CS 320: Concepts of Programming Languages

Lecture 10: Modules & Monads

Ankush Das

Nathan Mull

Oct 3, 2024

Abstractions of the Day

- Today, we will talk about modules and monads
- Module: this will help us organize big projects into multiple files
- Modules have *signatures* (like interfaces) and *structures* (like classes)
- Monad is not a new concept; it's a *design pattern* that's very common and helpful in programming
- We will primarily look at the *Maybe monad*
- Last concept before the midterm!

Modules

- You've been using modules all along without realizing it!
- Let's look at `assign04_02.ml`

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty =
  | Int
  | Bool

expr -> ty option
let type_of (exp : expr) = Some Int
```

Modules

- You've been using modules all along without realizing it!
- Let's look at `assign04_02.ml`

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty =
  | Int
  | Bool

expr -> ty option
let type_of (exp : expr) = Some Int
```

interface

Modules

- You've been using modules all along without realizing it!
- Let's look at `assign04_02.ml`

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty =
  | Int
  | Bool

expr -> ty option
let type_of (exp : expr) = Some Int
```

interface

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty = Int | Bool

val type_of : expr -> ty option
```

Modules

- You've been using modules all along without realizing it!
- Let's look at `assign04_02.ml`

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty =
  | Int
  | Bool

expr -> ty option
let type_of (exp : expr) = Some Int
```

`ocamlc -i assign04_02.ml`

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty = Int | Bool

val type_of : expr -> ty option
```

Signatures

- The file on the right describes the signature of the code on the left
- Collect all the type definitions and function types (and exceptions)

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty =
  | Int
  | Bool

expr -> ty option
let type_of (exp : expr) = Some Int
```

Signature

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty = Int | Bool

val type_of : expr -> ty option
```

Defining a Module

- You can put functions inside a module with the following syntax:

```
module MyList = struct
  int -> int list
  let rec generate n = if n = 0 then [] else n::(generate (n-1))

  'a list -> int
  let rec length l = match l with | [] -> 0 | _::t -> 1 + length t

  exception Failure

  'a list -> 'a
  let hd l = match l with | [] -> raise Failure | h::_ -> h

  'a list -> 'a list
  let tl l = match l with | [] -> raise Failure | _::t -> t
end
```

Using a Module

- You can use module functions using dot notation
- You can also "open" a module (as you've seen in assignments)

```
int list
let l = MyList.generate 100
int
let n = MyList.length l
int
let x = MyList.hd l
int list
let y = MyList.tl l
```

```
open MyList

int list
let l = generate 100
int
let n = length l
int
let x = hd l
int list
let y = tl l
```

Modules Have Types Too!

- Everything in OCaml has a type, including modules!
- Signature of a module is its type!

```
type expr =
  | True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty =
  | Int
  | Bool

expr -> ty option
let type_of (exp : expr) = Some Int
```

Type

```
type expr =
    True
  | False
  | Num of int
  | Or of expr * expr
  | Add of expr * expr
  | IfThenElse of expr * expr * expr

type ty = Int | Bool

val type_of : expr -> ty option
```

Type of MyList Module


```
module MyList = struct
  int -> int list
  let rec generate n = if n = 0 then [] else n::(generate (n-1))

  'a list -> int
  let rec length l = match l with | [] -> 0 | _::t -> 1 + length t

  exception Failure

  'a list -> 'a
  let hd l = match l with | [] -> raise Failure | h::_ -> h

  'a list -> 'a list
  let tl l = match l with | [] -> raise Failure | _::t -> t
end
```

Type

```
module MyList :
  sig
    val generate : int -> int list
    val length : 'a list -> int
    exception Failure
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
  end
```

User-Defined Module Types


```
module type STORE =
  sig
    type 'a t
    val new_store : 'a t
    val push : 'a t -> 'a -> 'a t
    val pop : 'a t -> 'a option * 'a t
    val top : 'a t -> 'a option
  end
```

- Let's implement modules which have this type!
- Two examples: Stack and Queue

Monads

- Last topic: Monads!
- Not a new topic; just looking at a very common programming pattern
- And we will introduce a module to make this pattern easier to program with
- We will do one example, famously called, the Maybe monad
- So, what's the common pattern?
- Let's see this in code!

Conclusion

- Last concept done!
- Next week: midterm review and some theorems and proofs!
- Read OCaml book 5.1, 5.2, 5.4 for modules
- Monads: https://ocaml.org/docs/monads
- Do as much practice as possible!
- Will release solutions to mock midterm by the end of this week