
PROGRAMMAZIONE 2

18. Tipi di dato, implementazione

A cosa servono?

- **Livello di progetto:** organizzano l'informazione
 - tipi diversi per concetti diversi
 - meccanismi espliciti dei linguaggi per l'**astrazione sui dati** (ad esempio **classi e oggetti**)
- **Livello di programma:** identificano e prevengono errori
 - i tipi sono controllabili automaticamente
 - costituiscono un “controllo dimensionale”
 - l'espressione **3+“pippo”** deve essere sbagliata
- **Livello di implementazione:** permettono alcune ottimizzazioni
 - **bool** richiede meno **bit** di **real**
 - strumenti per fornire informazioni necessarie alla macchina astratta per allocare spazio di memoria

Dati: classificazione

- **Denotabili**: se possono essere associati ad un **nome**
⌘ `let plusone = (fun x -> x + 1);;`
- **Esprimibili**: se possono essere il risultato della valutazione di **una espressione complessa** (diversa dal semplice nome)
⌘ `let pick_one n = if n = 0 then fun x -> x + 1
 else fun x -> x - 1;;`
- **Memorizzabili**: se possono essere memorizzati in una **variabile**
⌘ `Obj.val = Obj.val + 10;`

Esempio: le funzioni in ML (puro)

- Denotabili:
- `let plus (x, y) = x + y`
- Esprimibili
- `let plus = fun x -> fun y -> x + y`
- Memorizzabili:
- NO

Tipi di dato di sistema e di programma

- In una macchina astratta (e in una semantica) si possono vedere **due classi di tipi di dato** (o domini semantici)
 - i **tipi di dato di sistema**
 - ✓ definiscono lo **stato e le strutture dati** utilizzate nella simulazione di costrutti di controllo
 - i **tipi di dato di programma**
 - ✓ domini corrispondenti ai **tipi primitivi del linguaggio e ai tipi che l'utente può definire (se il linguaggio lo consente)**
- Tratteremo insieme le due classi anche se il componente “dati” del linguaggio comprende ovviamente solo i tipi di dato di programma

Tipi a tempo di compilazione e di esecuzione

1. Se l'informazione sui tipi è conosciuta completamente “a tempo di compilazione” (OCaml)
 1. il type checking è effettuato totalmente dal compilatore (type checking statico) sul testo del programma
2. Se l'informazione sui tipi è nota solo “a tempo di esecuzione” (JavaScript)
 1. sono necessari i **descrittori** per tutti i tipi di dato
 2. il type checking è effettuato totalmente a tempo di esecuzione (type checking dinamico)

Ogni valore deve avere un **descrittore** che viene usato dalla **macchina astratta** per verificare i tipi a run-time

Descrittori di dato

- Problema: per poter riconoscere il valore e interpretare correttamente la stringa di bit
 - è necessario (in via di principio) associare alla stringa un'altra struttura che contiene la descrizione del tipo (**descrittore di dato**), che viene usato ogni qualvolta si applica al dato un'operazione
 - ✓ per controllare che il tipo del dato sia quello previsto dall'operazione (type checking "dinamico")
 - ✓ per selezionare l'operatore giusto per eventuali operazioni overloaded

Descrittori per espressioni

```
type exp =  
  (* AST *)  
  | Eint of int  
  | Ebool of bool
```

```
type val=  
  (*Valori run-time*)  
  | Int of int  
  | Bool of bool
```

I descrittori dei tipi di dato sono espressi
tramite i costruttori **Int** e **Bool**

Uso dei descrittori

```
let plus(x, y) =  
  if typecheck("int", x) & typecheck("int", y)  
  then (match (x, y) with  
        (Int(u), Int(w)) -> Int (u + w))  
  else failwith ("type error")
```

```
let typecheck (x, y) = match x with  
  | "int" ->  
    (match y with  
     | Int(u) -> true  
     | _ -> false)
```

Tipi a tempo di compilazione e di esecuzione

1. Se l'informazione sui tipi è conosciuta completamente "a tempo di compilazione" (**OCaml**)
 1. si possono eliminare i descrittori di dato
 2. il type checking è effettuato totalmente dal compilatore (type checking statico)
2. Se l'informazione sui tipi è nota solo "a tempo di esecuzione" (**JavaScript**)
 1. sono necessari i descrittori per tutti i tipi di dato
 2. il type checking è effettuato totalmente a tempo di esecuzione (type checking dinamico)
3. Se l'informazione sui tipi è conosciuta solo parzialmente "a tempo di compilazione" (**Java**)
 1. i descrittori di dato contengono solo l'informazione "dinamica"
 2. il type checking è effettuato in parte dal compilatore e in parte dal supporto a tempo di esecuzione

Cos'è un tipo di dato e cosa vogliamo saperne

- Un **TD** è una collezione di valori
 - rappresentati da opportune strutture dati e da un insieme di operazioni per manipolarli
- Come sempre ci interessano due livelli
 - semantica
 - implementazione

Descrittori di dato

- Obiettivo: rappresentare una **collezione di valori** utilizzando quanto ci viene fornito da un linguaggio macchina
 - un po' di tipi numerici, caratteri
 - sequenze di celle di memoria
- Qualunque valore della collezione è alla fine una **stringa** di **bit**

Tipi scalari (esempi)

- **Booleani**

- val: true, false
- op: or, and, not, condizionali
- repr: un **byte**
- note: **C non ha un tipo bool**

- **Caratteri**

- val: a,A,b,B, ..., è,é,ë, ;, ', ...
- op: uguaglianza; code/decode; dipendono dal linguaggio
- repr: un **byte** (ASCII) o due **byte** (UNICODE)

Tipi scalari (esempi)

- **Interi**
 - val: 0,1,-1,2,-2,..., **maxint**
 - op: +, -, *, mod, div, ...
 - repr: alcuni **byte** (2 o 4); **complemento a due**
 - note: interi e interi lunghi (anche 8 **byte**); limitati problemi nella portabilità quando la lunghezza non è specificata nella definizione del linguaggio

- **Reali**
 - val: **valori razionali in un certo intervallo**
 - op: +, -, *, /, ...
 - repr: alcuni **byte** (4); **virgola mobile**
 - note: reali e reali lunghi (8 **byte**); problemi di portabilità quando la lunghezza non è specificata nella definizione del linguaggio

Tipi scalari (esempi)

- Il tipo **void**

- ha un solo valore
- nessuna operazione
- serve per definire il tipo di operazioni che modificano lo stato senza restituire alcun valore

void **f** (...) {...}

- il valore restituito da **f** di tipo **void** è sempre il solito (e dunque non interessa)

Tipi composti

- **Record**
 - collezione di campi (field), ciascuno di un (diverso) tipo
 - un campo è selezionato col suo nome
- **Record varianti**
 - record dove solo alcuni campi (mutuamente esclusivi) sono attivi a un dato istante
- **Array**
 - funzione da un tipo indice (scalare) a un altro tipo
 - array di caratteri sono chiamati stringhe; operazioni speciali
- **Insieme**
 - sotto-insieme di un tipo base
- **Puntatore**
 - riferimento (reference) a un oggetto di un altro tipo

Record

- » Introdotti per manipolare in modo unitario dati di tipo eterogeneo (C, C++, CommonLisp, Ada, Pascal, Algol68)
- » Java non ha tipi record. Perché'?
- » Esempio in C:

```
type studente =struct [  
    int matricola;  
    float altezza;]
```

- » Selezione di campo: `studente.matricola=100;`
- » Possono essere annidati
- » Generalmente sono esprimibili, denotabili e memorizzabili!!!

Record: implementazione

- Memorizzazione sequenziale dei campi
- Allineamento alla parola (16/32/64 bit)
 - spreco di memoria
- Pudding o packed record
 - disallineamento dei campi
 - accesso più costoso

Record: implementazione

```
struct x_  
{  
    char a;        // 1 byte  
    int b;         // 4 byte  
    short c;       // 2 byte  
    char d;        // 1 byte  
};
```

L'allineamento alla parola determina uno spreco di occupazione di memoria

Record: implementazione in OCaml

```
type label = Lab of string
type exp = ...
  | Record of (label * exp) list
  | Select of exp * label
```

```
Record([ (Lab "size", Int 7);
         (Lab "weight", Int 255) ])
```

Interprete

```
let rec eval e = match e with
  ...
  | Record (body) -> Record(evalRecord body)
  | Select (e1, l) -> match eval e1 with
    | Record(body) -> lookupRecord body l
    | _ -> raise TypeMismatch

let rec evalRecord body = match body with
  | [] -> []
  | (Lab l, e)::t ->
    (Lab l, eval e)::evalRecord t
```

Funzione ausiliaria di valutazione

```
let rec lookupRecord body (Lab l) =  
  match body with  
  | [] -> raise FieldNotFound  
  | (Lab l', v) :: t ->  
    if l = l' then v else  
      lookupRecord t (Lab l)
```

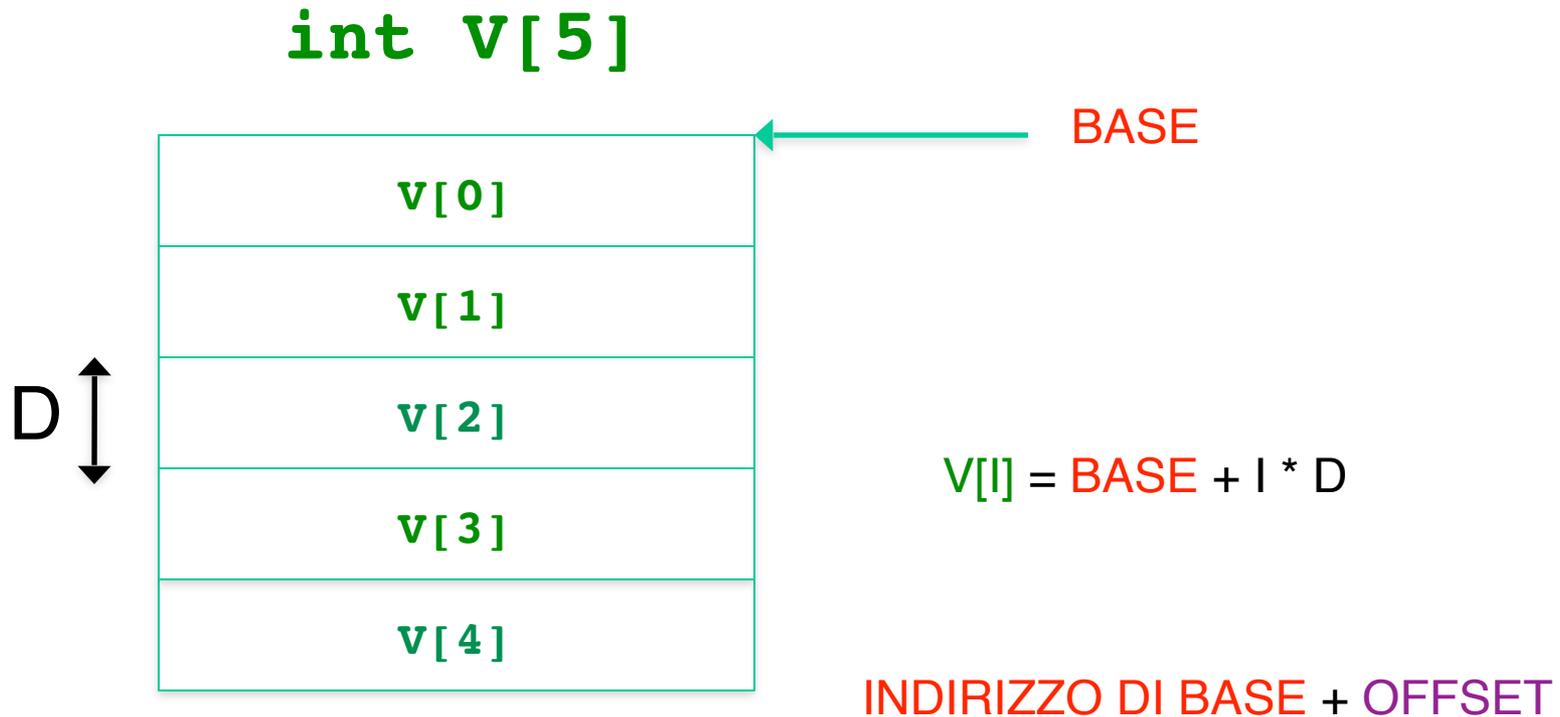
Array

- **Collezioni di dati omogenei**
 - funzione da un tipo indice al tipo degli elementi
 - indice: in genere discreto
 - elemento: “qualsiasi tipo” (raramente un tipo funzionale)
- **Dichiarazioni**
 - C: `int vet[30];` tipo indice tra 0 e 29
- **Array multidimensionali**
- **Principale operazione permessa**
 - selezione di un elemento: `vet[3], mat[10, 'c']`
 - attenzione: la modifica non è un'operazione sull'array, ma sulla locazione modificabile che memorizza un (elemento di) array

Array: implementazione

- Gli array sono di solito memorizzati in una porzione di memoria continua
- **Elementi memorizzati in locazioni contigue:**
 - ordine di riga: $v[1,1]; v[1,2]; \dots; v[1,10]; v[2,1]; \dots$
✓ maggiormente usato
 - ordine di colonna: $v[1,1]; v[2,1]; v[3,1]; \dots; v[10,1]; v[1,2]; \dots$
- **Formula di accesso (caso lineare)**
 - vettore $v[N]$ of **elem_type**
 - $v[n] = \text{base} + c * n,$
dove c è la dimensione per memorizzare un **elem_type**
- Un formula di accesso (più articolata) può essere stabilita anche per gli **array multidimensionali**

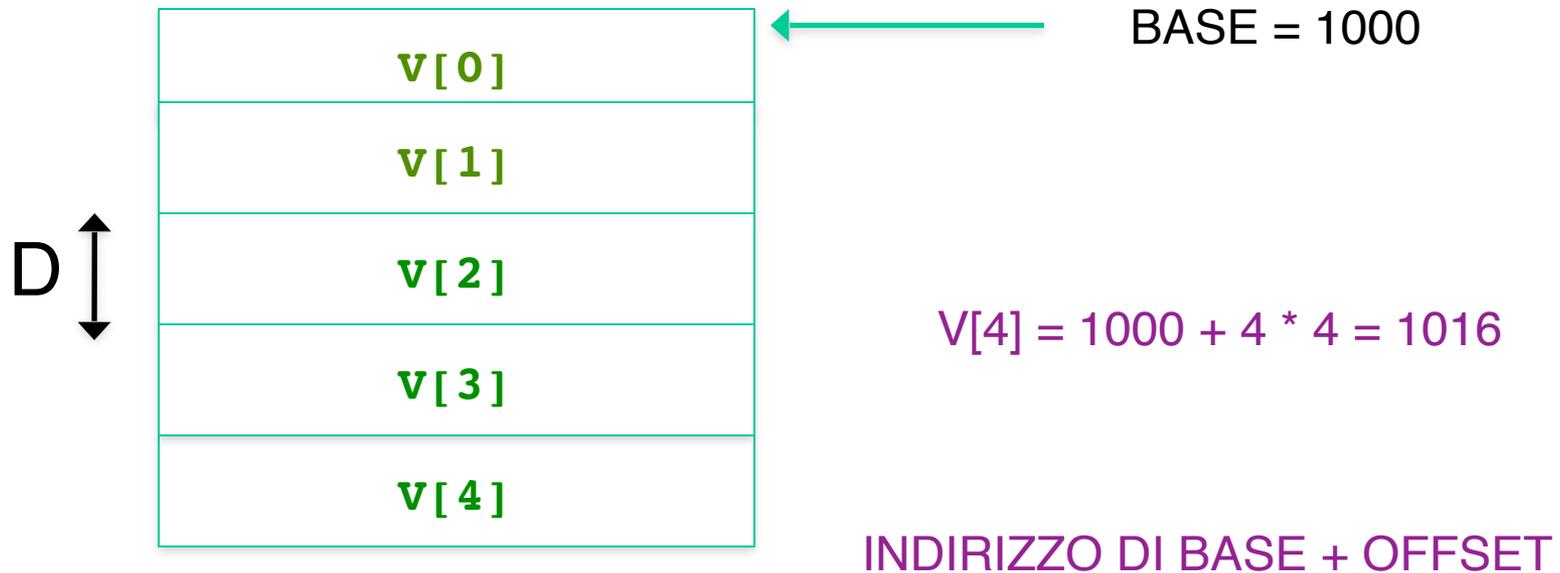
Accesso array: esempio



D dimensione in byte del tipo di base

Accesso array: esempio

int v[5]



D dimensione in byte del tipo di base = 4 byte

Il caso del C

- Il C non prevede controlli a runtime sulla correttezza degli indici di array
- Esempio: un array di 20 elementi di dimensione 2 byte allocato all'indirizzo 1000, l'ultima cella valida (indice 19) è allocata all'indirizzo 1038
- Se il programma, per errore, tenta di accedere l'array all'indice 40, il runtime non rileverà l'errore e fornirà un accesso scorretto alla locazione di memoria 1080

Puntatori

- **Valori: riferimenti + costante `null` (`nil`)**
- **Operazioni**
 - creazione
 - funzioni di libreria che **alloca e restituisce** un puntatore (e.g., **`malloc`**)
 - dereferenziazione
 - accesso al dato “puntato”: **`*p`**
 - test di uguaglianza
 - in specie test di uguaglianza con **`null`**

Il caso del C: array e puntatori

- Array e puntatori sono intercambiabili in C (!!)

```
int n;  
int *a;      // puntatore a interi  
int b[10];  // array di 10 interi  
...  
a = b;      // a punta all'elemento iniziale di b  
n = a[3];   // n ha il valore del terzo elemento di b  
n = *(a+3); // idem  
n = b[3];   // idem  
n = *(b+3); // idem
```

- Ma `a[3] = a[3]+1;`
modificherà anche `b[3]` (è la stessa cosa!)

Tipi di dato di sistema: un esempio

Pila non modificabile: Interfaccia

```
# module type Pila =  
  sig  
    type 'a pila  
    val create : int -> 'a pila  
    val push : 'a * 'a pila -> 'a pila  
    val pop : 'a pila -> 'a pila  
    val top : 'a pila -> 'a  
    val is_empty : 'a pila -> bool  
    val lungh : 'a pila -> int  
    exception Empty  
    exception Full  
  end
```

Pila NM: Semantica Algebrica

```
# module SemPila: Pila =
  struct
    type 'a pila =      New of int
                      | Push of 'a pila * 'a
    (*tipo algebrico *)
    exception Empty
    exception Full

    let create n = New n
    let rec max = function
      | New n -> n
      | Push (p, a) -> max p
    let rec lungh = function
      | New _ -> 0
      | Push (p, _) -> 1 + lungh(p)

    ...
  end
```

Pila NM: Semantica Algebrica

```
# module SemPila: Pila =
  struct
    type 'a pila = New of int | Push of 'a pila * 'a
                (*tipo algebrico *)

    let push(a, p) = if lungh (p) = max(p)
                    then raise Full else Push(p, a)
    let pop = function
        | Push (p, a) -> p
        | New n -> raise Empty
    let top = function
        | Push (p, a) -> a
        | New n -> raise Empty
    let is_empty = function
        | Push (p, a) -> false
        | New n -> true
  end
```

Pila NM: implementazione

```
# module ImpPila: Pila = struct
  type 'a pila =
    IPila of ('a option array) * int
    ...
end
```

- Il componente principale dell'implementazione è un **array** (astrazione della) memoria fisica in una implementazione in linguaggio macchina
- **Classica implementazione sequenziale**
 - utilizzata anche per altri tipi di dato simili alle pile (code)

Pila non modificabile: implementazione

```
# module ImpPila: Pila =
struct
  type 'a pila =
    IPila of ('a option array) * int
  let create n = IPila (Array.make n None, -1)
  let push(x, IPila(s,n)) = if n = (Array.length s - 1)
    then raise Full
    else (Array.set s (n + 1) (Some x) ;
          IPila (s, n + 1))
  let top(IPila(s,n)) = if n = -1 then raise Empty
    else (match Array.get s n with
          | Some y -> y)
  let pop(IPila(s,n)) = if n = -1 then raise Empty
    else IPila (s, n - 1)
  let is_empty(IPila(s,n)) = if n = -1 then true else false
  let lugh(IPila(s,n)) = n-1
end
```

Pila modificabile: interfaccia

```
# module type MPila =  
  sig  
    type 'a pila  
    val create : int -> 'a pila  
    val push : 'a * 'a pila -> unit  
    val pop : 'a pila -> unit  
    val top : 'a pila -> 'a  
    val is_empty : 'a pila -> bool  
    val lungh : 'a pila -> int  
    exception Empty  
    exception Full  
  
  end
```

Pila modificabile: implementazione

```
# module ImpMPila: type MPila =
  struct
    type 'a pila= ('a list) ref * int ref * int

    let create n = (ref [], ref 0, n)
    let push (x,(s,l,n)) = if !l = n then raise Full
                          else (s := x::!s; l := !l + 1)
    let top (s,l,n) = match !s with
                      | [] -> raise Empty
                      | x :: xs -> x
    let pop (s,l,n) = match !s with
                    | [] -> raise Empty
                    | x :: xs -> (s := xs; l := !l - 1)
    let is_empty (s,l,n) = if !l = 0 then true else false
    let lungh (s,l,n) = !l
  end
```

Meta-programmazione

- Un **meta-programma** è un programma che opera su altri programmi
- Esempi: **interpreti, analizzatori, debugger, ottimizzatori, compilatori, etc.**
- La meta-programmazione è utile soprattutto per definire nel linguaggio stesso
 - strumenti di supporto allo sviluppo
 - estensioni del linguaggio

Definizione di **tipi di dato**

- La programmazione di applicazioni consiste in gran parte nella definizione di “**nuovi tipi di dato**”
- Un qualunque **tipo di dato** può essere definito in qualunque linguaggio
 - anche in linguaggio macchina
- Gli aspetti importanti
 - quanto costa?
 - esiste il tipo?
 - il tipo è astratto?

Quanto costa?

- Il costo della simulazione di un “nuovo tipo di dato” dipende dal repertorio di strutture dati primitive fornite dal linguaggio
 - in linguaggio macchina, le sequenze di celle di memoria
 - in FORTRAN e ALGOL 60, gli array
 - in Pascal e C, le strutture allocate dinamicamente e i puntatori
 - in LISP, le s-espressioni
 - in ML e ProLog, le liste e i termini
 - in C++ e Java, gli oggetti

Esiste il tipo?

- Anche se abbiamo realizzato una **implementazione** delle liste (con **heap**, etc.) in **FORTRAN** o **ALGOL**
 - non abbiamo veramente a disposizione il tipo
- Poichè i tipi non sono **denotabili**
 - non possiamo “dichiarare” oggetti di tipo lista
- Stessa situazione in **LISP** e **ProLog**
- In **Pascal**, **ML** e **Java** i tipi sono **denotabili**, anche se con meccanismi diversi
 - dichiarazioni di tipo
 - dichiarazioni di classe

Dichiarazioni di classe

- Il meccanismo di C++ e Java (anche di OCaml)
- Il tipo è la **classe**
 - parametrico, con relazioni di sottotipo
- I valori del **nuovo tipo** (**oggetti**) sono creati con un'operazione di istanziamento della **classe**
 - non con una dichiarazione
- La parte struttura dati degli **oggetti** è costituita da un insieme di **variabili istanza** (o field) **allocati sullo heap**

Il tipo è astratto?

- Un **tipo astratto** è un insieme di valori
 - di cui non si conosce la rappresentazione (implementazione)
 - che possono essere manipolati solo con le operazioni associate
- Sono tipi astratti tutti i tipi primitivi forniti dal linguaggio
 - la loro rappresentazione effettiva non ci è nota e non è comunque accessibile se non con le operazioni primitive
- Per realizzare tipi di dato astratti servono
 - un meccanismo che permette di dare un nome al nuovo tipo (dichiarazione di tipo o di classe)
 - un meccanismo di “protezione” o **information hiding** che renda la rappresentazione visibile soltanto alle operazioni primitive
 - ✓ variabili d’istanza private in una classe
 - ✓ moduli e interfacce in C e Ocaml