Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 6 - Sequence 3: Functors

The-reference VS A-reference

- In our case study, the client has a reference to **the** module `Dict`.
- A more configurable client, hence a more reusable one, would let use choose the dictionary implementation externally.
- Using a **module functor**, the client will have a reference to **a** module.

A functorized client I

```
module type DictSig = sig
  type ('key, 'value) t
  val empty : ('key, 'value) t
  val add : ('key, 'value) t -> 'key -> 'value -> ('key, 'value) t
  exception NotFound
  val lookup : ('key, 'value) t -> 'key -> 'value
end;;
# module type DictSig =
  sig
    type ('key, 'value) t
    val empty : ('key, 'value) t
    val add :
      ('key, 'value) t ->
      'key -> 'value -> ('key, 'value) t
    exception NotFound
```

A functorized client II

```
  val lookup : ('key, 'value) t -> 'key -> 'value
end
```

A functorized client III

```
(* The client *)
module ForceArchive (Dict : DictSig) = struct
  let force = Dict.empty
  let force = Dict.add force "luke" 10
  let force = Dict.add force "yoda" 100
  let force = Dict.add force "darth" 1000
  let force_of_luke = Dict.lookup force "luke"
  let force_of_r2d2 = Dict.lookup force "r2d2"
end;;
# module ForceArchive :
  functor (Dict : DictSig) ->
    sig
      val force : (string, int) Dict.t
      val force_of_luke : int
      val force_of_r2d2 : int
    end
```

A functorized client IV

```
module Dict1 : DictSig = struct
  type ('key, 'value) t = ('key * 'value) list
  let empty = []
  let add d k v = (k, v) :: d
  exception NotFound
  let rec lookup d k =
    match d with
      | (k', v) :: d' when k = k' -> v
      | _ :: d -> lookup d k
      | [] -> raise NotFound
end;;
# module Dict1 : DictSig
```

A functorized client V

```
module Dict2 : DictSig = struct
  type ('key, 'value) t =
    | Empty
    | Node of ('key, 'value) t * 'key * 'value * ('key, 'value) t

  let empty = Empty

  let rec add d k v =
    match d with
    | Empty -> Node (Empty, k, v, Empty)
    | Node (l, k', v', r) ->
      if k = k' then Node (l, k, v, r)
      else if k < k' then Node (add l k v, k', v', r)
      else Node (l, k', v', add r k v)

  exception NotFound
```

A functorized client VI

```
  let rec lookup d k =
    match d with
      | Empty ->
        raise NotFound
      | Node (l, k', v', r) ->
        if k = k' then v'
        else if k < k' then lookup l k
        else lookup r k

end;;
# module Dict2 : DictSig
```

A functor is a function from modules to modules

- A functor is a module waiting for another module.
- The syntax of functors extends module definition with module arguments:

```
module SomeModuleIdentifier (SomeModuleIdentifier : SomeSignature) =
struct
  ...
end
```

- To apply a functor to a module:

```
SomeModuleIdentifier (SomeModule)
```

... where `SomeModule` is a module expression
(an identifier, an implementation or a functor application).

- The signature of a functor is written:

```
functor (ModuleIdentifier : SomeSignature) ->
  sig ... end
```

Another form of type parameterization

- The type parameter `'key` should be shared between the type of dictionaries and the declaration of the exception `NotFound`.
- That kind of type parameterization can be done with a functor.

Type parameterization by functorization I

```
module type DictSig = sig
  type key
  type 'value t
  val empty : 'value t
  val add : 'value t -> key -> 'value -> 'value t
  exception NotFound of key
  val lookup : 'value t -> key -> 'value
end;;
# module type DictSig =
  sig
    type key
    type 'value t
    val empty : 'value t
    val add : 'value t -> key -> 'value -> 'value t
    exception NotFound of key
```

Type parameterization by functorization II

```
  val lookup : 'value t -> key -> 'value
end
```

Type parameterization by functorization III

```
module Dict (Key : sig
  type t
  val compare : t -> t -> int
end) : DictSig = struct
  type key = Key.t
  type 'value t = (key * 'value) list
  let empty = []
  let add d k v = (k, v) :: d
  exception NotFound of key
  let rec lookup d k =
    match d with
      | (k', v) :: d' when Key.compare k k' = 0 -> v
      | _ :: d -> lookup d k
      | [] -> raise (NotFound k)
end;;
```

Type parameterization by functorization IV

```
# module Dict :
  functor
    (Key : sig type t val compare : t -> t -> int end) ->
    DictSig
```

Type parameterization by functorization V

```
module Dict1 = Dict (struct
  type t = string
  let compare k1 k2 = Pervasives.compare (String.lowercase k1)
    (String.lowercase k2)
end)
module ForceArchive (Dict : DictSig) = struct
  let force = Dict.empty
  let force = Dict.add force "luke" 10
  let force = Dict.add force "yoda" 100
  let force = Dict.add force "darth" 1000
  let force_of_luke = Dict.lookup force "luke"
  let force_of_r2d2 = Dict.lookup force "r2d2"
end;;
```

Type parameterization by functorization VI

```
# Characters 100-116:
    let compare k1 k2 = Pervasives.compare (String.lowercase k1)
    (String.lowercase k2)
                                             ^^^^^^^^^^^^^^^^
Warning 3: deprecated: String.lowercase
Use String.lowercase_ascii instead.
Characters 122-138:
    let compare k1 k2 = Pervasives.compare (String.lowercase k1)
    (String.lowercase k2)

    ^^^^^^^^^^^^^^^^
Warning 3: deprecated: String.lowercase
Use String.lowercase_ascii instead.
Characters 260-266:
    let force = Dict.add force "luke" 10
                                  ^^^^^^
```

Type parameterization by functorization VII

```
Error: This expression has type string
       but an expression was expected of type Dict.key
```

Type parameterization by functorization VIII

```
module Dict (Key : sig
  type t
  val compare : t -> t -> int
end) : DictSig with type key = Key.t = struct
  type key = Key.t
  type 'value t = (key * 'value) list
  let empty = []
  let add d k v = (k, v) :: d
  exception NotFound of key
  let rec lookup d k =
    match d with
      | (k', v) :: d' when Key.compare k k' = 0 -> v
      | _ :: d -> lookup d k
      | [] -> raise (NotFound k)
end;;
```

Type parameterization by functorization IX

```
# module Dict :
  functor
    (Key : sig type t val compare : t -> t -> int end) ->
    sig
      type key = Key.t
      type 'value t
      val empty : 'value t
      val add : 'value t -> key -> 'value -> 'value t
      exception NotFound of key
      val lookup : 'value t -> key -> 'value
    end
```

Type parameterization by functorization X

```
module ForceArchive (Dict : DictSig with type key = string) = struct
  let force = Dict.empty
  let force = Dict.add force "luke" 10
  let force = Dict.add force "yoda" 100
  let force = Dict.add force "darth" 1000
  let force_of_luke = Dict.lookup force "luke"
  let force_of_r2d2 = Dict.lookup force "r2d2"
end;;
# module ForceArchive :
  functor
    (Dict : sig
              type key = string
              type 'value t
              val empty : 'value t
              val add :
                'value t -> key -> 'value -> 'value t
```

Type parameterization by functorization XI

```
            exception NotFound of key
            val lookup : 'value t -> key -> 'value
          end) ->
  sig
    val force : int Dict.t
    val force_of_luke : int
    val force_of_r2d2 : int
  end
```

Type constraints on signatures

- Sometimes, the abstraction must be relaxed a little bit *a posteriori*.
- A **type constraint** expresses a fact about a type in a signature
- A type constraint restricts the use and definition of functors.
- In exchange, the functor client gets more guarantees about abstract types.
- Type constraints follow module signature:

```
some_signature
with type some_type_identifier = some_type
and type some_type_identifier = some_type
and ...
```