Lecture 25: Stacks
10:00 AM, Nov 1, 2019

Contents

1 Warm Up ................................................. 1
  1.1 Unit .................................................. 1

2 Writing Checks ....................................... 2
  2.1 checkExpect ......................................... 2
  2.2 checkError ........................................... 2

3 Module: Stacks ........................................... 3
  3.1 Module Types .......................................... 4
  3.2 Modules .............................................. 5

4 Summary ................................................. 7

1 Warm Up

You remember the take procedure. It consumes a list and a number and takes the first n items from the list. Sometime you have a 20 element list and you only want the first 5 so you take 5. It is going to be a recursive procedure that looks like this:

```ocaml
let rec take = (n : int, lst: list('a)): list('a) =>
switch (n,lst){
| (0, _) => []
| (n, []) => failwith("Tried to take more elements than available")
| (n, [hd, ...tl]) => [hd, ...take(n-1, tl)]
};
```

After compiling the line with (n, []) will give a warning because n isn’t used in the right hand side. You might want that n even though you are not using it because you know the Input will be a number and an empty list. You can write an underscore before the n that works like an underscore to avoid the warning.

This weeks homework we are providing you with take, drop and nth.
1.1 Unit

The type unit is a special type in Reason that has exactly one value associated to that type. You write that value as (). It is used in `print_string : string => unit`. A lot of the built-in procedures use unit to interact with the operating system. It is also used as the argument type for a function that takes no arguments. For example consider `let f = () => 4;` and `inline f = 4;` Why would the first over the latter? Let’s look at a situation where this matters: `let f : unit => int = () => 10/0;` versus `let g = 10/0;` g will fail to compile right away.

Since we are looking at weird things... what is the type of the `failwith` function?

```reason
let dizzy : int => int = fun
| 0 => failwith("can't divide by 0")
| n => 10/n;
```

According to this function `failwith` must output an int because of the type signature. Now let’s look at this function:

```reason
let tizzy : string => string = fun
| "abc" => failwith("No alphabets!")
| s => s;
```

According to this type signature, `failwith` must output a string. How can it be both a string and a number. Actually `failwith`’s type signature is `string => 'a`. Failwith is a function that produces a value. What value does it actually produce? the key is it doesn’t produce a value because it halts program execution so it doesn’t have to produce a value.

2 Writing Checks

2.1 checkExpect

checkExpect is a lovely thing in Drracket but we had to write our own in Reason. How would you write one?

```reason
checkExpect: ('a, 'a, string) => unit
let checkExpect: ('a, 'a, string) => unit = (a,b,s) =>
if (a == b) {()} else { print_endline(s)};
```

Since we are using `==` how do we know it works with functions? Let us say we have an addition function `f2`. We can’t write checkExpect(f2, +, "same") but we can do checkExpect(f2(3, 5), 3+ 5, "same") This doesn’t tell us if the functions are the same, we just know that they both do the same thing when 3, 5 are passed in which is closer than before.

2.2 checkError

checkError is very similar.
checkError:
let checkError: ('a, string, string) => unit = (x, msg, s) =>
if (???){()} else { print_endline(s)};

This is hard to write because if something produces an error, processing stops and we never get to testing. What we are actually going to do is pass in a function that produces a unit. The actual type signature is let checkError: () => 'a, string, string => unit; The use of unit to make a function lets us delay evaluation of the "bad part" until we can handle it.

How do we handle it? We will use try blocks. Where we can try something and if it produces a value then return that value but if it produces an error then return something else. They typically look like this:

try(100/x) {
| _ => 17
};

If you do this when x=5 then it will produce 20 but if x=0 then it will print the number 17. We can raise our own exceptions with Failure.

try(failwith("my message"); 17) {
| Failure("My message") => 22;
};

Here if the thing in the try raises an exception, we managed to catch it and produce a value, 22, instead. When writing checkError there are really three cases instead: 1. The raised exception is the expected one (pass)
2. An exception is raised, but it is not the right one (fail)
3. No exception is raised (fail)
The details of this code is in the CS17 setup support code. You will never have to use this in this course.

3 Module: Stacks

We have seen that Reason lets us define data types. In addition, Reason lets us define “operation” types. That is, we can declare a data type and specify the interface through which we access values of that type. This specification is called a module type in Reason, and its implementation is called a module. We illustrate the syntax for module types and modules by example.

Our first example of an abstract data type is a stack. A stack is based on the principle of LIFO: “last in, first out.” Imagine a face-up pile of cards. If you want to look at the top card, you can do so with very little effort. It’s similarly easy to remove the top card, or to put a new card on the pile, or to check if the pile is empty. But doing anything else—like looking at the middle card, for example—is harder (computationally). Does this remind you of anything?

it is similar to lists: Lists are a really good way to implement a stack. The empty list [] could be used as an empty stack, “cons” could be used to write push, “rest” could be used to write pop,
and “first” could be used to write top. But, this is only one implementation of a stack. There are a bunch of others, several of which you will see if you continue on to take CS 18. Though, these implementations do the same thing in the sense that they allow you to store and retrieve data in a sequential way.

The operations a stack supports are the following:

- check if the stack is_empty:
  
is_empty : stack('a) => bool

- push a datum onto the top of the stack:
  
push : ('a, stack('a)) => stack('a)

- pop the most-recently pushed datum off the stack; and
  
pop : stack('a) => stack('a)

- look at the datum on the top of the stack.
  
top : stack('a) => 'a

Typically stacks contain items of all the same kind. Stacks are most often used as 'processes' in your computer. The list of which functions are working on your computer are stored in a stack. A stack trace shows you the list of all the functions in your program that got you to the error.

3.1 Module Types

A module type, or a signature, is a specification of how you use an ADT. That is, it provides the interface to the ADT—the operations defined on data of that type.

A signature contains three things:

1. A data type: what name do we want to use for data of this type, and how (if at all) is this type parameterized? N.B.: A signature may actually contain many data type descriptions, but for our first couple, we'll just have one type in each. When we get to the Game project, a GAME will have a type for moves, a type for the game-progress (either “still in play” or “over”), a type for the game-state (i.e., where the checkers are on the checkerboard, or which squares have x’s or o’s in them for tic-tac-toe).

2. Constructors: how do we create values of this type?

3. Other operations: what can we do with values of this type once we’ve constructed them?

Let’s fill in our signature based on the answers to these three questions.

1. What name do we want to use to refer to our concrete representation of stacks? For the sake of simplicity, we’ll just use Stack. But that isn’t quite enough. Stacks, like lists, are containers, and so a stack must contain elements of some type. We want to be able to have stacks of numbers, stacks of strings, stacks of lists, and so on. So, we have to add a type parameter:
module type STACK =
    type stack('a);

This leads us to the following signature:

module type STACK = {
    type stack('a);

    /* Constructs an empty stack */
    let empty: stack('a);

    /* Tests whether a stack is empty */
    let is_empty: stack('a) => bool;

    /* Pushes an item onto a stack */
    let push: ('a, stack('a)) => stack('a);

    /* Pops an item off a stack */
    let pop: stack('a) => stack('a);

    /* Peeks at the top of a stack */
    let top: stack('a) => 'a;
};

In sum, a signature consists of a possibly (and in fact, usually) parametric type declaration, followed by several `val` declarations in which the constructor and various other operations provided by the ADT are specified in terms of the given type declaration.

### 3.2 Modules

We’ve now specified our ADT. But how do we implement it? To do so, we’re going to use yet another Reason construct, called **modules**. As you might expect, modules look very similar to signatures. In fact, we can use a signature as a template for writing a module.

Here is our signature, again:

module type STACK = {
    type stack('a);

    let empty: stack('a);

    let is_empty: stack('a) => bool;

    let push: ('a, stack('a)) => stack('a);

    let pop: stack('a) => stack('a);

    let top: stack('a) => 'a;
};
And here is a module template whose structure mimics that of our signature:

```ocaml
module ListStack = {
  type 'a stack = ...
  let empty = ...
  let is_empty (<stack('a)>) : bool = ...
  let push (<'a>, <stack('a)>) : stack('a) = ...
  let pop (<stack('a)>) : stack('a) = ...
  let top (<stack('a)>) : 'a = ...
}
```

Here, <'a> and <stack('a)> are placeholders for arguments of those types.

Signatures and modules look similar to one another. What are the key differences between them?

- You explicitly specify which signature the module is implementing (e.g., STACK).
- The module itself has a name (e.g., ListStack).
- The second line is struct instead of sig.
- Each occurrence of val in a signature corresponds to an occurrence of let in a module.
- All types and identifiers declared in the signature must be defined in the module. Whatever else is deemed necessary (e.g., helpers) can also be defined in the module; but such definitions won’t be part of the interface.

Now, let’s fill in our template. How do we want to implement stacks? Well, we need to store an arbitrary amount of data, so let’s try using lists.

```ocaml
module ListStack : STACK = {
  type stack('a) = Stack(list('a));
  let empty = Stack([]);
  let is_empty = s => s == empty;
  let push = (datum, Stack(l)) => Stack([datum, ...l]);
  let pop = (Stack(l)) =>
    switch (l) {
      | [] => failwith("empty stack")
      | [hd, ...tl] => Stack(tl)
    };
```
let top = (Stack(l)) =>
    switch (l) {
    | [] => failwith("empty stack")
    | [hd, ...tl] => hd
    };

How would we test our stack?

We are going to rename the module to TestListStack and include testing functions. Then write
module ListStack = TestListStack:Stack We can also write a function that will print out our stacks.

let rec printIntStack: Stack(int) => unit =
    fun
    | Stack([]) => print_endline("!")
    | Stack([hd, ...tl]) => (print_int(hd); print_string(", ");
    printIntStack(Stack(tl)));

This is helpful for printing out stacks to test implementation. If we did let d = ListStack.empty; reason will hide the type from us. This is actually a very useful thing that allows us to only see a ListStack through its signature. This lets us change our implementation of ListStack without breaking any program that uses it. You have been using things like this all along. It allows the implementers of reason to go back and change the implementation of plus to something more efficiently and you won’t even know.

4 Summary

Ideas

- ADTs offers implementation neutrality - in other words, we can switch data representation without affecting the rest of our program.

Skills

- We learned how to write a module type/signature - which is a specification of how to use an ADT.

- We learned to use modules, which look very similar to signatures. All types and identifiers declared in the signature must be defined in the module. Other helpers can be defined within a module, but won’t be part of the interface.

- We learned how to view data within abstract data types, for testing purposes.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).