The OCaml Type System

Fabrice LE FESSANT
INRIA – OCamlPro

November 6, 2013
Overview

1. The OCaml Programming Language
2. OCaml Core Types
3. OCaml Module System
4. OCaml Advanced Types
5. conclusion
Overview - The OCaml Programming Language

1. The OCaml Programming Language
2. OCaml Core Types
3. OCaml Module System
4. OCaml Advanced Types
5. Conclusion
Overview - OCaml History

1. The OCaml Programming Language
   - OCaml History
   - OCaml Principles
## 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**

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1991</td>
<td>Caml light (Xavier Leroy, Damien Doligez), light efficient bytecode C interpreter</td>
</tr>
<tr>
<td>1995</td>
<td>Caml special light (native code compiler + powerful module system)</td>
</tr>
<tr>
<td>1996</td>
<td>Objective Caml (objects and classes, by Jérôme Vouillon and Didier Remy)</td>
</tr>
<tr>
<td>1999</td>
<td>version 2.0, new more powerful class system</td>
</tr>
<tr>
<td>2000</td>
<td>version 3.0, merge with OLabel (labels, polymorphic variants) by Jacques Garrigue</td>
</tr>
<tr>
<td>2001</td>
<td>F#, an OCaml dialect by Microsoft</td>
</tr>
</tbody>
</table>
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 → namespaces ? type-classes ? annotations ?
Overview - OCaml Principles

1. The OCaml Programming Language
   - OCaml History
   - 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
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
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
OCaml uses strong type-checking

The compiler verifies that all operations performed on a value are allowed by the type of that value.
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 → need for a rich set of types to express everything a developer wants
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 → 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
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 → 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 → faster code, still safe
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)
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

1. The OCaml Programming Language
2. OCaml Core Types
3. OCaml Module System
4. OCaml Advanced Types
5. conclusion
2 OCaml Core Types

- Basic Types
- Functions and Polymorphism
- Records and mutable values
- Variants and Pattern-Matching
- Exceptions
Basic Types

Ints, Floats and Strings

OCaml version 4.01.0

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

# 11 + 2;;
- : int = 13

# let pi = 3.14;;
val pi : float = 3.14

val euclidian_div : int = 2

# let str = "Hello" ^ " " ^ "world";;
val str : string = "Hello world"
OCaml version 4.01.0

```ocaml
# 11 + 2;;
- : int = 13

# let pi = 3.14;;
val pi : float = 3.14

# let f = pi *. 2. +. 1.;;
val f : float = 7.28
```

```ocaml
# let euclidian_div = 5 / 2;;
val euclidian_div : int = 2

# let str = "Hello" ^ " " ^ "world";;
val str : string = "Hello world"
```
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
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

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

Basic Types

Booleans and Tuples

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

# let pair = (int_of_string("1"), "one");;
val pair : int * string = (1, "one")
```

Non-ambiguous syntax for basic constants and operators enables inference of basic types.

→ No operator overloading in OCaml
Basic Types

Booleans and Tuples

```ocaml
# 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

```ocaml
# let empty_list = [];;
val empty_list : 'a list = []
# 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.
```
The OCaml Type System

OCaml Core Types

Basic Types

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.
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. 
2 OCaml Core Types

- Basic Types
- Functions and Polymorphism
- Records and mutable values
- Variants and Pattern-Matching
- Exceptions
A function type is noted \( \text{arg} \rightarrow \text{res} \).

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

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

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

```ocaml
# add2 (1,3);;
- : int = 4
```

```ocaml
# let z = (2,3);;
add2 z;;
- : int = 5
```
One-argument Function

A function type is noted \( \text{arg} \rightarrow \text{res} \).

```ocaml
# 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 z = (2,3);;
add2 z;;
- : int = 5
```
One-argument Function

A function type is noted \( \text{arg} \rightarrow \text{res} \).

```ocaml
# 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>
```
A function type is noted \texttt{arg -> res.}

\begin{verbatim}
# 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
\end{verbatim}
Currification

One-argument Function

A function type is noted $\text{arg} \rightarrow \text{res}$.  

```ocaml
# 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

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

```ocaml
val add : int -> int -> int = <fun>
```

```ocaml
- : int = 4
```

```ocaml
Error: This expression has type 'a * 'b
but an expression was expected of type int
```
## Currification

### Multi-argument Function

```ocaml
# 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

```ocaml
# 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
```
Multi-argument Function

```ocaml
# 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

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

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

Lexical Scope

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

# let f x = 0;;
```

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

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

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

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

### Partial Application

```ocaml
# 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

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

```ocaml
# 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

Partial Application

```ocaml
# 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>
```

```ocaml
# let times2 = mulf 2.;
# times2 10.;
- : float = 20.
```
Closures

Partial Application

```ocaml
# 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

```ocaml
# 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.
```
The OCaml Type System

---

OCaml Core Types

---

Functions and Polymorphism

---

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
Closures

Functions as Values

```ocaml
# 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 as Values

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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

\[
\text{fold\_left } f \ x \ [a;b;c] = f \ (f \ (f \ x \ a) \ b) \ c
\]

```ocaml
# 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>
```

```ocaml
# let sum_list = fold_left add 0;;
val sum_list : int list -> int = <fun>
# sum_list [1;2;3;4];;
- : int = 10
```
Functions and Polymorphism

# Functions

## Recursive Functions

```ocaml
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>
```

```ocaml
# sum_list [1;2;3;4];;
- : int = 10
```
Functions

Recursive Functions

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

```ocaml
# 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
```
Polymorphic Functions

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

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

```ocaml
# 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.
Polymorphic Functions

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

```ocaml
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

```ocaml
# 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>
```
OCaml has some false polymorphic functions

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

```ocaml
# 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"]
```
OCaml has some false polymorphic functions

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

```ocaml
# 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

```ocaml
# let rec iter f list =
  match list with
  | [] -> ()
  | head :: tail ->
    f head;
    iter debug f tail;;
val iter : ('a -> unit) ->
  'a list -> unit
```
Combining recursivity and polymorphism need annotations

```ocaml
# 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
```
Combining recursivity and polymorphism need annotations

```ocaml
# 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 -> unit
but expr was expected of type int -> unit
```
Combining recursivity and polymorphism need annotations

```ocaml
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

2 OCaml Core Types

- Basic Types
- Functions and Polymorphism
- Records and mutable values
- Variants and Pattern-Matching
- Exceptions
Records

Immutable records

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

Immutable records are like tuples, but with field names.

Record types are inferred from field names.
Immutables records

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

```ocaml
# 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

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

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

Records and mutable values

**Mutable Records**

**Records with Mutable Fields**

```ocaml
# type t = { x : int; 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.
Mutable Records

Records with Mutable Fields

```ocaml
# 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.
A reference is a simple polymorphic record

```
# type 'a ref = { mutable content : 'a }
let ( := ) r x = r.content <- x
let ( ! ) r = r.content
```
A reference is a simple polymorphic record

```ocaml
# 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>
```
A reference is a simple polymorphic record

```ocaml
# 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

```ocaml
# 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.
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
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
```
Value Restriction

Only function let-definitions can be generalized:

```ocaml
# 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
```

Sometimes, it is necessary to eta-expanse functions.
Type-inference with mutable values

Value Restriction

Only function let-definitions can be generalized:

```ocaml
# 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:

```ocaml
# 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

2 OCaml Core Types
- Basic Types
- Functions and Polymorphism
- Records and mutable values
- Variants and Pattern-Matching
- Exceptions
Variants

Definition and Use

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

Definition and Use

```ocaml
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)
```
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

```ocaml
type 'a list =
    | []
    | :: of 'a * 'a list

type 'a option =
    None
    | Some of 'a
```
Overview - Exceptions

2 OCaml Core Types
- Basic Types
- Functions and Polymorphism
- Records and mutable values
- Variants and Pattern-Matching
- Exceptions
Exceptions

Exceptions are open variants

```ocaml
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

1. The OCaml Programming Language
2. OCaml Core Types
3. OCaml Module System
4. OCaml Advanced Types
5. conclusion
Overview - Signatures and Structures

3 OCaml Module System
  □ Signatures and Structures
  □ Functors
Structures

Group of definitions

```ocaml
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.

```ocaml
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
```
Access to types internal can be limited by signatures:

```ocaml
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
```
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
A Source File is a Structure: complex.ml

```ocaml
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

```ocaml
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

3 OCaml Module System
- Signatures and Structures
- Functors
How can we parameterize a data structure on a function?

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

```ocaml
code
module Make (H : sig
  type t
  val hash : t -> int
  val equal : t -> t -> bool
end) ->

sig
  let create ...
  ...
end
```
Modules in OCaml usually replace most needs for objects.
Overview - OCaml Advanced Types

1. The OCaml Programming Language
2. OCaml Core Types
3. OCaml Module System
4. OCaml Advanced Types
5. conclusion
Overview - Polymorphic Variants

4 OCaml Advanced Types
- Polymorphic Variants
- Labeled and Optional Arguments
- First Class Modules
- Generalized Algebraic Data Types
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

```ocaml
module A = struct

type colors =
| Red
| Green
| Blue

let to_string = function
| Red -> "red"
| Green -> "green"
| Blue -> "blue"

end
```
# 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
Polymorphic variants

```ocaml
# 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

```ocaml
# 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 ]
```
Polymorphic variants

```ocaml
# 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.
4 OCaml Advanced Types
- Polymorphic Variants
- Labeled and Optional Arguments
- First Class Modules
- Generalized Algebraic Data Types
Labeled Arguments

Functions can have labeled arguments

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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>
```
Labeled arguments can be made optional

```ocaml
# 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

```ocaml
# 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 can only exist with non-labeled ones

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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

4 OCaml Advanced Types
- Polymorphic Variants
- Labeled and Optional Arguments
- First Class Modules
- Generalized Algebraic Data Types
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.
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

```ocaml
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
```
# 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

4 OCaml Advanced Types
- Polymorphic Variants
- Labeled and Optional Arguments
- First Class Modules
- Generalized Algebraic Data Types
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

```ocaml
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

```ocaml
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)
```
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

1. The OCaml Programming Language
2. OCaml Core Types
3. OCaml Module System
4. OCaml Advanced Types
5. 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