Introduction to Functional Programming Using OCaml

Romain Bardou

July 6, 2011

Introduction

There are two ways to write error-free programs;
only the third one works.

— Alan J. Perlis

let's talk about

typing

Typing

types capture:

- invariants about variables
- design intents of the programmer

examples of such invariants:

- some variable x always contains an integer
- some variable l always contains a list
- some variable l' always contains a list of integers

types prevent errors such as:

- inserting a value into an integer (instead of a list)
- adding two lists together

Typing: Example

(all examples in this talk are written in OCaml)

```
let x: int = 5 in
let y: int = x + 10 in
let z: int list = y :: [ 1; 2; 3 ] in
...
```

(note: type annotations can be omitted thanks to type inference)

Typing: Example of Errors

you cannot add an integer and a list

```
5 + [ 1; 2; 3 ] (* type error *)
```

you cannot insert an integer into an integer

```
42 :: 69 (* type error *)
```

you cannot insert an integer into a list of lists of integers

```
let x: int list = [ 1; 2; 3 ] in
let y: int list list = [ [ 42; 69 ]; [ 4 ] ] in
let z: int list list = x :: y in
10 :: z (* type error *)
```

Typing is Done During Compilation

typing is done before the program is executed

errors are found before the program is executed

program is well-typed $\Longrightarrow$ whole category of errors prevented

let's talk about

algebraic types

Richer Types

types encapsulate invariants and design intents

the richer types are, the richer the invariants

types help the programmer to structure his data

$\Longrightarrow$ need richer types for more complex structures

Algebraic Types: Product Types (a.k.a. Records)

```
type complex =
  {
    re: float; (* real part *)
    im: float; (* imaginary part *)
  }

let add_complex (x: complex) (y: complex) =
  {
    re = x.re +. y.re;
    im = x.im +. y.im;
  }

let x: complex = { re = 0.; im = 10. }
let y: float = x.re
let z = add_complex x 2 (* type error *)
let t = x.toto (* type error *)
```

Algebraic Types: Product Types (a.k.a. Tuples)

instead of declaring a record type, you can use tuples

```
let add_complex (re1, im1) (re2, im2) =
  (re1 +. re2, im1 +. im2)
let x: (float * float) = (0., 10.)
let y = add_complex x 2 (* type error *)
```

use pattern-matching to read the components of the tuple

```
let ((z: float), (t: float)) = x
let (z, t) = x
let (_, _, u) = x (* type error *)
```

Algebraic Types: Sum Types (a.k.a. Variants)

```
type atom = H | He | Li | Be | B | C | N | O
type orbital = S | P | D | F

let orbital_of_atom (a: atom): (orbital * int) =
  match a with
    | H | Li → (S, 1)
    | He | Be → (S, 2)
    | B → (P, 1)
    | C → (G, 2) (* type error: G not an orbital *)
    | O → (P, 4)
    (* type error: we forgot atom N *)
```

Example: The Researcher Data Structure

possible solution: using a product type

```
type researcher =
  {
    student: bool; (* true iff the researcher is a student *)
    name: string;
    phd_students: string list;
  }
```

Example: The Researcher Data Structure

problem: ensure that students have no PhD students
solution: use a sum type

```
type researcher =
  | Student of string
  | Professor of string * string list
```

Example: The Researcher Data Structure

to read the list of PhD students, we need pattern-matching

```
let phd_students_of_researcher x =
  match x with
    | Student _ → []
    | Professor (_, l) → l
```

ensures students always have no PhD students
ensures the programmer considers all cases

Example: Lists

let's define our own list type!

```
type 'a mylist =
  | Empty
  | Cons of 'a * 'a mylist

let empty_list = Empty
let list_singleton = Cons ("coucou", Empty)
let list_1_2_3 = Cons (1, Cons (2, Cons (3, Empty)))
```

Example: Lists

compute the length of a list using a recursive function

```
let rec length x =
  match x with
    | Empty → 0
    | Cons (_, rem) → 1 + length rem
```

note: typing prevents me from forgetting the empty case
(not the case in languages with the NULL pointer)

Polymorphism

type of function length:

$$'a \text{ list} \rightarrow \text{int}$$

'a can be instanciated with any type

```
let x = length (Cons (1, Empty))
let y = length (Cons ("salut", Cons ("toi", Empty)))
```

avoid code duplication, avoid errors

Types: Conclusion

encode properties of your data structure in its type
the compiler ensures you preserve the properties
you thus avoid many programming errors

algebraic types are quite expressive
can often replace the heavy object paradigm

polymorphism avoids code duplication
... and, as a consequence, error duplication

let's talk about

side-effects

Variables Are Immutable

in OCaml, variables are immutable

```
let x = 1
```

the value of x is now 1 until the end of time

Variables Are Immutable

```
let x = 1
let x = 2
```

the first x is hidden
the second x is atually another variable
the code is comparable to:

```
let x_1 = 1
let x_2 = 2
```

References

a reference is a mutable value

```
let x = ref 0 in
x := 1;
x := !x + 3
```

Parenthesis: Initialization

must give an initial value to all variables, all references

avoids errors such as:

```
let x: int in (* not proper OCaml! *)
if x = 0 then (* probably an error: x is not initialized! *)
   ...
else
   ...
```

Mutable Records

record fields can be mutable

```
type complex =
  {
    mutable re: float;
    im: float;
  }

let x = { re = 0.; im = 10. } in
x.re ← 5.;
x.im ← 15.; (* type error: im is not mutable *)
```

References Are Mutable Records

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

let make_ref x = { contents = x }
let get x = x.contents
let set x y = x.contents <- y

let x = make_ref 0 in
set x (get x + 5)                          (* x := !x + 5 *)
```

While Loops

compute $\sum_{i=1}^{10} i$ with a **while** loop

```
let i = ref 1 in
let sum = ref 0 in
while !i <= 10 do
  sum := !sum + !i;
  i := !i + 1
done
```

For Loops

compute $\sum_{i=1}^{10} i$ with a **for** loop

```
let sum = ref 0 in
for i = 1 to 10 do
  sum := !sum + i
done
```

Issues With Side-Effects: Aliasing

```
let x = ref 0 in
let y = x in
x := 5;
```

what is the value of `!y` ?

$\Longrightarrow$ side-effects make it harder to reason about your program

Issues With Side-Effects: Concurrency

```
let x = ref 0 in
x := 5;
```

what is the value of `!x` if other programs can assign `x` at any time?

$\Longrightarrow$ side-effects are dangerous in the context of concurrency

Issues With Side-Effects: Not “Mathematical”

```
let i = ref 1 in
let sum = ref 0 in
while !i <= 10 do
  sum := !sum + !i;
  i := !i + 1
done
```

this is far from the mathematical definition of $\sum_{i=1}^{10} i$

$\Longrightarrow$ side-effects make it harder to reason about your program

let's talk about

functional programming

Functional Programming

the functional programming paradigm:

- no side-effects (i.e. pure programs)
- strict but rich type system
- no goto (gotos are evil)
- functions are values

brings the programming language closer to mathematics

Loops Are Recursive Functions

compute $\sum_{i=1}^{n} i$ using the functional approach

```
let rec sum n =
  if n <= 0 then
    0
  else
    n + sum (n - 1)
```

Loops Are Recursive Functions

compute $\sum_{i=1}^{n} i$ using the functional approach, again

```
let rec sum_aux acc n =
  if n <= 0 then
    acc
  else
    sum_aux (acc + n) (n - 1)

let sum n = sum_aux 0 n
```

Partial Application

```
let sum n = sum_aux 0 n
```

is equivalent to

```
let sum = sum_aux 0
```

The Type of Functions

```
let add a b = a + b
```

type of function add is

$$\text{int} \rightarrow \text{int} \rightarrow \text{int}$$

(read as int $\rightarrow$ (int $\rightarrow$ int))

function taking an integer argument a and returning another function taking an integer argument b and returning integer $a + b$

The Type of Functions and Partial Application

example: partial application of function add

```
let f = add 5
```

type of function f is

$$\text{int} \rightarrow \text{int}$$

function taking an integer argument b and returning integer $5 + b$

Functions Are Values

```
let add a b = a + b
```

is actually the same as:

```
let add a = fun b -> a + b
```

or as:

```
let add = fun a -> fun b -> a + b
```

Functions as Arguments

functions can be given as arguments to other functions

```
let f (g: int -> int) (x: int): int =
  g x + 10

let n = f (add 5) 3
let m = f (fun x -> 2 * x) 3
```

what is the value of *n* and *m*?

Iterator: List Mapping

```
let rec map (f: 'a → 'b) (l: 'a list): 'b list =
  match l with
    | [] → []
    | x :: rem → f x :: map f rem

let x = map (add 5) [ 1; 2; 3 ]
let y = map orbital_of_atom [ H | N | Be | Li ]
```

what is the value of x and y?

Iterator: List Folding

```
let rec fold
    (f: 'a → 'b → 'a)
    (a: 'a)
    (l: 'a list): 'b list =
  match l with
    | [] → a
    | x :: rem → fold f (f a x) rem

let x = fold add 0 [ 1; 2; 3 ]
```

what is the value of x?

Example: Sum

compute $\sum_{i=1}^{n} i$ using the functional approach, again again

```
let rec make_list n =
  if n <= 0 then
    []
  else
    n :: make_list (n - 1)

let sum n = fold add 0 (make_list n)
```

Combinator: Function Composition

```
let compose f g x =
  f (g x)
```

compute $\sum_{i=1}^{n} i$ using the functional approach, again again again

```
let sum = compose (fold add 0) make_list
```

$\Longrightarrow$ good combination properties

Matrix Product

recipe for a modular matrix product:

1. write a function which returns the product of two matrices
2. replace the use of `*` with a function argument
3. enjoy a more general polymorph product function

apply it to:

- integer multiplication `*` for integer matrices
- float multiplication `*.` for float matrices
- other operators for other algebras

Functional Programming: Conclusion

functional programming matters

- modular
- good compositional properties
- closer to a well-known language: mathematics
- less error-prone

let's talk about

numeric computation

OCaml and Numeric Computation

available libraries for OCaml (standard library):

- various integers
  - int
    31 bits (32-bits processors) or 63 bits (64-bits processors)
    default integers of OCaml, fast
  - int32, int64
    less efficient, but one more bit
  - arbitrary precision integers (modules Num and Big_int)
- floats
  native floats, fast under some conditions
- large arrays (module Bigarray) of various integers and floats;
  any dimension (vectors, 2D matrices, 3D matrices, and more);
  compatible with FORTRAN matrices

bindings for libraries of other languages (including FORTRAN)
may be written; some may already exist

Conclusion

should you use OCaml?

pros:

- less error-prone
- concise
- expressive (algebraic types, objects, modules and functors, labels, polymorphic variants)
- scalable (modular, compositional)
- maintainable
- fast to compile
- fast to execute

cons:

- young (less available libraries and tools)

References That Might, or Might Not, Be of Interest

OCaml website (download, documentation, community contents)
http://caml.inria.fr/

John Hughes
Why Functional Programming Matters

Emmanuel Chailloux, Pascal Manoury and Bruno Pagano
Developing Applications With Objective Caml

Guy Cousineau and Michel Mauny
The Functional Approach to Programming

Jon D. Harrop
OCaml for Scientists