



SEMANTICA DEI LINGUAGGI DI PROGRAMMAZIONE

1

Semantica e supporto a run time



- Le differenze fra i linguaggi di programmazione si riflettono in differenze nelle corrispondenti implementazioni
- Tutte le caratteristiche importanti per progettare un interprete o un supporto a tempo di esecuzione, si possono ricavare analizzando la semantica operativa del linguaggio

2

La nostra visione: Semantiche “eseguibili”



- Ocaml come metalinguaggio per esprimere la semantica dei linguaggi di programmazione
 - semantiche eseguibili che ci permettono di analizzare e valutare tutte le caratteristiche dei diversi paradigmi di programmazione e dei loro meccanismi di implementazione

3



E I DATI?

4

A cosa servono?



- 🦋 *Livello di progetto:* **Organizzano l'informazione**
 - Tipi diversi per concetti diversi:
 - Meccanismi espliciti dei linguaggi per l'astrazione sui dati (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 informazione necessaria alla macchina astratta per allocare spazio di memoria

5

Dati: classificazione



- 🦋 **Denotabili:** se possono essere associati ad un nome
- 🦋 **Esprimibili:** se possono essere il risultato della valutazione di una espressione complessa (diversa dal semplice nome)
- 🦋 **Memorizzabili:** se possono essere memorizzati in una variabile

6

Esempio: le funzioni in ML (puro)



☞ Denotabili:

- `let plus (x, y) = x+y`

☞ Esprimibili:

- `let plus = function(x: int) -> function(y:int) -> x+y`

☞ Memorizzabili: NO

7

TIPI



Un tipo è una collezione di valori dotata di un insieme di operazioni per manipolare tali valori.

La distinzione tra collezioni di valori che sono tipi o non sono tipi è una nozione che dipende dal linguaggio di programmazione.

Sistema di tipi



- I linguaggi moderni prevedono di associare tipi con i valori manipolati dai costrutti linguistici
- **Sistema di tipi**: il complesso delle informazioni che regolano i tipi nel linguaggio di programmazione
 - Tipi predefiniti
 - Meccanismi per definire e associare un tipo ai costrutti
 - Regole per definire equivalenza, compatibilità e inferenza

9

Sistema di tipi



- Un sistema di tipo per un linguaggio è detto **type safe** quando nessun programma può violare le distinzioni tra i tipi del linguaggio
 - Nessun programma durante l'esecuzione può generare un errore che derivi da una violazione di tipo

10



Type Checking

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

11



Regole di type checking

Regole di tipo definiscono quando un costrutto del linguaggio soddisfa i requisiti di tipo.

$$\frac{tenv \triangleright ehrs \Rightarrow tval \quad tenv[tval / x] \triangleright ebody \Rightarrow t}{tenv \triangleright \text{Let } x = ehrs \text{ in } ebody \Rightarrow t}$$

12



Inferenza di tipo

🦋 **Type inference:** il meccanismo di inferenza di tipi consente di dedurre il tipo associato a un programma senza bisogno di dichiarazioni esplicite

🦋 OCaml

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

13



Come opera l'inferenza?

```
# let f x = 2 + x;;  
val f : int -> int = <fun>
```

1. Qual è il tipo di f?
2. L'operatore + ha due tipi:
 $\text{int} \rightarrow \text{int} \rightarrow \text{int}$,
 $\text{real} \rightarrow \text{real} \rightarrow \text{real}$,
3. La costante 2 è di tipo int
4. Questo ci permette di concludere che $+$: $\text{int} \rightarrow \text{int} \rightarrow \text{int}$
5. Dal contesto di uso deriviamo che $x:\text{int}$
6. In conclusione $f(x:\text{int}) = 2+x$ ha tipo $\text{int} \rightarrow \text{int}$

ALGORITMO EFFETTIVO DI ML È PIÙ COMPLESSO.

14

Storia più articolata



- L'algoritmo di inferenza di tipo è stato originariamente introdotto da **Haskell Curry** e **Robert Feys** per il lambda calcolo tipato semplice nel 1958.
- Nel 1969, **Roger Hindley** ha esteso l'algoritmo dimostrando che restituisce il tipo più generale
- Nel 1978 **Robert Milner** introduce in modo indipendente un algoritmo, denominato W, per il linguaggio ML. L'algoritmo è in seguito dimostrato essere equivalente a quello proposto da Hindley.
- Nel 1982 **Luis Damas** dimostra la completezza dell'algoritmo per ML.

15

Inferenza di tipo e type checking



- 🔗 Java, C, and C++, C# utilizzano un meccanismo di type checking
 - Le annotazioni di tipo sono esplicite
- 🔗 ML, Ocaml, F#, Haskell utilizzano l'inferenza di tipo (ma lo usano anche C# 3.0, Visual Basic .Net 9.0)
 - Il compilatore determina il tipo più generale (*the most general type*).

16



Statico vs Dinamico

- JavaScript: controllo di tipo dinamico

```
js> var f= 3;  
js> f(2);  
typein:3: TypeError: f is not a function  
js>
```

- ML: controllo di tipo statico
f(x) f : A → B <fun> e x : A



Controlli statici e dinamici

- 🦋 **Controllo dinamico**: la macchina astratta deve controllare che ogni operazione sia applicata a operandi del tipo corretto
 - Overhead in esecuzione
- 🦋 **Controllo statico**: i controlli vengono effettuati dal compilatore prima della generazione del codice
 - Efficienza dovuta all'analisi statica
 - Prezzo da pagare: progettazione del linguaggio e compilazione più lenta



Consideriamo il seguente frammento di programma (ML-like)

```
let x = 1 in
  if (0 = 1) then x = "errore"
  else x = 5
```

Il sistema di tipo di ML invece lo segnala come non corretto

Ma il frammento non causa mai un errore per l'uso scorretto della variabile x

19



Esiste un metodo generale per stabilire se un generico programma può causare un errore di tipo a tempo di esecuzione?

Consideriamo il seguente programma, in cui assumiamo che P non possa causare errori di tipo:

```
int x;
P; //invocazione della procedura P
x = "errore"
```

Se esistesse un metodo generale, lo potremmo applicare, e potremmo decidere se P termina oppure no. **Ma questo è impossibile, come vedrete il prossimo anno.**

20

Linguaggi e Tipi



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

21

Polimorfismo



- Idea: una operazione può essere applicata ad espressioni di tipo diverso. Vari tipi:
 - **Polimorfismo ad hoc**: un operatore causa l'esecuzione di codice diverso a seconda del tipo degli argomenti
 - **Polimorfismo parametrico**: un operatore ha uno o più parametri di tipo (espliciti o impliciti)
 - **Polimorfismo di sottotipo**: un operatore con parametro di tipo T può essere applicato a un argomento di tipo T' se T' è sottotipo di T

22

Esempi di polimorfismo



- **Polimorfismo parametrico**
 - OCaml: staticamente, con inferenza dei tipi.
 - Se $f: 'a \rightarrow 'a$ allora $f(3): \text{int}$, $f(\text{true}): \text{bool}$, $f(\text{"ciao"}): \text{string}$
 - Java: metodi/tipi generici - staticamente, con parametri di tipo espliciti
- **Polimorfismo da sottotipo**
 - C++, Java, Eiffel, C#,...
 - basato su gerarchie di tipi
 - Java: gerarchia determinata da **extends** e **implements**

23

Esempi di polimorfismo



- **Polimorfismo ad hoc:** Uno stesso simbolo può corrispondere ad algoritmi differenti, a seconda del numero e/o del tipo degli argomenti
 - Operatori primitivi del linguaggio
 - Es: "+" in Java - somma di int, di double, ..., concatenazione di stringhe. Codice da eseguire determinato staticamente
 - Overloading: metodi con stesso nome ma firma diversa
 - Risolto staticamente in Java
 - Overriding: metodi con stessa firma in classi in relazione gerarchica
 - Risolto dinamicamente in Java con dynamic dispatching

24

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 strutture dati utilizzate nella esecuzione 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 permette)
- ✎ Tratteremo insieme le due classi anche se il componente "dati" del linguaggio comprende ovviamente solo i tipi di dato di programma

25

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



- ✎ Una collezione di valori
 - rappresentati da opportune strutture dati + un insieme di operazioni per manipolarli
- ✎ Come sempre ci interessano due livelli
 - Semantica
 - Implementazione

26

I Descrittori di Dato



- ✎ Obiettivo: rappresentare una collezione di valori utilizzando quanto ci viene fornito da un linguaggio macchina, cioè
 - un po' di tipi numerici, caratteri,
 - sequenze di celle di memoria
- ✎ Qualunque valore della collezione è alla fine una stringa di bits
- ✎ Problema: per poter riconoscere il valore e interpretare correttamente la stringa di bits
 - è necessario (in via di principio) associare alla stringa un'altra struttura che contiene la descrizione del tipo (*descrittore di dato*), che viene usato quando 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

27

Tipi a tempo di compilazione e a tempo di esecuzione



- ✎ Se l'informazione sul tipo è conosciuta completamente "a tempo di compilazione" (come in OCaml)
 - si possono eliminare i descrittori di dato
 - il type checking è effettuato totalmente dal compilatore (type checking statico)
- ✎ Se l'informazione sul tipo è nota solo "a tempo di esecuzione" (come in JavaScript)
 - sono necessari i descrittori per tutti i tipi di dati
 - il type checking è effettuato totalmente a tempo di esecuzione (type checking dinamico)
- ✎ Se l'informazione sul tipo è conosciuta solo parzialmente "a tempo di compilazione" (come in Java)
 - i descrittori di dato contengono solo l'informazione "dinamica"
 - il type checking è effettuato in parte dal compilatore e in parte dal supporto a tempo di esecuzione

28



INTERMEZZO

29

Termini



- Strutture ad albero composte da
 - simboli di variabile
 - simboli di funzione con corrispondente arietà n ($n \geq 0$)
 - ✓ un simbolo con arietà 0 è chiamato *costante*
- Un termine è
 - un simbolo di variabile o di costante
 - un simbolo di funzione n -aria applicato ad n termini
- I valori di un tipo ML definito per casi sono termini senza variabili (*ground*). Esempio:

```
type intlist = Empty | Cons of int * intlist
```

Definisce costante `Empty` e simbolo di funzione `Cons` di arietà 2
Esempi di termini (con costanti di tipo `int`):
`Cons(3,Empty), Cons(1, Cons(2,Empty))`

30

Termini con variabili e pattern matching



- Termini con variabili (come `Cons (n, m)`) “rappresentano” insiemi potenzialmente infiniti di strutture dati
- Pattern matching**
 - Confronta un **pattern p** (termine con variabili tutte distinte) con un termine **t** senza variabili
 - Cerca una **sostituzione**, cioè una funzione dalle variabili di **p** ai sottotermini di **t**
 - ✓ Se non esiste, il matching fallisce
 - ✓ Altrimenti ha successo, e le variabili vengono istanziate con la sostituzione in quel che segue
 - Esempio: il costrutto `match...with` in Ocaml

```
let add x = match x with
  (y, z) -> y + z ;;
val add : int * int -> int = <fun>
```
- Una funzione che restituisce un sottotermini del suo argomento è un **selettore**. Si possono definire facilmente usando il pattern matching

31



TIPI SCALARI

32



Tipi scalari (esempi)

Booleani

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

Caratteri

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

33



Tipi scalari (esempi)

Interi

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

Reali

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

34



Tipi scalari (esempi)

- Il tipo `void` (`void` in Java, `unit` in Ocaml)
 - ha un solo valore
 - nessuna operazione
 - serve per definire il tipo di operazioni che modificano lo stato senza restituire alcun valore
 - In Java:

```
void f (...) {...}
```

il valore restituito da `f` di tipo `void` è sempre il solito (e dunque non interessa)
 - In OCaml:

```
let f x = (x := !x + 1);;  
val f : int ref -> unit = <fun>
```

35



Tipi composti

- Record**
 - collezione di campi (fields), di tipi anche diversi
 - un campo è selezionato col suo nome
- Record varianti**
 - record dove solo alcuni campi (mutuamente esclusivi) sono attivi ad un dato istante
- Array**
 - funzione da un tipo indice (scalare) ad un altro tipo
 - array di caratteri sono chiamati stringhe; operazioni speciali
- Insiemi**
 - sottinsieme di un tipo base
- Puntatori**
 - riferimento (*reference*) a un oggetto di un altro tipo

36

Records



- Introdotti per manipolare in modo unitario dati di tipo eterogeneo
- C, C++, CommonLisp, Algol68, Pascal, ...
- Java: non ha tipi record, sussunti dalle classi
- Esempio, in C:

```
struct studente {
    char nome[20];
    int matricola; };
```
- Selezione di campo:

```
studente s;
s.matricola=343536;
```
- Record possono essere annidati
- memorizzabili, esprimibili e denotabili
 - ✓ Pascal non ha modo di esprimere “un valore record costante”
 - ✓ C lo può fare, ma solo nell’inizializzazione (initializer)
 - ✓ uguaglianza generalmente non definita (contra: Ada)

37

Records: implementazione



- 🦋 memorizzazione sequenziale dei campi
- 🦋 allineamento alla parola (16/32/64 bit)
 - accesso efficiente: posizione dei campi determinata staticamente
 - possibile spreco di memoria
- 🦋 Pudding o packed records
 - disallineamento
 - accesso più costoso: posizione di un campo va calcolata a runtime

38

Record implementazione



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

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

39

Record Implementazione



```
// effettivo "memory layout" (C COMPILER)  
struct x_  
{  
    char a;    // 1 byte  
    char _pad0[3]; // padding 'b' su 4 byte  
    int b;    // 4 bytes  
    short c;  // 2 bytes  
    char d;    // 1 byte  
    char _pad1[1]; // padding sizeof(x_)  
                // multiplo di 4  
};
```

40

Arrays



- ☞ 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

41

Array: implementazione



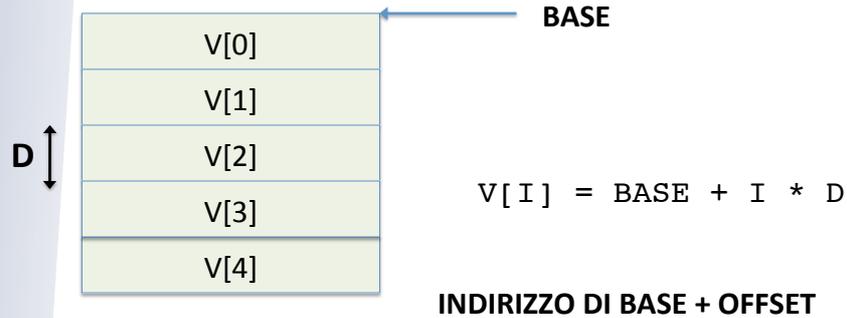
- ☞ Elementi memorizzati in locazioni contigue:
 - ordine di riga: `V[1,1];V[1,2];...;V[1,10];V[2,1];...`
 - ✓ maggiormente usato;
 - ordine di colonna: `V[1,1];V[2,1];V[3,1];...;V[10,1];V[1,2];...`
- ☞ Formula di accesso (caso lineare)
 - Vettore `V[N]` of `elem_type`
 - $V[I] = \text{base} + c * I$,
dove `c` è la dimensione per memorizzare un `elem_type`
- ☞ Formula di accesso (più articolata) puo' essere stabilita anche per gli array multidimensionali (dettagli nel libro di testo)

42



Accesso Array: esempio

`int v[5]`



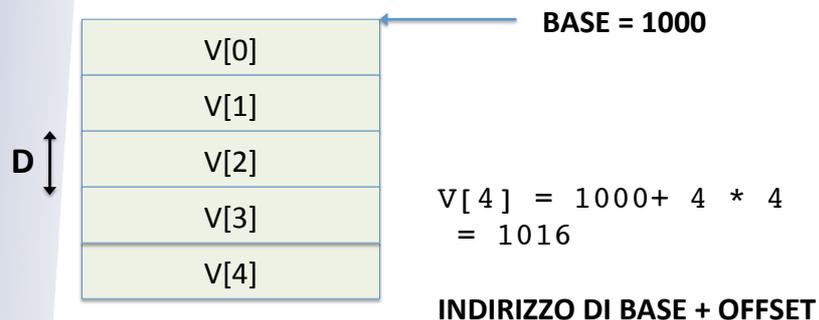
D dimensione in byte del tipo di base

43



Accesso Array: esempio

`int v[5]`



D dimensione in byte del tipo di base = 4 byte

44



Il caso del C

- ✎ Il C non prevede controlli a run-time 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 il vettore all'indice 40, il run-time non rileverà l'errore e fornirà un accesso scorretto alla locazione di memoria 1080.

45



Puntatori

- ✎ Valori : riferimenti; costante **null** (**nil**)
- ✎ Operazioni:
 - creazione
 - ✓ funzioni di libreria che alloca e restituisce un puntatore (eg, **malloc**)
 - dereferenziazione
 - ✓ accesso al dato "puntato": ***p**
 - test di uguaglianza
 - ✓ in specie test di uguaglianza con **null**

46



Array e puntatori in C

- 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 NB: aritmetica dei puntatori  
n = b[3];  // idem  
n = *(b+3); // idem
```

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

47



TIPI DI DATO DI SISTEMA

48

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
NB: Gli argomenti di emptystack determinano la lunghezza massima
della pila e il tipo degli elementi
```

49

Pila non modificabile: semantica



```
# module SemPila: PILA =
struct
  type 'a stack = Empty of int | Push of 'a stack * 'a (*tipo algebrico *)
  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
```

50

semantica algebrica



```
'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
```

Semantica “isomorfa” ad una
specifica in stile algebrico

semantica delle operazioni definita
da un insieme di equazioni fra
termini

il tipo di dato è un'algebra (iniziale)

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)
  (* crea un array di lunghezza nm, inizializzando gli elementi a x *)
  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
```

52

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
 - (astrazione della) memoria fisica in una implementazione in linguaggio macchina
- classica implementazione sequenziale
 - utilizzata anche per altri tipi di dato simili alle pile (code)

53

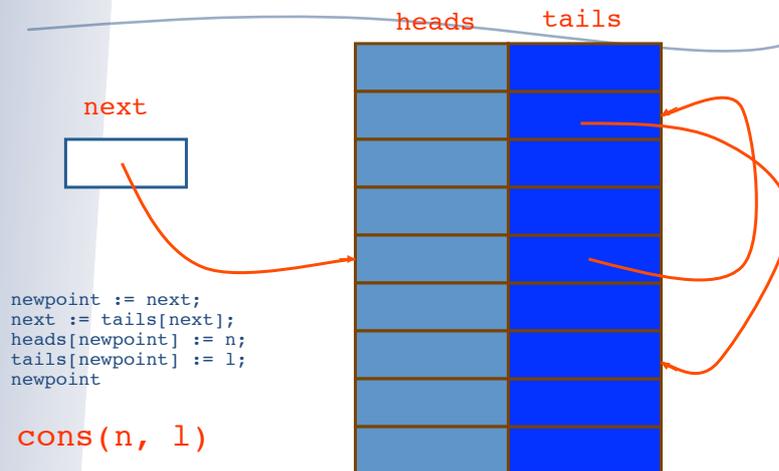
Lista (non polimorfa): interfaccia



```
# 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
```

54

Heap, lista libera, allocazione



55

Lista: implementazione a heap



```
# module ImpListaInt: LISTAINT =
  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 1; 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
```

56

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 lungh : 'a stack -> int
  val svuota : 'a stack -> unit
  val access : 'a stack * int -> 'a
  exception Emptystack
  exception Fullstack
  exception Wrongaccess
end
```

57

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 nofpops = lungh(x) - 1 - n in
    if nofpops < 0 then raise Wrongaccess else faccess(!x, nofpops)
end
```

58

Pila modificabile: implementazione

```
module ImpPila: 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 lugh( (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
  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

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 classe



- il meccanismo di C++ e Java (anche OCAML)
- il tipo è la classe
 - Parametrico, 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

66



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