Recitation 1 – *SML Module Language and Sequences*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2014)

January 14th, 2014

1 Welcome to 15-210!

- Lab 0 – *MiniLab*, has been released! It is due next Monday, January 20th. It's "*something light to get you started*", and shouldn't take too long. You'll get practice implementing and using our `SEQUENCE` library, and do a few Big-O proofs.
- In this class, we will be using SML as our default programming language, which you should be familiar with if you have taken 15-150 previously. For those who haven't (graduate students, for example), or if you need a bit of a refresher, we will be holding a crash course this week (see Piazza for updates).

2 Administrivia

Q: Where will the labs be posted?

A: We will be distributing the assignments for this course through Autolab (`https://autolab.cs.cmu.edu/`). You will also be submitting your assignments on Autolab.

Q: When will the labs be posted / when will they be due?

A: Labs usually go out on Mondays, and are due the following Monday by 11:59pm, with several exceptions. See the course schedule for updates.

Q: When are office hours?

A: Office Hours will be posted on the course webpage (`http://www.cs.cmu.edu/~15210/staff.html`). These times are subject to change. If you have time conflicts and cannot attend any of the listed office hours, please contact one of the course staff.

3 Plan

- The SML Module Language: Structures, Signatures, and Functors
- Sequence Cost Bounds

4 The SML Module Language

As you may have seen in 15-150, SML has a module language that includes *structures*, *signatures*, and *functors*. These concepts work together to make it the flexible functional programming platform it is today. Let's go over these quickly to make sure you understand how they work, as every lab in 15-210 will make use of them.

4.1 Structures

A structure is essentially a named *packaging* of functions and values, that are usually related either semantically or functionally (hah!) in some way. A structure is a single `struct` and `end` block, within which contains the necessary function implementations.

Consider the structure `YumPancakes` that encapsulates functions that are needed by *Nikki's Cafe* to serve pancakes to its customers. We denote a `pancake` by its weight, a `real`, and implement a `pile` of pancakes using the 210 `SEQUENCE` library.

Let's say that we have a file `YumPancakes.sml` containing the following structure:

```
structure YumPancakes =
struct
  open Seq

  type pancake = real  (* Pancake weights *)
  type pile = pancake seq

  fun remFrmBk (S : 'a seq) : 'a option =
      if length S = 0
      then NONE
      else SOME (drop (S, 1), nth S 0)

  fun serveTop (P : pile) : pancake option =
      case remFrmBk P of
          NONE => NONE
        | SOME (_, p) => SOME p

  fun serveBottom (P : pile) : pancake option =
      case remFrmBk P of
          NONE => NONE
        | SOME (P', p) => if length P' = 0
                          then SOME p
                          else serveBottom P'
end
```

To use this structure outside of its file (assuming we have SML/NJ's compilation manager `CM`[1] all set up), we can just, for example, say `YumPancakes.serveBottom P`, where `P` is a preexisting value of type `pile`.

[1]http://www.cs.cmu.edu/~15210/resources/cm.pdf

Q: What's wrong with this?

A: Clearly `remFrmBk` is an operation on stacks: what if we want to make use of the many different ways that stacks can be implemented? This would both remove the dependency on the `SEQUENCE` library, as well as allow for potentially more efficient implementations.

Q: What can we do instead then?

A: This is where signatures and functors come in.

4.2 Signatures

A signature is a specification of types and functions for structure(s) that implement it, with declarations between a `sig` and `end` block. We say that a structure *ascribes* to that signature.

Q: Have you seen something similar in imperative languages?

A: This is similar to the concept of an interface for `foo` in `foo.h`, implemented by its implementation in `foo.c` in the imperative world.

When a structure ascribes to a signature, every function and value declared in the signature must be implemented by the structure. In addition, when using the structure outside of its file, only the functions *exposed* by the signature can be used; any functions declared solely within the structure have scope limited in such a way (kind of like *public* versus *private* functions in a language like Java).

Now let's begin generalizing the `YumPancakes` structure by creating a `PANCAKES` signature in the file `PANCAKES.sig`. Note that the convention is to name a structure with a leading capital letter, and a signature in all caps.

```
signature PANCAKES =
sig
  type pancake
  type pile

  val serveTop : pile -> pancake option
  val serveBottom : pile -> pancake option
end
```

Q: Why are we returning `pancake options`?

A: Return `NONE` if we have an empty `pile`, `SOME p` otherwise.

Now our `YumPancakes` structure can ascribe to the `PANCAKES` signature:

```
structure YumPancakes : PANCAKES =
struct
  ...
  (* Same as above *)
  ...
end
```

Q: Why don't we define our types in `PANCAKES`?

A: We don't use the `Seq.seq` type anywhere in the signature, so that we are *free* from the `SEQUENCE` library.

Q: Why isn't `remFrmBk` part of the signature?

A: We are refactoring the data structure operation out of the interface, since **it has nothing to do with a pile of pancakes**.

Great! Now how do we implement `YumPancakes` using a stack? What a great segue into functors!

4.3 Functors

The last (but certainly not least!) part of the SML module language are functors. A functor creates a structure, *given* a structure. In other words, when provided a structure that ascribes to a specific signature, a functor returns an entire structure, that also ascribes to some signature. Because of what it does, a functor is also simply a `struct ... end` block.

Think of a functor as an awesome *structure factory*; when given differently-implemented structures that ascribe to the same signature, it spits out perfectly-typed structures that perform the same duties, just in different ways (this is of course assuming the functor typechecks!).

Back to our running example of pancakes,

Q: What structure can we pass into the functor that we're about to write?

A: A structure called `Stack` that ascribes to the signature `STACK`.

Q: What functions do we *absolutely* need in `STACK`? (assume simplicity)

A: `size`, `push` and `pop`. Maybe even `empty` and `singleton`, but we'll wing those this time.

```
signature STACK =
sig
  type 'a stack

  val size : 'a stack -> int
  val push : 'a stack * 'a -> 'a stack
  val pop : 'a stack -> ('a stack * 'a) option
end
```

This is a *polymorphic*, *abstract* data structure that lets us create stacks of any type `'a`. This is of course a very simplistic interface, but good enough for this illustration.

In the 15-210 labs, we maintain a convention that any module prefixed with "`Mk`" ("make") is a functor, anything in capitals is a signature, and anything else is a structure. Usually, the functor `MkFooBar` produces the structure `FooBar` that ascribes to the signature `FOO_BAR`.

Q: Aside: Make sense of `BARESEQUENCE`, `PARTIALSEQUENCE`, `SEQFUN`, `BareArraySequence`, `MkPartialArraySequence` and `MkSeqFun`.

A: These are the modules in *MiniLab*. `BareArraySequence` ascribes to `BARESEQUENCE`, and is passed into `MkPartialArraySequence` which returns a structure that ascribes to `PARTIALSEQUENCE`. `MkSeqFun` takes in a structure that ascribes to `PARTIALSEQUENCE` and returns a structure that ascribes to `SEQFUN`.

Let's now convert `YumPancakes` into a functor `MkPancakes` in `MkPancakes.sml`. This functor takes as input a structure that ascribes to `STACK`, and returns a structure that ascribes to `PANCAKES`.

```
functor MkPancakes (structure Stack : STACK) : PANCAKES =
struct
  open Stack

  type pancake = real  (* Pancake weights *)
  type pile = pancake stack

  fun serveTop P =
      case pop P of
          NONE => NONE
        | SOME (_, p) => SOME p

  fun serveBottom P =
      case pop P of
          NONE => NONE
        | SOME (P', p) => if size P' = 0
                          then SOME p
                          else serveBottom P'
end
```

Notice that now all operations are in terms of the input `Stack` structure, and a `pile` is now simply a `pancake stack`.

Q: Aside: If in the future we decide to represent a pancake by its unique *color* (or . . . *name*?), what can we do?

A: We can just create a separate functor `MkColorPancakes` that accepts and ascribes to the same signatures, and just change the type `pancake` to something like `string`.

Now let's implement a structure `SeqStack` that implements our `STACK` signature using sequences:

```
structure SeqStack : STACK =
struct
  type 'a stack = 'a Seq.seq

  val size = Seq.length  (* Note the currying *)

  fun push (S, e) =
      Seq.append (Seq.singleton e, S)

  fun pop S =
      if size S = 0
      then NONE
      else SOME (Seq.drop (S, 1), Seq.nth S 0)
end
```

Note that just as `MkPancakes` has nothing to do with the implementation of its `Stack` argument, `SeqStack` clearly has no reference to pancakes. This unambiguous separation of *abstract data type* and *implementation* is what makes the SML module language so powerful!

Q: What if we don't want to implement `STACK` using sequences?

A: This is the whole purpose of using a functor in the first place! Say we want to implement `STACK` using lists. We simply create another structure we're calling (*surprise, surprise*) `ListStack`.

```
structure ListStack : STACK =
struct
  type 'a stack = 'a list

  val size = List.length  (* Note the currying *)

  fun push (S, e) = e::S

  fun pop nil = NONE
    | pop (e::S) = SOME (S, e)
end
```

What's interesting here is that the code for `ListStack` is much more concise than that of `SeqStack`. Furthermore,

Q: What's the work of `Seq.append (S, T)`?

A: $O(|S|+|T|)$ according to our docs[2]

Q: What's the work of `List ::` (cons)?

A: $O(1)$

[2]http://www.cs.cmu.edu/~15210/docs/cost/ArraySequence.html

Q: How about the span for both?

A: $O(1)$ for both; no difference.

Q: So what's another benefit of using `ListStack`?

A: `ListStack.push` is not only more concise, but also more efficient.

4.4 Putting this all together

We can now finally put our functor (*structure factory*) to work:

```
structure SeqPancakes = MkPancakes (structure Stack =
                                     SeqStack)
structure ListPancakes = MkPancakes (structure Stack =
                                      ListStack)
```

Calling `SeqPancakes.serveBottom` and `ListPancakes.serveBottom` will result in *logically equivalent* results. That is to say, if passed the same `pile` value, they will both evaluate to the same value. The equivalence between `SeqPancakes` and `ListPancakes` holds true not just for `serveBottom`, but for any function exposed by the `PANCAKES` signature, which both structures ascribe to.

Note that this equivalence only holds because both `SeqStack` and `ListStack` implement stacks correctly. The `STACK` signature enforces the *existence* and *types* of functions in structures that ascribe to it, but not the *correctness* of their implementations. Consider what would happen if `ListStack.pop` was implemented as follows:

```
fun pop nil = NONE
  | pop e::S = NONE
```

Constructing `ListPancakes` with a functor as above would still work (type-check), but the resulting structure wouldn't serve pancakes as expected anymore! Modularization is very useful, but bear in mind that a module system that type-checks doesn't ensure the correctness of the implemented program.

5 Analyzing `SEQUENCE` Cost Bounds

A very important skill in this course is to be able to quickly and accurately determine the cost of a function by looking at its SML code or pseudocode. This is especially true when the function is implemented using our library functions.

Let's practice with some `SEQUENCE` functions; this will come in handy when you do *MiniLab*. Let S and T be some sequences, and $n = |S|$, $m = |T|$.

```
val a = map Int.toString S
(* Work: O(n) *)
(* Span: O(1) *)
```

```
fun doStuff ((a,b),(c,d)) =
    (Int.min(a,c), Int.max(b,d))
val b = reduce doStuff (0,0) (zip S T)
(* Work: O(min(n,m)) *)
(* Span: O(log(min(n,m))) *)
```

```
(* assume someSortedList contains >= 1 elements *)
val first = nth someSortedList 0
val c = iter (fn ((a, S), b) => if a = b then (a, S)
                                 else (b, append(S, singleton b)))
        (first, singleton first)
        someSortedList
(* Work: O(n^2) *)
(* Span: O(n) *)
```

```
val d = map (fn x => map (tabulate (op ~))
                      (append (x, %[1,2,3]))) S
(* Work: O(n + sum of all elements in list) *)
(* Span: O(1) *)
```

```
fun perms S =
    case length S of
      (0 | 1) => singleton S
    | _ =>
      let
        fun perms' (i, _) =
            map (fn S' => append(singleton(nth S i),  S'))
                (perms(append(take(S,i), drop(S,i+1))))
      in
        flatten (mapIdx perms' S)
      end
val e = perms S

(* Work: O(n^(n+2)) *) (* Span: O(??) *)
```