15-150 Fall 2015
Homework 09

Out: Wednesday, 04 November 2015
Due: Tuesday, 10 November 2015 at 23:59 EDT

1 Introduction

In this homework, you'll practice working with structures, signatures and functors even more. It will give you a chance to reason about representation independence and create several different implementations of interesting signatures.

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/09` directory should contain a file named exactly `hw09.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/09` directory (that contains a `code` folder and a file `hw09.pdf`). This should produce a file `hw09.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 `hw09.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 `hw09.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, 10 November 2015 at 23:59 EDT. 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
```

2 Bounding boxes

Consider the following signatures:

```
signature SCALAR =
sig
   type t
   val zero : t
   val one : t
   val add : t * t -> t
   val mult : t * t -> t
   val negate : t -> t
   val invert : t -> t
   val compare : t * t -> order
end;

signature SEQ =
sig
  type 'a seq
  exception Range
  val nth : int -> 'a seq -> 'a
  val tabulate : (int -> 'a) -> int -> 'a seq
  val map : ('a -> 'b) -> ('a seq -> 'b seq)
  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b
end;
```

Here is an implementation of `SCALAR` based on the real numbers:

```
structure Reals : SCALAR =
struct
   type t = real
   val zero = 0.0
   val one = 1.0
   val add = op+ : real * real -> real
   val mult = op* : real * real -> real
   fun negate x = ~1.0 * x
   fun invert x = 1.0/x
   val compare = Real.compare
end;
```

And here is an implementation of `SEQ`, which can be used for testing.

```
structure Seq : SEQ =
struct
```

```
   type 'a seq = 'a list;
   exception Range;
   fun nth _ [ ] = raise Range
    |  nth 0 (x::L) = x
    |  nth n (x::L) = nth (n-1) L;
   fun tabulate f n = List.tabulate(n,f);
   val map = List.map;
   fun reduce f z L = foldl f z L;
   fun mapreduce g z f L = reduce f z (map g L);
end;
```

We can make the following signature declaration:

```
signature BOX =
sig
   type point
   type box
   val inside : box -> point -> bool
   val union : box * box -> box
   val center : box -> point
   val vertices : box -> point Seq.seq
   val hull : point Seq.seq -> box option
end;
```

Task 2.1 (25 pts).
Implement the functor `Box` in `box.sml`, for building a structure with signature `BOX`, given an structure `S:SCALAR`, and using a fixed structure `Seq:SEQ` that we supply. You can assume that `S.compare` is a well-behaved comparison function of type `S.t * S.t -> order`. Your implementation should be based on the following ideas. We refer to values of type `S.t` as "scalars".

- A *point* (a value of type `point`) is a pair (x, y) of scalars.
- The *invariant* for this implementation is that in a pair (p_1, p_2) : `box` the x-coordinate of p_1 is less-than-or-equal to the x-coordinate of p_2, and the y-coordinate of p_1 is less-than-or-equal to the y-coordinate of p_2, with respect to `S.compare`.
- A *bounding box* (a value of type `box`) is a pair of points (p_1, p_2) representing the rectangular region of the plane for which p_1 is the bottom left-hand corner and p_2 is the top right-hand corner. For any rectangular region, there is *one* value (p_1, p_2) : `box` whose contents is exactly this rectangle, that satisfies the invariant.
- `fromPoint : point -> box` produces a bounding box that contains a single point.
- `vertices : box -> point seq` produces the sequence consisting of the four corners of a given bounding box, in the order bottom-left, top-left, top-right, bottom-right. (Note: the items in a sequence `s` are indexed starting from 0. The first item is `nth 0 s`, and so on.)
- `mid :point * point -> point` returns the midpoint of the straight line between the two given points.
- `center :box -> point` returns the center point of a bounding box.
- `inside :box -> point -> bool` checks if a point is inside a bounding box.
- `union :box * box -> box` returns the smallest bounding box that contains two given bounding boxes.
- `hull : point seq -> box option` returns `NONE` if applied to an empty sequence; otherwise it returns `SOME B`, where `B` is the smallest bounding box that contains all boxes in `S`.

3 Representation independence

A major advantage of modular programming and abstract types is that we can hide implementation details (how data is represented, how operations actually operate) from clients. A client of an abstract type implemented as a structure with a specific signature can only use the operations provided in that signature, and only knows how data is represented if the signature makes these details visible.

You will prove a "representation independence" result showing that two different implementations of the same signature are "observably equivalent", in that users are unable to distinguish between them.

Here's a signature for list-like structures with an append operation. It provides operations for going to from lists as well as one operation allowing us to append one value of type `'a List` to another.

```
signature APPENDABLE =
sig
   type 'a List
   val fromList : 'a list -> 'a List
   val toList   : 'a List -> 'a list
   val append   : 'a List -> 'a List -> 'a List
end
```

The reference implementation for `APPENDABLE` is just the standard libraries version of `list` and append.

```
structure Standard :> APPENDABLE =
struct
   type 'a List = 'a list

   fun fromList xs = xs
   fun toList xs = xs

   fun append xs ys = xs @ ys
end
```

Another implementation is

```
structure Fun :> APPENDABLE =
struct
   type 'a List = 'a list -> 'a list;

   fun fromList xs = fn ys => foldr op:: ys xs
   fun toList xs = xs []
```

```
   fun append xs ys = xs o ys
end
```

These two implementations represent lists very differently but it's quite easy to give a precise formulation of when a value of type `t Fun.List` is equivalent to a value of type `t Standard.List`. We'd like to say that they're equivalent if they agree on the output of `toList` but this won't be sufficient for many of the proofs we'll do. Instead we will say that `xs : t Standard.List` is equivalent to `ys : t Fun.List` (written $\texttt{xs} \equiv \texttt{ys}$) if and only if for all `L : t list`:

```
xs @ L = ys L
```

In the following proofs, you may need to use these lemmas.

1. For all `L1 L2 L3 : t list`, `(L1 @ L2) @ L3 = L1 @ (L2 @ L3)`
2. For all `L : t list`, `L @ [] = [] @ L = []`
3. For all `L1 L2 : t list` and `x : t`, `(x :: L1) @ L2 = x :: (L1 @ L2)`

Task 3.1 (10 pts). Prove that for all types `t` and all values `L : t list` that

$$\texttt{Standard.fromList L} \equiv \texttt{Fun.fromList L}$$

Task 3.2 (10 pts). Prove that for all types `t` and all values `L1, L2 : t Standard.List`, `L1', L2' : t Fun.List` if

$$\texttt{L1} \equiv \texttt{L1}' \wedge \texttt{L2} \equiv \texttt{L2}'$$

then

$$\texttt{Standard.append L1 L2} \equiv \texttt{Fun.append L1}' \texttt{ L2}'$$

.

Task 3.3 (5 pts). The implementation of `APPENDABLE` provided `Fun` is a well known technique for getting better performance for repeatedly appending to lists.

Argue (you do not need to rigorously prove) why its implementation may be more efficient than repeated application of `Standard.append`.

Hint: consider the cost of `(((L1 @ L2) @ L3) @ L4) @ L5` *versus that of using* `Fun.append`

4 Dictionaries as functions

Suppose we change the signature for dictionaries to include extra operations (`delete`, `map` and `filter`) and get rid of `trav`:

```
signature DICT =
sig
   type 'a dict
   structure Key : ORDERED
   val empty : 'a dict
   val insert : 'a dict -> (Key.t * 'a) -> 'a dict
   val lookup : 'a dict -> Key.t -> 'a option
   val delete : 'a dict -> Key.t -> 'a dict
   val map : ('a -> 'b) -> 'a dict -> 'b dict
   val filter : ('a -> bool) -> 'a dict -> 'a dict
end;
```

Task 4.1 (15 pts).
Write a functor `FunDict` using the following implementation for the type used to represent dictionaries:

```
type 'a dict = Key.t -> 'a option
```

Use opaque ascription to prevent users from exploiting the knowledge that dictionaries are functions!

The `empty`, `insert` and `lookup` functions should behave as before. The specifications for the new functions are:

- `delete D k` is a dictionary that behaves like `D` except that it has no entry with a key `EQUAL` to `k` (according to the relevant comparison function).
- Assuming that `f` is a total function, `map f D` is a dictionary with entries of the form `(k, f v)` for every entry `(k,v)` of `D`.
- Assuming that `p` is a total function, `filter p D` is a dictionary consisting of the entries `(k, v)` of `D` for which `p(v) = true`.

In this task, don't worry about efficiency: just develop a working implementation that meets the given specs.

5 Polynomials over a ring

Consider the following signature defining an ML version of the mathematical concept of a ring.

```
signature RING =
sig
  type t
  val zero : t
  val one : t
  val add : t * t -> t
  val mult : t * t -> t
end
```

These operations satisfy the properties that you would expect:

- `add` should be commutative and associative
- `zero` should be an identity for `add`
- `mult` should be associative
- `one` should be an identity for `mult`
- `mult` should distribute over `add`

These operations should obviously be total as well. In this section you'll be implementing several different structures satisfying these operations.

Task 5.1 (10 pts). Implement `RatRing:RING` in `rat.sml`. A value of type `RatRing.t` is a pair of integers `(a, b)` representing the rational number $\frac{a}{b}$.

Your implementation should define all the operations so that they correspond to 0, 1, +, and *.

- `zero` should represent the rational number 0
- `one` should represent the rational number 1
- if `a` represents x, and `b` represents y then `add (a, b)` should represent $x + y$.
- if `a` represents x, and `b` represents y then `mult (a, b)` should represent xy.

It is OK if there are multiple unequal representations of the same rational number

Task 5.2 (10 pts).
Implement the functor `PowerRing`, for building an implementation of *power series* over a given ring. That is, you'll be defining ring operations on series $c_0 + c_1X + c_2X^2$.... Here each c_i is an element of the ring supplied as an argument to the functor.

The underlying implementation of the powerseries type for your functor will be the function type `int -> R.t`. A function `f` of this type represents the series $\sum_{i=0}^{\infty} c_i X^i$ where for each i the value of `f(i)` is c_i. Note that here $\sum$, $+$, $*$, and all other math operations are interpreted as the appropriate ring operations for the argument structure.

- `zero` should represent the series $\sum_0^\infty 0X^i$
- `one` should represent the series $1 + \sum_1^\infty 0X^i$
- if `a` represents $\sum_0^\infty c_i X^i$, and `b` represents $\sum_0^\infty d_i X^i$ then `add (a, b)` should represent $\sum_0^\infty (c_i + d_i)X^i$.
- if `a` represents x, and `b` represents y then `mult (a, b)` should represent xy.

For multiplication of series, remember that

$$\left(\sum_0^\infty c_i X^i\right)\left(\sum_0^\infty d_i X^i\right) = c_0d_0 + (c_0d_1 + c_1d_0)X + (c_0d_2 + c_1d_1 + c_2d_0)X^2 + ...$$

When testing your powerseries, you must be sure to use series besides the ones defined by `zero` and `one` or you will not receive credit for your tests. Since your power series is potentially infinite, it may be helpful in your tests to compare coefficients at particular locations to ensure they are correct.

6 Red-Black Trees

Recall the definition of red-black trees:

> A red-black tree is a binary search tree where each node is colored either red or black. It must satisfy three additional properties
>
> 1. All the leaves are black
> 2. The child of any red node is black
> 3. Each path from the root to a leaf has the same number of black nodes

An example of a realization of this definition in SML is

```
datatype color = Red | Black
datatype 'a tree  = Leaf | Node of 'a tree * (color * 'a) * 'a tree
```

The goal of these constraints is to force a valid red-black tree to be roughly balanced. By not enforcing perfect balancing it's much easier to maintain the balancing properties. On the other hand, it needs to be proven that the constraints are not so weak the operations may become linear in the number of items in the tree.

Task 6.1 (5 pts). Prove that if `t = Node (l, (c, v), r)` is a valid red-black tree then so are both `l` and `r`.

Task 6.2 (5 pts). Prove that for any red-black tree with a black-height of `b` (that is, each path from the root to a leaf has `b` black nodes not counting the leaf) then $2^b \leq n$ where n is the number of nodes in the tree (counting all leaves).

Task 6.3 (5 pts). Prove that a red-black tree with `n` nodes has a height in $O(\log(n))$. To prove this, you must construct a constant so that the height is always below $c \log_2(n)$.

Hint: prove a fact relating the height of a tree to its black height as a small lemma.