15-150 Summer 2016
Lab 08

9 June 2016

1 Introduction

This lab is meant to give you a chance to experiment with the module system. We will see several implementations of simple signatures. The idea is to get comfortable working with them.

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`).

2.1 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.

2.2 The SML/NJ Build System

We will be using several SML files in this assignment. In order to avoid tedious and error-prone sequences of `use` commands, the authors of the SML/NJ compiler wrote a program that will load and compile programs whose file names are given in a text file. The structure `CM` has a function

```
val make: string -> unit
```

`make` reads a file usually named `sources.cm` with the following form:

```
Group is

$/basis.sml
file1.sml
file2.sml
file3.sml
...
```

Loading your code using the REPL is simple. Launch SML in the directory containing your work, and then:

```
$ sml
Standard ML of New Jersey v110.75 [built: Fri Feb  8 12:33:48 2013]
- CM.make "sources.cm";
[autoloading]
[library $smlnj/cm/cm.cm is stable]
[library $smlnj/internal/cm-sig-lib.cm is stable]
...
```

Simply call

```
CM.make "sources.cm";
```

at the REPL whenever you change your code instead of a `use` command like in previous assignments. The compilation manager offers a better interface to the command line. There is less typing and less of an issue with name shadowing between iterations of your code. In short, on this assignment, the development cycle will be:

1. Edit your source files.
2. At the REPL, type

   ```
   CM.make "sources.cm";
   ```

3. Fix errors and debug.

Be warned that `CM.make` will make a directory in the current working directory called `.cm`. This is populated with metadata needed to work out compilation dependencies, but can become quite large. The `.cm` directory can safely be deleted at the completion of this assignment.

It's sometimes the case that the metadata in the `.cm` directory gets in to an inconsistent state—if you run `CM.make` with different versions of SML in the same directory, for example. This often produces bizarre error messages. When that happens, it's also safe to delete the `.cm` directory and compile again from scratch.

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.

2.2.1 Emphatic Warning

`CM` will not return cleanly if any of the files listed in the sources have no code in them. Because we want you to learn how to write modules from scratch, we have handed out a few files that are empty except for a few place holder comments. That means that there are a few files in the `sources.cm` we handed out that are commented out, so that when you first get your tarball `CM.make "sources.cm"` will work cleanly.

You must uncomment these lines as you progress through the assignment! If you forget, it will look like your code compiles cleanly even though it almost certainly doesn't.

3 Sets of Sets of Sets

Recall from lecture, a *signature* is an interface specification that usually lists some types and values (that might use those types). In this task, you will write two implementations of the `INTSET` signature that can be found in `intset.sig`, and is given below:

The components of the signature have the following specifications:

- `set` is the type of the set of elements of type `int`.
- `empty` is a set that contains no elements.
- `find` is a function that takes a set and an element, and returns `true` if that element is in the set, or `false` if the element is not in the set.
- `insert` is a function that takes a set and an element and returns the set with the element added.
- `delete` is a function that takes a set and an element and returns the set with the element removed.
- `union` is a function that takes two sets X and Y and evaluates to the set $X \cup Y$, i.e. a set that contains all the elements in X and Y that results from performing a mathematical union of the two sets.
- `intersection` is a function that takes two sets X and Y and evaluates to the set $X \cap Y$, i.e. a set that contains the elements in both X and Y that results from performing a mathematical intersection of the two sets.
- `difference` is a function that takes two sets X and Y as input and evaluates to the set $X \backslash Y$, i.e. a set that contains all elements in X and not in Y, that results from performing the mathematical difference of the two sets.

There are many ways to implement the functionality of sets. We will implement sets in two different ways in this lab. The first should allow for duplicates to be inserted into the internal representation of the set, but care should be taken that (externally) the implementation still behaves like a set (recall that a set does not contain duplicate values for any given element). For example, deleting an element that occurs multiple times internally should externally delete that element completely. The second should not allow for any duplicates to be stored in the internal representation at all.

You should think about what invariants you want to have on your internal representation before starting to program; there are a few ways to do this.

Task 3.1 Implement the structure `SetKeepDuplicates` ascribing to `INTSET` found in `SetKeepDuplicates.sml` that allows for duplicate insertion into the set. Keep in mind that all functions implemented must account for the fact that the internal representation may contain duplicate values. This will be particularly important when removing elements from the set.

Task 3.2 Implement the structure `SetNoDuplicates` ascribing to `INTSET` found in `SetNoDuplicates.sml` that does not allow for duplicate insertion into the set. Keep in mind that all functions implemented must account for the fact that the internal representation must not contain duplicate values. This will be particularly important when adding elements to the set.

Have the TAs check your code before continuing!

Task 3.3 The two implementations of `INTSET` have various trade-offs. One of these trade-offs involves the work/span of `insert`. Find the work and span of `insert` in both `SetKeepDuplicates` and `SetNoDuplicates`. Discuss another function that has a different work/span based on the implementation.

4 Dictionaries

A *dictionary* is a datastructure that acts as a finite map from *keys* and to *values*. We represent a dictionary by a type

```
('k, 'v) dict
```

`dict` is a type constructor that takes two type arguments (unlike e.g. `'a list`, which takes only one). The first, `'k`, represents the type of keys, whereas the second, `'v` represents the type of values. For example, an `(int,string) dict` maps integers to strings.

There are many possible implementations of dictionaries, including using lists, trees, and functions. Since there are so many different ways to implement dictionaries, it would be nice if we could have an abstract interface to them that is the same regardless of the underlying implementation. This is where the module system comes in.

In `LabDict.sig`, we have provided the a signature for you to implement for dictionaries The type and values in it have the following specifications:

- `('k, 'v) dict` is an abstract type representing the type of the dictionary. Note that it is parametrized over two different types—`'k`, the type of keys, and `'v`, the type of values.
- `empty` is a dictionary that contains no mappings.
- `insert` is a function that takes a comparison function for keys[1], a dictionary, and a key-value pair and returns the dictionary with the mapping added. If the key is already in the dictionary, the new value supersedes the old one.
- `lookup` is a function that takes a comparison function, a dictionary, and a key, and returns `SOME v` if that key maps to the value `v` in the dictionary, or `NONE` if there is no mapping from the key.
- `modify` is a function that takes a comparison function, a dictionary, and a key and a value, and returns `SOME d` if the dictionary contains the key-value pair `(key,otherValue)` where `(key, otherValue)` is replaced with `(key,value)`. Otherwise, it evaluates to `NONE`.

We have called this signature `LABDICT` because it is a version of dictionaries that is small enough for you to implement in lab; a real dictionary library would provide more operations.

[1] You may feel a bit uncomfortable adding the comparison function as an argument to each of these functions, instead wondering why we cannot just abstract over it somehow. If so, good! We will go over this in a later lecture.

5 Implementation: BSTs

One way to implement a dictionary is using a binary search tree (BST). Recall the discussion of binary search trees from when we implemented mergesort on trees: the key invariant is that, for every `Node(l,x,r)`, everything in `l` is less than or equal to `x`, and everything in `r` is greater than or equal to `x`.

To implement dictionaries, we will store both a key and a value at each node, using the following datatype:

```
datatype ('k, 'v) tree =
    Leaf
  | Node of ('k, 'v) tree * ('k * 'v) * ('k, 'v) tree
```

In `Node(l,(k,v),r)`, `k` is the key and `v` is the value. Every key in `l` should be less than or equal to `k`, and every key in `r` should be greater than `k`. That is, the keys satisfy the BST invariant; the values are just along for the ride.

Task 5.1 In the file `TreeDict.sml`, implement a structure `TreeDict` matching the signature `LABDICT`. You should use the datatype above as the internal representation of a dictionary. Make sure you ascribe the signature LABDICT to make the type `dict` abstract!

To test your implementation, you can run the command

```
- CM.make "sources.cm";
```

from the REPL. Note that your code will not compile until you make `TreeDict.sml` and put a module in it.

To create tests for your code inside the SML file, you can do them normally as we have been doing all semester. You should put them inside the `TreeDict` structure.

To test your code from the REPL, you will need to refer to functions inside your `TreeDict` structure as components of the module. (i.e. as `TreeDict.<function_name>` where `<function_name>` is the name of the function you want to run). Recall that you can only refer to functions that have been defined in the signature.

Some notes about the compilation manager: If you compile your code using `CM.make`, the compilation manager will compile all of the files specified in the `.cm` file. One way to work is to use `CM.make` every time you want to compile.

However, this has the disadvantage that none of the project loads if there is a compilation problem anywhere, which can make debugging harder—you can't use the REPL to play around. An alternative is to `CM.make` when you first start working on a module, assuming your initial state is one where the `CM.make` succeeds (this is generally true for the support code we hand out). Then you can reload the file containing the module you are currently working on with `use` after you make updates to it. Note that this will shadow the modules in that file, and thus **not** update the modules "downstream". However, it is useful if you are

using emacs, and like to use the emacs command to load the current buffer: you can load one module repeatedly and use the REPL to test it. It's also useful if your current implementation of the module has a bug that causes later files in the `.cm` file to fail to compile. But when when you are done working on the module, you will want to run `CM.make` again to reload all the downstream modules, so that they refer to your new implementation.

Task 5.2 There is an issue with the trees as presented, though. Since you input a compare function every time you access or modify the dictionary, you can easily break the sortedness invariant of the tree by passing in different compare functions. In the files `BetterTreeDict.sml` and `BetterLabDict.sig` implement your own module that circumvents this issue.

Task 5.3 Phew. Finally, we will explore the concept of Information Hiding. First, compile your code and type `BetterTreeDict.empty` in the REPL. Note what you see. Now, ascribe the signature `BETTERLABDICT` opaquely to `BetterTreeDict`. Do the same as above. What differences do you see? What advantage does opacity have in hiding information in structures?

6 Fun With Integers

As you know, integers can be represented in many bases. All representations should, however, allow for addition and multiplication (and some other) operations on them. A good way to implement this would be to use a signature that provides an interface for the common behavior that all integer representations should have, and then implementing structures ascribing to this signature for the various bases.

In this task, you will implement integers in two different bases, and allow for addition and multiplication on them, following the `ARITHMETIC` signature in `arithmetic.sig`.

The components of the signature have the following specifications:

- `integer` is the type used to represent integers
- `rep` given an int, converts it to an integer type
- `display` displays the given integer as a string.
- `add` takes in two integers and returns the result of their addition.
- `mult` takes in two integers and returns the result of their multiplication.
- `toInt` takes an integer and converts it to an int. This may be useful for testing purposes.

In each of the following tasks, it might be challenging to write `add` and `multiply` without helper functions. For example, it might be useful to have a helper function keep track of the carry digits when implementing `add`.

Task 6.1 Implement the structure `Binary`, ascribing to `ARITHMETIC`, found in `Binary.sml` that implements the specified integer operations for integers in base 2. As an example, the `rep` function should convert the `int` 4 to an integer that represents the binary "100" (which is the binary form of that number).

Hint: the type `integer` should be `digit list`. With this type, it is possible to represent `ints` in two ways - one in which the least significant digit is stored as the first element in the digit list and one in which the least significant digit is stored as the last element in the digit list. One of these ways is easier to implement than the others. In addition, while it is possible to represent 0 as both the empty list and the list `[0]`, for simplicity, let 0 be represented by as the empty list.

Task 6.2 Implement the structure `Decimal`, ascribing to `ARITHMETIC` found in `Decimal.sml` that implements the specified integer operations for integers in base 10.