Lab 9: Abstract Data Types
12:00 PM, Nov 4, 2018

Contents

1 Introduction 1
2 Abstract Data Types 1
3 Signature 2
4 Example Module: Lists 3
5 Testing 4
6 Take 2: Sorted Lists 5
7 Analysis 6

Objectives

By the end of this lab, you will know:

- why ADTs are useful

By the end of this lab, you will be able to:

- implement sets as an abstract data type in OCaml
- represent sets in more than one concrete way

1 Introduction

We start off this lab with some definitions of common mathematical terms:

- A collection of items without duplicates is called a set.
- The items that comprise a set are called its elements.
- Two sets are said to be equal if they have the same elements regardless of order.
2 Abstract Data Types

So far in OCaml, the types we’ve defined have either been type aliases like `type 'a matrix = 'a list list`, which simply reference another existing type, or variant types, like `type pattern_element = Lit of word | One | Any`, which are types containing existing types. Both are useful; type aliases allow us to make it more clear what the inputs/outputs of a procedure are being used for, and variant types allow us to make new, more descriptive types.

However, we can’t control how these types are used. When defining `type 'a matrix = 'a list list`, we know that all of the lists need to be of equal length, but our type definition can’t enforce this. The user can create an `type 'a matrix = 'a list list` of whatever lists they want to, and OCaml won’t stop them. Even if our procedures throw errors in this case, the user can still call their list of lists an `type 'a matrix = 'a list list`, and might even believe it is one! And while variant types give us more control, since we can declare explicit new types, they don’t fix this issue; defining `type 'a matrix = Matrix of 'a list list` still lets the user make a Matrix of whatever lists they want to.

This is where abstract data types come in. Instead of just defining a type, like we do in the previous examples, we define a type, the way to make this type, and all of the ways to interact with this type. We do this in two parts. First, we define a signature (in this example, for matrices), which defines which types and procedures will exist in relation to matrices, but not how any of them will be implemented. Then, we write a model that implements these types and procedures.

The data type is abstract because the implementation details are hidden behind a signature. That is, when a user is using a module, they can only use the types and procedures given in the signature. Any helpers involved, and any variant types defined inside, are inaccessible to the user. This allows the writer of the module to control how the type is used. For example, let’s say we define the following signature for a matrix:

```
module type MATRIX =
sig
  type 'a matrix

  (* horizontally flips a matrix *)
  val horz_flip : 'a matrix -> 'a matrix

  (* vertically flips a matrix *)
  val vert_flip : 'a matrix -> 'a matrix

  (* transposes a matrix *)
  val transpose : 'a matrix -> 'a matrix

  (* creates a matrix from a list *)
  val matrix_of_list : 'a list list -> 'a matrix

  (* creates a list from a matrix *)
  val list_of_matrix : 'a matrix -> 'a list list

end
```

Regardless of what helpers we write while implementing this module, the user can only access these procedures. Additionally, if while implementing the module we write `type 'a matrix = Matrix of 'a list list`, the user will not have access to the Matrix constructor. If the user wants to make a
matrix, they’ll have to do it using `matrix_of_list`. This means we can control the dimensions of the matrix; if the user gives `list_of_matrix` a list containing lists of different lengths, we can fail right there, rather than waiting for them to transpose the matrix!

### 3 Signature

This is all very new, and probably seems pretty confusing right now. Don’t worry about this; this is a weird concept. Let’s take a look at a concrete example. In this lab, we’ll be writing an implementation of sets using abstract data types. Below, we have the signature for a set that we’ll be using.

A signature is a specification of how you can use an ADT. It describes all of the data types and procedures that the ADT has, but it doesn’t actually implement any of these things. In other words, a signature describes everything a set can do, but not how it does it.

The signature for sets is as follows:

```plaintext
module type SET =
  sig
    type 'a set

    (* The empty set *)
    val empty : 'a set

    (* Tests whether a set is empty *)
    val is_empty : 'a set -> bool

    (* Tests whether an element is in a set *)
    val member : 'a * 'a set -> bool

    (* Inserts a value into a set if it is not already there *)
    val insert : 'a * 'a set -> 'a set

    (* Deletes a value from a set if it is there *)
    val delete : 'a * 'a set -> 'a set

    (* Tests whether a first set is a subset of a second *)
    val is_subset : 'a set * 'a set -> bool

    (* Calculates the set difference between two sets *)
    val set_difference : 'a set * 'a set -> 'a set

    (* Creates a set from a list *)
    val set_of_list : 'a list -> 'a set

    (* Creates a list from a set *)
    val list_of_set : 'a set -> 'a list
  end
```

**Task:** You can find this signature in `/course/cs017/src/lab09/set_sig.ml`. Go ahead and copy this file into your `lab09` directory.
We also have an example module to help you get started in this lab. You can find this file in /course/cs017/src/lab09/list_set.ml

And while you’re at it, copy over /course/cs017/src/lab09/abstract.ml, and /course/cs0170/src/ocaml/CS17setup.ml as well.

4 Example Module: Lists

Now, we’ll take a look at an implementation of sets. Here, we have sets implemented using lists of unique items.

Take a look at the list_set.ml file that you just copied to your lab09 folder.

This is a module, ListSetI, which implements the signature SET with lists. As such, it will contain the instructions for how to run the procedures described in the set signature.

In other words, the signature describes everything a set can do. The module describes how it does it.

Task: Look at the top lines of list_set.ml. Specifically, these ones:

```ocaml
#use "set_sig.ml" ;;
module ListSetI : SET =
struct
```

This creates a module called ListSetI that implements SET. Notice how modules are declared almost the same way as signatures, but use the keyword struct instead of sig.

Task: Take a look at the rest of the file. Notice how we have implemented all of the data types and procedures that we specified in our signature.

Note: We used the Set constructor for our set type so that we don’t confuse lists and sets within our module.

Note: Here are explicit definitions of subset and set difference:

**Definition:** A set $P$ is a *subset* of $Q$ if every element of $P$ is an element of $Q$.

**Definition:** The *difference* between two sets $P$ and $Q$, written $P \setminus Q$, contains all elements of $P$ that are not also in $Q$.

5 Testing

If you write your test cases after ListSetI, you may notice that you cannot test any helpers you used in ListSetI. This is because ListSetI implements SET. To anything outside of the ListSetI module, ListSetI only describes how SET does things, and no more. You also can’t directly create a Set by doing Set [1; 2; 3], for example; you have to use list_to_set.

Therefore, in order to be able to test our helpers, we need to backtrack a bit. We need to change ListSetI so that it no longer implements SET. In order to do this, delete : SET after the name of
your module.

**Task:** Change ListSetI so that it no longer implements SET.

You can now write your test cases after the ListSetI module.

**Task:** Write your test cases after the ListSetI module. Run them to make sure they all pass.

**Note:** To access anything defined inside a module, you need to write ModuleName.procedure, instead of writing just the procedure’s name. This can get tedious after a while, though. To access everything in the procedure without prefixing the module’s name, add the following above your test cases: open ListSetI ;;

In other words, your tests could look something like this:

```plaintext
open ListSetI ;
check_expect insert ...
```

ListSetI does not implement SET anymore, which means that programmers do not have a guarantee that ListSetI contains all the procedures it needs to be a set. As such, if a programmer wants to take a set as input to a procedure they are writing, a ListSetI would not be considered valid input. It also means that everyone has access to ListSetI’s helpers. In this case, it doesn’t matter that much, but if you’re working on a complicated library, you may not want other programmers to see what goes on behind the scenes.

To address this problem in a way that still allows you to test your code, create a new module called ListSet like so:

```plaintext
module ListSet = (ListSetI : SET);;
```

This makes the module ListSet, and sets it equal to ListSetI that implements SET.

**Task:** Underneath your test cases, create ListSet.

Now, you have a module called ListSet that implements SET, and everything in it has been tested.

There’s one more thing: we wrote test cases that should pass for any implementation of a set in abstract.ml. If you run the file, it will test your implementation of ListSet. This is because AbstractSet is set equal to ListSet: SET, and thus, equal to your implementation of ListSet.

If you wanted to test a different implementation of SET, you would only have to change ListSet to the other module name. This is one example of how useful abstraction can be, and we’ll be using this method of testing on future assignments.

**Task:** Write more test cases in abstract.ml to further test your implementation.

**Note:** In these test cases, you’ll have to use list_of_set to create sets, and set_of_list to turn them back into lists if you want to check the elements.

You’ve reached a checkpoint! Please sign up to get a lab TA to review your work.

### 6 Take 2: Sorted Lists

Now it’s your turn! Change the way we represent sets: instead of storing sets as lists with no duplicates, we’ll store them as lists with no duplicates, in sorted, strictly increasing order. Assume that any list inputted into one of your procedures is sorted.
**Task:** Write a new set module, `SortedListSet` that uses this new representation. Follow the same process that we used for `ListSet`, i.e. first write `SortedListSetI`, and then set

```
SortedListSet = (SortedListSetI : SET);
```

Feel free to refer back to `list_set.ml` as you work! Here’s what your file should resemble as you get started:

```plaintext
#use "set_sig.ml" ;;
module SortedListSet : SET =
  struct
    type 'a set = Set of ...
    let empty : 'a set = ...
    let is_empty : 'a set -> bool = ...
    let member : 'a * 'a set -> bool = ...
    ...
  end ;;
```

**Note:** Because your lists are sorted, you should be able to do some tasks much faster. In order to do this, try not to use `member` in `is_subset` or `set_difference`.

Save your work in a file called `sorted_list_set.ml`, and include this line at the top of your `sorted_list_set.ml` file: `#use "set_sig.ml" ;;`

**Task:** Load the file `abstract.ml` once again—the extended version, with your own test cases. At the top, write

```plaintext
#use "sorted_list_set.ml" ;;
```

Now change the `AbstractSet` definition from

```
module AbstractSet = (ListSet : SET) ;;
```

to

```
module AbstractSet = (SortedListSet : SET) ;;
```

Rerun all the test cases. Do they all pass? They should! If they don’t, make sure that the order of the elements in the set matches your implementation of sorting!

The beauty of implementing sets as an abstract data type is that you can convert from one representation to another without impacting the functionality of other code that depends on that ADT. Beyond having to adopt your new representation (here, changing the definition of `AbstractSet`), all other parts of your code should run exactly as is!
7 Analysis

Next, let’s analyze the runtime of `is_subset` and `set_difference` with our two implementations of `Set`.

**Task**: Suppose the inputs to `is_subset` are the sets $P$ and $Q$, and suppose further that $P$ has $m$ elements and $Q$ has $n$ elements. What are the worst-case run times of `is_subset` and `set_difference` for ListSets in terms of $m$ and $n$?

**Task**: What are the worst-case run times of `is_subset` and `set_difference` for SortedListSets in terms of $m$ and $n$?

*Do the run times of either of these procedures vary with the choice of representation?*

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

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/csci10170/feedback](http://cs.brown.edu/courses/csci10170/feedback)