CS51 **Spring 2024**

Code Review 4: Modular Programming

1 Modules

Definition 1.1 (Module)**.** A **module** is the packaging of pieces of software such as functions and types.

A module is specified by placing the definitions of its keywords between the keywords `struct` and `end`:

```
module <modulename> =
  struct
   <definition1>
   <definition2>
   <definition3>
   ...
  end
```

We can define types for modules akin to how we define types for values and functions. A **module signature** defines the **interface** of a module, the types and values a module must provide.

To define a module signature, we use the following syntax, placing **type definitions** between the `sig` and `end` keyword:

```
module type <typename> =
 sig
  <defintion1>
  <definition2>
  <definition3>
  ...
 end
```

We can write that a module `<modulename>` follows a module signature `<moduletype>` by using the **:** operator, just as we do with `OCaml` expressions.

```
module <modulename> : <moduletype> =
  struct
   <definition1>
   <definition2>
   <definition3>
   ...
  end
```

Example 1.2 (Math Module)**.** In lab 7, we defined a `Math` module that satisfies the `MATH` type signature.

Listing 1: Math Module

```
module type MATH =
  sig
    (* the constant pi *)
    val pi : float
    (* cosine of an angle in radians *)
    val cos : float -> float
    (* sine of an angle in radians *)
    val sin : float -> float
    (* sum of two numbers *)
    val sum : float -> float -> float
    (* maximum value in a list; None if list is empty *)
    val max_opt : float list -> float option
  end ;;


module Math : MATH =
  struct
    let pi = 3.14159
    let cos = cos
    let sin = sin
    let sum = (+.)
    let max_opt lst =
      match lst with
      | [] -> None
      | hd :: tl -> Some (List.fold_left max hd tl)
  end ;;
```

The `val` keyword is used to declare the types of values within a module signature.

Remark 1.3 (Opening a module)**.** There are two ways that we'll typically open or use a module:

1. **Dot Notation**. For instance, we can define the we can use the `Math` module above to find the maximum element in a list.

```
Math.max_opt [5;9;3;2;1;6]
```

2. **Open keyword**. We can use the `open` keyword, and then use values from the module without prefixing with a dot notation and the module name.

```
(* Method 1 *)
let open Math in max_opt [5;9;3;2;1;6] ;;

(* Method 2 *)
open Math ;;
max_opt [5;9;3;2;1;6]
```

As a design and style note, we'll typically use `open` when we want to use the same values or functions from a module multiple times in a single file. However, even though prefixing with the module name can be longer syntactically, when we may have conflicts between function or value names, it is better to use dot notation, which follows from the **edict of intention**.

2 Abstract Data Types

Definition 2.1 (Abstract Data Type)**.** An **Abstract Data Type** (ADTs) is a type for objects whose behavior is **defined by its values and operations**. The definition of an ADT only provides the **type signatures of values and operations**, **not** how these operations are **implemented**. They are called "abstract" because the implementation details are hidden. [1]

Modules allow us to implement abstract data types and enforce **invariants**, properties of the ADTs that must hold true.

Example 2.2 (Stack)**.** An example of an abstract data type is a **stack**, a collection of elements that enforces the **Last-in-First-Out (LIFO)** invariant, which means that the last element *pushed* on to the stack must be the first element *popped* from the stack.

We can define a stack to support the following values and operations:

- `empty` : A stack with no elements
- `PUSH(i, stack)` : Pushes an element `i` onto `stack`
- `POP(stack)` : Pops an element from a `stack`
- `TOP(stack)` : Returns the element at the top of the `stack`

In lab 7, we first tried implementing a stack of integers, **internally as a list**.

Listing 2: IntListStack

```
module IntListStack =
 struct
 exception EmptyStack

 type stack = int list

 let empty : stack = []

 let push (i : int) (s : stack) : stack =
   i :: s

 let top (s : stack) : int =
   match s with
   | [] -> raise EmptyStack
   | h :: _ -> h

 let pop (s : stack) : stack =
   match s with
   | [] -> raise EmptyStack
   | _ :: t -> t
 end ;;
```

[1] https://www.geeksforgeeks.org/abstract-data-types/

There's a problem with this implementation. The **internal implementation of a stack is not hidden to a user of the `IntListStack` module**. Therefore, a user can break the `LIFO` invariant by calling a function like `List.rev`.

As shown in lab 7, the following code would break the invariant:

Listing 3: Bad Stack

```

let small_stack () : IntListStack.stack =
 let open IntListStack in
 empty
 |> push 5
 |> push 1 ;;

 let invert_stack : IntListStack.stack -> IntListStack.stack =
 List.rev ;;

 let bad_el = IntListStack.top (invert_stack (small_stack ())) ;;
```

Here, defining `bad_el` breaks the LIFO invariant. `invert_stack` **directly** reverses the elements in the stack; we shouldn't be able to invert a stack without pushing and popping the elements in last-in-first-out order (**see Problem 2.3**).

Problem 2.3. Create a proper implementation of a stack that allows you to correctly reverse the elements of a stack without breaking the LIFO invariant.

We can avoid this problem by **using a module signature**, proving an **abstract data type** that a user of the module can use.

Listing 4: INT_STACK

```
module type INT_STACK =
 sig
  exception EmptyStack
  type stack
  val empty : stack
  val push : int -> stack -> stack
  val top: stack -> int
  val pop : stack -> stack
 end ;;
```

Here, we **hide the internal implementation of** the `stack` abstract data type behind an **abstraction barrier** enforced using the `INT_STACK` module type. In other words, we provide the **abstract** data type `stack` without revealing the **concrete** implementation of the data type.

Using this module type signature, we can redefine or `IntListStack` module.

Listing 5: SafeIntListStack

```
module SafeIntListStack = (IntListStack : INT_STACK) ;;

let example_stack : IntListStack.stack =
  let open IntListStack in
  empty
  |> push 5
  |> push 1 ;;

// This would result in an error! Try testing this in the REPL!!
let example_element : int =
  example_stack
  |> List.rev
  |> IntListStack.top ;;
```

3 Polymorphic Modules

We can also define modules that implement **polymorphic** abstract data types.

Here's how we would define stacks, by providing the polymorphic abstract data type `'a stack`:

Listing 6: Polymorphic Stack

```
module type STACK =
  sig
    exception EmptyStack
    type 'a stack
    val empty : 'a stack
    val push : 'a -> 'a stack -> 'a stack
    val top : 'a stack -> 'a
    val pop : 'a stack -> 'a stack
  end ;;


module Stack : STACK =
  struct
    exception EmptyStack

    type 'a stack = 'a list (* We've chosen to implement stacks
                               internally as lists, a natural
                               and simple choice *)
    let empty : 'a stack = []

    (* push i s -- Adds an element i to the top of stack s *)

    let push (elt : 'a) (stk : 'a stack) : 'a stack =
      elt :: stk
```

```

  (* pop_helper s -- Returns a pair of the top element of the
     stack and a stack containing the remaining elements *)
  let pop_helper (stk : 'a stack) : 'a * 'a stack =
    match stk with
    | [] -> raise EmptyStack
    | hd :: tl -> (hd, tl)

  (* top s -- Returns the value of the topmost element on stack s,
     raising the EmptyStack exception if there is no element to be
     returned. *)
  let top (stk: 'a stack) : 'a =
    fst (pop_helper stk)

  (* pop s -- Returns a stack with the topmost element from s
     removed, raising the EmptyStack exception if there is no
     element to be removed. *)
  let pop (stk : 'a stack) : 'a stack =
    snd (pop_helper stk)
end ;;
```

See Lab 7 for more details.

4 Sharing Constraints

Definition 4.1 (Sharing constraints)**.** A **sharing constraint** adds additional information to a module signature, letting the user know about one or more type equalities. Consider Listing 4. Suppose, instead of a stack of `int`'s, we wanted to define a stack of `float`'s.

We could define a whole new signature for stacks of `float`'s.

Listing 7: FLOAT_STACK

```
module type FLOAT_STACK =
  sig
    exception EmptyStack
    type stack
    val empty : stack
    val push : float -> stack -> stack
    val top : stack -> float
    val pop : stack -> stack
  end ;;
```

However, this is repetitive. `INT_STACK` and `FLOAT_STACK` are identical, aside from type `int` being replaced with type `float`.

Instead, we can generalize this idea and create a `STACK` signature that works with any element type.

Listing 8: Stack

```
module type STACK =
  sig
    exception EmptyStack
    type element
    type stack
    val empty : stack
    val push : element -> stack -> stack
    val top : stack -> element
    val pop : stack -> stack
  end ;;
```

Now, suppose we create a module that satisfies the `STACK` signature.

Listing 9: IntListStack

```
module IntListStack : STACK =
  struct
    exception EmptyStack
    type element = int
    type stack = int list
    let empty : stack = []
    let push elt s = elt :: s
    let top s =
      match s with
      | [] -> raise EmptyStack
      | h :: _ -> h
    let pop s =
      match s with
      | [] -> raise EmptyStack
      | _ :: t -> t
  end ;;

  let example_stack : IntListStack.stack =
  let open IntListStack in
  empty
  |> push 5
  |> push 1 ;;

(*

This gives us an error:

Error: This expression has type int but an expression was expected of
   type IntListStack.element
*)
```

However, when we try to use the module, we get an error, even though the following is true:

The `STACK` signature says there will be a type `element` and a type `stack`, but does not say how these types are implemented. The `IntListStack` says that the type `element` will be an `int` and that the type `stack` will be an `element list`.

However, recall that users of our module only have access to what's provided in the type signature. So, the fact that `element` is an `int` is hidden behind the **abstraction barrier**, as the `element` is only defined to be an `int` in the module implementation in Listing 9 (which a user can't see publicly) and not in a module signature like Listing 8.

Here **sharing constraints** come into play as a solution.

Listing 10: IntListStack (Sharing Constraint Version)

```
module IntListStack : STACK with type element = int =
  struct
    exception EmptyStack
    type element = int
    type stack = int list
    let empty : stack = []
    (* ... remainder of the module ... *)
  end ;;
```

As shown in Listing 10, we add additional information to the type signature of module. `IntListStack` is a module of type `STACK` such that `type element = int`. Since, we've now specified the appropriate type of `type element` in the signature, the module can now be used.

Remark 4.2 (When to use Sharing Constraints)**.** Sometimes students are confused about when it's appropriate to use sharing constraints.

We can think of abstract data types as data types that are hidden behind the abstraction barrier or "from the outside world". Users interact with these abstract data types by using functions and values defined by the module.

However, in some instances, we must or want to allow users to directly access abstract data types, such as with the `element` type for the `Stack` module signature.

Sharing constraints are intended to **share** information "hidden behind the abstraction barrier", so whenever we want to allow users to directly access or know more about an abstract data type, we should use a sharing constraint.

5 Functors

Definition 5.1 (Functor)**.** The OCaml documentation defines a **functor** as a module that is parameterized by another module, similar to how functions are values parametrized by other values (i.e. arguments). In other words, we can think of a functor as a function that takes a module as input and produces a module as output.

Example 5.2 (Intervals)**.** We will consider the `MakeInterval` example from lab. First, we define a module type called `ORDERED_TYPE`.

Listing 11: ORDERED_TYPE

```
module type ORDERED_TYPE =
  sig
    type t
    val compare : t -> t -> int
  end ;;

(* Modules satisfying the `ORDERED_TYPE` signature will need to
   define a type `t`, as well as a function `compare` that compares
   two values for how they are ordered. *)

module IntOrderedType : ORDERED_TYPE =
  struct
    type t = int
    let compare = Stdlib.compare
  end ;;
```

Then, we define a module signature for intervals.

Listing 12: INTERVAL

```
module type INTERVAL =
  sig
    type interval
    type endpoint
    val create : endpoint -> endpoint -> interval
    val is_empty : interval -> bool
    val contains : interval -> endpoint -> bool
    val intersect : interval -> interval -> interval
  end ;;

(* Modules satisfying the `INTERVAL` signature will allow working
  with intervals of values of type `endpoint`. *)
```

Functors can take modules satisfying one signature as input and generate modules satisfying a different signature as output. We will have our `MakeInterval` functor accept a module satisfying `ORDERED_TYPE` as input and produce a module satisfying `INTERVAL`.

Listing 13: MakeInterval

```
module MakeInterval (Endpoint : ORDERED_TYPE)
    : (INTERVAL with type endpoint = Endpoint.t) =
  struct
   (* The functor defines a new module that contains a type 'interval',
      which uses the 'Endpoint.t' type from the input module. *)
    type interval =
      | Interval of Endpoint.t * Endpoint.t
      | Empty

    let create (low : Endpoint.t)
               (high : Endpoint.t)
             : interval =
      (* The functor also makes use of Endpoint.compare, a function
         defined by the input module. *)
      if Endpoint.compare low high > 0 then Empty
      else Interval (low, high)
    (* ... remainder of the module ... *)
  end ;;

(* Here's a functor that takes a module with the 'ORDERED_TYPE'
   signature and generates a module for an interval. *)
(* The input to the functor is a module 'Endpoint' that satisfies
   the 'ORDERED_TYPE' signature. *)
(* The output of the functor is a module that satisfies the
   'INTERVAL' signature. *)
(* We want the user to have access to the endpoint type, so we
   add a sharing constraint. *)
```

Now, we can generate new modules by applying our functor to an argument module.

Listing 14: MakeInterval

```
module IntInterval =
  MakeInterval (struct
                  type t = int
                  let compare = Stdlib.compare
                end) ;;
```

Remark 5.3 (Functors with Many Parameters)**.** Like functions, functors can have multiple arguments! We can even define higher-order functors that take in other functors, as you will see in Problem Set 6.

6 Practice Problems

Problem 6.1 (Inverting a Stack)**.** Consider the polymorphic `Stack` module as shown in Listing 6.

1. Define a function `invert_stack : 'a Stack.stack -> 'a Stack.stack` that inverts the elements of the stack.
2. Write a few unit tests outside the `Stack` module testing the functionality of `invert_stack`.

Problem 6.2 (Graph)**.** An (undirected) graph is a defined set of nodes and a set of edges, where each edge is a pair of different nodes.

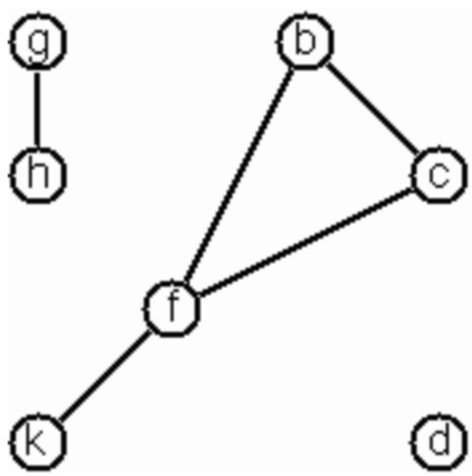

Figure 1: Graph Example

There are many ways to represent a graph. For this problem, we will consider the **adjacency list** representation, in which we represent a finite graph as a collection of unordered lists such that each list represents the neighbors (vertices that share an edge) with a particular vertex. For instance, we can represent the graph in Figure 1 as: `[('g', ['h']); ('h', ['g']); ('b', ['c', 'f']); ('c', ['b', 'f']); ('f', ['b', 'c', 'k']); ('k', ['f']); ('d', []) ]`.

Here, each element in the list is a pair, where the first element is a vertex and the second element is a list of neighbors.

We will work on defining a `Graph` module.

First, consider the following signatures for a `Vertex` and `Graph` module.

Listing 15: Vertex Module Signature

```
module type VERTEX =
  sig
    type t
    val compare : t -> t -> int
  end ;;
```

Listing 16: Graph Module Signature

```
module type GRAPH =
sig
  type v
  type graph
  exception VertexAlreadyExists
  exception VertexDoesNotExist
  exception EdgeAlreadyExists

  (* empty graph *)
  val empty: graph

  (* add a vertex to the graph *)
  val addVertex : v -> graph -> graph

  (* adds an edge between two vertices ‘u‘ and ‘v‘ *)
  val addEdge : v -> v -> graph -> graph

  (* Return a list of the vertices in the graph‘ *)
  val vertices : graph -> v list

  (* Return the neighbors of a vertex ‘v‘ *)
  val neighbors : v -> graph -> v list

  (* Return the number of vertices and edges in the graph *)
  val graph_size : graph -> int * int
end ;;
```

1. We will define a functor `MakeGraph` that takes in a module of type `Vertex` and returns a module of type `Graph`. What is the signature of this functor?

2. To define `MakeGraph`, start defining `type graph` based on the example given for Figure 1.

3. There are two main problems with this definition for `graph`. First, the `type` allows us to add the same vertex twice to the neighbors list for any vertex. Second, our list can contain the same vertex twice: for instance, the following is expressible `[('a', []); ('a', [])]`.

   Let's address the first problem by representing the neighbors as a **set**, which is an unordered collection of unique elements. Use the OCaml Set Module.

4. To address the second problem, suppose we instead represent the adjacency list using a **hash map** (or hash table). A hash map is a dictionary data structure in which we map unique keys to values. To do this, we'll use the OCaml Map Module, taking the keys to be the vertex and the values to be the neighbors set, as defined in part 3.

5. Define `addVertex` and `addEdge`. If the user attempts to add a vertex or edge that already exists, return the apporpriate exception.

6. Complete the implementation of the module by defining `vertices`, `neighbors`, and `graph_size`.

Problem 6.3 (Find Paths)**.**

1. Using the MakeGraph functor, define a module `IntGraph` such that the type of the vertices in the graph are integers.
2. Define a function `paths` that takes in an integer graph `g` and two vertices `a` and `b`, and returns a list of integer lists, such that each list represents an acyclic path from `a` to `b`. For instance, consider this graph:

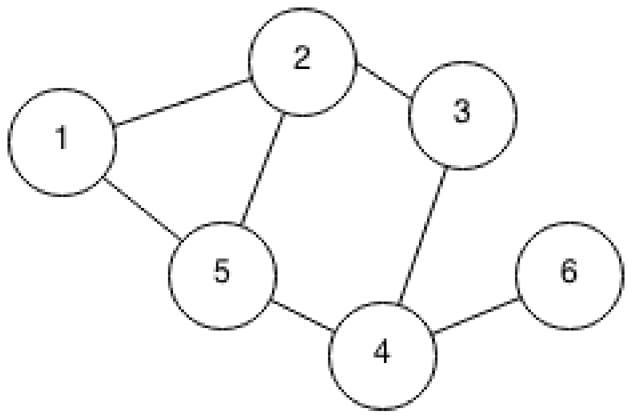

Figure 2: Integer Graph

`paths` should behave as follows (note order of list does not matter):

```
paths int_graph 2 5
- : int list list =
[[2;5]; [2;1;5]; [2;3;4;5]]
```

Problem 6.4 (Set)**.** Let's implement the `Set` module from scratch. Sets are an unordered list of elements with no duplicates.

1. Define a module signature for an `ORDERED_TYPE` which specifies the type of the elements, a function to compare elements, and a function to convert each of the elements into strings.
2. Create a module type for `Set` that includes the following values and operations on sets are supportable.
   (a) `empty` : the empty set
   (b) `add` : add an element to a set
   (c) `take` : a function that takes an element in the set and returns the rest of the elements in the set as a pair : (h, t), where h is the extract element and t are the rest of the elements.
   (d) `mem` : check if an element is a member of a set
   (e) `union` : return the union of two sets
   (f) `intersection` : return the intersection of two sets
   (g) `print_set` : convert a set into a string.

3. Write a functor `MakeSet` that takes in a module of `ORDERED_TYPE` and returns a module of type `SET`.

4. Use `MakeSet` to define the following modules:
   (a) A set of integers
   (b) A set of strings
   (c) A set of int lists
   (d) A set of integer sets

5. Define a function `power_set` that returns a set of all the subsets of the original set. For this problem, you can create a function that returns the power set for a set of integers. For instance, the power set of $\{1,2,3,4\}$ is

```
{{}, {1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4},
   {3, 4}, {1, 2, 3}, {1, 3, 4}, {1, 2, 4}, {2, 3, 4},  {1, 2, 3,
   4}}
```