The OCaml Type System

Fabrice LE FESSANT
INRIA – OCamlPro

November 6, 2013

Overview

Overview - The OCaml Programming Language

Overview - OCaml History

OCaml History: 1970-1990

The Foundations

- 1973: ML for LCF, by Robin Milner
- 1980: team Formel at INRIA works on le_ML (Gérard Huet, Guy Cousineau, Larry Paulson)
- 1984: CAM = Categorical Abstract Machine
- 1984: Standard ML definition
- 1985: work on the implementation of CAML
  - target the Coq proof assistant
  - does not want to be constrained by a standard
- 1987: first release of CAML (Guy Cousineau, Michel Mauny, Ascander Suarez, Pierre Weis)

OCaml History: 1990-2001

From Caml to Objective-Caml

- 1991: Caml light (Xavier Leroy, Damien Doligez), light efficient bytecode C interpreter
- 1995: Caml special light (native code compiler + powerful module system)
- 1996: Objective Caml (objects and classes, by Jérôme Vouillon and Didier Remy)
- 1999: version 2.0, new more powerful class system
- 2000: version 3.0, merge with OLabl (labels, polymorphic variants) by Jacques Garrigue
- 2001: F#, an OCaml dialect by Microsoft

OCaml History: 2007-2013

From Objective-Caml to OCaml

- 2007: new camlp4 preprocessor, ocamlbuild, OCamlJava (OCaml-to-JVM compiler)
- 2009 : CompCert, certified C compiler in Coq
- 2010: first-class modules, explicit polymorphism and polymorphic recursion
- 2010: js_of_ocaml: OCaml-to-Javascript compiler
- 2011: renaming from Objective-Caml to OCaml
- 2012: GADT
- 2014 $\rightarrow$ namespaces ? type-classes ? annotations ?

Overview - OCaml Principles

OCaml Philosophy

OCaml Design Choices

- A language to reason about programs
  Functional, strong type-checking, strict

OCaml Philosophy

OCaml Design Choices

- A language to reason about programs
  Functional, strong type-checking, strict
- A language to program with
  Type inference, pattern-matching, mutations

OCaml Philosophy

OCaml Design Choices

- A language to reason about programs
  Functional, strong type-checking, strict
- A language to program with
  Type inference, pattern-matching, mutations
- A language for intensive symbolic computations
  Fast garbage collector, native-code optimizing compiler

OCaml Philosophy

OCaml Design Choices

- A language to reason about programs
  Functional, strong type-checking, strict
- A language to program with
  Type inference, pattern-matching, mutations
- A language for intensive symbolic computations
  Fast garbage collector, native-code optimizing compiler
- A language for general use
  Bytecode, native-code, JVM, Javascript

Type-Checking

Type-Checking

Dynamic type-checking

Validity of operations is checked at runtime.
Examples: Python, Ruby, Lua, bash

Type-Checking

Dynamic type-checking

Validity of operations is checked at runtime.
Examples: Python, Ruby, Lua, bash

Weak type-checking

The compiler verifies the validity of some operations, but allows either explicit or implicit coercions to be tested at runtime.
Examples: C, C++, Java, C#, Scala, F#

Type-Checking

Dynamic type-checking

Validity of operations is checked at runtime.
Examples: Python, Ruby, Lua, bash

Weak type-checking

The compiler verifies the validity of some operations, but allows either explicit or implicit coercions to be tested at runtime.
Examples: C, C++, Java, C#, Scala, F#

Strong type-checking

The compiler verifies that all operations performed on a value are allowed by the type of that value.
Examples: OCaml, SML, Haskell

Strong Type-Checking

OCaml uses strong type-checking

The compiler verifies that all operations performed on a value are allowed by the type of that value.

Strong Type-Checking

OCaml uses strong type-checking

The compiler verifies that all operations performed on a value are allowed by the type of that value.

- All types must be known at compile-time
- Everything must be typed $\rightarrow$ need for a rich set of types to express everything a developer wants

Strong Type-Checking

OCaml uses strong type-checking

The compiler verifies that all operations performed on a value are allowed by the type of that value.

- All types must be known at compile-time
- Everything must be typed $\rightarrow$ need for a rich set of types to express everything a developer wants
- Full type-inference: the compiler *guesses* the types, the developer does not need to annotate variables with types

Strong Type-Checking

OCaml uses strong type-checking

The compiler verifies that all operations performed on a value are allowed by the type of that value.

- All types must be known at compile-time
- Everything must be typed $\rightarrow$ need for a rich set of types to express everything a developer wants
- Full type-inference: the compiler *guesses* the types, the developer does not need to annotate variables with types
- All tests are done at compile-time, no tests done at runtime $\rightarrow$ faster code, still safe

OCaml Type Inference

OCaml Type Inference

Full type-inference on the core language

- Based on syntax for basic types
- Propagation by unification

OCaml Type Inference

Full type-inference on the core language

- Based on syntax for basic types
- Propagation by unification

Advanced types can require type annotations

- Subtyping with objects and polymorphic variants
- Polymorphic methods and polymorphic recursion
- Generalized Algebraic Data Types (GADT)

Overview - OCaml Core Types

Overview - Basic Types

Basic Types

Ints, Floats and Strings

```
        OCaml version 4.01.0
# 11 + 2;;
- : int = 13
```

Basic Types

Ints, Floats and Strings

```
        OCaml version 4.01.0
# 11 + 2;;
- : int = 13
# let pi = 3.14;;
val pi : float = 3.14
```

Basic Types

Ints, Floats and Strings

```
        OCaml version 4.01.0
# 11 + 2;;
- : int = 13
# let pi = 3.14;;
val pi : float = 3.14
# let f = pi *. 2. +. 1.;;
val f : float = 7.28
```

Basic Types

Ints, Floats and Strings

```
            OCaml version 4.01.0
# 11 + 2;;
- : int = 13
# let pi = 3.14;;
val pi : float = 3.14
# let f = pi *. 2. +. 1.;;
val f : float = 7.28
# let euclidian_div = 5 / 2;;
val euclidian_div : int = 2
```

Basic Types

Ints, Floats and Strings

```
        OCaml version 4.01.0
# 11 + 2;;
- : int = 13
# let pi = 3.14;;
val pi : float = 3.14
# let f = pi *. 2. +. 1.;;
val f : float = 7.28
# let euclidian_div = 5 / 2;;
val euclidian_div : int = 2
# let str = "Hello" ^ " " ^ "world";;
val str : string = "Hello world"
```

Basic Types

Booleans and Tuples

```
# let truth = true || false && true;;
val truth : bool = true
```

Basic Types

Booleans and Tuples

```
# let truth = true || false && true;;
val truth : bool = true
# let pair = (int_of_string("1"), "one");;
val pair : int * string = (1, "one")
```

Basic Types

Booleans and Tuples

```
# let truth = true || false && true;;
val truth : bool = true
# let pair = (int_of_string("1"), "one");;
val pair : int * string = (1, "one")
# let tuple = (1, 1.0, "one", 't');;
val tuple : int * float * string * char
```

Non-ambiguous syntax for basic constants and operators enables inference of basic types.
→ No operator overloading in OCaml

Lists

'a list

```
# let empty_list = [];;
val empty_list : 'a list = []
```

Lists

'a list

```
# let empty_list = [];;
val empty_list : 'a list = []
# let list123 = 1 :: 2 :: 3 :: []
val list123 : int list = [1; 2; 3]
```

Lists

'a list

```
# let empty_list = [];;
val empty_list : 'a list = []
# let list123 = 1 :: 2 :: 3 :: []
val list123 : int list = [1; 2; 3]
# let list123 = [1;2;3];;
val list123 : int list = [1; 2; 3]
```

Lists

'a list

```
# let empty_list = [];;
val empty_list : 'a list = []
# let list123 = 1 :: 2 :: 3 :: []
val list123 : int list = [1; 2; 3]
# let list123 = [1;2;3];;
val list123 : int list = [1; 2; 3]
# let hetero_list = [1;2; 3.0 ];;
                          ^^^
Error: This expression has type float but
  an expression was expected of type int
```

Lists must be filled with the same type.

Overview - Functions and Polymorphism

Currification

One-argument Function

A function type is noted `arg -> res`.

```
# let incr = function x -> x + 1;;
val incr : int -> int = <fun>
```

Currification

One-argument Function

A function type is noted `arg -> res`.

```
# let incr = function x -> x + 1;;
val incr : int -> int = <fun>
# let add2 = function (x,y) -> x + y;;
val add2 : int * int -> int = <fun>
```

Currification

One-argument Function

A function type is noted `arg -> res`.

```
# let incr = function x -> x + 1;;
val incr : int -> int = <fun>
# let add2 = function (x,y) -> x + y;;
val add2 : int * int -> int = <fun>
# let add2 (x,y) = x + y;;
val add2 : int * int -> int = <fun>
```

Currification

One-argument Function

A function type is noted `arg -> res`.

```
# let incr = function x -> x + 1;;
val incr : int -> int = <fun>
# let add2 = function (x,y) -> x + y;;
val add2 : int * int -> int = <fun>
# let add2 (x,y) = x + y;;
val add2 : int * int -> int = <fun>
# add2 (1,3);;
- : int = 4
```

Currification

One-argument Function

A function type is noted `arg -> res`.

```
# let incr = function x -> x + 1;;
val incr : int -> int = <fun>
# let add2 = function (x,y) -> x + y;;
val add2 : int * int -> int = <fun>
# let add2 (x,y) = x + y;;
val add2 : int * int -> int = <fun>
# add2 (1,3);;
- : int = 4
# let z = (2,3);;
  add2 z;;
- : int = 5
```

Currification

Multi-argument Function

```
# let add = fun x y -> x + y;;
val add : int -> int -> int = <fun>
```

Currification

Multi-argument Function

```
# let add = fun x y -> x + y;;
val add : int -> int -> int = <fun>
# let add x y = x + y;;
val add : int -> int -> int = <fun>
```

Currification

Multi-argument Function

```
# let add = fun x y -> x + y;;
val add : int -> int -> int = <fun>
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 1 3;;
- : int = 4
```

Currification

Multi-argument Function

```
# let add = fun x y -> x + y;;
val add : int -> int -> int = <fun>
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 1 3;;
- : int = 4
# add (1,3);;
Error: This expression has type 'a * 'b
 but an expression was expected of type int
```

Closures

Lexical Scope

```
# let f x = x + 10
  let add_f x = add x (f x);;
  add_f 3;;
- : int = 16
```

Closures

Lexical Scope

```
# let f x = x + 10
  let add_f x = add x (f x);;
  add_f 3;;
- : int = 16
# let f x = 0;;
```

Closures

Lexical Scope

```
# let f x = x + 10
  let add_f x = add x (f x);;
  add_f 3;;
- : int = 16
# let f x = 0;;
  add_f 3;;
- : int = 16
```

Bindings cannot be modified in OCaml:
Variables and functions can only be redefined, without impacting previously defined functions.

Closures

Partial Application

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
```

Closures

Partial Application

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# let incr = add 1;;
val incr : int -> int = <fun>
```

Closures

Partial Application

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# let incr = add 1;;
val incr : int -> int = <fun>
# incr 3;;
- : int = 4
```

A function with type `arg1 -> arg2 -> res`
is equivalent to `arg1 -> (arg2 -> res)`.

Closures

Partial Application

```
# let mulf = fun x -> (fun y -> x *. y);;
val mulf : float -> float -> float = <fun>
```

Closures

Partial Application

```
# let mulf = fun x -> (fun y -> x *. y);;
val mulf : float -> float -> float = <fun>
# let mulf = fun x -> fun y -> x *. y;;
val mulf : float -> float -> float = <fun>
```

Closures

Partial Application

```
# let mulf = fun x -> (fun y -> x *. y);;
val mulf : float -> float -> float = <fun>
# let mulf = fun x -> fun y -> x *. y;;
val mulf : float -> float -> float = <fun>
# let mulf x y = x *. y;;
val mulf : float -> float -> float = <fun>
```

Closures

Partial Application

```
# let mulf = fun x -> (fun y -> x *. y);;
val mulf : float -> float -> float = <fun>
# let mulf = fun x -> fun y -> x *. y;;
val mulf : float -> float -> float = <fun>
# let mulf x y = x *. y;;
val mulf : float -> float -> float = <fun>
# let times2 = mulf 2.;;
# times2 10.;;
- : float = 20.
```

Closures

Functions as Values

```
# let f_x_x f x = f x x;;
val f_x_x : ('a -> 'a -> 'b) -> 'a -> 'b
```

Closures

Functions as Values

```
# let f_x_x f x = f x x;;
val f_x_x : ('a -> 'a -> 'b) -> 'a -> 'b
# let square_f = f_x_x mulf;;
val square_f : float -> float = <fun>
```

Closures

Functions as Values

```
# let f_x_x f x = f x x;;
val f_x_x : ('a -> 'a -> 'b) -> 'a -> 'b
# let square_f = f_x_x mulf;;
val square_f : float -> float = <fun>
# square_f 3.0;;
- : float = 9.
```

Closures

Functions as Values

```
# let f_x_x f x = f x x;;
val f_x_x : ('a -> 'a -> 'b) -> 'a -> 'b
# let square_f = f_x_x mulf;;
val square_f : float -> float = <fun>
# square_f 3.0;;
- : float = 9.
# let square_i = f_x_x ( * );;
val square_i : int -> int = <fun>
```

Closures

Functions as Values

```
# let f_x_x f x = f x x;;
val f_x_x : ('a -> 'a -> 'b) -> 'a -> 'b
# let square_f = f_x_x mulf;;
val square_f : float -> float = <fun>
# square_f 3.0;;
- : float = 9.
# let square_i = f_x_x ( * );;
val square_i : int -> int = <fun>
# square_i 5;;
- : int = 25
```

Functions

Recursive Functions

```
fold_left f x [a;b;c] = f (f (f x a) b) c /*
```

Functions

Recursive Functions

`fold_left f x [a;b;c] = f (f (f x a) b) c` /* */

```
# let rec fold_left f acc = function
      [] -> acc
    | head :: tail ->
              fold_left f (f acc head) tail;;
```

Functions

Recursive Functions

```
fold_left f x [a;b;c] = f (f (f x a) b) c /* */
```

```
# let rec fold_left f acc = function
      [] -> acc
    | head :: tail ->
              fold_left f (f acc head) tail;;
val fold_left : ('a -> 'b -> 'a) ->
                'a -> 'b list -> 'a = <fun>
```

Functions

Recursive Functions

```
fold_left f x [a;b;c] = f (f (f x a) b) c /* */
```

```
# let rec fold_left f acc = function
      [] -> acc
    | head :: tail ->
            fold_left f (f acc head) tail;;
val fold_left : ('a -> 'b -> 'a) ->
            'a -> 'b list -> 'a = <fun>
# let sum_list = fold_left add 0;;
val sum_list : int list -> int = <fun>
```

Functions

Recursive Functions

```
fold_left f x [a;b;c] = f (f (f x a) b) c /* */
```

```
# let rec fold_left f acc = function
      [] -> acc
    | head :: tail ->
                fold_left f (f acc head) tail;;
val fold_left : ('a -> 'b -> 'a) ->
                 'a -> 'b list -> 'a = <fun>
# let sum_list = fold_left add 0;;
val sum_list : int list -> int = <fun>
# sum_list [1;2;3;4];;
- : int = 10
```

Polymorphism

Polymorphic Functions

`List.fold_left` can be applied on any type of list:

```
val fold_left : ('a -> 'b -> 'a) ->
                 'a -> 'b list -> 'a = <fun>
```

Polymorphism

Polymorphic Functions

`List.fold_left` can be applied on any type of list:

```
val fold_left : ('a -> 'b -> 'a) ->
                 'a -> 'b list -> 'a = <fun>
# let mulf_list = List.fold_left ( *. ) 1.;;
val mulf_list : float list -> float = <fun>
# mulf_list [1.0 ;2.0 ;3.0 ;4.0];;
- : float = 24.
```

Such polymorphic functions are very useful to increase the sharing of *generic code* in an application, to avoid the maintenance of several copies of the code.

Polymorphism

Polymorphism is very common in libraries

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.iter;;
- : ('a -> unit) -> 'a list -> unit = <fun>
# List.sort;;
- : ('a -> 'a -> int) -> 'a list -> 'a list
# List.rev;;
- : 'a list -> 'a list = <fun>
```

Polymorphism

OCaml has some false polymorphic functions

A function is *truely polymorphic* if it does not access the content of the value on which type it is polymorphic.

```
# compare;;
- : 'a -> 'a -> int = <fun>
# List.sort compare [ 8; 40; 5 ];;
- : int list = [5; 8; 40]
# let list = List.sort compare ["8";"40";"5"];;
val list : string list = ["40"; "5"; "8"]
```

Polymorphism

OCaml has some false polymorphic functions

A function is *truely polymorphic* if it does not access the content of the value on which type it is polymorphic.

```
# compare;;
- : 'a -> 'a -> int = <fun>
# List.sort compare [ 8; 40; 5 ];;
- : int list = [5; 8; 40]
# let list = List.sort compare ["8";"40";"5"];;
val list : string list = ["40"; "5"; "8"]
# list > [ "20" ] && list < [ "8"; "1" ];;
- : bool = true
# list = List.rev ["8"; "5"; "40"];;
- : bool = true
```

Polymorphism

Combining recursivity and polymorphism need annotations

```
# let rec iter f list =
    match list with
      [] -> ()
    | head :: tail ->
       f head;
       iter debug f tail;;
  val iter : ('a -> unit) ->
                           'a list -> unit
```

Polymorphism

Combining recursivity and polymorphism need annotations

```
# let rec iter debug f list =
    if debug then begin
      iter false print_int [ 1; 2; 3 ];
    end;
    match list with
      [] -> ()
    | head :: tail ->
       f head;
       iter debug f tail;;
val iter : bool -> (int -> unit) ->
                       int list -> unit
```

Polymorphism

Combining recursivity and polymorphism need annotations

```
# let rec iter debug f list =
    if debug then begin
      iter false print_int [ 1; 2; 3 ];
      iter false print_string [ "x"; "y" ];
    end;
    match list with
      [] -> ()
    | head :: tail ->
       f head;
       iter debug f tail;;
Error: expr "print_string" has type string -> uni
 but expr was expected of type int -> unit
```

Polymorphism

Combining recursivity and polymorphism need annotations

```
let rec iter :
  'a. bool -> ('a -> unit) -> 'a list -> unit
  = fun debug f list ->
  if debug then begin
      iter false print_string [ "x"; "y" ];
      iter false print_int [ 1; 2; 3 ];
  end;
  match list with
    [] -> ()
  | head :: tail ->
     f head;
     iter debug f tail;;
```

Overview - Records and mutable values

Records

Immutable records

```
# type t = { x : int; name : string; }
```

Records

Immutable records

```
# type t = { x : int; name : string; }
# let to_string z =
    Printf.sprintf "{ x = %d; name = %S }"
        z.x z.name
```

Records

Immutable records

```
# type t = { x : int; name : string; }
# let to_string z =
    Printf.sprintf "{ x = %d; name = %S }"
       z.x z.name
# let change_name z new_name =
    { z with name = new_name }
val change_name : t -> string -> t = <fun>
```

Immutable records are like tuples, but with field names.
Record types are inferred from field names.

Mutable Records

Records with Mutable Fields

```
# type t = { x : int;a
    mutable name : string; }
```

Mutable Records

Records with Mutable Fields

```
# type t = { x : int;a
     mutable name : string; }
# let to_string z =
   Printf.sprintf "{ x = %d; name = %S }"
       z.x z.name
```

Mutable Records

Records with Mutable Fields

```
# type t = { x : int;a
     mutable name : string; }
# let to_string z =
    Printf.sprintf "{ x = %d; name = %S }"
        z.x z.name
# let change_name z new_name =
    z.name <- new_name
val change_name : t -> string -> unit = <fun>
```

In OCaml, copying a record might be faster than mutating it.

References

A reference is a simple polymorphic record

```
# type 'a ref = { mutable content : 'a }
  let ( := ) r x = r.content <- x
  let ( ! ) r = r.content
```

References

A reference is a simple polymorphic record

```
# type 'a ref = { mutable content : 'a }
  let ( := ) r x = r.content <- x
  let ( ! ) r = r.content
  let fact n =
    let res = ref 1 in
    for i = 2 to n do res := i * !res; done;
    !res
val fact : int -> int = <fun>
```

References

A reference is a simple polymorphic record

```
# type 'a ref = { mutable content : 'a }
  let ( := ) r x = r.content <- x
  let ( ! ) r = r.content
  let fact n =
    let res = ref 1 in
    for i = 2 to n do res := i * !res; done;
    !res
val fact : int -> int = <fun>
# let rec fact n =
    if n = 1 then 1 else n * fact (n-1)
```

Type-inference with mutable values

Unsafe behavior

```
# let list = ref [];;
val list : 'a list ref
# list := [1 ; 2 ; ];;
(* !list contains an int list *)
# List.iter print_string !list;;
Segmentation Fault
```

OCaml must be careful with mutable values.

Type-inference with mutable values

Safe behavior

```
# let list = ref [];;
val list : '_a list ref
# list := [1 ; 2 ; ];;
(* !list contains an int list *)
# List.iter print_string !list;;
                         ^^^^^
Error: This expression has type int list
  but an expression was expected of type
  string list
```

Type-inference with mutable values

Value Restriction

Only function let-definitions can be generalized:

```
# let map_length list =
    List.map List.length list;;
val map_length : 'a list list -> int list
```

Type-inference with mutable values

Value Restriction

Only function let-definitions can be generalized:

```
# let map_length list =
    List.map List.length list;;
val map_length : 'a list list -> int list
# let map_length = List.map List.length;;
val map_length : '_a list list -> int list
```

Type-inference with mutable values

Value Restriction

Only function let-definitions can be generalized:

```
# let map_length list =
    List.map List.length list;;
val map_length : 'a list list -> int list
# let map_length = List.map List.length;;
val map_length : '_a list list -> int list
# map_length [ [1;2]; []; [ 1; 0; 5 ]];;
- : int list = [2; 0; 3]
```

Type-inference with mutable values

Value Restriction

Only function let-definitions can be generalized:

```
# let map_length list =
    List.map List.length list;;
val map_length : 'a list list -> int list
# let map_length = List.map List.length;;
val map_length : '_a list list -> int list
# map_length [ [1;2]; []; [ 1; 0; 5 ]];;
- : int list = [2; 0; 3]
# map_length [ [ 'a']; []; [ 'b'; 'v' ]];;
Error: This expression has type char but an
  expression was expected of type int
```

Sometimes, it is necessary to eta-expanse functions.

Overview - Variants and Pattern-Matching

Variants

Definition and Use

```
type xml =
    Tag of string * (string *string) list * xml
  | PCData of string
  | List of xml list
```

Variants

Definition and Use

```
type xml =
    Tag of string * (string *string) list * xml
  | PCData of string
  | List of xml list

let link url name =
    Tag("a", [ "href", url ], PCData name)
```

Variants

Pattern-Matching

```
let rec remove_ahref = function
  | Tag("a", attrs, body) when
     List.mem_assoc "href" attrs ->
                              remove_ahref body
  | Tag(tag, attrs, body) ->
          Tag(tag, attrs, remove_ahref body)
  | (PCData _ ) as doc -> doc
  | List xmls ->
           List (List.map remove_ahref xmls)
```

Variants

Predefined Variants

```
type 'a list =
    []
  | :: of 'a * 'a list
type 'a option =
    None
  | Some of 'a
```

Overview - Exceptions

Exceptions

Exceptions are open variants

```
exception X of int
exception Y of string
let f (a,b,c) =
  try
    if a then raise (X b);
    raise (X c);
    failwith "Should not happen !"
  with X n -> string_of_int n
     | Y s -> s
```

Overview - OCaml Module System

Overview - Signatures and Structures

Structures

Group of definitions

```
module Complex = struct
  type t = {
     mutable re : float;
     mutable im : float }
  let create re im = { re; im }
  let add x y = { re = x.re +. y.re;
                  im = x.im +. y.im }
  let set_re x re = x.re <- re
  let set_im x im = x.im <- im
end
let two = Complex.add (Complex.create 0. 0.)
                      (Complex.create 1. 1.)
```

Signatures

Module Types

Module types specify what is exported by a module.

```
module Complex : sig
  type t =  {
     mutable re : float;
     mutable im : float }
  val create : float -> float -> t
  val add : t -> t -> t
  val set_re : t -> float -> unit
  val set_im : t -> float -> unit
end =
  struct ... end
```

Signatures

Abstract Types

Access to types internal can be limited by signatures:

```
module Complex : sig
  type t


  val create : float -> float -> t
  val add : t -> t -> t
  val set_re : t -> float -> unit
  val set_im : t -> float -> unit
end =
  struct ... end
```

Signatures

Abstract Types

Access to values and functions can be removed.

```
module Complex : sig
  type t


  val create : float -> float -> t
  val add : t -> t -> t



end =
  struct ... end
```

Files and Modules

A Source File is a Structure: complex.ml

```
type t = {
     mutable re : float;
     mutable im : float }
let create re im = { re; im }
let add x y = { re = x.re +. y.re;
                im = x.im +. y.im }
let set_re x re = x.re <- re
let set_im x im = x.im <- im
```

Files and Modules

An Interface File is a Signature: complex.mli

```
type t
val create : float -> float -> t
val add : t -> t -> t
```

Interfaces can be defined at a project first steps, so that teams can work either at implementing the module, or at using the module.

Overview - Functors

Functors

Functions on Modules

How can we parameterize a data structure on a function ?

```
# type key = { id : string;
               mutable atime : float; }
# let table = Hashtbl.create 13
# Hashtbl.add table
      { id = "x"; atime = 0.0 } "Hello";;
val table : (key, string) Hashtbl.t
```

Only the key should be hashed...

Functors

Functions on Modules

The `Hashtbl` module defines a functor to create new types of hash tables.

```
module Make(H : sig
       type t
       val hash : t -> int
       val equal : t -> t -> bool
     end) ->
    sig
      let create ...
      ...
    end
```

Functors

Functions on Modules

```
# module H = Hashtbl.Make(struct
    type t = key
    let hash key = Hashtbl.hash key.id
    let equal k1 k2 = k1.id = k2.id
  end)
# let table = H.create 13
# H.add table
      { id = "x"; atime = 0.0 } "Hello";;
val table : string H.t
```

Modules in OCaml usually replace most needs for objects.

Overview - OCaml Advanced Types

Overview - Polymorphic Variants

Variants

Advantages

- Variants can express different states of data in a short and clean way
- Pattern-matching on variants is:
  - efficient (compiled in an optimal number of tests)
  - safe (warnings are displayed on non-exhaustive pattern-matchings)

Variants

Limitations

```
module A = struct
  type colors =
   | Red
   | Green
   | Blue
  let to_string = function
   | Red -> "red"
   | Green -> "green"
   | Blue -> "blue"
end
```

Variants

Limitations

```
# module B = struct
    type more_colors =
     | Red | Green | Blue (* same as A *)
     | White | Black
    let to_string = function
     | White -> "white"
     | Black -> "black"
     | colors  -> A.to_string colors
  end                               ^^^^^^
Error: This expression has type more_colors
but an expression was expected of type A.colors
```

Variants

Polymorphic variants

```
# module A = struct
  type colors = [ `Red | `Green | `Blue ]
  let to_string = function
   | `Red -> "red"
   | `Green -> "green"
   | `Blue -> "blue"
  end;;
module A : sig
    type colors = [ `Blue | `Green | `Red ]
    val to_string :
     [< `Blue | `Green | `Red ] -> string
  end
```

Variants

Polymorphic variants

```
# module B = struct
    type more_colors = [
     | A.colors
     | `White | `Black ]
    let to_string = function
     | `White -> "white"
     | `Black -> "black"
     |  c -> A.to_string c
  end;;                   ^^^
Error: This expr has type [> `Black | `White ]
 but an expr was expected of type [< A.colors ]
```

Variants

Polymorphic variants

```
# module B = struct
    type more_colors = [
     | A.colors
     | `White | `Black ]
    let to_string = function
     | `White -> "white"
     | `Black -> "black"
     | #A.colors as c -> A.to_string c
  end;;
```

Works, but a type annotation is needed.

Overview - Labeled and Optional Arguments

Labeled Arguments

Functions can have labeled arguments

```
# let rec concat ~sep ~list =
    match list with
      [] -> ""
    | head :: tail ->
     head ^ sep ^ concat ~sep ~list:tail;;
val concat : sep:string -> list:string list
    -> string = <fun>
```

Labeled Arguments

Functions can have labeled arguments

```
# let rec concat ~sep ~list =
    match list with
      [] -> ""
    | head :: tail ->
     head ^ sep ^ concat ~sep ~list:tail;;
val concat : sep:string -> list:string list
    -> string = <fun>
# concat ~sep:"/" ~list:[ "a"; "b"; "c" ]
- : string = "a/b/c"
# concat ~sep:"/" ~list:[ "a"; "b"; "c" ]
- : string = "a/b/c"
```

Labeled Arguments

Labeled arguments can be reordered

```
# let rec concat ~sep ~list =
    match list with
      [] -> ""
     | head :: tail ->
      head ^ sep ^ concat ~sep ~list:tail;;
val concat : sep:string -> list:string list
    -> string = <fun>
# concat ~sep:"/" ~list:[ "a"; "b"; "c" ]
- : string = "a/b/c"
# concat ~list:[ "a"; "b"; "c" ] ~sep:"+"
- : string = "a+b+c"
```

Optional Arguments

Labeled arguments can be made optional

```
# let rec concat ?sep ~list =
    let separator = match sep with
      None -> "/" | Some sep_o -> sep_o in
    match list with [] -> ""
    | head :: tail ->
     head ^ separator ^ concat ?sep ~list:tail;;
val concat : ?sep:string -> list:string list
    -> string = <fun>
```

Optional Arguments

Labeled arguments can be made optional

```
# let rec concat ?sep ~list =
    let separator = match sep with
      None -> "/" | Some sep_o -> sep_o in
    match list with [] -> ""
    | head :: tail ->
     head ^ separator ^ concat ?sep ~list:tail;;
val concat : ?sep:string -> list:string list
    -> string = <fun>
# concat ~list:[ "a"; "b"; "c"];;
- : ?sep:string -> string = <fun>
```

Optional Arguments

Optional arguments can only exist with non-labeled ones

```
# let rec concat ?sep list =
    let separator = match sep with
      None -> "/" | Some sep_o -> sep_o in
    match list with [] -> ""
    | head :: tail ->
     head ^ separator ^ concat ?sep tail;;
val concat : ?sep:string -> string list
    -> string = <fun>
```

Optional Arguments

Optional arguments can only exist with non-labeled ones

```
# let rec concat ?sep list =
    let separator = match sep with
      None -> "/" | Some sep_o -> sep_o in
    match list with [] -> ""
    | head :: tail ->
     head ^ separator ^ concat ?sep tail;;
val concat : ?sep:string -> string list
    -> string = <fun>
# concat [ "a"; "b"; "c"];;
- : string = "a/b/c/"
```

Optional Arguments

Optional arguments can only exist with non-labeled ones

```
# let rec concat ?sep list =
    let separator = match sep with
      None -> "/" | Some sep_o -> sep_o in
    match list with [] -> ""
    | head :: tail ->
     head ^ separator ^ concat ?sep tail;;
val concat : ?sep:string -> string list
    -> string = <fun>
# concat [ "a"; "b"; "c"];;
- : string = "a/b/c/"
# concat [ "a"; "b"; "c" ]  ~sep:"+";;
- : string = "a+b+c"
```

Optional Arguments

Default values can be specified for optional arguments

```
# let rec concat ?(sep="/") list =
    match list with [] -> ""
     | head :: tail ->
      head ^ sep ^ concat ~sep tail;;
val concat : ?sep:string -> string list
    -> string = <fun>
# concat [ "a"; "b"; "c" ]  ~sep:"+";;
- : string = "a+b+c"
# concat [ "a"; "b"; "c"];;
- : string = "a/b/c/"
```

Overview - First Class Modules

First Class Modules

Modules as Values

It can be useful to be able to manipulate dynamically modules. For that, OCaml has introduced first class modules, i.e. modules that can be used as values.

First Class Modules

Signature

```
module type Filename = sig
  type t
  val of_string : string -> t
  val dirname : t -> t
  val concat : t -> string -> t
  val to_string : t -> string
end
```

First Class Modules

Structures

```
module WinFilename : Filename = struct
  type t = { partition : string option;
             absolute : bool;
             files : string list; }
  let to_string t = ...
  let dirname t = ...
  ...
end
module UnixFilename : Filename = struct
  ...
end
```

First Class Modules

Modules as Values

```
# module File =
    (val
        match Sys.os_type with
        | "win32" ->
          (module WinFilename: Filename)
        | _       ->
          (module UnixFilename: Filename)
    );;
module File : Filename
```

`File` is dynamically associated to the specific implementation of filenames for the current operating-system.

Overview - Generalized Algebraic Data Types

Generalized Algebraic Data Types (GADT)

Yet another kind of variants !

Sometimes, a function might return different types of values, depending on its arguments.
This is not possible with variants or polymorphic variants.
It is possible in OCaml with GADTs.

GADT definition

AST for a simple evaluator

```
type _ term =
  | Int : int                         -> int term
  | Bool : bool                       -> bool term
  | Add : int term * int term         -> int term
  | And : bool term * bool term       -> bool term
  | If : bool term * 'a term * 'a term
                                      -> 'a term
  | Pair : 'a term * 'b term
                                -> ('a * 'b) term
  | Fst : ('a * 'b) term              -> 'a term
  | Snd : ('a * 'b) term              -> 'b term
```

GADT definition

Evaluation Function

```
let rec eval : type a . a term -> a =
  function
  | Int n -> n
  | Bool b -> b
  | Add (a, b) -> eval a + eval b
  | And (a, b) -> eval a && eval b
  | If (t, c, a) ->
      if eval t then eval c else eval a
  | Pair (a, b) -> eval a, eval b
  | Fst p -> fst (eval p)
  | Snd p -> snd (eval p)
```

GADT definition

Evaluation

```
# let t = Snd (Pair (Bool false,
                     Pair (Int 1, Int 2)));;
val t : (int * int) term = <gadt>
# eval t;;
- : int * int = (1, 2)
# let t2 = If(Bool true, Fst t, Int 3);;
val t2 : int term = <gadt>
# eval t2;;
- : int = 1
```

Overview - conclusion

Conclusion

OCaml Principles

- Full type-inference on the core language
- One of the richest type-systems
- Still improving: for next versions
  - Namespaces
  - Runtime Types
  - Type-classes