Objective Caml per programmatori Python

Paolo Donadeo[1]

[1]Studio Associato 4Sigma, Milano

19 Febbraio 2010

Passione per la programmazione

Programmare per passione

- inizio: BASIC del VIC-20
- C, C++ e Modula-2
- Ingegneria: Perl e Java :-(
- post laurea: Ada95 e Python
- per caso: Objective Caml, Javascript, Lua
- Scheme :-)

Programmare per lavoro

- embedded in C e desktop (QT) in C++
- Galialeo Avionica, Progetto Typhoon
- consulente in Comelz (Python e C++)
- PHP (**DISGUSTOSO!**), OCaml, Javascript, Python (Django)

Convinzioni personali (1)

Idee forti

- tipizzazione statica
- correttezza formale
- manutenibilità
- refactoring
- no guerre di religione!⇒ Vim rulez! :-)

Obiezioni

- importanza dei test
- codice generico
- complessità inutile
- REPL (Read-eval-print loop)

Convinzioni personali (2)

Cattive abitudini

- stato mutabile globale
- programmazione thread-oriented

Buone idee

- principio DRY (Don't Repeat Yourself)
- principio KISS (Keep It Simple, Stupid)

Esperto OCaml?

In OCaml ho scritto

- tool per applicazioni web
- giocattolino basato su Cairo
- Ex-nunc: un framework web
- un piccolo framework web MTV
- tool di stitema ad uso personale
- faccio parte del team di Batteries

Non sono un mago OCaml

- non conosco **per nulla** Camlp4
- non conosco **bene** ocamlbuild

Perle di saggezza

> This is the same argument you tend to hear for learning Latin. It won't get you a job, except perhaps as a classics professor, but it will improve your mind, and make you a better writer in languages you do want to use, like English.
>
> But wait a minute. This metaphor doesn't stretch that far. The reason Latin won't get you a job is that no one speaks it. If you write in Latin, no one can understand you. But Lisp is a computer language, and computers speak whatever language you, the programmer, tell them to.

Paul Graham, Beating The Averages

Programmazione Funzionale

Idee forti

- solo valori costanti
- le **funzioni** sono un buon modello
- **funzione** in senso **matematico**
- computazione $\Rightarrow$ applicazione funzionale
- strutture dati persistenti
- garbage collection
- no boilerplate! (HOF)
- Tail Call Optimization

Alcune conseguenze

- nessun problema sincronizzazione
- functional patterns
- TCO $\Rightarrow$ Continuation-passing style

Bonus slide (1)

Se puoi scrivere un Y-Combinator, il linguaggio è funzionale?

Y-Combinator in OCaml

```
type 'a mu = In of ('a mu → 'a)
let out (In x) = x

let y f = (fun x a → f (out x x) a)
          (In (fun x a → f (out x x) a))

let fac f = function 0 → 1 | n → n * f (n-1)
List.map (y fac) (range 10)

let fib f = function
              | 0 → 0
              | 1 → 1
              | n → f (n-1) + f (n-2)
List.map (y fib) (range 10)
```

Bonus slide (2)

In Python è possibile scrivere il Y-Combinator, senza difficoltà:

Y combinator in Python

```
Y = lambda f: (lambda x: x(x)) (lambda y: f(lambda
    *args: y(y)(*args)))

fac = lambda f: lambda n: (1 if n<2 else n*f(n-1))
[ Y(fac)(i) for i in range(10) ]

fib = lambda f: lambda n: 0 if n == 0 else (1 if n
    == 1 else f(n-1) + f(n-2))
[ Y(fib)(i) for i in range(10) ]
```

Tuttavia, Python non è un linguaggio funzionale!

Alcuni fatti

- Nome: Objective (come in Objective-C) Caml (come il cammello)
- pronuncia: oh-KAM-əl
- logo: il cammello, ovviamente!
- Sito http://caml.inria.fr/ocaml/index.en.html
- Autori: Xavier Leroy, Jérôme Vouillon, Damien Doligez, Didier Rémy e Jacques Garrigue (e molti altri)
- Mantenuto da: INRIA
- Modello di sviluppo: cattedrale (come Lua)
- Licenza: compilatore e tools: Q Public License 1.0
  libreria standard: LGPL2 + linking exception

Disponibilità

OCaml in Debian/Ubuntu

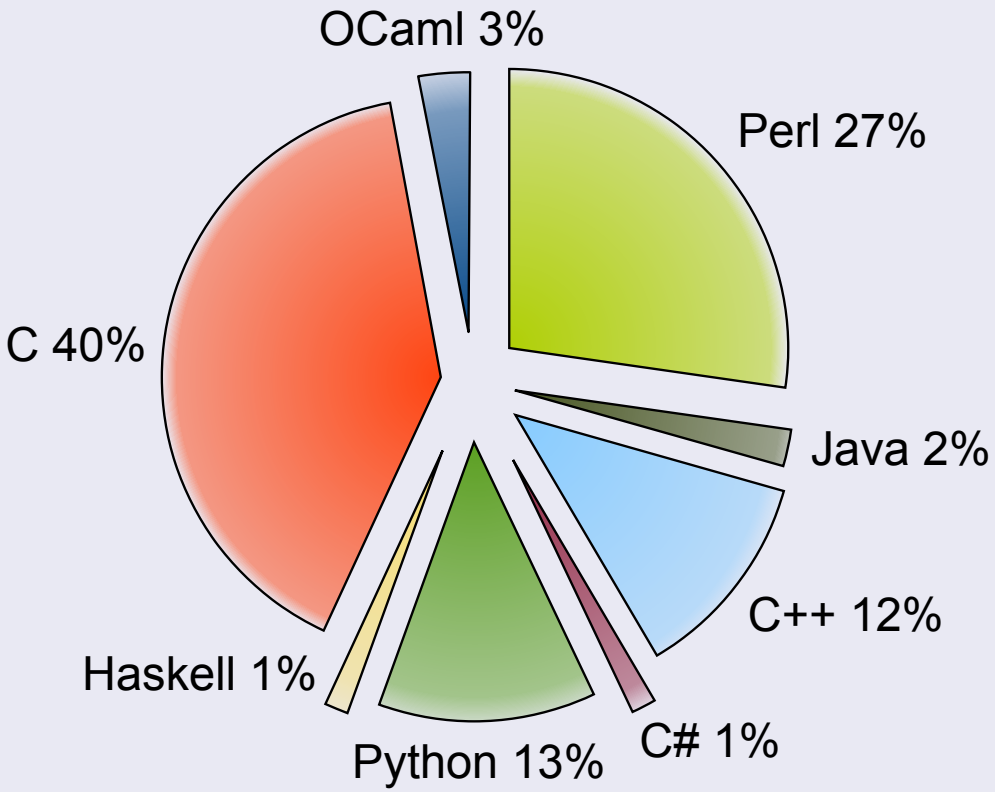
La presenza di OCaml in Debian/Ubuntu è assicurata dalla Debian OCaml Task Force, composta da membri attivi della comunità OCaml (Stefano Zacchiroli, Sylvain Le Gall. . .)

OCaml in RedHat/Fedora

La presenza in RedHat/Fedora è assicurata da Richard Jones, dipendente RedHat, anch'egli attivo nella comunità OCaml. Richard Jones è anche autore di tool ufficiali RedHat scritti in OCaml.

Diffusione

Diffusione in Debian/Ubuntu (tag “implemented-in”)



C: 2466
Perl: 1672
Python: 772
C++ 745
OCaml: 178
Java: 130
C#: 87
Haskell: 83

Panoramica

- efficienza
- pragmatismo (costrutti imperativi)
- linguaggio multi-paradigma (come Python)
- tipizzazione statica
- inferenza di tipo
- tipi aperti (ci torniamo)
- valutazione **strict** (**lazy** se occorre)
- programmazione “in the large”: moduli, tipi privati, abbreviazioni, funtori, etc. . .
- Camlp4 (Pre-Processor-Pretty-Printer)

Moduli e oggetti

Il module system

- elemento semantico
- **module type**, ovvero firme di moduli
- funtori (moduli parametrici)

L'object system

- oggetti immediati
- semantica molto ricca
- **ereditarietà** $\neq$ **sottotipo**

Premessa

Nessuna “batteria” inclusa.

Motivazioni

- Motivazione economica
- motivazione “purista”
- motivazione legale

Vantaggi

- libreria molto stabile, virtualmente bug-free
- manutenzione semplice da parte di INRIA

Svantaggi

- l’installazione **di fatto** non autocontentuta
- problemi sotto Windows (pochi)

Moduli standard

ABC della programmazione

- **Pervasives**: operatori e funzioni aritmetici e booleani, **compare**, utility minime, conversioni basilari, I/O basilare, stringhe di base.
- **Strutture dati**: Array, Buffer, Hashtbl, List, Map, Queue, Set, Lazy, Stack, String, Stream, Weak
- **Operazioni sui dati**: Char, Complex, Int32, Int64, Nativeint, Oo, Str
- **I/O evoluto**: Format, Printexc, Printf, Scanf, Marshal
- **Algoritmi (terra terra)**: Digest, Random, Sort

Moduli di supporto

- **Interazione col sistema**: Arg, Callback, Filename, Gc, Sys, Unix, Threads
- **Utility per linguaggi**: Genlex, Lexing, Parsing
- **Varie**: Num, Bigarray, LablTk, Graphics, Str, Dynlink

Strumenti per tutti i giorni (1)

Compilare

- **ocamlc**: byte code
- **ocamlopt**: compilatore nativo (**non** ottimizzante)
- **ocamldep**: calcolo delle dipendeze tra moduli
- **ocamlmklib**: creazione librerie OCaml/C
- **ocamlobjinfo**: dump delle informazioni in una compilation unit
- **ocamldoc**: compila la documentazione

Creare linguaggi

- **ocamllex**: generatore di lexer
- **ocamlyacc**: generatore di parser
- **CamlP4**: (Pre-Processor-Pretty-Printer) estensioni sintattiche

Strumenti per tutti i giorni (2)

Debugging e profiling

- **ocamldebug**: debugger interattivo
- **ocamlcp**: compilatore per profiling attivo
- **ocamlprof**: profiler per ocamlcp
- **profiling**: si può anche usare **gprof**

Compilazione semi automatica con ocamlbuild

- compilatore di progetti, “orchestra” la compilazione
- calcola dipendenze
- generalmente fa tutto da solo
- estensibile in OCaml
- è in beta e poco documentato :-(

Ambienti “integrati”

Emacs e Tuareg-mode (pacchetti .deb e .rpm)

- indentazione automatica (secondo le OCaml programming guidelines)
- permette di eseguire frammenti di codice
- permette di compilare, eseguire, saltare sugli errori
- C-c C-t $\Rightarrow$ mostra il tipo sotto il cursore

OcaIDE, ODT

- plugin di Eclipse
- ben integrati con Eclipse
- supporto (parziale) al completamento automatico e all’help contestuale
- suggeriscono il tipo delle espressioni

Nulla di confrontabile a Visual Studio + F#. Io uso GVim :-)

Fonti ufficiali

- Il manuale di riferimento del linguaggio
- mailing list “beginners”
- mailing list ufficiale
- manuale utente ocamlbuild
- wiki ocamlbuild
- lista di risorse su sito INRIA
- OCaml FAQ
- Making code run fast
- Writing efficient numerical code in Objective Caml

Fonti non ufficiali

Pubblicazioni

- Developing applications with Objective Caml
- The OCaml Journal (60£ per 6 mesi)
- Introduction to Objective Caml di Jason Hickey
- OCaml for Scientists di Jon Harrop

Siti Internet

- The OCaml Tutorial
- PLEAC-Objective CAML
- Planet OCaml
- I miei link su Delicious

Espressioni costanti di tipi nativi

- interi (int) a 31 (63) bit con segno
- floating point (float, IEEE 754)
  attenzione: non c'è overloading, quindi:
  1 + 1, ma 3.14 +. 2.72
  1 + 3.14 è un errore!
- caratteri (char): 'a', 'b', ...
- stringhe (string): "foo", "pippo"
- bool: **true** e **false**; operatori: &&, ||, not
  = uguaglianza strutturale (negazione: <>)
  == uguaglianza fisica (negazione: !=)
- unit: tipo con un unico valore, (), usato per funzioni imperative

I tipi strutturati più semplici

Prodotto cartesiamo (tuple)

- le tuple si creano come in Python:

```
# (65, 'B', "ascii");;
- : int * char * string = 65, 'B', "ascii"
```

- funzioni di proiezione: fst e snd (per le coppie)

```
# fst;;
- : 'a * 'b → 'a = <fun>
# fst ("Foo", 42);;
- : string = "Foo"
```

La lista

- si indicano con le [] e sono polimorfe
- [1; 2; 3] ⇒ int list ; [1; "foo"; 3] ⇒ errore!;
- 1::2::3::[] ⇒ [1; 2; 3]; [1; 2] @ [3] ⇒ [1; 2; 3]

Pausa: la struttura del programma

- modello computazionale: $\lambda-$calcolo (Alonzo Church, 1932)
- l'espressione λ è una funzione, di un solo parametro
- il programma è composto di espressioni che vengono valutate
- $\lambda x.3x$ in Python si scrive: **lambda** x: 3 $*$ x
- le strutture di controllo (if, while, for, try...with) sono espressioni

if e let

Struttura di controllo condizionale

- **if** expr1 **then** expr2 **else** expr3
- **if** 3 = 4 **then** 0 **else** 4 vale 4 (intero)
- **if** 3 = 4 **then** "0" **else** "4" vale 4 (stringa)
- **if** 3 = 4 **then** 0 **else** "4" $\Rightarrow$ errore di tipo!
- (**if** 3 = 5 **then** 8 **else** 10) + 5 vale 15

Dichiarazione di valori, let binding

- dichiarazione globale: **let** name = expr;;
- dichiarazione locale: **let** name = expr1 **in** expr2

```
# let foo =
    let bar = 21 and baz = 2 in
      (bar*baz);;
val foo : int = 42
```

Espressioni funzionali e funzioni (1)

- l'espressione λ si scrive **function** x $\rightarrow$x$*$x
- applicazione funzionale:

```
# (function x → x * x) 5;;
- : int = 25
```

- funzione di più parametri: **function** x $\rightarrow$**function** y $\rightarrow$x$*$y
- applicazione parziale di una λ:

```
# (function x → function y → x*y) 5;;
- : int → int = <fun>
```

- sintassi alternativa per funzioni di arità *n*:
  fun $p_1 \ldots p_n \rightarrow$expr
- funzione con nome:
  let prodotto x y = x$*$y;;
- operatori infissi (notare argomento tupla e gli spazi):
  let (**) (x1, y1) (x2, y2) = x1 *. x2 +. y1 *. y2;;

Espressioni funzionali e funzioni (2)

- definizione ricorsiva: **let rec** *name* $p_1 \ldots p_n$ = *expr*

```
# let rec fattoriale n =
    if n = 0 then 1
    else n*(fattoriale (n - 1));;
val fattoriale : int -> int = <fun>
```

- polimorfismo parametrico: se le funzioni lavorano su strutture, il tipo è una variabile libera

```
# let make_pair a b = (a, b);;
val make_pair : 'a -> 'b -> 'a * 'b = <fun>

# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# let ( |> ) x f = f x;;
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>
```

- **let** res = f1 (f2 (f3 x)) $\Rightarrow$ **let** res = x |> f3 |> f2 |> f1

Higher order functions (HOF)

Permettono di combinare tra loro funzioni per crearne di nuove

```
# let ( |- ) f g x = g (f x);;
val ( |- ) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c =
    <fun>

# let ( / ) p1 p2 =
    join_path [p1; p2] |>
      (split_path |- norm_path |- join_path);;
val ( / ) : string -> string -> string = <fun>

# "/" / "home" / "paolo" / "bin" / ".." / "doc";;
- : string = "/home/paolo/doc"
```

Comporre funzioni secondo lo schema bottom-up. Vedi prefazione di "On LISP" di Paul Graham.

Concetti introduttivi

- il **tipo** è un cencetto centrale in OCaml
- costruire astrazioni (Java, C++ ma anche Python)
- verificare invarianti a compile time!

Caratteristiche di un nuovo tipo

- tipi “somma” e tipi “prodotto” (varianti vs. tuple e record)
- tipi “chiusi” e tipi “aperti” (record e varianti monomorfici vs. oggetti e varianti polimorfici)
- tipi “costanti” e tipi “parametrici” (uso di **type variables**, con applicazioni anche avanzate come i **phantom types**)

Type expressions

- **type** *name* = *typedef* ;;
- ricorsivi: **type** $name_1$ = $typedef_1$ **and** $name_2$ = $typedef_2$;;
- parametrico: **type** ’a *name* = *typedef* ;;
- più parametri: **type** (’a_1,. . .,’a_n) *name* = *typedef* ;;

Record (prodotto chiuso di tipi)

Sintassi: **type** *name* = { $name_1 : t_1; \ldots; name_n : t_n$ };;

Esempi

- molto semplice, stile **struct** in C:

```
type p2d = { x : float; y : float; }
type x2d = { a : p2d; b : p2d; c : p2d; }
let origine = { x = 0.0; y = 0.0 }
```

- uso semplice dei parametri:

```
type 'a http_session = {
    key : string;
    data : 'a;
    ctime : float;
    mtime : float;
    expire : float option;
  }
```

Tipi somma (varianti monomorfici, somma chiusa di tipi)

Tuple e **record** $\Rightarrow$ prodotto cartesiano di tipi
Tipi somma $\Rightarrow$ unione insiemistica di tipi (C union, più o meno...)

Sintassi generica

type *name* =
  | $Name_i$ ...
  | $Name_j$ **of** t_j ...
  | $Name_k$ **of** t_k * ... * t_l ...;;

Esempio

```
type seme = | Cuori | Fiori | Picche | Quadri;;
type figura = | Fante | Donna | Re;;
type carta = | Figura of figura * seme | Carta of
    int * seme | Jolly;;
# let due_di_picche = Carta (2, Picche);;
val due_di_picche : carta = Carta (2, Picche)
```

Pattern Matching

Riconoscere pattern nella struttura dei dati

Esempio

```
# let bool_and a b = match a, b with
    | (true, true) → true
    | (true, false) → false
    | (false, true) → false
    | (false, false) → false;;
val bool_and : bool → bool → bool = <fun>
```

Wild card "_"

```
# let bool_and a b = match a, b with
    | (true, true) → true
    | _ → false;;
val bool_and : bool → bool → bool = <fun>
```

Esempio: albero binario

```
# type 'a bin_tree =
    | Empty
    | Node of 'a bin_tree * 'a * 'a bin_tree;;

let rec list_of_tree tree = match tree with
  | Empty → []
  | Node(lb, r, rb) →
      (list_of_tree lb) @ (r::(list_of_tree rb));;
val list_of_tree : 'a bin_tree → 'a list = <fun>

# let rec insert x tree = match tree with
    | Empty → Node(Empty, x, Empty)
    | Node(lb, r, rb) →
      if x < r then Node(insert x lb, r, rb)
               else Node(lb, r, insert x rb);;
val insert : 'a → 'a bin_tree → 'a bin_tree = <fun>
```

Tipi somma (varianti polimorfici, somma aperta di tipi)

Come i tipi somma, ma con possibilità di raffinamenti successivi

Sintassi generica

```
type name = [ `Name_i ...
   | `Name_j of t_j ...
   | `Name_k of t_k * ... * t_l ... ];;
```

Esempio

```
# `Open;;
- : [> `Open] = `Open
# let print_lock lck =
    match lck with
      | `Open → print "The lock is open"
      | `Close → print "The lock is closed";;
val print_lock : [< `Close | `Open] → unit = <fun>
```

Oggetti (prodotto aperto di tipi) (1)

Object system

- prodotto **aperto**
- ereditarietà $\neq$ sottotipo
- meccanismi di visibilità: quelli del linguaggio

Semplice esempio

```
# class p2d x_init y_init =
  object
    val mutable x = x_init
    val mutable y = y_init
    method get_x = x
    method get_y = y
    method move (dx,dy) =
      x <- x +. dx; y <- y +. dy
  end;;
```

Oggetti (prodotto aperto di tipi) (2)

La risposta del "toplevel"

```
class p2d : float → float →
  object
    val mutable x : float
    val mutable y : float
    method get_x : float
    method get_y : float
    method move : float * float → unit
  end

# let origin = new p2d 0.0 0.0;;
val origin : p2d = <obj>
```

Interfaccia di una class

In OCaml ciò che conta è l'interfaccia inferita, non le "parentele" tra classi

class type

```
# class type point_type =
  object
    method get_x : float
    method get_y : float
    method move : float * float → unit
  end;;

# let print_point p =
    Printf.printf "(%f,%f)\n" p#get_x p#get_y;;
val print_point :
    < get_x : float; get_y : float; .. > → unit =
      <fun>
```

Programmazione imperativa

- programmazione funzionale vincente su manipolazione dati
- performance $\Rightarrow$ feature imperative
- variabili (sì, esistono!)
- strutture mutabili: array, hash table, weak reference, code
- strutture di controllo: cicli **for** e **while**

```
let esco = ref false;;
while not !esco do
  print_string "Un altro giro? (y/n) ";
  let str = read_line () in
    if str.[0] = 'n' then
      esco := true
done;;
```

Applicativi (famosi) in OCaml

- FFTW: Trasformata di Fourier più veloce del West!
- Unison: sincronizzazione bidirezionale
- MIdonkey
- The haXe compiler
- Frama-c: analisi statica di sorgenti C
- Coccinelle: patch semantiche, usato nel kernel di Linux
- Hevea: conversione LaTeX in HTML
- Polygen: genera frasi casuali da una grammatica, esempio
- MTASC: il primo compilatore ActionScript 2 open source al mondo

Librerie essenziali

- Findlib: tool per installazione
- Camomile: libreria Unicode
- Batteries: estensione stdlib, INRIA-blessed
- Core: estensione stdlib, molto vasta
- Ocamlnet: toolkit di rete molto completo, un vero framework!
- PXP: toolkit XML, validazione da DTD
- JSON-static, Sexplib,. . . serializzazione type safe
- compressori: ZIP, BZIP, GZIP. . .
- binding ai principali DB
- OUnit: unit test alla JUnit
- Lwt: thread cooperativi, “monade della continuazione”
- niente link: tutti pacchettizzati Debian/Ubuntu/RedHat!

Cataloghi software

- The OCaml hump
- OCamlCore forge
- la pagina di Xavier Leroy
- la pagina di Markus Mottl
- la pagina di Alain Frisch
- la pagina di Jean-Christophe Filliâtre
- la pagina di Martin Jambon

Qualcuno usa davvero?

- Jane Street Capital: analisi finanziarie a Wall Street. Solo OCaml, 1×10^6 LOC in produzione!
- Flying Frog Consultancy Ltd.: consulenze scientifiche, libri e pubblicazioni, ricerca e sviluppo
- Skydeck: servizi telefonici
- MyLife: motore di ricerca e network sociale, tipo LinkedIn
- MLstate: applicazioni SaaS, autore di OPA, un framework web completo
- Merjis Ltd.: analisi business e ottimizzazione del ritorno sull'investimento.
- elenco per area geografica

Conclusione

Punti di forza

- linguaggio sintetico ed espressivo
- tool di grande qualità, performance
- forte comunità, elevatissima qualità del codice

Debolezze

- documentazione (a volte) immatura
- comunità “frammentata”, manca sito di riferimento
- manca un toolkit grafico decente, no QT :-(

Grazie per la pazienza

Queste slide sono distribuite sotto la (solita) licenza Creatice Commons (Attribuzione, condividi allo stesso modo).
Se avete altre domande o cercate aiuto (su OCaml :-)) scrivetemi a questo indirizzo: **p.donadeo@4sigma.it**