Learning To Learn OCaml Lecture 2

Alexander Berenbeim

2016-10-18 Tue

Contents

1 Building Programs By Building Functions

1.1 Naming

- It is inefficient to re-enter sub-expressions within the same program.
- `OCaml` allows us to define a name to stand in for a result of a single evaluated expression with the `let ... = ... in` construct

1.2 Example of Naming

```
# 100*100*100;;
-: int = 1000000
# let x = 100;;
val x : int = 100
# let x = 100 in x * x * x;;
- : int = 1000000
# let cube x = x * x * x;;
val cube : int -> int = <fun>
# cube 100;;
- : int = 1000000
```

1.3 The Limits of Naming

- Using our `cube` function from the previous slide

```
# cube false;;
Error: This expression has type bool
but an expression was expected of type int
```

- `OCaml` is a powerful *and* challenging language to use because of its **type** inference system
- `OCaml` is statically typed and checks types at compile-time, so we can catch immediately what is wrong with our example. In this case, cube will only accept as an input an argument of type `int`.
- As we will see in a bit, `OCaml` cannot recognize all errors; a **good** programmer will anticipate that **exceptions** will need to be raised.

1.4 Building Boolean Predicates :: Pattern Recognition

- A classic programming application is **boolean** classification.
- `OCaml` allows us to define Boolean classifications for all types using pattern recognition and the inductive constructions of types

```
# let isvowel c =
c = 'a' || c = 'e' || c = 'i' || c = 'o' || c = 'u';;
val isvowel : char -> bool = <fun>
```

1.5 Building Boolean Predicates :: Algebraic Identity

```
# let addtotwentyone a b =
a + b = 21;;
val addtoten : int -> int -> bool = <fun>
# addtotwentyone 8 12;;
- : bool = false
# addtotwentyone 9 12;;
- : bool = true
```

1.6 Recursive functions

- Recursion is powerful.
- Recursive constructions are explicitly defined by invoking the `rec` constructor in `OCaml`

```
# let rec gcd a b =
#if b = 0 then a else gcd b (a mod b);;
val gcd : int -> int -> int = <fun>
# gcd 64000 3456;;
- : int = 128
# let rec factorial a =
if a = 1 then 1 else a * factorial (a - 1);;
val factorial : int -> int = <fun>
```

1.7 Recursive Constructions :: Lists

- Lists are a **polymorphic** construction in `OCaml` with two pre-defined operators `::` (the "cons" operator) and `@` (the "append" operator) used to put witnesses of a type α an α list or attach an α list to an α list

```
# 1::[2;3;4];;
- : int list = [1;2;3;4];;
# [false;true] @ [true;true];;
- : bool list = [false;true;true;true];;
# let rec length l =
match l with
[] -> 0
| h :: t -> 1 + length t;;
val length : 'a list -> int = <fun>
```

1.8 Some List Operations Example

```
let rec take n l =
if n < 1 then [] else
match l with
h :: t -> h :: take (n-1) t;;
- : int -> 'a list -> 'a list = <fun>
let rec drop n l =
if n < 1 then l else
match l with
h :: t -> drop (n - 1) t;;
- : int -> 'a list -> 'a list = <fun>
```

1.9 Knowing When Things Will Go Wrong

- The operations defined on the previous slide aren't pattern exhaustive; we shouldn't trust that it will work because of what we know about the integers and the fixed points in those arguments.
- `take 2 [true]` will **Raise an Exception**, ="Match Failure"=.
- Exceptions are how `OCaml` reports such **run-time** errors.
- `OCaml` has some built-in exceptions like `Division_by_zero`, although we will often have to make these exceptions ourselves.

```
# 10 / 0;;
Exception : Division_by_zero.
```

- We make these exceptions ourselves with the **raise** constructor.

1.10 Raising Exceptions

- We fix our functions on lists as follows:

```
let rec take n l =
match l with
[] -> if n = 0
then []
else raise (Invalid_argument "take")
| h :: t -> if n  < 0
then raise (Invalid_argument "take")
```

```
else if n = 0 then [ ] else
h::take (n - 1) t
- : int -> 'a list -> 'a list = <fun>
```

- Additional Exercise: Fix the `drop` function.

1.11 More Exceptions

- We can do more than raising exceptions in functions; we can define them.
- These two examples *carry an integer* along with the exception; that is, we can define Exceptions that use the `of` construct to introduce the type of information that the exception carries.

```
# exception Undefined of int;;
exception Undefined of int
# let f x = if x = 0 then raise (Undefined 0)
else 100 / x;;
val f : int -> int = <fun>
# f 6 = 16;;
- : int = 16
# f 0 ;;
Exception Undefined 0.
```

1.12 Handling Exceptions

- We may not only raise exception; we may **handle exceptions** with an **exception handler** written using the `try ... with` contruct

```
# let safe_divide x y =
try x / y with
 Division_by_zero -> 0;;
# val safe_divide : int -> int -> int = <fun>
```

1.13 Dictionaries

- One common structure is the **dictionary** which associates **keys** with **values**
- We can think of dictionaries as a witness of an $\alpha \times \beta$ `list`

- Product types have naturally defined projection maps:

```
# let fst p = match p with (x,_) -> x;;
# val fst : 'a * 'b -> 'a = <fun>
# let snd p = match p with (_,y) -> y;;
# val snd : 'a * 'b -> 'b = <fun>
```

1.14 Example: Reading a Dictionary

```
# let rec lookup x l =
match l with
[] -> raise Not_found
| (k,v) :: t ->
if k = x then v else lookup x t;;
# val lookup : 'a -> ('a * 'b) list -> 'b = <fun>
```