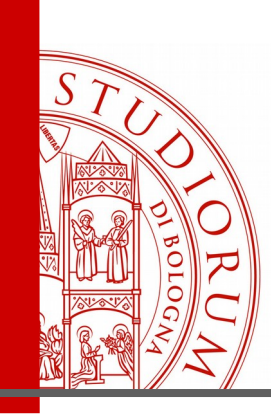
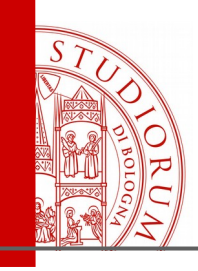


Algebraic Data Types
e
Meccanismi per Evitare null Pointers



Algebraic Data Types (ADT)

Un concetto emerso:

- Introdotti in ML (fine anni 70)
- Linguaggi funzionali: ML, Standard ML, OCaml, Miranda (1985), Haskell (1987), F#, Elm (2012), Idris, Racket, …
- Altri linguaggi: Scala (2003), Kotlin (2011), Rust (2010), Swift (2017), …
- Assenti in Go….

ADT: perché?

Un ADT è

- un'unione disgiunta (coprodotto), taggata da atomi/costruttori, di tuple/record (prodotti) di tipi
- può essere mutualmente ricorsivo

In matematica

- prodotti/coprodotti sono nozioni fondamentali in algebra, teoria delle categorie, … con ricca teoria matematica
- coprodotti di prodotti sono una tipica forma canonica completa (i prodotti distribuiscono sui coprodotti)

ADT: implementazione

Running example:

```
type tagged_union =
    Ka of A
  | …
  | Kz of Z

match Ka a with
    Ka a => “a”
 | …
 |  Kz z => “z”
```

ADT: implementazione

In set theory: le unioni disgiunte si riducono alle unioni fra insiemi taggati

$$A + \ldots + Z == \{1\} \times A \cup \ldots \cup \{n\} \times Z$$

```
typedef struct {
        int tag ;
        union {
            A a;
            …
            Z z;
        } f
} tagged_union;
```

```
A a;
C c;
tagged_union u = {3, .c = c};   // creazione

u.tag = 1; u.f.a;               // assegnamento

switch (u.tag) {                // pattern matching
    case 1:
        A a = u.f.a ;
        ….
        break;
}
```

ADT: implementazione in Erlang

```
case { ka, a } of
    { ka, A } => “a” ;
    …
    { kz, Z} => “z”
end
```

ADT: encoding via oggetti + chiusure

```
interface tagged_union {
   T match<T> (fa: A → T, …, fz: Z → T);
};

class IA(A x) {
     private A a = x
     T match<T> (fa: A → T, …, fz: Z → T) {
          return fa(a);
      }
}

new IA(a).match<String>((A a) { return “a”;}, …, (Z z) { return “z”; })
```

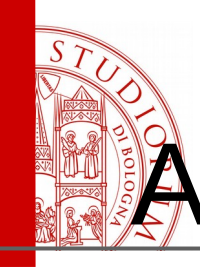
ADT: encoding via oggetti + visitor pattern

```
interface Visitor<T> {
      T visit_a (A a);
      ....
      T visit_z (Z z);
   }
```

```
interface tagged_union {
      T match<T> (Visitor<T> visitor);
};
```

```
class IA(A x) {
    private A a = x
    T match<T> (Visitor<T> visitor) {
        return visitor.visit_a(a);
    }
}
```

```
new IA(a).match<String>(object implements Visitor<String> {
   String visit_a (A a) { return "a";};
   …
   String visit_z (Z z) { return "z"; }});
```



ADT: confronto implementazioni

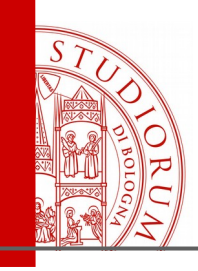
- No run-time overhead in C/OCaml/Haskell/Erlang/..
  Chiamate di metodi via dynamic dispatching quando implementate via oggetti

- Closed world assumption (C, Ocaml, Haskell) vs open world (via oggetti)
  (con visitor = via di mezzo)
- Open world + sealing = closed world in OO

Sealing

- In OO sealing un'interfaccia significa impedire di aggiungere nuove implementazioni all'interfaccia.

- Nuove keyword (il linguaggio si complica) vs vari encoding (il codice si complica)



Sealing in Java

```
public abstract class tagged_union {
    private tagged_union() {};  // private: subclasses disabled
                                // except inner classes

    public static final class Aa extends tagged_union {
        private final A a;
        public Aa(A x) { a = x; }  // public in inner class
    }
}
```

+ visitor da aggiungere

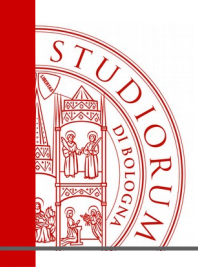
ADTs in Scala

```
sealed trait Tagged_union
case class Aa(a: A) extends Tagged_union;
…
case class Zz(z : Z) extends Tagged_union;

val u: Tagged_union = Aa(a)
def analyse(u : Tagged_union) : String =
    u match {
       case Aa(a) => “a”
       …
       case Zz(z) => “z”
    }
```

ADTs in Scala

- traits are like interfaces
- sealed: disallow further extensions
- case class: syntactic sugar to define fields + constructor that set the fields + pattern matching
- exhaustivity checks implemented
- everything is compiled onto classes



ADT in Haskell

```
data Tagged_union =
    Ka a | … | Kz z

case Ka z of
    Ka a → “a”
    …
    Kz z → “z”
```

ADT in Kotlin

ADT =
- sealing +
- case statements over class type +
- implicit type cast during case statement +
- exhaustivity check

Kotlin è linguaggio ad oggetti ma linguisticamente e come implementazione è più simile alla soluzione C/OCaml (ogni oggetto punta alla sua classe che gioca il ruolo del tag in C)

ADT in Kotlin

```
sealed class Tagged_union {
   data class Aa(val a : A) : Tagged_union()
    …
   data class Zz(val z : Z) : Tagged_union()
}

fun process(u: Tagged_union) {
    return when (u) {  // qui u ha tipo Tagged_union
       is Aa → “a”       // qui u ha tipo Aa; posso fare u.a
       …
       is Zz → “z”       // qui z ha tipo Zz; posso fare u.z
    }
}
```

ADT vs Enumerated Types

- molti linguaggi imperativi (Pascal, Ada, Go, C, C#, ...) hanno la nozione di enumerated type

- in Pascal:

```
type
   cardsuit = (clubs, diamonds, hearts, spades);
var
   trump : cardsuit
```

- a ogni entry è associato univocamente un intero (come gli atomi in Erlang)

- in genere associati a case/switch statement

- Enumerated values = atomi, es. clubs
  ADT values = tuple taggate con un atomo es. { ka, a }

ADT in Swift

```
enum Tagged_union {                  // enumerated type that really is an ADT
        case aa (a : A)              // i costruttori
        …
        case zz(z : Z)

    func elabora() {                 // un metodo...
      switch self {
        case let .aa(a):
          return “a”
        ….
        case let .zz(z):
          return “z”
      }
    }
}

let u : Tagged_union = .cc(c)
```

ADT in Rust

```
enum Tagged_union {
        Aa(a),
        …
        Az(z)
}

let u = Tagged_union::Aa(a)

match u {
     Tagged_union::Aa(a) => “a”
      …
      Tagged_union::Zz(z) => “z”
}
```

Riassunto

- gli ADT sono ormai un costrutto mainstream nella maggior parte dei linguaggi
- prima visione: ADT come tipi enumerati che trasportano valori, sintassi concisa, semantica ovvia, implementazione semplice/efficiente
- seconda visione: ADT come classi con particolari proprietà, sintassi e semantiche barocche, implementazioni inefficienti

Null values

In C/Java/…:

- ogni tipo ha un valore di default
  0 per int, NULL per pointer, …
- variabili non inizializzate possono prendere tale valore (o ricevere un valore casuale: dipende dal linguaggio)
- accedere al valore non inizializzato è quasi sempre errato
- NULL pointer dereferencing è uno dei bug più tipici dei programmi C (ma anche Java, etc.)

Optional values

Alternativa via ADT:

OCaml:

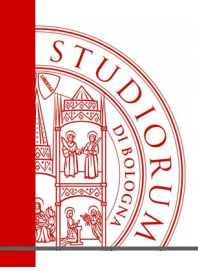
type ‘a option = None | Some **of** ‘a

Rust:

enum Option<T> { None, Some(T) }

Haskell:

data Maybe a = Nothing | Just a



Working with optional values

```
let read filename =                  // optional file
 let stream =                        // stream or error
    match filename with
       None => Some stdin
     | Some n => open_in n in
 match stream with                   // data or error
      None => None
   | Some str => read 100 str
```

Working with optional values

Pros:

- the compiler forces the user to consider the bad cases
- changing a type from/to an optional one will trigger error message where changes need to be taken in account
- (‘a option) option != ‘a option e può codificare informazione differente (e.s. un valore opzionale o un errore)
- invarianti diventano espliciti e documentati

Working with optional values

Quando il programmatore conosce un invariante che dice che un valore opzionale non può essere null:

```
let p' =
 match p with
    Some x → x
  | None → assert false in (* p cannot be null
                              because … *)
```

Working with optional values

Cons:

- a run-time un valore opzionale occupa più spazio (per il tag) e può essere boxed (p.e. in Ocaml/Haskell/…)
- accesso al valore opzionale più lento
- codice più verboso

  (ma può essere mitigato)

Mitigazione della verbosità

- ‘a option è un container per valori di tipo ‘a
- tipiche operazioni sui container:

  val map : (‘a → ‘b) → ‘a option → ‘b option

  val iter : (‘a → unit) → ‘a option → unit

  val bind :

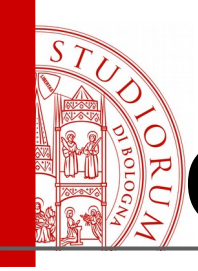
  (‘a → ‘b option) → ‘a option → ‘b option

  val default : ‘a option → ‘a → ‘a

  ...
- Some + bind forniscono a option la struttura di una monade: introdurremo le monadi più avanti

Working with optional values

```
(* takes an optional filename in input;
   The Some of the integer read from the file or
    None in case of error *)
(* val read : string option → int option *)
let read filename =
  let filename' = default filename "/tmp/log" in
  let chan = open_in filename' in   (* open_in : string → chan option*)
  let str = bind (read 1) chan in    (* read : chan → string option *)
  iter close chan ;                  (* close : chan → unit *)
  bind parse_int str                 (* parse_int : string → int option *)
```



Optional values vs exceptions

- Optional values are similar to exceptions
- None corresponds to throwing an exception
- map/bind/… correspond to exception propagation
- default/match to catching exceptions

Pros:

- no uncatched exceptions, programmer forced to reason on errors

Cons:

- still more verbose, more memory used, (much) slower at run-time, puts pressure on garbage collection

Nullable references in Kotlin

- Se T è un reference type, T? è un nullable reference type
- T e T? hanno la stessa rappresentazione a run-time (no overhead, T?? non ammesso)
- Il compilatore inibisce l'uso di T? come valore, l'assegnamento di null a T

```
val a : String = "ciao"  // mandatory assignment
a = null                 // compilation error
fun (b: String?) : Char { return x[1] } // comp. error
```

Nullable references in Kotlin

- Kotlin promotes a T? to a T after an explicit check that T is not null

```
fun foo(x : String?) : Char {
  if (x != null) return x[1]  // here x : String
  else return 'a'             // here x : String?
}
```

Nullable references in Kotlin

- se o : T?, o?.f è equivalente a

  if (o != null) o.f **else** null

- detto altrimenti, o?f è la (map f o) in OCaml
- o?.f?.g(x)?.h(y) funziona a dovere
- o?. let { println(it) } esegue la chiusura { .. }

  solo se o è non null; o promosso a T viene passato alla chiusura (di cui it è il parametro implicito)
- detto altrimenti, o?. let f è la (iter f o) di OCaml

Nullable references in Kotlin

- se o : T?, o ?: d restituisce o se o != null e d altrimenti
- in altre parole, o ?: d è (default o d) in Ocaml
- esempio idiomatico interessante:

```
fun (n : Node) {
    val p = n.getParent() ?: throw CalledOnRoot()
    ...
}
```

Nullable references in Kotlin

- se o : T?, o!! solleva un'eccezione se o è null, altrimenti ritorna o promosso a T
- casts:
  - T y = o as T
    solleva un'eccezione se o non è un T oppure o è null
  - T? y = o as T?
    solleva un'eccezione se o non è un T
  - T? y = o as? T
    ritorna sempre un T? che è null se o non è un T o è null

Casts (or lack of) in Kotlin

- Come nel caso della nullità, il compilatore di Kotlin tiene conto dei test di tipo per promuovere automaticamente gli oggetti

```
fun test(x: Any) {
    if (x is String) print(x.length) // x : String qui
    else …                           //  x : Any qui
}
```

Casts (or lack of) in Kotlin

Altro esempio che usa l'analisi per casi:

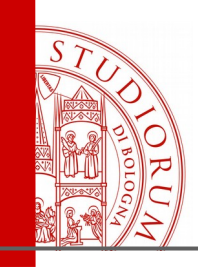
```
when(x) {
        is Int → print(x + 1)
        is String → print(x.length + 1)
}
```

Le colpe dei padri...

- Kotlin è implementato sulla JVM e rimane 100% compatibile con le librerie Java
- in Java i generics furono aggiunti in seguito con una serie di hacks…
- in particolare a run-time le informazioni di tipo sono erased: List<String> e List<List<String>> sono indistinguibili e rappresentati come List<*>
- questo inibisce alcuni dei meccanismi visti prima

Riassumendo

- nel 2018 non ci sono più scusanti per linguaggi con null pointer exception
- ADT option permettono di rappresentare altro oltre alla nullità dei puntatori, ma sono pesanti quando usati a tale scopo
- le null references di Kotlin sono linguisticamente molto vicine all'ADT option, ma con un uso specializzato e non introducono alcun overhead a run-time
- Kotlin traccia le condizioni per prevenire casts e null-pointer checks in una maniera decisamente interessante



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Claudio Sacerdoti Coen

Dipartimento di Informatica: Scienza e Ingegneria (DISI)

claudio.sacerdoticoen@unibo.it

www.unibo.it