15-150 Summer 2016
Lab 10

25 March 2015

1 Introduction

This lab is meant for you to delve further into the module system and get experience with functors which are functions on modules. A lot of this lab will be working off of what you saw in last week's lab.

1.1 CM

In this lab, because we're dealing with lots of files containing structures, everything will be orchestrated by CM (SML's Compilation Manager). This means you'll be editing the `sources.cm` file as needed, and running `CM.make "sources.cm"` to load your code.

One quirk of `CM` is that it will give you warnings if you have code that exists outside of a structure. It will evaluate that code and fail to compile if it doesn't type check, but it will never introduce any bindings from it into the environment. To avoid this annoying behavior, it's best to just put everything inside structures—even if they don't ascribe to a signature.

Remember that the names of some files (in this case `setofsets.sml`,`pairset.sml`, and `psetfunctor.sml`) **are commented out in your** `sources.cm` **file. You must uncomment these lines after implementing the code or it will not be compiled.**

1.2 Getting Started

Update your clone of the `git` repository to get the files for this weeks lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

1.3 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have REQUIRES and ENSURES clauses and tests.

2 Sets With Functors

Recall from lecture, a *functor* is analogous to a function that takes in a module as an argument and returns another module. In lab last week, you implemented sets in two different ways as structures that ascribed to the SET signature. In those implementations, you were told to assume that the set contained integers and so elements of the set could be compared by `op=`.

An interesting problem to think about is how you would compare elements in a set if you did not know what their type will be. This problem can be solved by the useful idea of packaging up a type with a function that checks equality. An example of this is the EQUAL signature that can be seen in `equal.sig`.

This is an example of a type class, which is an important use of signatures where you describe a type equipped with a (probably non-exhaustive) collection of operations on it.

When the type of the element being stored is not known, a set can be implemented as a functor that accepts a structure that ascribes to the signature `EQUAL` and returns a structure that ascribes to the signature `SET`. However, we're going to take a different approach - implementing them as dictionaries where the keys are the elements of the sets which all map to unit.

A dictionary is implemented as a functor that takes in a structure ascribing to signature `EQUAL` and returns a structure that ascribes to signature `DICT`. A set will then be a functor whose argument will be a structure that ascribes to signature `DICT` and returns an implementation of the `SET` signature. The implementation of a dictionary is given to you in `dict.sml` which you should know about and use later in the lab but for the purpose of creating your set you only need to refer to `dict.sig`.

Task 2.1 Implement the functor `DictSet` (found in `dictset.sml`) that takes a structure ascribing to `DICT` as an argument and returns an implementation of the `SET` signature.

3 Fun With Sets

Now that you have implemented sets that work for a generic type, you can perform some interesting set operations. However, some of these operations would require you to implement additional types of sets. In this problem we're going to construct those types of sets, but not actually use them to implement any new set operations.

One example of this is the power set (which is a set containing sets). As you know, for some set S, it's power set, $\mathcal{P}(S)$, is the set of all possible subsets of S. However, to take the power set of a set, you need a way to represent a set of sets.

Task 3.1 Implement a set of sets as a functor `SetOfSets` (found in `setofsets.sml`) that takes in a structure S with signature `SET` and creates another structure with signature `SET` that represents a set of sets (where each inner set uses the set implementation S).

Task 3.2 Test your implementation of `SetOfSets` by passing sets of different types into the functor as arguments.

Another interesting type of set is the Cartesian Product set. The Cartesian product of two sets A and B denoted $A \times B$ is the set of all (a, b) ordered pairs such that $a \in A$ and $b \in B$. To take a Cartesian product you need to be able to represent a set of pairs, where each pair has one element from A and one from B.

Task 3.3 Write a functor `PairOfEqual` that ascribes to the signature `EQUAL`, that, given a structure `PE` ascribing to signature `PAIR_OF_EQUAL`, produces a structure that checks if elements of such pairs are equal.

The structure `PE` contains two structures `E1` and `E2`, each ascribing to `EQUAL` (implementing equality for types `t1` and `t2` respectively). This signature is defined in `equal.sig`.

Task 3.4 Implement sets of pairs as a functor `PairSet` (found in `pairset.sml`) that takes in a structure `PE` ascribing to signature `PAIR_OF_EQUAL` and creates another structure with signature `SET` that represents sets of pairs with type `(t1 * t2)`. You may want to use the `PairOfEqual` functor you just wrote, along with some other functors.

Note that SML has a syntax for writing functors with multiple arguments (so we could have made the two arguments `E1` and `E2`). However, in our experience many find this syntax confusing, so we've made all functors take only one argument.

Task 3.5 Test your implementation of `PairSet` by passing sets of different types into the functor as arguments. This will require making some structures of signature `PAIR_OF_EQUAL`.

4 Category Theory

Note: this problem contains more mathematical background than is necessary to write the code. Skip to the last paragraph before Task 4.1 if you just want to know what you're supposed to do.

SML functors share their name with a concept in the branch of mathematics called category theory. A mathematical category is an algebraic structure that has some collection of "objects" and a set of "structure-preserving" mappings between them. (When we say "structure-preserving", we mean that relationships between objects are preserved by the mapping. For example, if we choose the collection of groups to be our objects, our mappings are group homomorphisms.) This abstract view provides us with a way to generalize mathematical objects and their relationships that connects and unifies often very different mathematical disciplines.

Categories are defined by two properties

1. Composition of mappings is associative
2. Every object has an identity mapping

A simple category, and the one that we will use for this problem, is the category of sets (by convention, denoted as **Set**). Our objects are sets and our mappings are mathematical functions. Function composition is associative, and the identity function is just $f(x) = x$, so the two properties are indeed satisfied.

Now let's get back to functors. In category theory, a functor is a structure-preserving mapping between categories. This means that for a functor F between categories C and D

- For each object $X \in C$ there is an associated $F(X) \in D$
- $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ for every $X \in C$ and its identity mapping id_X
- $F(g \circ f) = F(g) \circ F(f)$ for all mappings $f : X \to Y$ and $g : Y \to Z$ in C

Your task will be to implement the power set functor, $P : \mathbf{Set} \to \mathbf{Set}$. To do this for real, we would need to actually define categories in SML, which is possible, but too complicated for this lab. Instead, you'll be writing a functor that takes in a structure S ascribing to `SET` and creates two functions, one to map sets to their powersets, and one to map functions between sets to functions between powersets.

Using the `SetOfSets` functor you implemented in Task 3.1, implement a function `make_powerset : set -> powerset` that takes a set s and returns its powerset, $\mathcal{P}(s)$, the set of all subsets of s. Then implement a function `make_powerfun : (elem -> elem) -> set -> set`. A function that takes a function f from `elem` to `elem` and returns a function g from a set of `elem` to a set of `elem` such that g applies f to every element in its argument.

We see that `make_powerset` applies P to our objects, and `make_powerfun` applies P to our mappings. Note that you will have to provide the appropriate type definitions for `elem`, `set`, and `powerset`.

Task 4.1 Implement the `PowerSetFunctor` functor (in `psetfunctor.sml`) that takes in a structure S with signature `SET` and creates a structure with signature `POWERSETFUNCTOR` that implements `make_powerset` and `make_powerfun` as described above.

Task 4.2 Test your implementation of `PowerSetFunctor` by passing sets of different types into the functor as arguments, using the result to actually construct powersets and interesting "power functions."