```

Task 8.3 (3 pts). We now would like to define a function which allows us to move between different states. First, write a function

```
delete_all : 'a list -> 'a list -> 'a list
```

such that `delete_all (A, B)` evaluates to the list obtained from `B` by deleting all items that are equal to a member of `A`.

For example:

```
delete_all [2,1] [1,2,3,3,2,4,2] = [3,3,4]
delete_all [2,2] [2] = [ ]
```

You may define this recursively, and should use the following function as a helper.

```
delete : 'a -> 'a list -> 'a list

fun delete _ [ ] = [ ]
  | delete x (y::L) = if x=y then delete x L else y :: delete x L
```

Task 8.4 (6 pts). Write a function

```
is_ok : move -> state -> bool
```

that checks if a given move is allowed in a given state. Be careful to handle properly any special cases in which the move involves the table.

Task 8.5 (6 pts). Write a function

```
move : move -> state -> state
```

such that when `S` is a legal state and `m` is a move allowed in state `S`, `move m S` evaluates to a legal state corresponding to the result of doing move `m` from `S`. When `m` is not an allowed move, or the state is not a legal collection of facts, it does not matter what your function does as long as it is type-correct.

Task 8.6 (7 pts). Implement a function

```
run : move list -> state -> state option
```

such that when `S` is a legal state and `M` is a list of moves,

- `run M S` evaluates to `SOME S`, where `S` is the state reached by doing the moves in `M` in sequential order, from start state `S`, if this is allowed.
- `run M S` evaluates to `NONE` if this sequence of moves is not allowed.

Task 8.7 Extra Credit (5 pts). We would like to convert a valid Blocks World state back into a list of towers of blocks. Write a function

```
towers : state -> block list list
```

that produces, from a legal state, a list of the towers of blocks from the configuration described by the state. Recall that a tower has the form `[b1, ... , bk, 0]` where `b1` through `bk` are blocks, `b1` is clear, and each one is on top of the next one in the list.