

# CS 51 Code Review 4

## Modules and Functors in OCaml

---

Sam Green and Gabbi Merz

Harvard University

# Table of contents

1. Modules and Abstraction
2. Functors
3. Binary Heaps (for Ps4)

# Modules and Abstraction

---

- A **module** is a collection of values (and remember, functions are values) and types.
- A **module signature** or **module type** describes the contents of a module.
- While not precisely true, this analogy may help:

$$\frac{\text{type}}{\text{value}} \cong \frac{\text{signature}}{\text{module}}$$

# Modules

Here's a definition of the Math module:

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

Important syntax here: `module`, `struct`, `end`. What are these for? Are they analogous to other syntax we've seen so far?

# Module Signatures

The type analog for modules is the module signature. For example:

```
# module type TF =  
  sig  
    type info  
    val info : info  
    val hometown : string  
    val print_info : unit -> unit  
    val grade_assignment : int -> string  
    val favorite_function : float -> float -> float  
    val fold : int list -> int -> int  
  end ;;
```

We could then apply this signature to the Sam or Gabbi module.  
(Remember, files are module by default!)

```
module TFGabbi = Gabbi : TF ;;
```

## A BigNum Module.

What happened when you typed `#mod_use 'ps3.ml'` into utop while you were working on problem set 3?

Something like this (output omitted for space):

```
module Ps3 :  
  sig  
    type bignum = { neg : bool; coeffs : int list; }  
    val base : int  
    val negate : bignum -> bignum  
    val equal : bignum -> bignum -> bool  
  ...
```

(Notice that Ps3 has an anonymous module signature applied to it! This signature is the default, which exposes every defined type and values. It's inferred automatically by the compiler!)

What are some undesirable design properties of this default?

## Exercise: A `BigNum` Module Signature

The solution to these problems is to write a module signature and use it to enforce an abstraction barrier between client code and the `Ps3` module.

Let's imagine that the only values we wanted to include were `toInt`, `fromInt`, `plus`, `negate`, and `times`. What would the signature be? How would we create a `BigNum` module out of this signature and the `Ps3` module?



# Functors

---

# Functor: Definition and Purpose

Informally, a **functor** is a “function” from modules to modules. More precisely, a functor is a module that is **parameterized** by another module.

Some possible uses:

- Enforce abstractions.
- Make code more generic.
- Make code more extensible.

## Functor: Motivation

Imagine we needed a stack data structure as part of a system we were building. What's the “simplest” way to use the following signature for stacks with elements of several different types?

```
# module type STACK =  
  sig  
    exception Empty  
    type element  
    type stack  
    val empty : unit -> stack  
    val push : element -> stack -> stack  
    val top : stack -> element  
    val pop : stack -> stack  
    val serialize : stack -> string  
  end ;;
```

## Functor: Example

```
module type SERIALIZE =  
  sig  
    type t  
    val serialize : t -> string  
  end ;;  
  
module MakeStack (Element: SERIALIZE)  
  : (STACK with type element = Element.t) =  
  struct  
    type element = Element.t ;;  
    ...  
  end ;;
```

## Exercise: Functors

Create a module to handle stacks of `(int, int)` values.

# Binary Heaps (for Ps4)

---

# Why Binary Heaps?

The conceptual idea for this week's problem set is about modules and functors. It's also the first foray into a real abstraction.

The motivating problem is a priority queue. The implementation progression is:

1. List based.
2. Binary search tree based.
3. Binary heap based.

The binary heap implementation allows for  $O(\log n)$  operations that could be worst case  $O(n)$  in other implementations.

# Binary Heap: Definition

A binary (min)heap is a binary search tree that satisfies additional representation invariants.<sup>1</sup>

The first is an **ordering** invariant:

*The value stored at the root of any subtree (including the root of the whole tree) must be smaller than all values stored in the subtrees below the root.*

The second is a **balance** invariant:

*For any node, its left child tree is either the same size as (in number of nodes) or exactly one node larger than its right child tree.*

---

<sup>1</sup>Recall the definition of representation invariant!



# Heap Operations

The biggest challenge in implementing the binary heap is understanding the rules for inserting and popping so that these invariants are enforced. So let's this sequence of operations to get comfortable:

1. Insert 10.
2. Insert 6.
3. Insert 6.
4. Insert 5.
5. Insert 11.
6. Insert 1.
7. Take.

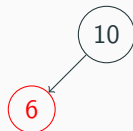
# Insert 10

1. Insert 10.

10

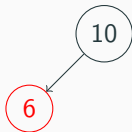
## Insert 6 for Balance

1. Insert 10.
2. Insert 6.

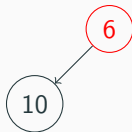


## Fix for Order

1. Insert 10.
2. Insert 6.



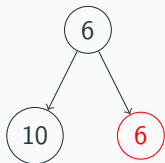
**(a)** Unfixed.



**(b)** Fixed.

## Add another 6

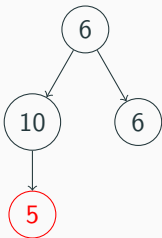
1. Insert 10.
2. Insert 6.
3. Insert 6.



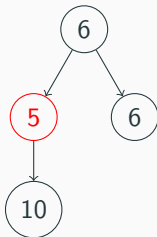
No fix required.

# Add a 5

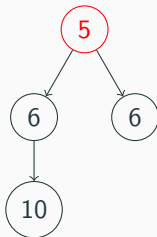
1. Insert 10.
2. Insert 6.
3. Insert 6.
4. Insert 5.



**(a)** Add 5 here for balance.



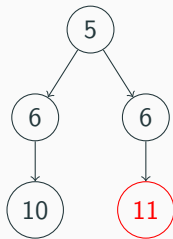
**(b)** 5 bubbles up.



**(c)** 5 bubbles up again.

# Add an 11

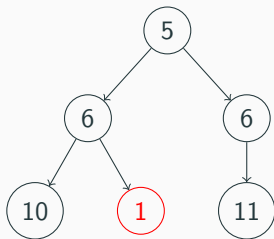
1. Insert 10.
2. Insert 6.
3. Insert 6.
4. Insert 5.
5. Insert 11.



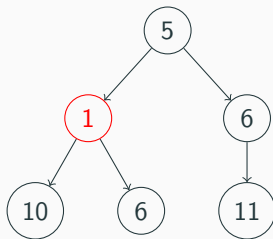
Add 11 here for balance, no fix required.

# Add a 1

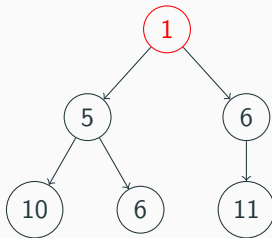
1. Insert 10.
2. Insert 6.
3. Insert 6.
4. Insert 5.
5. Insert 11.
6. Insert 1.



**(a)** Add 1 here for balance.



**(b)** Swap 1 and 6 for ordering

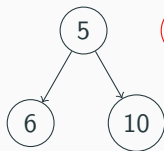
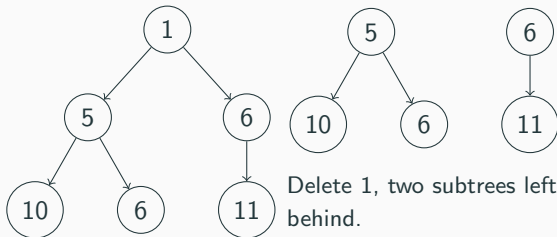


**(c)** Swap 1 and 5 for ordering

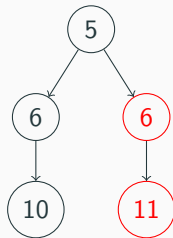


# Take the Lowest Element.

1. Insert 10.
2. Insert 6.
3. Insert 6.
4. Insert 5.
5. Insert 11.
6. Insert 1.
7. Take.



**(a)** Pick a subtree as new tree and fix it.



**(b)** Add leftover elements to the new tree.

# Takeaways

1. Understand the invariants.
2. Enforce the invariants separately.
3. Identify the recursive structure. Note that invariant properties apply to subtrees! (Hint.)
4. Draw pictures.
5. (Try not to code until you understand the data structure)

**Questions?**

**Remember to fill out the form!**

`http://sa.muel.green/cs51`