Core OCaml For C/C++ Programmers

Part II – Designing Types

OCaml PRO – Benjamin Canou

March 9-13 2015

Outline

Defining custom types

Type aliases

Simple (or combinations of) predefined types can be renamed.

- For documentation.
- For use in annotations.

Syntax `type name = type-expr` as in:

```
type point = float * float
type vector = float * float
let translate ((x, y) : point) ((u, v) : vector) : point =
  (x +. u, y +. v)
```

Note: the two types are still compatible.
The annotation are only for humans.

Records

A set of named values grouped in an allocated block memory.

- Syntax `type name = { field-name : type-expr ; ... }`.
- Construction syntax: `{ field = value ; ... }`.
- `{ x = x ; y = y }` can be shortened as `{ x ; y }`.
- Access syntax: `record.field`.

The same example:

```
type point = { x : float ; y : float }
type vector = { u : float ; v : float }
let translate (p : point) (v : vector) : point =
  { x = p.x +. v.u ; y = p.y +. v.v }
```

Note: the two types are distincts.
The annotations are not necessary here.
They become necessary when several records share the same field names.

Records 2 / 2

Record patterns can be used to deconstruct records:

- Only interesting fields can be specified, other are catch-alls.
- `{ x = x ; y = y }` can also be written `{ x ; y }` in patterns.

Simplified example using record patterns for the arguments

```
type point = { x : float ; y : float }
type vector = { u : float ; v : float }
let translate { x ; y } { u ; v } =
  { x = x +. v ; y = y +. v }
```

Mutable fields

Fields can be declared mutable:

- Using the keyword `mutable` before their name in the definition.
- Assignment syntax is `record.field <- value`.

Example:

```
type point = { mutable x : float ; mutable y : float }
type vector = { mutable u : float ; mutable v : float }
let translate ({ x ; y } as p) { u ; v } =
  p.x <- x +. v ;
  p.y <- y +. v
```

A finite set of named cases unified as one type.

- For defining simple tag sets (enums).
- For unifying several types as one (unions).
- And everything in-between.
- The perfect combination with pattern matching.

Simple enums

A finite set of constant constructors.

Syntax:

- Definition: `type name = Constructor | ... | Constructor`
- Construction and pattern: `Constructor`
- Constructors must start with a capital.

Example:

```
type axis = Vertical | Horizontal
let mirror point axis =
  match axis with
  | Vertical -> point.x <- -. point.x
  | Horizontal -> point.y <- -. point.y
```

Exhaustivity is also checked on user types.

Constructors with arguments

Constructors can be used as stand-alone tags or attached to values.

Syntax:

- Definition: `type name = Constructor of type | Constructor | ...`
- Construction and pattern: `Constructor expr`

Example:

```
type axis = Vertical | Horizontal
type operation = Mirror of axis | Scale of float | Identity
let apply point operation =
  match operation with
  | Mirror Vertical -> point.x <- -. point.x
  | Mirror Horizontal -> point.y <- -. point.y
  | Scale s ->
    point.x <- s *. point.x ;
    point.y <- s *. point.y
  | Identity -> ()
```

Exceptions

The type `exn` is actually an extensible sum type.

- The `exception` keyword adds a constructor to `exn` .
- Constructors can take arguments, as with sum types.

Example: boxing a result in an exception for loop breaking.

```
exception Position of int

let find_in_array arr v =
  try
    for i = 0 to Array.length arr - 1 do
      if arr.(i) = v then raise (Position v)
    done ;
    None
  with Position p -> Some p
```

Exceptions

A note on exceptions in OCaml:

- `try` has $O(1)$ cost (not zero)
- `raise` has $O(1)$ cost (not $O(\text{stack size})$)

Example: fast stack rewind using an exception.

```
exception Zero

let mult_ints l =
  try
    let rec loop = function
      | [] -> 1
      | 0 :: _ -> raise Zero
      | v :: vs -> v * loop vs in
    loop l
  with Zero -> 0
```

Type definitions are always recursive (unlike `let` bindings).

- You cannot use a previous `t` when defining `t`.
- Type aliases cannot use recursion, only constructed types.

Syntax `type t1 = ... and t2 = ...` makes t1 and t2 recursive.

Values of recursive types are often treated by recursive functions.
As with functional recursion, don't forget the base case !

Example with records

```
type point =
  { x : float ;
    y : float ;
    base : point option }

let rec shorten = function
  | { base = None } as p -> p
  | { x ; y ; base = Some { x = x' ; y = y' ; base } } ->
    let { x = x' ; y = y' } = shorten base in
    { x = x + x' ; y = y + y' ; base = None }

let rec shorten = function
  | { base = None } as p -> p
  | { x ; y ; base = Some { x = x' ; y = y' ; base } } ->
    shorten { x = x + x' ; y = y + y' ; base }
```

Recursive types

Example with sums

```
type roadmap =
  | Stop
  | Take of string * roadmap
  | Drive of int * roadmap

let format_roadmap roadmap =
  let rec loop mile = function
    | Stop ->
      printf "At mile %d, stop" mile
    | Continue (n, rest) ->
      printf "Continue for %d miles to mile" n (mile + n) ;
      loop (mile + n) rest
    | Take (name, rest) ->
      printf "At mile %d, take exit %s" mile name;
      loop mile rest in
  loop 0 roadmap
```

Mixed example

```ocaml
type road =
  { name : string ;
    exits : (int * exit) list }
and exit =
  | Exit of road | Toll of float * road | Service

let find_service_area_before_toll_booth road =
  let rec find { exits } =
    let rec find_exit = function
      | [] -> None
      | (_, Toll _) :: rest -> find_exit rest
      | (mile, Service) :: _ -> Some (Continue (mile, Stop))
      | (mile, Exit road) :: rest ->
        match find (mile + mile') road with
        | None -> find_exit rest
        | Some cont ->
          Some (Continue (mile, Take (road.name, cont))) in
    find_exit exits in
  find road
```

Polymorphic types

Types can have parameters:

- For applying a same structure to different base types.
- For building polymorphic containers.
- For factorizing code.

Syntax:

- Single parameter: `type 'a t = ...`
- Several parameters: `type ('a, 'b) t = ...` .

Restrictions:

- Variables appearing on the right must be declared as parameters.
- Parameters should appear on the right.

We define a minimal expression language.
The idea is to write little expressions such as:

```
(add 1 2 3 4 5 6)
(letin x (input) (add 1 input))
```

And evaluate them over different numerical domains.

The output from the parser is as follows:

```
type ast =
  | Const of string                    (* const *)
  | Operation of string * ast list     (* (op _ _ ...) *)
  | Var of string                      (* var *)
  | Letin of string * ast * ast        (* (let var _ _) *)
```

We want to evaluate them to an OCaml value.
We define an intermediate language for evaluation to a type `'a`.

```
type 'a expr =
  | Const of 'a
  | Operation of ('a list -> 'a) * 'a expr list
  | Var of string
  | Letin of string * 'a expr * 'a expr
```

The evaluator:

```
let eval expr =
  let rec eval env = function
    | Const x -> x
    | Operation (f, args) ->
      f (List.map (eval env) args)
    | Var v -> List.assoc v env
    | Letin (n, v, b) ->
      let env = (n, eval env v) :: env in
      eval env b in
  eval [] expr
```

We want to parse expression over different domains.
For this we write a generic parsing algorithm.
We abstract the specificities in a record.

```
type 'a parsers =
  { parse_const : string -> 'a ;
    parse_operation : string -> ('a list -> 'a) }
```

The parser will be:

```
let rec parse
  : 'a parsers -> ast -> 'a expr
  = fun parsers -> function
    | Const s ->
      Const (parsers.parse_const s)
    | Operation (n, args) ->
      Operation (parsers.parse_operation n,
                 List.map (parse parsers) args)
    | Var n ->
      Var n
    | Letin (n, v, b) ->
      Letin (n, parse parsers v, parse parsers b)
```

So we can have an `int` instance:

```
let int_parsers : int parsers =
  { parse_const = int_of_string ;
    parse_operation = function
      | "add" ->
        (function
          | [] -> invalid_arg "add"
          | x :: xs -> List.fold_left (+) x xs)
      | "sub" ->
        (function
          | [ x ] -> - x
          | [ x ; y ] -> x - y
          | _ -> invalid_arg "sub")
      | _ -> raise Not_found }
```

Types in the Standard Library

References

The type definition:

```
type 'a ref = { mutable contents : 'a}
```

The operators

```
let (!) r = r.contents
let (:=) r v = r.contents <- v
```

Exercise: find another way !

Lists

The type:

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

The operations:

```
let hd = function
  | Nil -> invalid_arg "hd"
  | Cons (hd, _) -> hd
let tl = function
  | Nil -> invalid_arg "tl"
  | Cons (_, tl) -> tl
```

Exercise: rewrite all the combinators !

Interesting public types in the library

- Complex encodes complex numbers as a record type.
- Unix encode C structures as records and enums as sum types.
- Graphics uses an enum for event listening flags, and a record for status.
- Arg uses a sum type to describe command line arguments.

The Type System

Type Inference in OCaml

OCaml's type inference

- May only fail or return a correct type.
- Will always return the principal (most generic) type.
- May fail on non erroneous programs.

Using Hindley–Milner style unification.

Intuively

- The inference starts by giving generic types to everything.
- If uses the types of called functions and referenced values
  - to check that the local use is compatible ;
  - conversely, to narrow the types of local expressions ;
- In a single pass called unification:
  - Two generic types are unified as one.
  - Two different primitive types are not unified and lead to an error.
  - A generic type and a primitive type unifiy to the primitive.

The inference algorithm is a recursive descent on the program.

For each sub-expression it:

- Allocates fresh variables for all its subexpressions.
- Applies predefined typing rules for relating the subexpressions:
    - The condition of an `if` is unified with `bool`;
    - The branches of a match are unified together;
    - The function parameters and arguments are unified pairwise; etc.

  by calling the unification recursively.
- Finally unifies the expression result type with the expected one.

Destructive unification is used:

- Types variables are references.
- Unifying a type variable with a primitive type destroys the variable.
- Trying to update an already destroyed variable leads to a type clash.
- Updating one occurence will update all
  (so a variable can only be used with one type).

When the body of a `let` is completely typed, potential type variables are generalized, the result is polymorphic.

Typing rules

Out of curiosity, the type system is described using rules that looks like:

$$\frac{x : \sigma \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau}[\mathsf{Var}] \qquad \frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'}[\mathsf{App}]$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\, x\, .\, e : \tau \to \tau'}[\mathsf{Abs}] \qquad \frac{\Gamma \vdash e_0 : \tau \qquad \Gamma, x : \bar{\Gamma}(\tau) \vdash e_1 : \tau'}{\Gamma \vdash \mathtt{let}\ x = e_0\ \mathtt{in}\ e_1 : \tau'}[\mathtt{Let}]$$

A derivation of these rules over a program is proves it is well typed.

For instance, `let x = 2 in 2 + x` is well typed because:

$$\frac{\Gamma \vdash 2 : \mathit{int} \qquad \dfrac{\vdash (+) : \mathit{int} \to \mathit{int} \to \mathit{int} \quad \vdash 2 : \mathit{int} \quad \Gamma, x : \mathit{int} \vdash x : \mathit{int}}{\Gamma, x : \mathit{int} \vdash 2 + x : \mathit{int}}[\mathsf{App}]}{\Gamma \vdash \mathtt{let}\ x = 2\ \mathtt{in}\ 2 + x : \mathit{int}}[\mathtt{Let}]$$

The inference algorithm's goal is to try and compute such a tree.

Troubleshooting

The way inference works explains:

- Why some annotations seem ignored (previous example with aliases).
- Why some functions end up more polymorphic than expected.
- Why some error message are a little cryptic...

```
# let maybe_three x = if x then 3 ;;
Error: This expression has type int
       but an expression was expected of type unit
# let maybe_three x = if x then [] ;;
Error: This variant expression is expected to have type unit
       The constructor [] does not belong to type unit
# let f b x = if b then print_int x else print_float x ;;
Error: This expression has type int
       but an expression was expected of type float
# List.map succ [ 1. ; 2. ; 3. ] ;;
Error: This expression has type float
       but an expression was expected of type int
```

Disambiguation of constructors and fields:

- Necessary when several records have the same name.
- The inference takes the last definition by default.
- Solved with annotations

Weak type variables.

- Some values have type variables that cannot be generalized.
- Example: `let storage = ref [].`
    - Should be of type `'a list ref.`
    - Unsafe since one could insert heterogenous values.
- The toplevel will display a weak type variable: `'_a list ref.`
- This kind of variable is updated upon its first monomorphic usage.

Limitations

Value restriction:

- Once upon a time, only constants and functions were generalized.
- In OCaml this is a lot relaxed, yet correct.
- But the heuristics is not always perfect.
- OCaml may not accept to generalize truly polymorphic values.

The solution is often to turn these values into functions.

Polymorphic recursion:

- The type of a recursive function is unified with its own usage.
- It cannot be polyorph is used recursively in a monmorphic way.

Example:

```
# let rec first x l =
    match l with  [] -> x | l :: _ -> l
  and f () =
    (first 1 [], first 1.  []) ;;
Error: This expression has type float
           but an expression was expected of type int
```

Solution

```
# let rec first : 'a. 'a -> 'a list -> 'a
    = fun x l -> match l with [] -> x | l :: _ -> l
  and f () = (first 1 [], first 1. []) ;;
val first : 'a -> 'a list -> 'a = <fun>
val f : unit -> int * float = <fun>
```

Outline