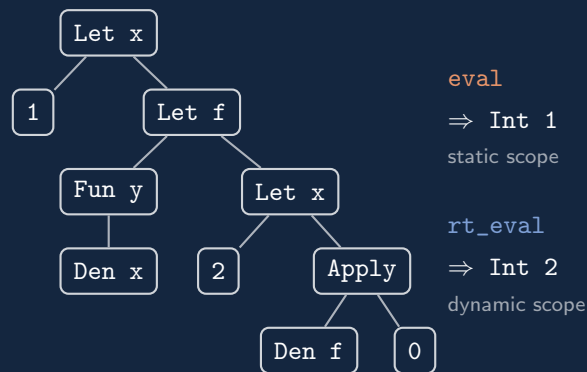




UNIVERSITÀ DI PISA

INTERPRETER IN OCAML

syntax • environments • closures • scoping



ApplyOver(Fun("x", Sum(Den "x", CstInt 3)), d)

OCaml • pattern matching • algebraic data types

An Interpreter for a Functional Language in OCaml: Static and Dynamic Scoping, with Dictionaries

Project Report

Giuseppe Muschetta

B.Sc. in Computer Science

Course: Programming II • Course Instructor: Prof. Gian-Luigi Ferrari

Abstract

This report presents the OCaml part of the Programming II project: an interpreter for a small functional language, extended with a dictionary type of its own and written in two versions, one for static and one for dynamic scoping. The language has integers, booleans and strings, arithmetic, logical and comparison operators, conditionals, local declarations, first-class functions and recursive functions; its abstract syntax is an OCaml algebraic data type with 30 constructors, and each interpreter is a recursive function that gives meaning to every constructor by pattern matching. The run-time support represents environments as association lists and functions as closures, and checks the types of the operands of every primitive operation at run time. Dictionaries map identifiers to values and come with six operations: creation, lookup, insertion, removal, clearing and the application of a function to every value; each one returns a new dictionary and signals its errors with dedicated exceptions. The two interpreters, `eval` and `rt_eval`, differ only in how functions are represented and applied, and the project's test program shows the consequence: the same expression evaluates to 1 under static scoping and to 2 under dynamic scoping. All the test expressions were run with both interpreters, together with further checks of recursion, error handling and scoping inside dictionary operations.

Contents

1	Introduction	2
2	Abstract syntax	2
3	Run-time support	3
3.1	Values	3
3.2	Environments	3
3.3	Dynamic type checking	3
4	Static scoping: eval	4
5	Dynamic scoping: rt_eval	5
5.1	The difference on one program	5
6	Dictionaries	6
7	Testing	6
7.1	The test expressions	6
7.2	Further checks	7
8	Conclusion	8

1 Introduction

An interpreter is the most direct way to define what the programs of a language mean: for every construct of the language it says how to compute a value from it. In a functional language such as OCaml this definition can be written almost literally. The programs to be interpreted are data, described by an algebraic type with one constructor per construct, and the interpreter is a recursive function that analyzes them by pattern matching, with one case per constructor.

The second part of the Programming II project starts from the didactic functional language presented in the course and asks for two extensions:

1. a **dictionary** type, a collection of key-value pairs with operations to create a dictionary, read the value of a key, add and remove pairs, empty it and apply a function to all its values;
2. two interpreters, one for **static** and one for **dynamic** scoping, so that the same programs can be run under both rules and compared.

The whole project is a single OCaml file, `OCAML-interpreter.ml`, of about 540 lines: the abstract syntax (Section 2), the run-time support (Section 3), the interpreter with static scoping, `eval` (Section 4), the one with dynamic scoping, `rt_eval` (Section 5), and a set of test expressions. Section 6 describes the dictionaries, which both interpreters share, and Section 7 the tests. The project's first part, a secure data container written in Java, is described in a separate report.

2 Abstract syntax

Programs are represented by their *abstract syntax*, the tree structure that remains once parsing has dealt with concrete details such as parentheses and precedence. The project does not include a parser: test programs are written directly as syntax trees, as values of the type `exp`.

Identifiers are strings, `type ide = string`, and the type `exp` has 30 constructors, grouped in Table 1. Dictionaries need a syntax of their own, because a dictionary written in a program is a sequence of pairs whose values are still expressions to evaluate:

Listing 1: The syntax of dictionaries: a list of pairs (key, expression).

```
and dictionary = Empty | Item of (ide * exp) * dictionary;;
```

Table 1: The 30 constructors of the abstract syntax `exp`.

Group	Constructors
Constants and names	<code>CstInt</code> of <code>int</code> , <code>CstTrue</code> , <code>CstFalse</code> , <code>CstStr</code> of <code>ide</code> , <code>Den</code> of <code>ide</code>
Arithmetic	<code>Sum</code> , <code>Diff</code> , <code>Times</code> , <code>Div</code> , <code>Mod</code> (binary), <code>Minus</code> (unary)
Comparison	<code>Equals</code> , <code>Minor</code> , <code>Greater</code> , <code>Minoreq</code> , <code>Greatereq</code>
Logic	<code>And</code> , <code>Or</code> , <code>Not</code>
Control and binding	<code>Ifthenelse</code> of <code>exp * exp * exp</code> , <code>Let</code> of <code>ide * exp * exp</code>
Functions	<code>Fun</code> of <code>ide * exp</code> , <code>Apply</code> of <code>exp * exp</code> , <code>Letrec</code> of <code>ide * ide * exp * exp</code>
Dictionaries	<code>Dict</code> of <code>dictionary</code> , <code>Get</code> of <code>exp * ide</code> , <code>Add</code> of <code>exp * ide * exp</code> , <code>Remove</code> of <code>exp * ide</code> , <code>Clear</code> of <code>exp</code> , <code>ApplyOver</code> of <code>exp * exp</code>

Functions have one parameter: `Fun("x", body)` is the anonymous function of `x`, and `Letrec(f, x, body, e)` declares a recursive function `f` of parameter `x`, visible both in its own body and in the expression `e`. For example, the factorial of 5 is written

```
Letrec("fact", "n",
      Ifthenelse(Equals(Den "n", CstInt 0), CstInt 1,
                  Times(Den "n", Apply(Den "fact", Diff(Den "n", CstInt 1)))),
      Apply(Den "fact", CstInt 5))
```

3 Run-time support

3.1 Values

Evaluating an expression produces an *expressible value*, of type `evT` (Listing 2). Besides integers, booleans and strings, the values include dictionaries and four representations of functions: two for each scoping rule, one for ordinary and one for recursive functions. `Unbound` is the value associated with an identifier that has no binding.

Listing 2: The expressible values.

```
type evT =
| Unbound
| Int of int
| Bool of bool
| String of string
| Closure of ide * exp * evT env          (* static scope *)
| RecClosure of ide * ide * exp * evT env  (* static scope, recursive *)
| DynFun of ide * exp                     (* dynamic scope *)
| RecDynFun of ide * ide * exp             (* dynamic scope, recursive *)
| Dictionary of (ide * evT) list;;
```

At run time a dictionary is simply a list of pairs of keys and *values*: the expressions of its syntax have been evaluated once, when the dictionary was created.

3.2 Environments

An environment associates identifiers with values. It is represented as an association list, type `'t env = (ide * 't) list`, with two operations: `bind` adds a pair at the head of the list, and `lookup` returns the value of the first pair with the given identifier. Because the search stops at the first match, a new binding of a name *shadows* the older ones without destroying them: when the evaluation leaves the scope of a declaration, the environment that was in use before it is still intact. This is what makes a purely functional representation of environments possible: nothing is ever updated in place.

3.3 Dynamic type checking

The language is dynamically typed: the interpreter checks the type of the operands of every primitive operation when it executes it. A function `typecheck` tests whether a value belongs to one of the types `"int"`, `"bool"` and `"dictionary"`, and each primitive operation uses it before computing its result:

Listing 3: A primitive operation with its type check: integer division.

```
let div x y =
  if typecheck "int" x && typecheck "int" y
  then (
    match (x, y) with
    | (Int n, Int u) → if u = 0 then failwith ("Division by zero")
                      else Int(n / u)
    | (_, _) → raise (GenericError "div function")
  )
```

```
else failwith("Type error");;
```

Run-time errors are reported with the exception **Failure** and a message that names the problem: a type mismatch ("Type error"), a guard that is not a boolean ("Non boolean guard"), the application of something that is not a function ("Non functional value") and division or modulo by zero. Dictionaries have two exceptions of their own, **KeyNotFound** and **ExistingKey**, described in Section 6.

4 Static scoping: eval

Under **static** (or lexical) scoping, a free variable in the body of a function refers to the binding that was visible *where the function was defined*. To implement it, a function value must remember that environment: it is a **closure**: its parameter, its body and its environment of definition.

The interpreter **eval e r** computes the value of the expression **e** in the environment **r**. Most of its cases are a direct translation of the meaning of each construct: a constant evaluates to itself, **Den i** looks **i** up in **r**, a primitive operation evaluates its operands and applies the corresponding function of the run-time support, and **Let(i, e1, e2)** evaluates **e2** in **r** extended with **i** bound to the value of **e1**. The cases that define scoping are those of functions:

Listing 4: The function cases of **eval**.

```
| Fun (param, fbody) → Closure (param, fbody, r)
| Letrec (f, param, fbody, lbody) →
    let bEnv = bind r f (RecClosure (f, param, fbody, r)) in
    eval lbody bEnv
| Apply (eF, eArg) →
    let fclosure = eval eF r in (
    match fclosure with
    | Closure (arg, fbody, fDecEnv) →
        let aVal = eval eArg r in
        let aEnv = bind fDecEnv arg aVal in
        eval fbody aEnv
    | RecClosure (f, arg, fbody, fDecEnv) →
        let aVal = eval eArg r in
        let rEnv = bind fDecEnv f fclosure in
        let aEnv = bind rEnv arg aVal in
        eval fbody aEnv
    | _ → failwith ("Non functional value")
    )
```

The same rules can be written in the notation of big-step operational semantics, where $r \vdash e \Downarrow v$ reads “in the environment r , the expression e evaluates to the value v ”, and $r[x \mapsto v]$ is r extended with the binding of x to v :

$$\frac{}{r \vdash \text{Fun}(x, b) \Downarrow \text{Closure}(x, b, r)}$$

$$\frac{r \vdash e_1 \Downarrow \text{Closure}(x, b, r_d) \quad r \vdash e_2 \Downarrow v \quad r_d[x \mapsto v] \vdash b \Downarrow v'}{r \vdash \text{Apply}(e_1, e_2) \Downarrow v'}$$

The body of the function is evaluated in its environment of definition r_d , extended with the parameter: the environment r of the call is used only to evaluate the argument.

Recursion. A recursive function needs to see its own name in its body, but its closure cannot contain itself directly. The interpreter solves the problem at the time of the call: **Letrec** creates a **RecClosure** that records the name of the function, and every application of a **RecClosure** first binds that name to the closure itself in the environment of definition, then binds the parameter. Each recursive call therefore finds the function exactly where static scoping says it should be.

5 Dynamic scoping: `rt_eval`

Under **dynamic** scoping, a free variable in the body of a function refers to the most recent binding *active at the time of the call*. A function no longer needs to remember an environment: its value is just its parameter and its body, `DynFun(param, body)`, and the body is evaluated in the environment of the caller.

The interpreter `rt_eval` has the same cases as `eval`, and differs only where functions are created and applied:

Listing 5: The function cases of `rt_eval`: no environment is captured, and the body runs in the caller's environment `r`.

```
| Fun (param, fbody) → DynFun (param, fbody)
| Letrec (f, param, fbody, lbody) →
    let bEnv = bind r f (RecDynFun (f, param, fbody)) in
    rt_eval lbody bEnv
| Apply (eF, eArg) →
    let fval = rt_eval eF r in (
    match fval with
    | DynFun (arg, fbody) →
        let aVal = rt_eval eArg r in
        let aEnv = bind r arg aVal in
        rt_eval fbody aEnv
    | RecDynFun (f, arg, fbody) →
        let aVal = rt_eval eArg r in
        let rEnv = bind r f fval in
        let aEnv = bind rEnv arg aVal in
        rt_eval fbody aEnv
    | _ → failwith ("Non functional value")
    )
```

$$\frac{}{r \vdash \text{Fun}(x, b) \Downarrow \text{DynFun}(x, b)}$$

$$\frac{r \vdash e_1 \Downarrow \text{DynFun}(x, b) \quad r \vdash e_2 \Downarrow v \quad r[x \mapsto v] \vdash b \Downarrow v'}{r \vdash \text{Apply}(e_1, e_2) \Downarrow v'}$$

The only difference from the static rule is the environment in the last premise: r , the environment of the call, instead of r_d , the environment of the definition.

5.1 The difference on one program

The project's test `e8` isolates the difference in four lines: a function `f` reads the variable `x`, and `x` is declared again between the definition of `f` and its call.

Listing 6: The scoping test `e8`, in the abstract syntax and in OCaml-like notation.

```
let e8 = Let("x", CstInt 1, Let("f", Fun("y", Den "x"),
    Let("x", CstInt 2, Apply (Den "f", CstInt 0))));;

(* let x = 1 in
   let f = fun y → x in
   let x = 2 in f 0 *)
```

With static scoping, `f` is a closure that captured the environment where `x` is 1, and the call returns `Int 1`. With dynamic scoping, the body of `f` is evaluated in the environment of the call, where the most recent binding of `x` is 2, and the call returns `Int 2`. Figure 1 shows the two environments in which the body `Den "x"` is evaluated.

Static scoping (eval)

environment of definition of `f`, extended with `y`

`y = 0` `x = 1` $\Rightarrow$ Int 1

Dynamic scoping (rt_eval)

environment of the call, extended with `y`

`y = 0` `x = 2` `f` `x = 1` $\Rightarrow$ Int 2

Figure 1: The environments in which the body of `f` reads `x` in the test `e8`, newest binding on the left. The lookup stops at the first binding of `x`.

6 Dictionaries

A dictionary associates keys, which are identifiers, with values of any type of the language, functions included. Both interpreters implement the six operations in the same way, and each operation on an existing dictionary first checks, with `typecheck "dictionary"`, that its argument evaluates to a dictionary. No operation modifies a dictionary: each one returns a *new* dictionary value, so a dictionary bound to a name never changes, in keeping with the functional nature of the language.

`Dict(d)` creates a dictionary from its syntax, evaluating its values in the current environment. A key may appear only once: if a key is repeated, the creation fails.

`Get(d, k)` returns the value associated with the key `k`, and raises `KeyNotFound` if there is none.

`Add(d, k, e)` returns the dictionary extended with the pair of `k` and the value of `e`, at the end; if the key is already present it raises `ExistingKey` instead of overwriting the existing value.

`Remove(d, k)` returns the dictionary without the pair of key `k`, and raises `KeyNotFound` if the key is absent.

`Clear(d)` returns the empty dictionary.

`ApplyOver(f, d)` applies the function `f` to every value of `d` and returns the dictionary of the results, with the same keys in the same order. The function is evaluated, checked to be a function under the current scoping rule, and applied to each value with the same rules as `Apply`.

The functions that work on the run-time representation are short recursive functions on lists, such as the one that implements `Add`:

Listing 7: Adding a pair: the list is copied up to the end, and an existing key raises an exception.

```
let rec add dict key value =
  match dict with
  | [] → (key, value)::[]
  | (k, v)::tail → if k = key
                    then raise (ExistingKey "Key already in the dictionary")
                    else (k, v)::(add tail key value);;
```

Since `ApplyOver` applies its function with the scoping rule of the interpreter that runs it, it is also a second place where the two rules can give different results: Section 7 shows an example.

7 Testing

7.1 The test expressions

The source file ends with nine test expressions, `e0` to `e8`: the first eight exercise the dictionaries, and `e8` is the scoping test of Listing 6. For this report, each of them was evaluated in the empty environment with both interpreters, by loading the unchanged source file in the OCaml toplevel (version 5.5). Table 2 reports the results.

Table 2: The project's test expressions, evaluated in the empty environment with `eval` and `rt_eval`.

Test	Expression	Result (both interpreters, except e8)
e0	a dictionary with the key "two" repeated	fails with Failure : a dictionary cannot have duplicate keys
e1	Dict of one, two, three, four $\mapsto 1, 2, 3, 4$	[("one", 1); ("two", 2); ("three", 3); ("four", 4)]
e2	Get(e1, "three")	Int 3
e3	Add(e1, "five", CstInt 5)	e1 followed by ("five", 5)
e4	Add(e3, "six", Sum(CstInt 2, CstInt 4))	e3 followed by ("six", 6)
e5	Remove(e4, "three")	[("one", 1); ("two", 2); ("four", 4); ("five", 5); ("six", 6)]
e6	Clear(e3)	Dictionary []
e7	ApplyOver(Fun("x", Sum(Den "x", CstInt 3)), e4)	[("one", 4); ("two", 5); ("three", 6); ("four", 7); ("five", 8); ("six", 9)]
e8	the scoping test (Listing 6)	eval: Int 1 rt_eval: Int 2

The results also show that the operations do not modify their arguments: `e4` and `e6` are both computed from `e3`, and `e5` and `e7` from `e4`, yet each of them finds its argument exactly as it was built by the previous test.

7.2 Further checks

A second group of expressions, written for this report and evaluated in the same way, covers recursion, the run-time errors and scoping inside dictionary operations (Table 3).

Table 3: Further checks, run with the unchanged interpreter.

Expression	Result
factorial of 5 with Letrec (Section 2)	Int 120 with both interpreters
Get(e1, "ten"), Remove(e1, "ten")	KeyNotFound
Add(e1, "two", CstInt 9)	ExistingKey
Div(CstInt 7, CstInt 0)	Failure "Division by zero"
Ifthenelse(CstInt 1, CstInt 2, CstInt 3)	Failure "Non boolean guard"
Apply(CstInt 3, CstInt 4)	Failure "Non functional value"
Sum(CstInt 1, CstTrue)	Failure "Type error"
Mod(CstInt 17, CstInt 5), Minus(CstInt 4)	Int 2, Int -4
ApplyOver of a function that reads a free variable (below)	eval: values +10 rt_eval: values +100

The last check combines dictionaries and scoping. A function `addk` adds the value of a free variable `k` to its argument; `k` is 10 where `addk` is defined and 100 where `ApplyOver` uses it:

```
Let("k", CstInt 10,
  Let("addk", Fun("v", Sum(Den "v", Den "k")),
    Let("k", CstInt 100, ApplyOver(Den "addk", e1))))
```

With static scoping `addk` keeps its environment of definition, and the dictionary `e1` becomes [("one", 11); ("two", 12); ("three", 13); ("four", 14)]; with dynamic scoping it reads the binding of `k` active when `ApplyOver` applies it, and the result is [("one", 101);

(`"two"`, 102); (`"three"`, 103); (`"four"`, 104)]. The same function, applied to the same dictionary, gives two different dictionaries: the scoping rule matters for every construct that applies functions, not only for `Apply`.

8 Conclusion

The interpreter shows how little machinery is needed to define a functional language precisely: an algebraic type for the syntax, a list for the environment, a type of values and one recursive function. Pattern matching makes each case of the interpreter read like the definition of a construct, and OCaml's exhaustiveness checks make it hard to forget one. Writing the language twice, once for each scoping rule, made the difference between the two rules concrete. It lies entirely in what a function value contains and in which environment its body is evaluated: only the cases that create and apply functions change, a small part of the interpreter, yet enough to change the result of a program. The dictionary extension, finally, showed how a new data type enters a language: a syntax for its literals, a run-time representation, a type check, and operations whose errors are reported with exceptions of their own.

References

- [1] M. Gabbrielli and S. Martini. *Programming Languages: Principles and Paradigms*, second edition. Springer, 2023.
- [2] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy and J. Vouillon. *The OCaml System: Documentation and User's Manual*. INRIA. <https://ocaml.org/manual/>.
- [3] G. Kahn. Natural Semantics. In *STACS 87*, Lecture Notes in Computer Science 247, pages 22–39. Springer, 1987.
- [4] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.