OCaml

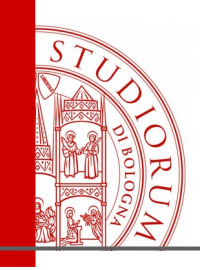
IL PRESENTE MATERIALE È RISERVATO AL PERSONALE DELL'UNIVERSITÀ DI BOLOGNA E NON PUÒ ESSERE UTILIZZATO AI TERMINI DI LEGGE DA ALTRE PERSONE O PER FINI NON ISTITUZIONALI

Da ML a OCaml

- ML (Meta-Language), 1973, Robin Milner et alt.
  - Linguaggio per l'implementazione del dimostratore di teoremi LCF
  - Forte enfasi sul calcolo simbolico e sui tipi di dato astratti
- Standard ML (1990, revised 1997)
  - New Jersey ML, Moscow ML, Mlton, Poly/ML, SML.NET, Alice, ...
- @INRIA: Caml Heavy (in lisp), poi Caml Light (1995 circa), poi Ocaml (1996-...)
- F# = OCaml in .NET + asynchronous/parallel/actor + ...

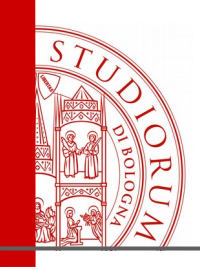
OCaml buzzwords

- Functional (higher order, impure)
- Eager + lazy thunks
- Multi-paradigm: mutable heap cells, objects, etc.
- Efficient (compiler and generated code)
- Bytecode, native code, javascript, java bytecode, ...
- Powerful type system supporting type inference
- Safety
- Sophisticated module system
- Widely used
- Symbolic computation (algebraic data types, ...)



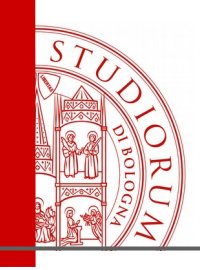
Ocaml: the good

- Concisione, produttività, correttezza del codice
- Ampia scelta di costrutti non standard (varianti polimorfe, oggetti funzionali, funtori, moduli di prima classe, tipi di dati algebrici generalizzati, …)
- IL linguaggio per implementare compilatori, type-checker, etc. in maniera efficiente



Ocaml: the bad

- No SMP support, bad with concurrency
- Il linguaggio è cresciuto molto e caoticamente
  - molte feature, alcune poco usate
  - sintassi a volte terribile
- Ha a lungo sofferto per lo sviluppo in ambiente accademico (modello a cattedrale, bazar tardivo e poco fruttuoso)
  - librerie carenti, copertura parziale, librerie multiple, interoperabilità, ...
- Sorpasso da parte di Haskell nel mondo accademico e come numero/qualità di librerie



Il nucleo funzionale

OCaml	Erlang
fun x → fun y → x + y fun x y → x + y	fun (X,Y) → X + Y
f 3 4 (ma anche f 3)	f(3,4)
f	fun f/2
varname	Varname
Empty Node(3,4)	empty {node, 3, 4}
let pattern = expr in expr2	pattern = expr, expr2

match expr with
  Node X → Some X
| Empty → None
	case expr of
	  {node, X} → {some, X} ;
	  empty → none
	end

Il nucleo funzionale

OCaml	Erlang
let g x = x * 2 ;; let rec f x y = f x 0 + y	G(X) → X * 2. F(X,Y) → F(X,0) + Y.
let g x y = x + y in g 2 2	G = fun (X,Y) → X + Y end, G(2,2)
let f = function None → 0 | Some x → x	F none → 0 ; F {some, X} → X.
Ok(x) as y when x < 0 → Good(y,x)	Y = { ok, X } when X < 0 → {good, Y, X}
try throw (Foo 3) with Foo X → X	try throw({foo, 3}) catch {foo, X} → X

Il nucleo funzionale

… e così via

Rispetto a Erlang:

- eccezioni e costruttori (atomi e tuple etichettate) vanno dichiarati prima
- tutte le funzioni sono tipate (ma il compilatore inferisce il tipo più generale)
- no limitazioni arbitrarie di Erlang sulle guardie
- tipi di dati predefiniti e/o loro sintassi differente

Concorrenza, distribuzione

Nessun supporto linguistico alla concorrenza

Hindley-Milner type inference

- no ad-hoc polymorphism (overloading)

  1 + 2 1.0 +. 2.0 “hello” ^ “world”

- vantaggio: il tipo delle variabili viene inferito dal loro uso

  let f x y = 3.14 *. x *. float_of_int y

  val f : float → int → float

  let f g x = g (g x) + 1

  val f : (int → int) → int → int

Hindley-Milner type inference

- il compilatore ritorna un errore di tipo quando le variabili sono usate in maniera incoerente

let f g = g 3, g “ciao”

Error: This expression has type string but an expression was expected of type int

- in caso di errore il tipo inferito dal compilatore può essere sorprendente e inserire tipi espliciti aiuta

Hindley-Milner type inference

- in assenza di vincoli o quando i vincoli sono laschi vengono generati schemi di tipi, ovvero i tipi iniziano con quantificazioni universali su variabili di tipo indicate con “lettere greche” ‘a, ‘b, ‘c, …

let f g (x,y) = g x, (y,y)

val f : (‘a → ‘b) → ‘a * ‘c → ‘b * (‘c * ‘c)

Hindley-Milner type inference

- i tipi possono essere **ristretti** esplicitamente
  * per documentazione
  * per trovare errori
  * quando estensioni al sistema di tipi lo richiedono

let f g x y = g x, y
val f : (‘a → ‘b) → ‘a → ‘c → ‘b * ‘c

let f (g : ‘a → int) (x : int) y = g x, (y : ‘a * ‘d)
val f: (int → int) → int → int * ‘d → int * (int * ‘d)

Hindley-Milner type inference

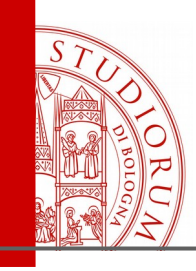
- uno schema di tipi viene generato solamente per le definizioni introdotte da un let-in (globali o locali)
- in particolare: le funzioni hanno tipo polimorfo, ma i parametri sono sempre monomorfi

let f x = x **in** f 3, f “ciao”
val - : int * string

let f g = g 3, g “ciao”
Error: This expression has type string but an expression was expected of type int

Hindley-Milner type inference

- il sistema di tipi di Hindley-Milner è un fragile gioiello
    * è decidibile
      (esiste un algoritmo che determina se un programma è ben tipato)
    * ammette most general unifier
      (il programma calcola un tipo di cui tutti gli altri sono istanza)
    * iper-conciso: no tipaggio di parametri/valori di ritorno, no istanzazioni esplicite alla

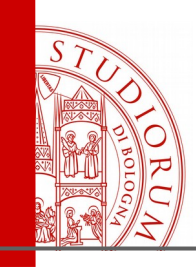


Hindley-Milner type inference

A run-time:

* nessuna rappresentazione dei tipi (come C!) quindi nessun overhead
* stessa rappresentazione dei dati di Erlang (il codice è naturalmente uniformemente polimorfo)

Polimorfismo alla Hindley-Milner chiamato anche polimorfismo uniforme o generico.



Hindley-Milner type inference

Critica: i tipi sono la prima forma di documentazione del codice!

- un editor evoluto può visualizzare i tipi di tutte le espressioni
- le funzioni esportate da un modulo (vedi dopo) sono esplicitamente tipate nella signature del modulo
- in caso di funzioni dal tipo astruso, l'utente può restringere il tipo esplicitamente

Abbreviazione di tipi

Esempi:

type currency = int
type ‘a set = ‘a list (* param. a sinistra *)
type (‘a,’b) mappa = (‘a * ‘b) list (* tipo con due param. *)

Utili per abbreviare tipi, ma l’uguaglianza di tipi è **strutturale**

let f (g: int → int) (x: currency) = g x
val f : (int → int) → currencty → int

ma anche

val f : (currency → int) → int → currency

Tipi di dati algebrici

Esempi:

```
type ('a, 'b) tree =
    Leaf of 'b
  | Node of ('a * 'b) tree * 'a * ('a, 'b) tree
```

- tree è un tipo di dato algebrico parametrico su 'a e 'b
- Leaf e Node sono costruttori rispettivamente unario e ternario
- come { leaf, X } e { node, T1, K, T2 } in Erlang

Tipi di dati algebrici :(

- i costruttori NON possono essere applicati parzialmente

```
Node;;
```

```
Error: The constructor Node expects 3
   argument(s), but is applied here to 0
   argument(s)
```

- K1 of ‘a * ‘b è binario
  K2 of (‘a * ‘b) è unario e prende una coppia

Tipi di dati algebrici :)

- il compilatore controlla che tutti i pattern match siano **esaustivi** e **privi di duplicati**

```
let f = function
      Node(t1,k,t2) → k
    | Node(t1,k,t2) when k < 0 → 0
```

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Leaf _
Warning 11: this match case is unused.
```

Tipi di dati algebrici :)

- in caso di guardie il problema del controllo dell'esaustività e dell'assenza duplicati diventa indecidibile ==> possibile falsi positivi

```
let f = function
    x when x < 0 → 0
  | x when x >= 0 → 1
```

Warning 8: this pattern-matching is not exhaustive.
All clauses in this pattern-matching are guarded.

Tipi di dati algebrici :)

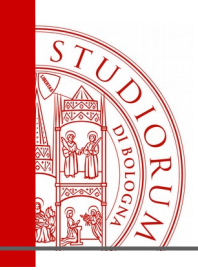
- il controllo di esaustività è grandioso
  * nessun caso dimenticato per sbaglio
    (se si evitano pattern catch-all tipo _ → …)
  * in caso di aggiunta di un costruttore al tipo
    il compilatore enumera tutti i punti dove
    occorre aggiornare il codice…
  * … e l'editor ci porta automaticamente al punto
    dove fare la modifica
- confronta: funzioni come metodi, costruttori come classi, aggiunta di un nuovo metodo/funzione o di una nuova classe/costruttore

Tipi di dati algebrici vs ??

- ADT = closed world assumption
  - Tutte le possibili forme sono note
    Es.: protocolli, strutture dati, …
  - I cambiamenti nelle possibili forme sono rari/inaspettati e richiedono ripensamento generale
  - Mantenere i giusti invarianti è fondamentale
  - Mantenere compatte le definizioni di funzione ne agevola la comparensione

Tipi di dati algebrici vs ??

- ?objects? = open world assumption
  - Tutte le possibili forme sono note
    Es.: protocolli, strutture dati, …
  - I cambiamenti nelle possibili forme sono rari/inaspettati e richiedono ripensamento generale
  - Mantenere i giusti invarianti è fondamentale
  - Mantenere compatte le definizioni di funzione ne agevola la comparensione



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Claudio Sacerdoti Coen

Dipartimento di Informatica: Scienza e Ingegneria (DISI)

claudio.sacerdoticoen@unibo.it

www.unibo.it