

Dati e Tipi di dato



- ✦ Linguaggi prevedono delle opportune strutture di dati
- ✦ Esempi
 - Struct in C
 - List in OCAML
 - Classi e Oggetti in Java

1

Sistemi di tipo



- ✦ I linguaggi moderni prevedono di associare tipi con i valori manipolati dai costrutti linguistici
- ✦ Sistema di tipo
 - Meccanismo per definire e associare un tipo ai costrutti
 - Regole per definire equivalenza, compatibilita' e inferenza

2

Type Checking



- ✎ Strumento che assicura che un programma segue le regole di compatibilità dei tipi.
- ✎ Linguaggio è strongly typed se evita l'uso non conforme ai tipi richiesti delle operazioni del linguaggio
- ✎ Linguaggio è statically typed se è strongly typed e il controllo dei tipi viene fatto staticamente

3

Esempi



- ✎ OCAML: strongly typed la maggior parte dei controlli è statica
- ✎ C difficilmente fa controlli a run-time
- ✎ Linguaggi di scripting moderni (Python) sono fortemente tipati con controllo dinamico

4

Polimorfismo



- ✎ Idea di base e' fare in modo che una operazioni possa essere applicata ad un insieme di tipi
- ✎ Esempio la funzione map di OCAML
- ✎ ML supporta il polimorfismo staticamente mediante un meccanismo per l'inferenza di tipo
 - Esiste un assegnamento di tipi che permette di controllare staticamente la correttezza dell'applicazione di una operazione.
- ✎ Polimorfismo di sottotipo. Una variabile X di tipo T puo' essere usata in tutti quei contesti in cui e' previsto un tipo T' derivato da T
 - C++, Java, Eiffel, C#

5

Tipi di Dato



- ✎ descrittore, tipi, controllo e inferenza dei tipi
- ✎ specifica (semantica) e implementazione di tipi di dato
 - implementazioni "sequenziali"
 - ✓ pile non modificabili
 - implementazione delle liste con la heap
 - termini con pattern matching e unificazione
 - ambiente, tabelle
 - tipi modificabili
 - ✓ pile modificabili
 - programmi come dati e metaprogrammazione
- ✎ meccanismi per la definizione di nuovi tipi
- ✎ astrazioni sui dati

6

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**
 - ✓ domini semantici che definiscono lo stato e strutture dati utilizzate nella simulazione di costrutti di controllo (**esempio: l'ambiente**)
 - **i tipi di dato di programma**
 - ✓ domini corrispondenti ai tipi primitivi del linguaggio ed ai tipi che l'utente può definire (**esempio le dichiarazioni di classi in Java**)
- ✦ tratteremo insieme le due classi anche se il componente "dati" del linguaggio comprende ovviamente solo i tipi di dato di programma

7

Cos'è un Tipo di Dato e cosa vogliamo sapere di lui



- ✦ una collezione di valori
 - rappresentati da opportune strutture dati +
- ✦ un insieme di operazioni per manipolarli
- ✦ Due livelli importanti
 - **semantica**
 - ✓ Una semantica algebrica "implementata" con i meccanismi per la definizione di nuovi tipi (mediante termini) in OCAML
 - **implementazione**
 - ✓ altre implementazioni in OCAML, date tipicamente in termini di strutture dati "a basso livello" (array)

8

Le stringhe (tipo primitivo)



- ✦ C and C++
 - Non sono primitivi
 - Stringa e' un array di **char** con le operazioni associate (libreria di sistema)
- ✦ SNOBOL4 (a string manipulation language)
 - Primitive
 - Valanga di operazioni incluso il pattern matching
- ✦ Fortran - Python
 - Tipo primitivo (assegnamento e altre operazioni)
- ✦ Java
 - Tipo primitivo: la classe `String`
- ✦ Perl, JavaScript, Ruby, ePHP
 - built-in pattern matching, e regular expressions (`s*`, `s+`

9

I Descrittori di Dato



- ✦ immaginiamo di voler rappresentare una collezione di valori utilizzando un linguaggio macchina
 - un po' di tipi numerici, caratteri, etc.
 - sequenze di celle di memoria
 - qualunque valore è alla fine una codifica (stringa di bits)
- ✦ per poter riconoscere il valore e interpretare correttamente la rappresentazione (la stringa di bits)
 - è necessario associare alla stringa un'altra struttura che contiene la descrizione del tipo (*descrittore di dato*), che viene usato ogniqualvolta 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

10



Le stringhe

- ✎ Static length: sufficiente il descrittore a tempo di compilazione
- ✎ Dynamic length: e' indispensabile il descrittore a runt-time;
 - allocation/deallocation e' un problema non banale di implementazione

11



Stringhe

✎ Descrittore statico

✎ Descrittore dinamici

| |
|---------------|
| Static string |
| Length |
| Address |

| |
|------------------------|
| Limited dynamic string |
| Maximum length |
| Current length |
| Address |

12

Tipi a tempo di compilazione e a tempo di esecuzione



- ✎ se l'informazione sul tipo è conosciuta completamente "a tempo di compilazione"
 - si possono eliminare i descrittori di dato,
 - il type checking è effettuato totalmente dal compilatore (type checking statico, ML)
- ✎ se l'informazione sul tipo è nota solo "a tempo di esecuzione"
 - sono necessari i descrittori per tutti i tipi di dato
 - il type checking è effettuato totalmente a tempo di esecuzione (type checking dinamico)

13

✎ se l'informazione sul tipo è conosciuta solo parzialmente "a tempo di compilazione"

- i descrittori di dato contengono solo l'informazione "dinamica"
- il type checking è effettuato in parte dal compilatore ed in parte dal supporto a tempo di esecuzione
- JAVA



14

Tipi a tempo di compilazione: specifica o inferenza?



- ✎ l'informazione sul tipo viene di solito fornita con delle asserzioni (specifiche)
 - nelle dichiarazioni di costanti e variabili
 - nelle dichiarazioni di procedura (tipi dei parametri e tipo del risultato)
 - Tipi dinamici non costringono a specificare tipi ovunque (è palloso) e rende il linguaggio poco flessibile, ma
- ✎ in alternativa (o in aggiunta) alle asserzioni fornite nel programma, il linguaggio può essere dotato di un algoritmo di analisi statica che riesce ad inferire il tipo di ogni espressione
 - tipico dei linguaggi funzionali moderni (esempio, ML)

15

Tipi come valori esprimibili e denotabili



- ✎ importante strumento per la definizione di astrazioni sui dati
 - manca del tutto nei linguaggi che ignorano i tipi (LISP e PROLOG) e nei linguaggi antichi (FORTRAN, ma anche ALGOL)
 - nasce con PASCAL
 - sempre più importante nei linguaggi funzionali, imperativi e object-oriented moderni
- ✎ La nozione di tipo è uno degli argomenti importanti del nostro percorso all'interno dei linguaggi di programmazione

16

Semantica dei tipi di dato



- useremo per la semantica semplicemente OCAML
 - utilizzando il meccanismo dei tipi algebrici (costruttori, etc.) per definire (per casi) un tipo (generalmente ricorsivo)
 - Alcuni esempi
 - ✓ due tipi di pila (non modificabile e modificabile)
 - ✓ liste (non polimorfe)
 - ✓ termini, sostituzioni, unificazione, pattern matching
 - ✓ S-espressioni (LISP)

17

Pila non modificabile: interfaccia



```
# module type PILA =
sig
  type 'a stack
  val emptystack : int * 'a -> 'a stack
  val push : 'a * 'a stack -> 'a stack
  val pop : 'a stack -> 'a stack
  val top : 'a stack -> 'a
  val empty : 'a stack -> bool
  val lungh : 'a stack -> int
  exception Emptystack
  exception Fullstack
end
```

18

Pila non modificabile: semantica



```
# module SemPila: PILA =
struct
  type 'a stack = Empty of int | Push of 'a stack * 'a
  exception Emptystack
  exception Fullstack
  let emptystack (n, x) = Empty(n)
  let rec max = function
    | Empty n -> n
    | Push(p,a) -> max p
  let rec lungh = function
    | Empty n -> 0
    | Push(p,a) -> 1 + lungh(p)
  let push (a, p) = if lungh(p) = max(p) then raise Fullstack else Push(p,a)
  let pop = function
    | Push(p,a) -> p
    | Empty n -> raise Emptystack
  let top = function
    | Push(p,a) -> a
    | Empty n -> raise Emptystack
  let empty = function
    | Push(p,a) -> false
    | Empty n -> true
end
```

19

Pila non modificabile: semantica algebraica



☛ semantica “isomorfa” ad una specifica in stile algebrico

```
'a stack = Empty of int | Push of 'a stack * 'a
emptystack (n, x) = Empty(n)
lungh(Empty n) = 0
lungh(Push(p,a)) = 1 + lungh(p)
push(a,p) = Push(p,a)
pop(Push(p,a)) = p
top(Push(p,a)) = a
empty(Empty n) = true
empty(Push(p,a)) = false
```

- ☛ tipo (ricorsivo) definito per casi (con costruttori)
- ☛ semantica delle operazioni definita da un insieme di equazioni fra termini
- ☛ il tipo di dato è un'algebra (iniziale)

20

Pila non modificabile: implementazione



```
# module ImpPila: PILA =
  struct
    type 'a stack = Pila of ('a array) * int
    exception Emptystack
    exception Fullstack
    let emptystack (nm,x) = Pila(Array.create nm x, -1)
    let push(x, Pila(s,n)) = if n = (Array.length(s) - 1) then
      raise Fullstack else
      (Array.set s (n +1) x;
       Pila(s, n +1))
    let top(Pila(s,n)) = if n = -1 then raise Emptystack
      else Array.get s n
    let pop(Pila(s,n)) = if n = -1 then raise Emptystack
      else Pila(s, n -1)
    let empty(Pila(s,n)) = if n = -1 then true else false
    let lungh(Pila(s,n)) = n
  end
```

21

Pila non modificabile: implementazione



```
# module ImpPila: PILA =
  struct
    type 'a stack = Pila of ('a array) * int
    .....
  end
```

- ✎ il componente principale dell'implementazione è un array
 - memoria fisica in una implementazione in linguaggio macchina
- ✎ classica implementazione sequenziale
 - utilizzata anche per altri tipi di dato simili alle pile (code)

22

Lista (non polimorfa): interfaccia



```
# module type LISTAINT =
  sig
    type intlist
    val emptylist : intlist
    val cons : int * intlist -> intlist
    val tail : intlist -> intlist
    val head : intlist -> int
    val empty : intlist -> bool
    val length : intlist -> int
    exception Emptylist
  end
```

23

Lista: semantica



```
# module SemListaInt: LISTAINT =
  struct
    type intlist = Empty | Cons of int * intlist
    exception Emptylist
    let emptylist = Empty
    let rec length = function
      | Empty -> 0
      | Cons(n,l) -> 1 + length(l)
    let cons (n, l) = Cons(n, l)
    let tail = function
      | Cons(n,l) -> l
      | Empty -> raise Emptylist
    let head = function
      | Cons(n,l) -> n
      | Empty -> raise Emptylist
    let empty = function
      | Cons(n,l) -> false
      | Empty -> true
  end
```

24

Lista e Pila: stessa “semantica”

```
intlist = Empty | Cons of int * intlist
emptylist = Empty
length(Empty) = 0
length(Cons(n,l)) = 1 + length(l)
cons (n, l) = Cons(n, l)
tail(Cons(n,l)) = l
head(Cons(n,l)) = n
empty(Empty) = true
empty(Cons(n,l)) = false

'a stack = Empty of int | Push of 'a stack * 'a
emptystack (n, x) = Empty(n)
lungh(Empty n) = 0
lungh(Push(p,a)) = 1 + lungh(p)
push(a,p) = Push(p,a)
pop(Push(p,a)) = p
top(Push(p,a)) = a
empty(Empty n) = true
empty(Push(p,a)) = false
```

25

Stessa “implementazione”?

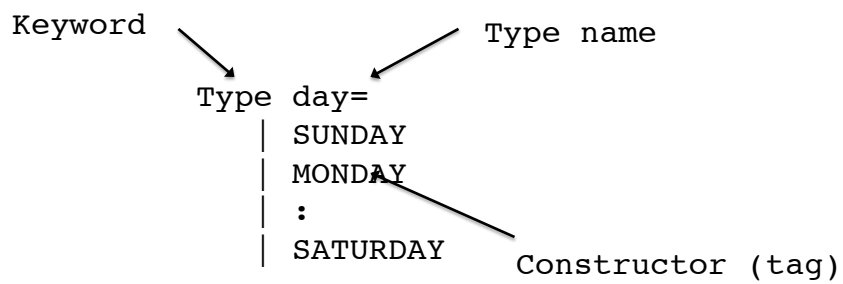
- ✎ non conviene implementare una lista con un array
 - vorremmo una implementazione sequenziale in cui un unico array viene utilizzato per rappresentare “tante” liste
 - la heap!
- ✎ l’array unico va bene per la pila perché è un tipo di dato di sistema
 - ne esistono un numero piccolo e predefinito nella implementazione del linguaggio
- ✎ la lista è tipicamente un tipo di dato d’utente
 - che ne può costruire un numero arbitrario nei propri programmi
- ✎ mostriamo l’implementazione delle liste con una heap senza operazioni per “disallocare”

26

Intermezzo



🐘 Come definiamo tipi di dato in OCAML?



Costruttori sono il valore del tipo

27

Tipi e pattern matching



```
let to_print = (n : day): string =
  begin match n with
  | SUNDAY -> "domenica"
  | MONDAY -> "lunedì"
  | :
  | SABATO -> "sabato"
  end
```

Abbiamo un caso per costruttore:
Il pattern matching e' definito sulla sintassi
del tipo

28

Tipi



Costruttori possono "trasportare" valori

```
Type stack_int =  
  | Empty  
  | Push of 'stack_int * int
```

Valori: `Push(Push(Empty, 6), 7)`

29

Tipi e ricorsione



```
type my_string_list =  
  | Nil  
  | Cons of string * my_string_list
```

30

Il perche' del tutto



- ✎ Non un'opinione personale ma una valutazione generale
- ✎ Il punto chiave: data abstraction
 - Astrarre dai dettagli di implementazione
 - Meccanismo di specifica
 - Strumento per ragionare sui programmi e sul progetto di programmi

31

Altre motivazioni



- ✎ Programmare non significa solamente codificare algoritmi furbi
- ✎ Bisogna organizzare e manipolare dati
 - Progettare strutture dati
 - Scrivere il codice per accedere e manipolare le strutture dati
- ✎ A volte cambiare una struttura dati e' un problema

32

Opinione condivisa



- ✚ Pensare a un tipo di dato come a un insieme di operazioni astraendo da come sono realizzate
- ✚ Forzare il programma a accedere alla struttura dati solamente con le operazioni previste.
- ✚ Tre aspetti concettuali da considerare
 - Interfaccia che specifica le operazioni permesse
 - Implementazione che descrive la realizzazione delle operazioni con opportune strutture dati
 - Utente che utilizza solamente le operazioni previste

33

Vantaggi



- ✚ Clienti possono accedere ai dati solamente mediante le operazioni previste
- ✚ Reappresentazione e' nascosta
- ✚ Modo furbo per organizzare e gestire problemi di grosse dimensioni
 - Compilazioni separate
 - Librerie
 - Riutilizzo di codice
- ✚ Meccanismo di astrazione

34

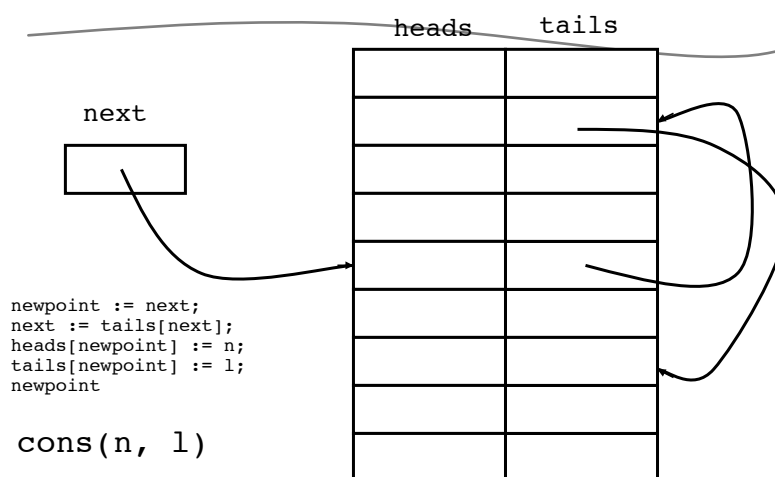
Lista (non polimorfa): interfaccia



```
# module type LISTAINT =
  sig
    type intlist
    val emptylist : intlist
    val cons : int * intlist -> intlist
    val tail : intlist -> intlist
    val head : intlist -> int
    val empty : intlist -> bool
    val length : intlist -> int
    exception Emptylist
  end
```

35

Heap, lista libera, allocazione



36

Lista: implementazione a heap



```
# module ImpListInt: LISTINT =
struct
  type intlist = int
  let heapsize = 100
  let heads = Array.create heapsize 0
  let tails = Array.create heapsize 0
  let next = ref(0)
  let emptyheap =
    let index = ref(0) in
    while !index < heapsize do
      Array.set tails !index (!index + 1); index := !index + 1
    done;
    Array.set tails (heapsize - 1) (-1); next := 0
  exception Fullheap
  exception Emptylist
  let emptylist = -1
  let empty l = if l = -1 then true else false
  let cons (n, l) = if !next = -1 then raise Fullheap else
    ( let newpoint = !next in next := Array.get tails !next;
      Array.set heads newpoint n; Array.set tails newpoint l; newpoint)
  let tail l = if empty l then raise Emptylist else Array.get tails l
  let head l = if empty l then raise Emptylist else Array.get heads l
  let rec length l = if l = -1 then 0 else 1 + length (tail l)
end
```

37

Termini



- 🏔 La prendiamo da lontano e partiamo dalla logica
- 🏔 In un linguaggio logico (FOL)
 - Costanti: c, c' ...
 - Variabili: x, y, \dots
 - Funzioni: f, g, \dots
 - Predicati: P
- 🏔 Definizione di termine (ground = senza variabili)
 - ✓ Le costanti sono termini
 - ✓ Se f è una funzione di arietà n , e t_1, \dots, t_n sono termini allora $f(t_1, \dots, t_n)$ è un termine
 - ✓ Ogni altro termine è costruito applicando le due regole precedenti

38



TERMINI E ALBERI

39

Termini



- ✎ strutture ad albero composte da simboli
 - di variabile
 - di funzione ad n argomenti ($n \geq 0$) (costruttori)
- ✎ un termine è
 - un simbolo di variabile
 - un simbolo di funzione n -aria applicato ad n termini
- ✎ i valori di un tipo ML definito per casi sono termini senza variabili
 - `type intlist = Empty | Cons of int * intlist`
 - `Cons(3,Empty), Cons(1, Cons(2,Empty))`
- ✎ i pattern ML usati per selezionare i casi sono termini con variabili
 - `let head = function`
 - | `Cons(n,l) -> n`

40

Termini con variabili e pattern matching



☞ i termini con variabili

- `Cons(n,l)`

“rappresentano” insiemi possibilmente infiniti di strutture dati

☞ il pattern matching

- struttura dati (termine senza variabili) vs. pattern può essere utilizzato per definire “selettori”
- `let head = function`

| `Cons(n,l) -> n`

☞ le variabili del pattern vengono “legate” a sottotermini (componenti della struttura dati)

41

Termini e unificazione



☞ unificazione

- tra due termini con variabili

le variabili in uno qualunque dei due termini possono essere “legate” a sottotermini

☞ l’algoritmo di unificazione calcola la sostituzione più generale che rende uguali i due termini oppure fallisce

- la sostituzione è rappresentata da un insieme di equazioni fra termini “in forma risolta”
 - ✓ equazioni del tipo `variabile = termine`
 - ✓ `variabile` non occorre in nessun `termine`

42

Termini, sostituzioni: interfaccia



```
# module type TERM =
  sig
    type term
    type equationset
    exception MatchFailure
    exception UndefinedMatch
    exception UnifyFailure
    exception OccurCheck
    exception WrongSubstitution
    val unifyterms : term * term -> equationset
    val matchterms : term * term -> equationset
    val instantiate : term * equationset -> term
  end
```

43

Termini, sostituzioni: 1



```
# module Term: TERM =
  struct
    type term = Var of string | Cons of string * (term list)
    type equationset = (term * term) list
    exception MatchFailure
    exception UndefinedMatch
    exception UnifyFailure
    exception OccurCheck
    exception WrongSubstitution

    let rec ground = function
      | Var _ -> false
      | Cons(_, t1) -> ground1 t1
    and ground1 = function
      | [] -> true
      | t::t11 -> ground(t) & ground1(t11)
    .....
  end
```

44

Termini, sostituzioni: 2



```
# module Term: TERM =
struct
  type term = Var of string | Cons of string * (term list)
  type equationset = (term * term) list
  .....
  let rec occurs (stringa, termine) = match termine with
    | Var st -> stringa = st
    | Cons(_,t1) -> occurs1 (stringa, t1)
  and occurs1 (stringa, t1) = match t1 with
    | [] -> false
    | t::t11 -> occurs(stringa,t) or occurs1(stringa, t11)
  let rec occursset (stringa, e) = match e with
    | [] -> false
    | (t1,t2)::e1 -> occurs(stringa, t1) or occurs(stringa, t2)
      or occursset(stringa, e1)
  let rec rplact (t, v, t1) = match t with
    | Var v1 -> if v1 = v then t1 else t
    | Cons(s,t1) -> Cons(s, rplact1(t1,v,t1))
  and rplact1 (t1, v, t1) = match t1 with
    | [] -> []
    | t::t11 -> rplact(t, v, t1) :: rplact1(t11, v, t1)
  .....
```

45

Termini, sostituzioni: 3



```
# module Term: TERM =
struct
  type term = Var of string | Cons of string * (term list)
  type equationset = (term * term) list
  .....
  let rec replace (eqset, stringa, termine) = match eqset with
    | [] -> []
    | (s1, t1) :: eqset1 ->
      (rplact(s1,stringa,termine),rplact(t1,stringa,termine))::
      replace(eqset1, stringa, termine)

  let rec pairs = function
    | [], [] -> []
    | a::l1, b::l2 -> (a,b)::pairs(l1, l2)
    | _ -> raise UnifyFailure
  .....
```

46

Termini, sostituzioni: 4



```
# module Term: TERM =
struct
  type term = Var of string | Cons of string * (term list)
  type equationset = (term * term) list
  .....
  let rec unify (eq1, eq2) = match eq1 with
  | [] -> eq2
  | (Var x, Var y) :: eq11 -> if x = y then unify(eq11, eq2) else
    unify(replace(eq11,x,Var y),(Var x,Var y)::(replace(eq2,x,Var y)))
  | (t, Var y) :: eq11 -> unify((Var y,t)::eq11,eq2)
  | (Var x, t) :: eq11 -> if occurs(x,t) then raise OccurCheck else
    unify(replace(eq11,x,t),(Var x,t)::(replace(eq2,x,t)))
  | (Cons(x,x1), Cons(y,y1)) :: eq11 -> if not(x = y) then
    raise UnifyFailure else unify(pairs(x1,y1)@eq11,eq2)

  let unifyterms (t1, t2) = unify([(t1,t2)],[])
  .....
```

47

Termini, sostituzioni: 5



```
# module Term: TERM =
struct
  type term = Var of string | Cons of string * (term list)
  type equationset = (term * term) list
  .....
  let rec pmatch (eq1, eq2) = match eq1 with
  | [] -> eq2
  | (Var x, t) :: eq11 -> if occursset(x, eq11@eq2) then
    raise UndefinedMatch else pmatch(eq11,(Var x, t)::eq2)
  | (Cons(x,x1), Cons(y,y1)) :: eq11 -> if not(x = y) then
    raise MatchFailure else pmatch(pairs(x1,y1)@eq11, eq2)
  | _ -> raise UndefinedMatch

  let matchterms (pattern, termine) =
    if not (ground(termine)) then raise UndefinedMatch else
    pmatch([(pattern,termine)],[])

  let rec instantiate (t, e) = match e with
  | [] -> t
  | (Var x, t1) :: e1 -> instantiate (rplact(t, x, t1), e1)
  | _ -> raise WrongSubstitution
end
```

48

Ambiente (env)



- ✎ tipo (polimorfo) utilizzato nella semantica e nelle implementazioni per mantenere una associazione fra stringhe (identificatori) e valori di un opportuno tipo
- ✎ la specifica definisce il tipo come funzione
- ✎ l'implementazione che vedremo utilizza le liste
- ✎ è simile il dominio store

49

Ambiente: interfaccia



```
# module type ENV =
  sig
    type 't env
    val emptyenv : 't -> 't env
    val bind : 't env * string * 't -> 't env
    val bindlist : 't env * (string list) * ('t list)
      -> 't env
    val applyenv : 't env * string -> 't
    exception WrongBindlist
  end
```

50

Ambiente: semantica



```
# module Funenv:ENV =
struct
  type 't env = string -> 't
  exception WrongBindlist
  let emptyenv(x) = function y -> x
  let applyenv(x,y) = x y
  let bind(r, l, e) =
    function lu -> if lu = l then e else applyenv(r,lu)
  let rec bindlist(r, il, el) = match (il,el) with
  | ([],[]) -> r
  | i::i1l, e::e1l -> bindlist (bind(r, i, e), i1l, e1l)
  | _ -> raise WrongBindlist
end
```

51

Ambiente: implementazione



```
# module Listenv:ENV =
struct
  type 't env = (string * 't) list
  exception WrongBindlist
  let emptyenv(x) = [("",x)]
  let rec applyenv(x,y) = match x with
  | [(_,e)] -> e
  | (il,e1) :: x1 -> if y = i1 then e1 else applyenv(x1, y)
  | [] -> failwith("wrong env")
  let bind(r, l, e) = (l,e) :: r
  let rec bindlist(r, il, el) = match (il,el) with
  | ([],[]) -> r
  | i::i1l, e::e1l -> bindlist (bind(r, i, e), i1l, e1l)
  | _ -> raise WrongBindlist
end
```

52

Tipi di dato modificabili



- ✎ a livello semantico, riconduciamo la modificabilità alla nozione di variabile
 - lo stato “modificabile” corrispondente sarà in realtà modellato con il dominio store
- ✎ per l’implementazione usiamo varie strutture dati modificabili come l’array

53

Pila modificabile: interfaccia



```
# module type MPILA =
sig
  type 'a stack
  val emptystack : int * 'a -> 'a stack
  val push : 'a * 'a stack -> unit
  val pop : 'a stack -> unit
  val top : 'a stack -> 'a
  val empty : 'a stack -> bool
  val length : 'a stack -> int
  val svuota : 'a stack -> unit
  val access : 'a stack * int -> 'a
  exception Emptystack
  exception Fullstack
  exception Wrongaccess
end
```

54

Pila modificabile: semantica



```
# module SemMPila: MPILA =
struct
  type 'a stack = ('a SemPila.stack) ref
  exception Emptystack
  exception Fullstack
  exception Wrongaccess
  let emptystack (n, a) = ref(SemPila.emptystack(n, a) )
  let lungh x = SemPila.lungh(!x)
  let push (a, p) = p := SemPila.push(a, !p)
  let pop x = x := SemPila.pop(!x)
  let top x = SemPila.top(!x)
  let empty x = SemPila.empty !x
  let rec svuota x = if empty(x) then () else (pop x; svuota x)
  let rec faccess (x, n) =
    if n = 0 then SemPila.top(x) else faccess(SemPila.pop(x), n-1)
  let access (x, n) = let nof pops = lungh(x) - 1 - n in
    if nof pops < 0 then raise Wrongaccess else faccess(!x, nof pops)
end
```

55

Pila modificabile: implementazione



```
module ImpMPila: MPILA =
struct
  type 'x stack = ('x array) * int ref
  exception Emptystack
  exception Fullstack
  exception Wrongaccess
  let emptystack(nm, (x: 'a)) = ((Array.create nm x, ref(-1)): 'a stack)
  let push(x, ((s,n): 'x stack)) = if !n = (Array.length(s) - 1) then
    raise Fullstack else (Array.set s (!n + 1) x; n := !n + 1)
  let top(((s,n): 'x stack)) = if !n = -1 then raise Emptystack
    else Array.get s !n
  let pop(((s,n): 'x stack)) = if !n = -1 then raise Emptystack
    else n:= !n - 1
  let empty(((s,n): 'x stack)) = if !n = -1 then true else false
  let lungh( (s,n): 'x stack) = !n
  let svuota ((s,n): 'x stack) = n := -1
  let access (((s,n): 'x stack), k) =
    (* if not(k > !n) then *)
    Array.get s k
    (* else raise Wrongaccess *)
end
```

56

S-espressioni



- 👁️ la struttura dati fondamentale di LISP
 - alberi binari con atomi (stringhe) sulle foglie
 - modificabili
- 👁️ vedremo solo interfaccia e semantica
 - l'implementazione (a heap) simile a quella delle liste

57

S-espressione: interfaccia



```
# module type SEXPR =
  sig
    type sexpr
    val nil : sexpr
    val cons : sexpr * sexpr -> sexpr
    val node : string -> sexpr
    val car : sexpr -> sexpr
    val cdr : sexpr -> sexpr
    val null : sexpr -> bool
    val atom : sexpr -> bool
    val leaf : sexpr -> string
    val rplaca: sexpr * sexpr -> unit
    val rplacd: sexpr * sexpr -> unit
    exception NotALeaf
    exception NotACons
  end
```

58

S-espressione: semantica

```
# module SemSexpr: SEXPR =
  struct
    type sexpr = Nil | Cons of (sexpr ref) * (sexpr ref) | Node of string
    exception NotACons
  exception NotALeaf
    let nil = Nil
    let cons (x, y) = Cons(ref(x), ref(y))
    let node s = Node s
    let car = function Cons(x,y) -> !x
      | _ -> raise NotACons
    let cdr = function Cons(x,y) -> !y
      | _ -> raise NotACons
    let leaf = function Node x -> x
      | _ -> raise NotALeaf
    let null = function Nil -> true
      | _ -> false
    let atom = function Cons(x,y) -> false
      | _ -> true
    let rplaca = function (Cons(x, y), z) -> x := z
      | _ -> raise NotACons
    let rplacd = function (Cons(x, y), z) -> y := z
      | _ -> raise NotACons
  end
```

59

Programmi come dati

- la caratteristica fondamentale della macchina di Von Neumann
 - i programmi sono un particolare tipo di dato rappresentato nella memoria della macchina
- permette, in linea di principio, che, oltre all'interprete, un qualunque programma possa operare su di essi
- possibile sempre in linguaggio macchina
- possibile nei linguaggi ad alto livello
 - se la rappresentazione dei programmi è visibile nel linguaggio
 - e il linguaggio fornisce operazioni per manipolarla
- di tutti i linguaggi che abbiamo nominato, gli unici che hanno questa caratteristica sono LISP e PROLOG
 - un programma LISP è rappresentato come S-espressione
 - un programma PROLOG è rappresentato da un insieme di termini

60

Metaprogrammazione



- ✎ un metaprogramma è un programma che opera su altri programmi
- ✎ esempi: interpreti, analizzatori, debuggers, ottimizzatori, compilatori, etc.
- ✎ la metaprogrammazione è utile soprattutto per definire, nel linguaggio stesso,
 - strumenti di supporto allo sviluppo
 - estensioni del linguaggio

61

Meccanismi per la 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?

62

Quanto costa? 1



- ✎ 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 arrays
 - in PASCAL e C, le strutture allocate dinamicamente ed i puntatori
 - in LISP, le s-espressioni
 - in ML e Prolog, le liste ed i termini
 - in C++ e Java, gli oggetti

63

Quanto costa? 2



- ✎ è utile poter disporre di
 - strutture dati statiche sequenziali, come gli arrays e i records
 - un meccanismo per creare strutture dinamiche
 - ✓ tipo di dato dinamico (lista, termine, s-espressione)
 - ✓ allocazione esplicita con puntatori (à la Pascal-C, oggetti)

64

Esiste il tipo?



- ✎ anche se abbiamo realizzato una implementazione delle liste (con heap, lista libera, 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, Java i tipi sono denotabili, anche se con meccanismi diversi
 - dichiarazioni di tipo
 - dichiarazioni di classe

65

Dichiarazioni di tipo



`type nometipo = espressione di tipo`

- ✎ il meccanismo di PASCAL, C ed ML
- ✎ il potere espressivo dipende dai meccanismi forniti per costruire espressioni di tipo (costruttori di tipo)
 - PASCAL, C
 - ✓ enumerazione
 - ✓ record, record ricorsivo
 - ML
 - ✓ enumerazione
 - ✓ prodotto cartesiano
 - ✓ iterazione
 - ✓ somma
 - ✓ funzioni
 - ✓ ricorsione
- ✎ ci possono essere tipi parametrici
 - in particolare, polimorfi (parametri di tipo tipo)

66

Dichiarazioni di classe



- ✦ il meccanismo di C++ e Java (anche OCAML)
- ✦ il tipo è la classe
 - parametrico in OCAML
 - con relazioni di sottotipo
- ✦ i valori del nuovo tipo (oggetti) sono creati con un'operazione di istanziazione della classe
 - non con una dichiarazione
- ✦ la parte struttura dati degli oggetti è costituita da un insieme di variabili istanza (o fields) allocati sulla heap

67

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 istanza private in una classe
 - ✓ moduli e interfacce in C ed ML

68