15-150 Fall 2014
Lab 06

Thursday 2nd October, 2014

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.

1 Introduction

1.1 Getting Labs

We will be distributing the text and any starter code for the labs using Autolab. Each week's lab will start being available at the beginning of class.

On the Autolab page for the course, the current lab is the last entry under `Lab`. Say it is called "`labnn`". Click on this link. There, two links matter

- View writeup: this is the text of the lab in PDF format.
- Download handout: this is the starter code, if any, for the lab. It will always be distributed as a compressed archive (in `.tgz` format). Uncompressing it will create the following directories:

`lab`*nn*`/`	Directory for lab *nn*
`code/`	Code directory for lab *nn*
`*.sml`	Starter files for lab *nn*
`handout.pdf`	Copy of writeup for lab *nn*

1.2 Code Structure

On this and future assignments, we will be grading your programs on more than just their input-output behavior. It's not enough to have programs that happen to work: they need to clearly state what they do, have some empirical evidence that they work as advertised, and be easy for other people to read and reason about.

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

1. In the first line of comments, write a call template of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function (include type annotations for the arguments and result of the function)
5. Provide test cases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

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

(* Tests: *)

val 1 = factorial 0
val 720 = factorial 6
```

1.3 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 running *either*

- `sml -m sources.cm` from the terminal prompt, *or*
- `CM.make "sources.cm"` from the SML prompt

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.

2 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.
- `member` 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., the set containing all the elements in X and Y that results from performing the 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., the set containing the elements in both X and Y that results from performing the 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., the set containing all elements in X and not in Y, that results from performing the mathematical difference of the two sets.
- `equal` is a function that takes two sets X and Y as input and returns `true` if they contain the same elements, and `false` otherwise.

There are many ways to implement such sets. We will implement sets in two different ways in this lab. Both take the type `set` to be `int list`. 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. 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 2.1 Implement the structure `SetKeepDuplicates` ascribing to `INTSET` found in the file `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 2.2 Implement the structure `SetNoDuplicates` ascribing to `INTSET` found in the file `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.

Checkout point!

Completing everything up to here in the lab assignment will guarantee credit for this lab.

Click here or go to the class schedule and click on a `Check me in` button.

3 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 implement 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 `integer`'s and returns the result of their addition.
- `mult` takes in two `integer`'s 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 3.1 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 `int`'s 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 3.2 Implement the structure `Decimal`, ascribing to `ARITHMETIC` found in `Decimal.sml` that implements the specified integer operations for integers in base 10.