# Modules and encapsulation

UNIVERSITY OF MINNESOTA
Driven to Discover®

# Outlook

- Files as Modules
- Signatures for opacity
- Abstract types
  - Motivation: balanced trees

# Files as modules: ocamlc

- Files are a way of structuring code (in every programming language)
- Suppose we have file1.ml with these definitions:
  - let foo x = x + 1
- Now in file2.ml we can write:
  - #load “file1.ml” — proper way to load a file
  - let _ = print_int (File1.foo 3) — using foo from file1.ml

# Files as modules: utop

- In utop you may have been used to
- #use “file1.ml”

- That’s fine for utop, but #load is preferred when using multiple files

# Aside: dune

- dune is a tool that:
  - calls ocamlc for you
  - allows you to specify: which files are for testing, which are a code library for reuse, and which file makes up the executable (plus some other things)
  - allows you to better organise your code in directories
  - can call utop for you while you're developing code
  - can call a documentation generation tool for you

# Files as modules: homeworks

- You’ve been creating modules all along.
- When you submit a file, here’s how we can test it:
  - #load “hw3.ml”
  - module type hw3 = sig (.. some signature ..) end
  - module _ : hw3 = Hw3
  - let () = assert (Hw3.some_function some_args = …)

# Back to signatures…

- Recall that this was okay:

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

# Back to signatures…

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

- When we write the above, there is no RGB.inc
- Recall that ‘RGB_sig’ also stands for the promise not to use anything outside that signature

# Opacity

- Hiding everything outside the signature is referred to as opacity
- We might want to use this to:
  - Hide helper functions that nobody needs
  - Hide functions that are likely to change in the future
  - Hide functions that break things when used incorrectly
- Note that hiding is the *default*. (Why?)

# Hiding types

- We saw this on Monday:.

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

# Why hide types?

- Many data-structures satisfy invariants:
  - Priority-sorted queue is sorted
  - Balanced (/red-black/avl) trees are balanced(-ish)
  - Let’s look at an example for rationals

# Creating a library for rationals

- Rational numbers are (typically) represented by two integers, called a numerator and a denominator
- Let’s write Ratio (n, d) to stand for n/d
- We’d like equality to work as expected: 4/8 = 1/2
  - 4 / 8 and -1 / -2 should both be expressed as 1 / 2.
  - Normalization: denominator should be positive, divide out common factors
- We’ll again pretend that int does not have overflows

# Rational

```
type rational = Ratio of (int*int)
let zero = Ratio (0,1)
exception Division_by_zero
let rec gcd a b =
  if b = 0 then a else gcd b (a mod b)
let normalize n d = let g = gcd n d in
  Ratio (n / g, d / g)
let make n d =
  if d = 0 then raise Division_by_zero else
  if d > 0 then normalize n d else
    normalize (-n) (-d)
let greater (Ratio (n1,d1)) (Ratio (n2,d2)) =
  n1*d2 > n2*d1
```

# Which functions should be visible?

```
type rational = Ratio of (int*int)
let zero = Ratio (0,1)
exception Division_by_zero
let rec gcd a b =
  if b = 0 then a else gcd b (a mod b)
let normalize n d = let g = gcd n d in
  Ratio (n / g, d / g)
let make n d =
  if d = 0 then raise Division_by_zero else
  if d > 0 then normalize n d else
    normalize (-n) (-d)
let greater (Ratio (n1,d1)) (Ratio (n2,d2)) =
  n1*d2 > n2*d1
```

# What does this return?

```
let greater (Ratio (n1,d1)) (Ratio (n2,d2)) =
  n1*d2 > n2*d1
```

- greater (Ratio (1,2)) (Ratio (3,-2));;

# What does this return?

```
let greater (Ratio (n1,d1)) (Ratio (n2,d2)) =
  n1*d2 > n2*d1
```

- greater (Ratio (1,2)) (Ratio (3,-2));;
- val _ : bool = false
- Is this a bug in our code?

# What does this return?

```
let greater (Ratio (n1,d1)) (Ratio (n2,d2)) =
  n1*d2 > n2*d1
```

- greater (Ratio (1,2)) (Ratio (3,-2));;
- val _ : bool = false
- Our code was using that the denominator is always positive…
- The input (Ratio (3,-2)) is wrong here!
- ocaml has a way to prevent wrong inputs…

# The right signature

```
module type rational =
  sig
    type t
    val zero : t
    exception Division_by_zero
    val make : int -> int -> t
    val greater : t -> t -> bool
  end
```

# Using the signature…

```
type rational = Ratio of (int*int)
module RationalInternal = struct
  type t = rational
  let zero = Ratio (0,1)
  exception Division_by_zero
  let rec gcd a b = if b = 0 then a else gcd b (a mod b)
  let normalize n d = let g = gcd n d in Ratio (n/g, d/g)
  let make n d =
    if d = 0 then raise Division_by_zero else
    if d > 0 then normalize n d else
      normalize (-n) (-d)
  let greater (Ratio (n1,d1)) (Ratio (n2,d2)) =
    n1*d2 > n2*d1
end
module Rational : rational = RationalInternal
```

# Some tips to improve testability

```
utop # Rational.zero;;
- : Rational.t = <abstr>
```

- ocaml takes the opacity thing very seriously: utop isn't able to print the internal structure of Rational.t
- As a consequence, we made some decisions in the previous slide:
  - Define types outside the module
  - Define an alias for the type inside a module that is called '..Internal'
  - Define the opaque module on a separate line

# Compare:

```
utop # RationalInternal.zero;;
- : rational = Ratio (0, 1)
```

- If we had put the type within the module:

```
utop # RationalInternal.zero;;
- : RationalInternal.t =
RationalInternal.Ratio (0, 1)
```

- (not bad, but slightly verbose)

# Extending a module / signature

- Suppose we wanted to add ‘plus’
  - Easiest (and recommended) way:
    add ‘plus’ to the signature, add ‘plus’ to the module
  - Suppose we need to do it in a separate file:
    - create a new signature that is the old one,
      but with ‘plus’ added
    - create a new implementation that is the old one,
      but with ‘plus’ added
    - How do we add without repeating ourselves?

# Extending a module / signature

```
module type rationalPlus = sig
  include rational
  val plus : t -> t -> t
end
module RationalInternalPlus = struct
  include RationalInternal
  let plus (Ratio (n1,d1)) (Ratio (n2,d2)) =
    normalize (n1*d2 + n2*d1) (d1*d2)
end
module RationalPlus : rationalPlus = RationalInternalPlus
```

- If we did this in a separate file, we could even choose to use (at the risk of confusing ourselves and others):

```
module Rational : rationalPlus = RationalInternalPlus
```

# Why doesn't this work?

```
module RationalPlus = struct
  include Rational
  let plus (Ratio (n1,d1)) (Ratio (n2,d2)) =
    normalize (n1*d2 + n2*d1) (d1*d2)
end
```

# Why doesn’t this work?

```
module RationalPlus = struct
  include Rational
  let plus (Ratio (n1,d1)) (Ratio (n2,d2)) =
    normalize (n1*d2 + n2*d1) (d1*d2)
end
```

- This uses ‘Ratio’ where something of type ‘t’ is required.
- We know that they are the same, but ‘Rational’ hides this.
- This uses ‘normalize’, but it’s not a (visible) part of ‘Rational’
- ocaml takes opacity very seriously!
- (this is another reason to separate the ..Internal module)

# Here is the risk!

```
module RationalPlus = struct
  include Rational
  let plus (Ratio (n1,d1)) (Ratio (n2,d2)) =
    normalize (- (n1*d2 + n2*d1)) (- (d1*d2))
end
```

- This breaks the invariant that denominators cannot be negative
- Ocaml protects against it!

# Here is the risk!

```
module RationalPlus = struct
  include RationalInternal
  let plus (Ratio (n1,d1)) (Ratio (n2,d2)) =
    normalize (- (n1*d2 + n2*d1)) (- (d1*d2))
end
```

- This breaks the invariant that denominators cannot be negative
- Ocaml doesn’t protect against it…
- … because we used RationalInternal to circumvent the protection!

# Staying safe…

- Here's a safe minimal rational number signature:

```
module type rational =
  sig
    type t
    exception Division_by_zero
    val make : int -> int -> t
    val unmake : t -> (int * int)
  end
```

- This signature suffices to build all further functions!
- Reason: 'make' is the safe Ratio constructor, and 'unmake' is the counterpart to a pattern match

# Staying safe…

- Here's an example on how we can expand it:

```
module type rational =
  sig
    type t
    exception Division_by_zero
    val make : int -> int -> t
    val unmake : t -> (int * int)
  end

(insert definition of module Rational)

module RationalPlus = struct
  include Rational
  let plus a b = match (unmake a, unmake b) with
    | ((n1,d1), (n2,d2))
         -> make (n1*d2 + n2*d1) (d1*d2)
end
```

# In case you're interested… (overly generalized example)

- More generally (for any canonically representable type), we can replace 'unmake' by a 'fold_right':

```
module type rational =
  sig
    type t
    exception Division_by_zero
    val make : int -> int -> t
    val fold : (int -> int -> 'a ) -> t -> 'a
  end

(* How to get 'unmake' back: *)
unmake = Rational.fold (fun a b -> (a,b))
```

# Key take-aways

- Use signatures to hide implementation details that are:
  - unnecessary helper functions
  - unsafe helper functions
  - types that should not be used directly
- Use a separate module whose name ends with Internal for easy testing. Don't apply a signature to this module!
- The …Internal approach also allows others to easily extend your module in the future, which usually is desirable.

# Outlook

- Note that we could write integer exponentiation on rationals (r^n, n is an integer and r is a rational)
- The procedure would use multiplication and inverses
- This procedure would work for other data-types as well (like matrix exponentiation: M^n), provided they have multiplication and inverses
- We can (and will) write code that is polymorphic-ish:
  exp : ‘a_ish -> int -> ‘a_ish,
  provided that ‘a_ish is something for which multiplication and inverse is defined.