Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 6 - Sequence 1: Information hiding

Publishing the interface of a module

- A module usually provides a **well-delimited set of features**.
- These features should come with some kind a **user manual** to indicate to clients:
  1. functions preconditions that **must be verified**;
  2. data invariants that **must be preserved** ;
  3. definitions on which the user **must not rely**
     (because they may change in the future).
- As a **contract that must be respected in exchange for the module features**.

A **module signature** represents that contract.
The type checker will enforce points 2 and 3.

Signature manipulation

- To constrain a module with a specific signature:
  `module M : sig ... end = struct ... end`

- A signature can be named:
  `module type S = sig ... end`

- These names can also be used in module definitions:
  `module M : S = struct ... end`

Writing down a (bad) signature for stacks I

```
module Stack : sig
  type 'a t = 'a list
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> ('a * 'a t) option
end = struct
  type 'a t = 'a list
  let empty = []
  let push x s = x :: s
  let pop = function
    | [] -> None
    | x :: xs -> Some (x, xs)
end;;
```

Writing down a (bad) signature for stacks II

```
# module Stack :
  sig
    type 'a t = 'a list
    val empty : 'a t
    val push : 'a -> 'a t -> 'a t
    val pop : 'a t -> ('a * 'a t) option
  end
```

A badly designed signature I

```
module Naturals : sig
  (* Invariant: A value of type t is a positive integer. *)
  type t = int
  val zero : t
  val succ : t -> t
  val pred : t -> t
end = struct
  type t = int
  let zero = 0
  (* The functions maintain the invariant. *)
  let succ n = if n = max_int then 0 else n + 1
  let pred = function 0 -> 0 | n -> n - 1
end;;
```

A badly designed signature II

```
# module Naturals :
  sig
    type t = int
    val zero : t
    val succ : t -> t
    val pred : t -> t
  end
```

A badly designed signature III

```
open Naturals
let rec add : t -> t -> t = fun x y ->
  if x = zero then y else succ (add (pred x) y);;
# val add : Naturals.t -> Naturals.t -> Naturals.t = <fun>
let i_break_the_abstraction = pred (-1);;
# val i_break_the_abstraction : Naturals.t = -2
```

Signature manipulation

- The invariant is broken because the programmer should not use the fact that "`type t = int`".
- A comment could be written to ask him not to use that fact.
- But, this would require some discipline and it may be hard to resist to the temptation.
- **Abstract types** will give no choice to the client but to respect this rule.

A well designed signature I

```
module Naturals : sig
  (* Invariant: A value of type t is a positive integer. *)
  type t
  val zero : t
  val succ : t -> t
  val pred : t -> t
end = struct
  type t = int
  let zero = 0
  (* The functions maintain the invariant. *)
  let succ n = if n = max_int then 0 else n + 1
  let pred = function 0 -> 0 | n -> n - 1
end;;
```

A well designed signature II

```
# module Naturals :
  sig
    type t
    val zero : t
    val succ : t -> t
    val pred : t -> t
  end
open Naturals
let rec add : t -> t -> t = fun x y ->
  if x = zero then y else succ (add (pred x) y);;
# val add : Naturals.t -> Naturals.t -> Naturals.t = <fun>
```

Abstract types

- We have **hidden** the definition of the type `t`.
- The signature does not publish this fact anymore.
- The typechecker ensures that the clients cannot use that fact.
- The type `t` is called an **abstract type**.

Private definitions

- The programmer can choose **not to export** some definitions.
- This is convenient to hide private internal functions.

Hiding internal functions I

```ocaml
module Naturals : sig
  type t
  val zero : t
  (* val return_natural : int -> t *)
  val succ : t -> t
  val pred : t -> t
end = struct
  type t = int
  let zero = 0
  (* The following function is for internal purpose only. *)
  let return_natural n = assert (n >= 0 && n <= max_int); n
  let succ n = if n = max_int then 0 else return_natural (n + 1)
  let pred = function 0 -> 0 | n -> n - 1
end;;
```

Hiding internal functions II

```
# module Naturals :
  sig
    type t
    val zero : t
    val succ : t -> t
    val pred : t -> t
  end
```

```
Naturals.return_naturals 0;;
```

```
# Characters 1-25:
  Naturals.return_naturals 0;;
  ^^^^^^^^^^^^^^^^^^^^^^^^
Error: Unbound value Naturals.return_naturals
```