CSCI 2041: Modules

Lecture 14

October 9th 2023

University of Minnesota
Driven to Discover®

Outline

- Modules
- Signatures
- A particular module

… we’ve already seen a Module!

- Recall that functions like fold_right and map on lists are built-in:
  List.fold_right
  List.map

```
utop # #show_module List;;
module List = List
module List :
  sig
    val length : 'a list -> int
    val compare_lengths : 'a list -> 'b list -> int
    val compare_length_with : 'a list -> int -> int
    val cons : 'a -> 'a list -> 'a list
    (…)
```

Creating a Module

- Unlike List, most Modules do not come pre-loaded
- We get to write them ourselves!

- Here's an example:

```
module RGB = struct
  type primary_color = Red | Green | Blue
  let inc x = match x with
    | Red -> Green
    | Green -> Blue
    | Blue -> Red
end
```

Creating a Module (with utop output)

```
module RGB = struct
  type primary_color = Red | Green | Blue
  let inc x = match x with
    | Red -> Green
    | Green -> Blue
    | Blue -> Red
end;;
module RGB :
  sig
    type primary_color = Red | Green | Blue
    val inc : primary_color -> primary_color
  end
```

Using a Module (with utop output)

```
utop # RGB.inc RGB.Red;;
- : RGB.primary_color = RGB.Green
```

Why Modules?

- Using Modules this way:
  - gives a way to structure code
  - provides a way to keep namespaces separate

- We’ll see more reasons to use Modules this week!

Namespace issue

- Q: Why do we need this?
  wouldn’t we get the same benefits from writing “RGB_inc” instead of “RGB.inc”?

- A1: Not quite! Check this out:

```
utop # RGB.(inc Red);;
- : RGB.primary_color = RGB.Green
```

Namespace issue

- Q: Why do we need this?
  wouldn’t we get the same benefits from writing “RGB_inc” instead of “RGB.inc”?

- A2: Not quite! Check this out:

```
utop # module R = RGB;;
module R = RGB
utop # R.inc RGB.Red;;
- : R.primary_color = R.Green
```

Useful in utop, avoid otherwise…

- There's an alternative to using the module name with .
- The book teaches it, I don't like it:

```
utop # open RGB;;
utop # inc Red;;
- : primary_color = Green
```

- The book teaches it, I don't like it because:
  - It breaks the namespace separation (i.e. it's confusing)
  - There's no "close" counterpart

Modules as a way to structure code…

- Structuring code has two aspects:
  - where do I put my functions?
  - from where do I call functions?
- In ocaml, we can write these agreements as “signatures”

Utop delivers a signature:

```
module RGB = struct
  type primary_color = Red | Green | Blue
  let inc x = match x with
    | Red -> Green
    | Green -> Blue
    | Blue -> Red
end;;
module RGB :
  sig
    type primary_color = Red | Green | Blue
    val inc : primary_color -> primary_color
  end
```

Writing a signature without a module

```
module type RGB_sig =
  sig
    type primary_color = Red | Green | Blue
    val inc : primary_color -> primary_color
  end
```

Why would you write signatures?

- Signature represents an agreement between a code-writer and code-user
- Signatures can be re-used
  - Documentation can be tied to signatures
  - Consequently, documentation can be reused

Use case of signatures: compilation checks

```
module type RGB_sig =
  sig
    type primary_color = Red | Green | Blue
    val inc : primary_color -> primary_color
  end
module RGB = struct
  type primary_color = Red | Green | Blue
end
module RGB_checked : RGB_sig = RGB ;;
```

```
Error: Signature mismatch:
       Modules do not match:
         sig type primary_color = RGB.primary_color = Red | Green | Blue end
       is not included in
         RGB_sig
       The value `inc' is required but not provided
```

Use case of signatures: compilation checks (2)

```
module type RGB_sig =
  sig
    type primary_color = Red | Green | Blue
    val inc : primary_color -> primary_color
  end
module RGB = struct
  type primary_color = Red | Green | Blue
end
module RGB_checked = (RGB : RGB_sig);;
```

```
Error: Signature mismatch:
       Modules do not match:
         sig type primary_color = RGB.primary_color = Red | Green | Blue end
       is not included in
         RGB_sig
       The value `inc' is required but not provided
```

Use case of signatures: compilation checks (3)

```
module type RGB_sig =
  sig
    type primary_color = Red | Green | Blue
    val inc : primary_color -> primary_color
  end
module RGB : RGB_sig = struct
  type primary_color = Red | Green | Blue
end;;
```

```
Error: Signature mismatch:
       Modules do not match:
         sig type primary_color = RGB.primary_color = Red | Green | Blue end
       is not included in
         RGB_sig
       The value `inc' is required but not provided
```

Compilation check 2:

- This is okay:

```
module type RGB_sig =
  sig
    type primary_color = Red | Green | Blue
  end
module RGB = struct
  type primary_color = Red | Green | Blue
  let inc x = match x with
    | Red -> Green
    | Green -> Blue
    | Blue -> Red
end
module RGB_checked : RGB_sig = RGB;;
```

Q: Why is one thing okay, the other not?

- module A : B, for module A and signature B, is okay if B has fewer definitions than A, but not vice versa.
- What reason might the OCaml designers have to allow more in code, but not more in signatures?

Q: Why is one thing okay, the other not?

- module A : B, for module A and signature B, is okay if B has fewer definitions than A, but not vice versa.

- A signature is seen an agreement between programmers (possibly between you and yourself):
  - The implementer agrees to implement *at least* the functions in the signature
  - The code-user agrees to use *at most* the functions in the signature
- Let's apply this to some other conditions

Is this allowed?

```
module type sig1 = sig
  val inc : int -> int
end
module M1 : sig1 = struct
  let inc (x : 'a) : 'a = x
end
```

Is this allowed?

```
module type sig1 = sig
  val inc : int -> int
end
module M1 : sig1 = struct
  let inc (x : 'a) : 'a = x
end
```

Yes, the implementation is more general!

Is this allowed?

```
module type sig2 = sig
  val inc : 'a -> 'a
end
module M2 : sig2 = struct
  let inc (x : int) : int = x
end
```

Is this allowed?

```
module type sig2 = sig
  val inc : 'a -> 'a
end
module M2 : sig2 = struct
  let inc (x : int) : int = x
end
```

No, the implementation is less general!

Is this allowed?

```
module type sig1 = sig
  type foo = Bar | Baz
end
module M : sig1 = struct
  type foo = Bar
end
```

Is this allowed?

```
module type sig1 = sig
  type foo = Bar | Baz
end
module M : sig1 = struct
  type foo = Bar
end
```

No, the constructor M.Baz is not implemented

Is this allowed?

```
module type sig1 = sig
  type foo = Bar
end
module M : sig1 = struct
  type foo = Bar | Baz
end
```

Is this allowed?

```
module type sig1 = sig
  type foo = Bar
end
module M : sig1 = struct
  type foo = Bar | Baz
end
```

No, a complete pattern match for ‘foo’
would become incomplete when using M

.. but this is allowed

```
module type sig1 = sig
  type foo
end
module M : sig1 = struct
  type foo = Bar
end
```

- Reason: the line ‘type foo’ should be read as:
  - the type foo exists
  - we’re not saying what constructors foo has

- We’ll see why this is useful when we do encapsulation

A particular use-case

```
utop # #require "core.top";;
No such package: core.top
```

- (install core from command line: ‘opam install core’. This take a while)

```
let ( let* ) = Core.Option.(>>=) in
let* x1 = double 3 in
let* x2 = plus x1 3 in
double x2
;;
- : int option = Some 18
```

(under suitable definitions of double and plus)

Outlook

- Wednesday: Encapsulation, a reason to hide types
- Next week: Proofs about code (finally!)
- Week after: Combining proofs and modules
- Nov 6th: second midterm