Lecture 29: Modules & Stacks
10:00 AM, Nov 12, 2018

Contents

1 OCaml Surprises 1
   1.1 Redefinition .............................................. 1
   1.2 Type Redefinition ........................................... 1
   1.3 check_error ............................................... 1

2 Modules and Stacks 2
   2.1 Modules as a Tool for Organization ................. 2
   2.2 Examples of Modules ..................................... 2

3 An Example: Stacks 3
   3.1 Quiz? .................................................... 4

1 OCaml Surprises

1.1 Redefinition

Let’s define a variable, say a, to be some value, say 5:

```ocaml
let a = 5;;
```

and then define some function to operate on a:

```ocaml
let f = function x -> x + a;;
```

When invoked, this function will always return \(x + 5\) regardless of any redefinition of a. That is, the function uses the value of a that existed when it was created rather than the current value of a.

1.2 Type Redefinition

When we redefine types, the resulting instances of these types are not equal. Consider the following:

```ocaml
type shade = Gray of int;;
let g = Gray 10;;
type shade = Gray of int;;
let h = Gray 10;;
g = h;;
```

The last line here will throw an error because g and h have different types.
1.3 check_error

check_error takes in two arguments; the second is the expected contents of the exception produced by “failwith”. The function can be used as follows:

```ocaml
check_error (function () -> (transpose []))
  "A matrix can't be 0-dimensional."
```

The first argument is a procedure which, when called, should fail. You may think of () as a tuple with no elements. The proper term for this is a “unit”. If instead we just ran

```ocaml
check_error (transpose []) "A matrix can't be 0-dimensional"
```

then when evaluated, transpose would throw an error, preventing check_error from ever being run!

What type does failwith have? Well, it depends on the output type of our function. In order for our function to have a consistent output type, the failwith must take on the same type as whatever our function is supposed to return. For example, if we had a function:

```ocaml
let divide_ten : int -> int = function
  | 0 -> failwith "Cannot divide by zero."
  | x -> 10 / x
```

then the “type” of this failwith would be int.

However, what nothing is actually produced. When failwith is invoked, processing halts.

2 Modules and Stacks

2.1 Modules as a Tool for Organization

When software gets big, it gets messy and impossible for one single person to manage, and taming the messiness is one of the big goals of Computer Science. One way to do this is using interfaces. An ATM, for example, is a real life interface; it provides you with useful functionality of a bank, without the user ever needing to understand what is going on behind the scenes to execute your transactions. Furthermore, the way you interact with an ATM are standard across different banks and locations. Essentially, interfaces define the way two entities interact. In OCaml, such interfaces are called modules.

2.2 Examples of Modules

One module we have been using is List, which consists of a collection of defined functions, such as List.hd, List.tl, List.nth, List.map, etc., that all operate on the same type of data (lists).

An OCaml module is therefore an assembly of types and procedures (and sometimes values). In code, a module is called a structure, or struct.
Sometimes we promise to provide a service but it doesn’t matter how, just that it gets done. For example, when we asked you to transpose a matrix in the homework, some people took a row wise approach by transposing the rest of the matrix, and then cons-ing something onto the front of each resulting row. Others took a column wise approach by mapping rest over the matrix to get the last \( n-1 \) columns, and then transposed that and cons-ed on one last row. Both ways of solving the problem got the job done! If someone decided to reimplement the transpose procedure, the procedure will still work for the user and not change anything.

Similarly, the procedure specification and type definitions in a module constitute an interface. The user does not need to know how things work, just that they do work. In OCaml, such a specification is called a module signature. Your module would also include procedures that actually implement all the procedures the signature requires. A signature is similar to a type signature, but for modules rather than procedures.

Modules can be defined as follows:

\[
\text{module Foo = struct let i = 7 end;}
\]

### 3 An Example: Stacks

We have seen that OCaml lets us define data types. In addition, OCaml 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 OCaml, 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).

The operations a stack supports are the following:

- check if the stack is empty;
  
  \[
  \text{is_empty : 'a stack -> bool}
  \]

- **push** a datum onto the top of the stack;

  \[
  \text{push : 'a -> 'a stack -> 'a stack}
  \]

- **pop** the most-recently pushed datum off the stack; and

  \[
  \text{pop : 'a stack -> 'a stack}
  \]

- look at the datum on the top of the stack.

  \[
  \text{top : 'a stack -> 'a}
  \]
Question: How is this any different than a list?

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.

So how do we implement a stack?

module ListStack =
  struct
    type 'a stack = Stack of 'a list
    let empty = Stack []
    let is_empty : 'a stack -> bool = function s -> (s = empty)
    let push : 'a * 'a stack -> 'a stack =
      function datum, Stack lst -> Stack (datum,lst)
    ...
  
A module type, or a signature, is a specification of how you use an the module. The module type says what a module must contain, but doesn’t say anything about how the elements are implemented.

Let’s define a signature for a stack. We start like this:

module type STACK =
  sig
    ...
  end

By convention, module type names are written in all capital letters. Here is the signature for stacks:

module type STACK =
  sig
    type 'a stack
    val empty: 'a stack
    val is_empty: 'a stack -> bool
    val push: 'a stack * 'a stack -> 'a stack
    val pop: 'a stack -> 'a stack
    val top: 'a stack -> 'a
  end

3.1 Quiz!

The task for today’s quiz was to write a signature for the queue data structure. The signature must include the following elements:

- empty, which starts with an empty queue
• `is_empty`, which tests if a queue is empty
• `enqueue`, which adds an element to the beginning of a queue
• `peek`, which looks at the last element of the queue
• `dequeue`, which removes an element from the end of a queue

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)