OCaml

In the previously lectures:

- basic syntax: fun. application, let-s, arith. ops.
- data types: strings, chars, variants, records, arrays
- code organization: modules, module types
- evaluation order: applicative, strict by default

ocamldebug demo

```
let _ = print_endline "Hi!"

let add x y = x + y

let main () =
  let z = 5 + 5 in
  print_int (add z 7);
  print_endline "text"

let _ = main ()
```

```
>ocamlc -g demo.ml
(ocd) break @ Demo 6
(ocd) run
(ocd) ... {help, list, step, back, bt, up, down, print, goto}
(ocd) quit
```

Labeled function arguments

```
open StdLabels
String.sub : string -> pos:int -> len:int -> string
```

Labeled function arguments

```
open StdLabels
String.sub : string -> pos:int -> len:int -> string
```

So you can write

Example

```
String.sub "Hello" ~pos:0 ~len:4
```

or even

```
String.sub "Hello" ~len:4 ~pos:0
```

Labeled function arguments (cont.)

```
let myconcat ~front:x ~back:y = x^y
let _ =
  myconcat ~back:"World!" ~front:"Hello "
```

Labeled function arguments (cont.)

```
let myconcat ~front:x ~back:y = x^y
let _ =
  myconcat ~back:"World!" ~front:"Hello "
```

or even

```
let myconcat ~front ~back = front^back
let _ =
  myconcat ~back:"World!" ~front:"Hello "
```

Optional function arguments

```
let rec concatStrings ?sep:(s=", ") = function
  | []    -> ""
  | [x]   -> x
  | x::xs -> x ^ s ^ concatStrings ~sep:s xs

let _ =
  assert(concatStrings           ["AA";"B"] =  "AA, BB");
  assert(concatStrings ~sep:"; " ["AA";"B"] =  "AA; BB")
```

Use case: Maps

```
module type OrderedType =
sig
  type t
  val compare: t -> t -> int
end

module type S =
sig
  type key
  type 'a t
  val empty: 'a t
  val add  : key -> 'a -> 'a t -> 'a t
  ...
end

module Make:
  functor (Ord : OrderedType) ->
    S with type key = Ord.t
```

```
module IntPairs =
struct
 type t = int * int
 let compare (x0,y0) (x1,y1) =
   match Pervasives.compare x0 x1 with
         0 -> Pervasives.compare y0 y1
       | c -> c
end

module PairsMap = Map.Make(IntPairs)
open PairsMap

let m = empty |> add (0,1) "hello"
              |> add (1,0) "world"
```

Use case: Maps

- similar to template code in C++
- defines immutable data structures
- implemented with balanced binary trees
- fixed type for the key; polymorphic over the value

Use case: Hash tables

```
module type HashedType =
sig
  type t
  val equal : t -> t -> bool
  val hash : t -> int
end

module type S =
sig
  type key
  type 'a t
  val create : int -> 'a t
  val add : 'a t -> key -> 'a -> unit
  ...
end
module Make:
  functor (H : HashedType) ->
    S with type key = H.t
```

```
module IntHash =
struct
 type t = int
 let equal (x:int) y = x=y
 let hash (i:int)    = i
end

module IntH = Hashtbl.Make(IntHash)

let h = IntH.create 17

let _ =
  IntH.add h 12 "hello"
```

Use case: Hash maps

- defines a mutable data structure
- arg. to create is the initial size of the table
- implemented using an array of lists
- fixed type for the key; polymorphic over the value

Use case: Hashcons

```
type 'a hobj = private {
    obj   : 'a;
    tag   : int;
    hcode : int;
}
module type Table =
sig
  type key
  type t
  val create : int -> t
  val clear : t -> unit
  val hashcons : t -> key -> key hobj
  ...
end

module MakeTable:
  functor (HT : HashedType) ->
    Table  with type key = HT.t
```

```
module PMTabel = MakeTable (PairsMap)

let tbl = PMTabel.create 10

let f () : PairsMap.t = ...

let f' () : PairsMap.t hobj =
  Table.hashcons tbl (f ())
```

Use case: Hashcons

- a mutable data structure
- used for sharing values that are structurally equal
- originates from Lisp
- implemented using hash-maps of weak references

Use case: Printf

Example

```
open Printf

let _ =
  printf "Hello World!";
  printf "Error %d: %s\n" 5 "file not found"
```

Use case: Printf

Example

```
open Printf

let _ =
  printf "Hello World!";
  printf "Error %d: %s\n" 5 "file not found"
```

- Works as expected, but how?

Type problem

String constants in OCaml have a special sub-type

Example

```
"Hello World!"    : ('a, 'b, 'a) format
"Error %d: %s\n" : (int -> string -> 'a, 'b, 'a) format
```

Type problem

String constants in OCaml have a special sub-type

Example

```
"Hello World!"    : ('a, 'b, 'a) format
"Error %d: %s\n" : (int -> string -> 'a, 'b, 'a) format
```

that are used by the printer functions

```
val printf : ('a, out_channel,   unit) format -> 'a
val sprintf: ('a,         unit, string) format -> 'a
val fprintf: out_channel -> ('a,out_channel,unit) format -> 'a
val bprintf: Buffer.t    -> ('a,   Buffer.t,unit) format -> 'a
```

Type problem

String constants in OCaml have a special sub-type

Example

```
"Hello World!"    : ('a, 'b, 'a) format
"Error %d: %s\n" : (int -> string -> 'a, 'b, 'a) format
```

that are used by the printer functions

```
val printf : ('a, out_channel,   unit) format -> 'a
val sprintf: ('a,         unit, string) format -> 'a
val fprintf: out_channel -> ('a,out_channel,unit) format -> 'a
val bprintf: Buffer.t    -> ('a,   Buffer.t,unit) format -> 'a
```

Example

```
printf  "Error %d: %s\n" : int -> string -> unit
sprintf "Error %d: %s\n" : int -> string -> string
```

The %a format

Used for printing user-defined types:

Example

```
type mytype = ...
let myval : mytype = ...
let myformat : out_channel -> mytype -> unit = ...

let _ =
  printf "%d%a\n" 10 myformat myval
```

The %a format

Used for printing user-defined types:

Example

```
type mytype = ...
let myval : mytype = ...
let myformat : out_channel -> mytype -> unit = ...

let _ =
  printf "%d%a\n" 10 myformat myval
```

Example

```
"%a"    : (('b -> 'c -> 'a) -> 'c -> 'a, 'b, 'a) format
"%t"    : (('b -> 'a) -> 'a, 'b, 'a) format

printf "%d%a": int -> (out_channel -> 'a -> unit) -> 'a -> unit
```

How do you write a function:

```
val printf : ('a, out_channel,   unit) format -> 'a
```

How do you write a function:

```
val printf : ('a, out_channel,   unit) format -> 'a
```

Remember that the first arg. of printf is a string.

- Step through each character, and
- cast the result into the right type.

Example (pseudocode)

```
val magic : 'a -> 'b (* cast function of OCaml *)

let rec printf = function
  | ...
  | '%'::'d'::xs -> magic (fun (i:int) -> ...)
  | ...
```

Versions of 'format'

- format

```
type ('a, 'b, 'c) format = ('a, 'b, 'c, 'c) format4
```

- format4

```
type ('a, 'b, 'c, 'd) format4 =
       ('a, 'b, 'c, 'c, 'c, 'd) format6
```

- format6

```
type ('a, 'b, 'c, 'd, 'e, 'f) format6 =
  | Format of ('a, 'b, 'c, 'd, 'e, 'f) fmt * string
```

- fmt ...

The new definition ;-)

```
type ('a, 'b, 'c, 'd, 'e, 'f) fmt =
| Char : ('a0, 'b0, 'c0, 'd0, 'e0, 'f0) fmt -> (char -> 'a0, 'b0, '
| Caml_char : ('a1, 'b1, 'c1, 'd1, 'e1, 'f1) fmt -> (char -> 'a1, '
| String : ('x, string -> 'a2) padding
* ('a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmt -> ('x, 'b2, 'c2, 'd2, 'e2, 'f
| Caml_string : ('x0, string -> 'a3) padding
* ('a3, 'b3, 'c3, 'd3, 'e3, 'f3) fmt -> ('x0, 'b3, 'c3, 'd3, 'e3, '
| Int : int_conv
* ('x1, 'y) padding
* ('y, int -> 'a4) precision
* ('a4, 'b4, 'c4, 'd4, 'e4, 'f4) fmt -> ('x1, 'b4, 'c4, 'd4, 'e4, '
| Int32 : int_conv
* ('x2, 'y0) padding
* ('y0, int32 -> 'a5) precision
* ('a5, 'b5, 'c5, 'd5, 'e5, 'f5) fmt -> ('x2, 'b5, 'c5, 'd5, 'e5, '
| Nativeint : int_conv
* ('x3, 'y1) padding
* ('y1, nativeint -> 'a6) precision
* ('a6, 'b6, 'c6, 'd6, 'e6, 'f6) fmt -> ('x3, 'b6, 'c6, 'd6, 'e6, '
| Int64 : int_conv
* ('x4, 'y2) padding
* ('y2, int64 -> 'a7) precision
* ('a7, 'b7, 'c7, 'd7, 'e7, 'f7) fmt -> ('x4, 'b7, 'c7, 'd7, 'e7, '
| Float : float_conv
```

The new definition ;-)

```
type ('a, 'b, 'c, 'd, 'e, 'f) fmt =
| Char : ('a0, 'b0, 'c0, 'd0, 'e0, 'f0) fmt -> (char -> 'a0, 'b0, 'c0, 'd0, 'e0, 'f0) fmt
| Caml_char : ('a1, 'b1, 'c1, 'd1, 'e1, 'f1) fmt -> (char -> 'a1, 'b1, 'c1, 'd1, 'e1, 'f1) fmt
| String : ('x, string -> 'a2) padding
* ('a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmt -> ('x, 'b2, 'c2, 'd2, 'e2, 'f2) fmt
| Caml_string : ('x0, string -> 'a3) padding
* ('a3, 'b3, 'c3, 'd3, 'e3, 'f3) fmt -> ('x0, 'b3, 'c3, 'd3, 'e3, 'f3) fmt
| Int : int_conv
* ('x1, 'y) padding
* ('y, int -> 'a4) precision
* ('a4, 'b4, 'c4, 'd4, 'e4, 'f4) fmt -> ('x1, 'b4, 'c4, 'd4, 'e4, 'f4) fmt
| Int32 : int_conv
* ('x2, 'y0) padding
* ('y0, int32 -> 'a5) precision
* ('a5, 'b5, 'c5, 'd5, 'e5, 'f5) fmt -> ('x2, 'b5, 'c5, 'd5, 'e5, 'f5) fmt
| Nativeint : int_conv
* ('x3, 'y1) padding
* ('y1, nativeint -> 'a6) precision
* ('a6, 'b6, 'c6, 'd6, 'e6, 'f6) fmt -> ('x3, 'b6, 'c6, 'd6, 'e6, 'f6) fmt
| Int64 : int_conv
* ('x4, 'y2) padding
* ('y2, int64 -> 'a7) precision
* ('a7, 'b7, 'c7, 'd7, 'e7, 'f7) fmt -> ('x4, 'b7, 'c7, 'd7, 'e7, 'f7) fmt
| Float : float_conv
* ('x5, 'y3) padding
* ('y3, float -> 'a8) precision
* ('a8, 'b8, 'c8, 'd8, 'e8, 'f8) fmt -> ('x5, 'b8, 'c8, 'd8, 'e8, 'f8) fmt
| Bool : ('a9, 'b9, 'c9, 'd9, 'e9, 'f9) fmt -> (bool -> 'a9, 'b9, 'c9, 'd9, 'e9, 'f9) fmt
| Flush : ('a10, 'b10, 'c10, 'd10, 'e10, 'f10) fmt -> ('a10, 'b10, 'c10, 'd10, 'e10, 'f10) fmt
| String_literal : string * ('a11, 'b11, 'c11, 'd11, 'e11, 'f11) fmt -> ('a11, 'b11, 'c11, 'd11, 'e11, 'f11) fmt
| Char_literal : char * ('a12, 'b12, 'c12, 'd12, 'e12, 'f12) fmt -> ('a12, 'b12, 'c12, 'd12, 'e12, 'f12) fmt
| Format_arg : pad_option
* ('g, 'h, 'i, 'j, 'k, 'l) fmtty
* ('a13, 'b13, 'c13, 'd13, 'e13, 'f13) fmt -> (('g, 'h, 'i, 'j, 'k, 'l) format6 -> 'a13, 'b13,
'c13, 'd13, 'e13, 'f13)
fmt
| Format_subst : pad_option
* ('g0, 'h0, 'i0, 'j0, 'k0, 'l0, 'g2, 'b14, 'c14, 'j2, 'd14, 'a14)
fmtty_rel
* ('a14, 'b14, 'c14, 'd14, 'e14, 'f14) fmt -> (('g0, 'h0, 'i0, 'j0, 'k0, 'l0) format6 -> 'g2,
'b14, 'c14, 'j2, 'e14, 'f14)
fmt
| Alpha : ('a15, 'b15, 'c15, 'd15, 'e15, 'f15) fmt -> (('b15 -> 'x6 -> 'c15) -> 'x6 -> 'a15, 'b15, 'c15, 'd15, 'e15, 'f15)
fmt
| Theta : ('a16, 'b16, 'c16, 'd16, 'e16, 'f16) fmt -> (('b16 -> 'c16) -> 'a16, 'b16, 'c16, 'd16, 'e16, 'f16)
fmt
```

Actually it's not that bad

Example

```
"{%d}" :
  Format
   (Char_literal ('{',
     Int (Int_d,
      No_padding,
      No_precision,
      Char_literal ('}',
       End_of_format))),
    "{%d}")

"hello" :
  Format
   (String_literal ("hello",
     End_of_format),
    "hello")
```

- using Generalised algebraic datatypes

Comments on OCaml

- Standard library too small. Use Batteries!

Comments on OCaml

- Standard library too small. Use Batteries!
- Only single-threaded!

Comments on OCaml

- Standard library too small. Use Batteries!
- Only single-threaded!
- Decent infrastructure: ocamllex, ocamlyacc, ocamlbuild, ocamldebug, opam.

Comments on OCaml

- Standard library too small. Use Batteries!
- Only single-threaded!
- Decent infrastructure: ocamllex, ocamlyacc, ocamlbuild, ocamldebug, opam.
- But forget about profiling! (ocamlprof)

Comments on OCaml

- Standard library too small. Use Batteries!
- Only single-threaded!
- Decent infrastructure: ocamllex, ocamlyacc, ocamlbuild, ocamldebug, opam.
- But forget about profiling! (ocamlprof)
- *OOP in OCaml is not usable!*