Strutturare i dati

I tipi di dato

Tipo di dato

- collezione di valori omogenei
- ad ogni tipo di dato si associa:
  - un insieme di operazioni
  - una rappresentazione a basso livello, usata dal compilatore-interprete

A cosa servono i tipi?

Scrittura del codice: organizzazione concettuale dei dati

- divido i dati in diverse categorie
- ad ogni categoria associo un tipo
- tipi come “commenti formali” su identificatori, espressioni

Analisi semantica: identificano e prevengono errori

- controllo che i dati siano usati coerentemente con il loro tipo
- controllo automatico
- costituiscono un “controllo dimensionale”:
  - 3+“Pippo” può essere sbagliato

Generazione del codice: determinano l’allocazione dei dati in memoria

Il sistema di tipi di un linguaggio:

- tipi predefiniti
- meccanismi per definire nuovi tipi
- meccanismi relativi al controllo dei tipi:
  - equivalenza
  - compatibilità
  - inferenza
  - polimorfismo

Sistemi di tipo statici o dinamici

Quando avviene il controllo di tipo, type checking :

- statico: a tempo di compilazione,
  si eseguono gran parte dei controlli __ C, Java, Haskell
- dinamico: durante l'esecuzione del codice
  Python, JavaScript, Scheme

Separazione non netta,

- quasi tutti i linguaggi fanno dei controlli dinamici
- alcuni controlli di tipo (es. dimensione degli array), possono avvenire solo a tempo di esecuzione.

Statici: vantaggi e svantaggi

- vengono anticipati gli errori
- maggiore efficienza: meno controlli di tipo a tempo di esecuzione, ma alcuni fanno fatti
- a volte più prolissi, bisogna introdurre informazioni di tipo nel codice,
  (ma si documenta meglio il codice)
  alcuni linguaggi (Python, ML) usano type inference per rendere le dichiarazioni di tipo facoltative
- meno flessibili, per prevenire possibili errori di tipo, si impedisce codice perfettamente lecito,

```
(define (mix g) (cons (g 7) (g #t)))
(define pair_of_pairs (mix (lambda (x) (cons x x))))
```

- a costo di una certa complessità, programmi equivalenti, possono avere tipo statico

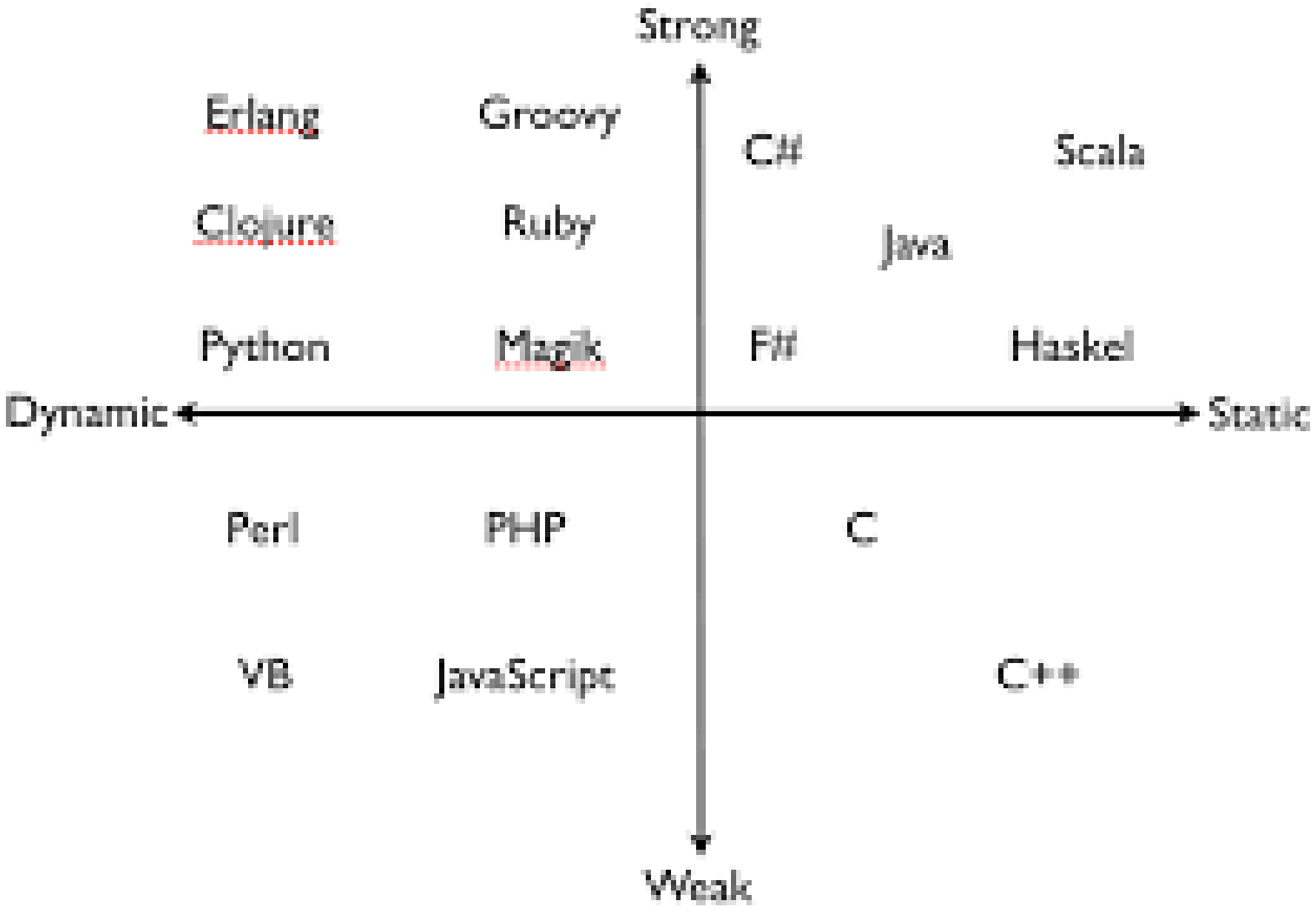
Dinamici: vantaggi e svantaggi

- bisogna eseguire il codice per trovare l'errore
  (i test vanno comunque fatti anche per i s.t. statici)
- informazioni sul tipo da inserire nei dati e controlli di tipo da fare a tempo di esecuzione
  (test non pesanti, ottimizzazione possibile)
- più concisi (meno definizioni di tipo nel codice)
- più flessibili

Strong - weak type system

- Strong type system impediscono, rendono difficile, che errori di tipo non vengano rilevati
  (type safe)
- Weak type system permettono una maggiore flessibilità a costo della sicurezza,
  - errori di tipo
    es.: uso una sequenza di 4 caratteri come un numero intero possono essere non rilevati (né al tempo di esecuzione, né al tempo di compilazione)

Concetti indipendenti:



Una catalogazione dei tipi e dei loro valori:

Possiamo distinguere i tipi in:

- denotabili: con valori rappresentabili mediante identificatori
- esprimibili: con valori rappresentabili mediante espressioni (diverse da identificatori - dichiarazioni)
- memorizzabili: con valori che possono essere salvati in memoria, assegnati a una variabile

Linguaggi diversi fanno scelte diverse

- se un specifico tipo sia denotabile, esprimibile memorizzabile:
- (es. funzioni)

Tipi predefiniti, scalari

Scalari: i valori possono essere ordinati, messi in scala

Booleani

val: true, false (Java), (Scheme: `#t, #f`)
op: or, and, not, condizionali
repr: un byte
note: C non ha un tipo Boolean

Caratteri

val: `a,A,b,B, ..., è,é,ë, ; , ‘,`
op: uguaglianza; code/decode; dipendenti dal linguaggio
repr: un byte: (ASCII - C), due byte: (UNICODE - Java), lunghezza variabile: (UTF8 - opzione in Java)
codifiche a lunghezza variabile rendono complesso accedere ai singoli caratteri di una stringa, tramite indice: necessario usare funzioni di libreria

Tipi scalari

Interi

val: 0,1,-1,2,-2,. . . ,maxint

op: +, -, *, mod, div, . . . , più un buon numero di funzioni definite in varie librerie.

repr:

- alcuni byte (1, 2, 4, 8, 16), complemento a due
  - in alcuni linguaggi, più tipi interi per specificare il numero di byte della rappresentazione
    C: (`byte` o `char`, `short`, `int`, `long`, `long long`) C
    Go: (int8, int16, int32, int64)
  - spesso possibile specificare versione unsigned: binario puro
    Go: (uint8, uint16, uint32, uint64)
  - Scheme e altri linguaggi dispongono di interi di dimensione illimitata.

Tipi scalari

Floating point

val: valori razionali in un certo intervallo
op: +, -, *, /, . . . , più un buon numero di funzioni definite in varie librerie.
repr: alcuni byte (4, 8, 16?); (`float`, `double`, `Quad` o `float128`), notazione standard IEEE-754, virgola mobile

- Nota: alcuni linguaggi non specificano la lunghezza della rappresentazione usata per interi e floating-point, questo può creare problemi nella portabilità tra compilatori diversi.

Tipi numerici

Tipi numerici non supportati dall'hardware,
non presenti in tutti i linguaggi:

- Complessi
  - repr: coppia di reali
  - presenti in Scheme, Ada, definiti tramite librerie in altri linguaggi.
- Fixed point
  - val: razionali ma non in notazione esponenziale
  - repr: alcuni byte (2 o 4); complemento a due o BCD (Binary Coded Decimal), virgola fissa
  - presenti in Ada.
- Razionali
  - rappresentazione esatta dei numeri frazionari
  - repr: coppie di interi
  - presenti in Scheme

Tipi numerici in Scheme

Scheme considera 5 insiemi di numeri

```
number
complex
real
rational
integer
```

ciascuno sovrainsieme dei sottostanti

- rappresentazione interna nascosta, lasciata all'implementazione
  per gli interi normalmente rappresentazione a lunghezza arbitraria
- si distingue tra `exact` (integer, rational)
  e `inexact` numbers (real, complex).

Tipi Scalari

Void

- val: un solo valore: * oppure ()
- op: nessuna operazione
- repr: nessun bit

permette di trattare procedure, comandi
come casi particolari di funzioni, espressioni

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

la procedura `f` vista come funzione,
deve restituire un valore (e non nessuno)

il valore restituito da `f`, di tipo void, è sempre lo stesso,
non è possibile e non serve specificarlo nell'istruzione `return`

Una prima classificazione

Tipi ordinali (o discreti):

- booleani, interi, caratteri
- ogni elemento possiede un succ e un prec (eccetto primo/ultimo)
- altri tipi ordinali:
  - enumerazioni
  - intervalli (subrange)
- uso
  - vi si può “iterare sopra”
  - indici di array

Tipi scalari non ordinali

- floating-point, razionali

Enumerazioni

- introdotti in Pascal
-

```
type Giorni = (Lun,Mar,Mer,Giov,Ven,Sab,Dom);
```

- i programmi diventano più leggibili,
  - non codifico il giorno della settimana con un intero, uso una rappresentazione leggibile
- valori ordinati: `Mar < Ven`
- iterazione sui valori: `for i:= Lun to Sab do ...`
- op: `succ, pred`, confronto <=
- rappresentati con un byte (`char, byte`)

Enumerazioni

In C, Java:

```
enum giorni = {Lun,Mar,Mer,Giov,Ven,Sab,Dom};
```

In C la definizione precedente è equivalente a

```
typedef int giorni;
const giorni Lun=0, Mar=1, Mer=2, ...,Dom=6;
```

- Java distingue tra `Mar` e `1`,
  valori non compatibili di tipi distinti
- C non distingue

Intervalli (subrange)

- introdotti in Pascal, non presenti in in C, Scheme o Java
  in Java implementabili come sottoggetti
- i valori sono un intervallo dei valori di un tipo ordinale (il tipo base dell'intervallo)
- Esempi:

```
type MenoDiDieci = 0..9;
type GiorniLav = Lun..Ven;
```

- rappresentati come il tipo base
- perché usare un tipo intervallo invece del suo tipo base:
  - documentazione "controllabile" (con type checking dinamico)
  - potenzialmente è possibile risparmiare memoria,

Tipi composti, o strutturati, o non scalari

Record o struct

- collezione di campi (field), ciascuno di un (diverso) tipo
- un campo è selezionato col suo nome

Record varianti o union

- 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

Tipi composti, o strutturati, o non scalari

Insieme

- sottoinsieme di un tipo base

Puntatore

- riferimento (reference) ad un oggetto di un altro tipo

Funzioni, procedure, metodi, oggetti.

Structure o Record

- raggruppare dati di tipo eterogeneo
- C, C++, CommonLisp, Algol68: struct (structure)
- Esempio, in C:

```
struct studente {
char nome[20];
int matricola; };
```

- Selezione di campo:

```
struct studente s;
s.nome = "Mario";  // errore in C
strcpy(s.nome, "Mario");
s.matricola=343536;
```

- i record possono essere annidati

Structure in Java

- Java: non ha tipi record, sussunti dalle classi
  record: classe senza metodi
- le definizioni precedenti sono simulate in Java come:

```
class Studente {
public String nome;
public int  matricola;
}

Studente s = new Studente();
s.nome = "Mario";
s.matricola = 343536;
```

Structures - Record

- memorizzabili, denotabili, ma non sempre esprimibili
  - non è generalmente possibile scrivere un'espressione, diversa da un identificatore, che definisca un record
  - C lo può fare, ma solo nell'inizializzazione di variabile record,
  - uguaglianza generalmente non definita (eccezione: Ada)
  - struct sono valori esprimibili in Scheme

```
struct studente s = {"Mario", 343536};
```

Struct in Scheme

Posso implementare i record come liste e definire le funzioni per:

- costruire i record
- accedere ai campi
- testare il tipo di record

```
(define  (book title authors)  (list 'book title authors))
(define  (book-title b) (car (cdr b)))
(define  (book-? b) (eq? (car b) 'book))
```

usate come

```
(define bazaar
  (book
   "The Cathedral and the Bazaar"
   "Eric S. Raymond" ))
(define titoloBazar (book-title bazaar))
```

Generazione automatica: MIT-Scheme

Le funzioni per gestire i record possono essere create in modo automatico

parte non standard, dipendente dall'implementazione

MIT-Scheme:

```
(define-structure book title authors)
(define bazaar
  (make-book
   "The Cathedral and the Bazaar"
   "Eric S. Raymond" ))
(define titoloBazar (book-title bazaar))

(book? bazaar)
```

Costruttori automatici: Racket

Racket:

```
(struct  book (title authors))

(define bazaar
  (book
   "The Cathedral and the Bazaar"
   "Eric S. Raymond" ))
(define titoloBazar (book-title bazaar))

(book? bazaar)
```

Costruttori automatici

L'esempio studente diventa

```
(struct studente (nome matricola))

(define s (studente "Mario" 343536))

(studente-nome s)
```

- la definizione `(define-struct studente (nome matricola))` porta alla creazione delle opportune funzioni di
  - creazione `make-studente`
  - selezione campo `studente-nome`
  - test `studente?`

Record: memory layout

- memorizzazione sequenziale dei campi
- allineamento alla parola (4, 8 byte)
  - spreco di memoria

```
struct record {char init; int matr; char last};
```

- packed record
  - disallineamento
  - accesso più complesso (in assembly)
- il riordino dei campi può permettere:
  - risparmio di spazio
  - mantenimento dell'allineamento delle parole di memoria

```
struct record {char init, last; int matr;};
```

Union - Record con varianti

In un record variante alcuni campi sono alternativi tra loro:
a seconda del dato, solo alcuni

```
type studente = record
nome : packed array [1..6] of char;
matricola: integer;
case fuoricorso : Boolean of
    true: (ultimoanno: 2020..maxint);
    false:(anno:(primo,secondo,terzo);
           inpari:Boolean;)  end;
var s: studente;
s.fuoricorso := true;
s.ultimoanno:= 2021;
```

- I due campi (le due varianti) `ultimoanno` e `anno` possono condividere la stessa locazione di memoria
- il campo case, tag (`fuoricorso`) può avere un qualsiasi tipo ordinale

Record varianti: memory layout

```
type studente = record
nome : packed array [1..6] of char;
matricola: integer;
case fuoricorso : Boolean of
    true: (ultimoanno: 2000..maxint);
    false:(anno:(primo,secondo,terzo);
           inpari:Boolean;)
end;
```

più versioni di tipo `studente`:

- alloco la dimensione massima
- a seconda della classe di `studente`, stessa zona di memoria usata per tipi di dato diversi

Union type

Tipi unione sono presenti in molti linguaggi,

```
union Data {
    int i;
    float f;
    char str[20];
};


union Data y;

y.i = 3;
```

`y` può assumere tipi diveri

Record varianti

Possibili in molti linguaggi C: union + structure

```
struct studente {
    char nome[6];
    int matricola;
    bool fuoricorso;
    union {
        int ultimoanno;
        struct { int anno;
                 bool inpari;} studInCorso;
       }campivarianti  };
```

- Pascal (Modula, Ada) unisce unioni e record con eleganza
  - in Pascal: `s.anno`
  - in C: `s.campivarianti.studInCorso.anno`
    nessuna correlazione esplicita tra campo `fuoricorso` e campi 'varianti'

Union, Variant e type safety

I tipi unione permettono facilmente di aggirare i controlli di tipo

```
union Data {
    int i;
    float f;
    char str[20];
} data;
float y;
...
data.str = "abcd";
y = data.f;
```

- la codifica ASCII di “abcd”, trattata come numero reale, senza alcun avviso da parte del sistema

Type safety, motivazioni

Variant del Pascal con il campo case, agevolano la scrittura di codice di controllo ma

- nessuna garanzia
- nessun controllo forzato
- vincoli di tipo facilmente aggirabili

Java non prevede i tipi unione.

Motivazione dei tipi unione,

- risparmio di spazio, attualmente meno importante
- alcuni tipi naturalmente descritti da tipi unione
  - lista: lista vuota o elemento seguito da lista
  - albero binario: foglia o (nodo, sottoalbero sinistro, sottoalbero destro)
  -
- poter interpretare la stessa stringa di bit in modi differenti (operazione a basso livello)

Versioni sicure dei tipi unione

Algol 68, Haskell, ML, Rust:

- esistono tipi equivalenti ai tipi unione più alternative possibili
- ogni alternativa viene etichettata
- analizzo i tipi unione attraverso un costrutto `case`,
  - per ogni possibile caso, definisco il codice che lo gestisce.

```
case data in
(int i) : a = i + 1
(float j) : x = sqrt(j)
```

Rust example

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String)}

fn process_message(msg: Message) {
    match msg {
        Message::Quit => println!("Quit variant"),
        Message::Move { x, y } => println!("Move to x: {}, y:
        Message::Write(text) => println!("Text message: {}", t
}}
```

Union type in Rust

```rust
union Data {
    f1: u32,
    f2: f32,
}

fn main() {
    let u = Data { f1: 1 };
    unsafe {
        println!("f1: {}", u.f1); // Outputs 1
        println!("f2 as float: {}", u.f2);
    }
}
```

Struct (e quasi Union) in Java con le sottoclassi

```
enum AnnoCorso {PRIMO, SECONDO, TERZO};
class Studente {
   public String  nome;
   public int matricola;
}
class StudenteInCorso extends Studente {
    public AnnoCorso anno;  }
class StudenteFuoriCorso extends Studente {
    public int ultimoAnno; } ....
      StudenteInCorso mario = new StudenteInCorso();
      mario.nome = "Mario";
      mario.matricola = 456;
      mario.anno = AnnoCorso.TERZO;
```

- non posso trasformare `mario` in un `StudenteFuoriCorso`
- non posso definire funzioni con parametro un `StudenteInCorso` o un `StudenteFuoriCorso`

Array

Collezioni di dati omogenei:

- funzione da un tipo indice al tipo degli elementi, rappresentata in modo estensionale
- indice: tipo ordinale, discreto
- elemento: “qualsiasi tipo memorizzabile” (raramente un tipo funzionale)

Dichiarazioni

```
int vet[30];      // tipo indice: tra 0 e 29, C
int[] vet;        // Java, notazione preferita
int vet[];        // Java, notazione alternativa
var vet : array [0..29] of integer;  // Pascal
```

Array multidimensionali

Due possibili definizioni:

- array con più indici
- array con elementi altri array

In Pascal le due definizioni sono equivalenti

```
var mat : array [0..29,'a'..'z'] of real;
var mat : array [0..29] of array ['a'..'z'] of real;
```

Possibili entrambe ma non sono equivalenti in Ada:

- la seconda permette slicing (selezionare una parte di array)

C e Java non posso dichiarare array con più indici.

```
int mat[30][26]    // C
int[][] mat        // Java
```

In Python, array multidimensionali solo con la libreria numpy

Array: operazioni

Principale operazione permessa:

- selezione di un elemento: `vet[3]` `mat[10,'c']` `mat[10][12]`
  - l'elemento può essere letto o modificato

Alcuni linguaggi permettono slicing:

- selezione di parti contigue di un array
- Esempio: in Ada, con

```
mat : array(1..10) of array (1..10) of real;
```

`mat(3)` indica la terza riga della matrice quadrata `mat`

- possibile anche in C `mat[3]`
- slicing più sofisticati, non necessariamente intere righe, Python

```
y=(1,2,3,4)
x = y[:-1]
```

Calcolo vettoriale

- l'assegnazione tra array
  - normalmente per riferimento
  - per valore disponibile solo in alcuni linguaggi
    implementabile con un ciclo di assegnazioni
- operazioni vettoriali, operazione aritmetiche estese agli array
- usate in: calcolo scientifico, grafica, criptografia
- presenti anche in assembly.
- Fortran90: A+B somma gli elementi di A e B (dello stesso tipo)

Memorizzazione degli array

Elementi memorizzati in locazioni contigue.

Per array multidimensionali, tre alternative:

- dati contigui:
  - ordine di riga:
    `V[0,0] V[0,1] ...V[0,9] V[1,0] ...`
    maggiormente usato;
    le righe, sub-array di un array, memorizzate in locazioni contigue
  - ordine di colonna:
    `V[0,0] V[1,0] V[2,0]... V[13,0];V[0,1];...`
- array multidimensionale come vettore di puntatori (Java)

Ordine rilevante per efficienza in sistemi con cache, per algoritmi di scansione del vettore

- vettore memorizzato per righe, scansione per colonne:
  - esamino punti distanti in memoria
  - genero un numero più alto di cache miss

Array: calcolo indirizzi

Calcolo locazione corrispondente a `A[i,j,k]` (per riga)

- `i` indice piano, `j` riga, `k` colonna
- gli indici partono da `0`
- `A : array[l1, l2, l3]` of `elemType`
- `elemType A[l1][l2][l3]`
  - `S3`: dimensione di (un elemento di) `elem_type`
  - `S2` = `l3 * S3` dimensione di una riga
  - `S1` = `l2 * S2` dimensione di un piano
- locazione di `A[i,j,k]` è:

```
a              indirizzo di inizio di A
+ i*S1      = ind di inizio del piano di A[i,j,k], sottoma
+ j*S2      = ind di inizio della riga di A[i,j,k], sottov
+ k*S3      = ind di A[i,j,k]
```

Array: shape

- Forma (o shape): numero di dimensioni e intervalli dell'indice per ogni dimensione

Può essere determinata in vari istanti:

- forma statica (definita a tempo di compilazione)
- forma fissata al momento dell'esecuzione della dichiarazione, chiamata della procedura e definizione delle variabili locali (per vettori locali a una procedura)
- forma dinamica, le dimensioni variano durante l'esecuzione (Python, JavaScript)

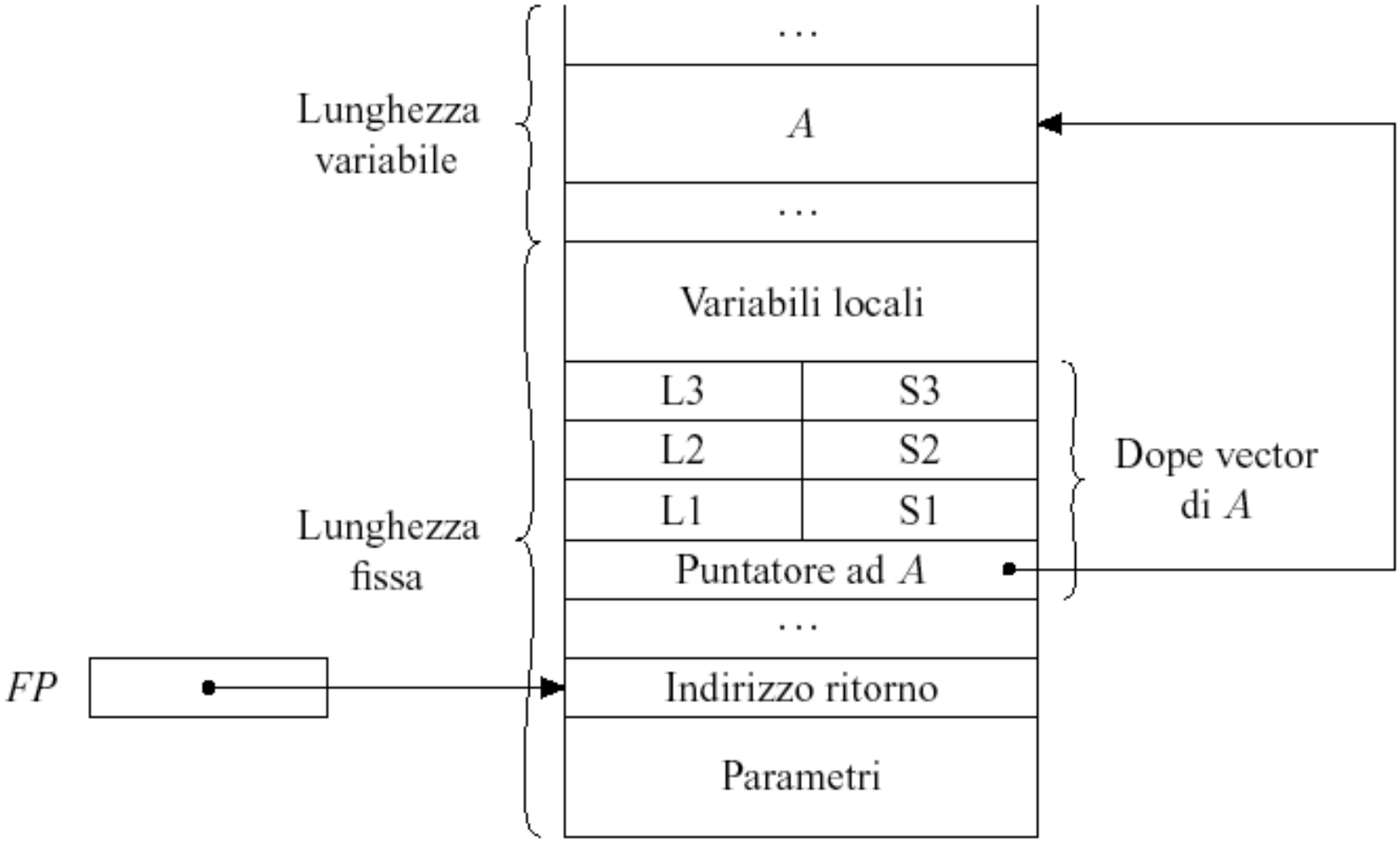
Dope vector (descrittore del vettore)

Come si accede a un vettore?

- Con forma statica, informazioni sulla forma dell'array è mantenuta dal compilatore
- Se la forma non statica, info mantenuta in un descrittore dell'array detto dope vector che contiene:
  - puntatore all'inizio dell'array (nella parte variabile)
  - numero dimensioni
  - limite inferiore (se a run-time vogliamo controllare anche l'out-of-bound anche limite superiore)
  - occupazione per ogni dimensione (valore Si)
- Il dope vector è memorizzato nella parte fissa del RdA
- Per accedere al vettore, si calcola l'indirizzo a run-time, usando il dope vector

Esempio: RdA con dope vector

```
A : array[l1..u1, l2..u2, l3..u3] of elem_type
```



Memorizzazione dell'array

Dove memorizzare un array dipende dalla forma:

- Statica: RdA
- Fissata al momento della dichiarazione: dope vector nella parte iniziale, vettore in fondo al RdA
- Forma dinamica: heap, solo il dope vector nel RdA

Per Java:

- gli array sono oggetti
- memorizzati nella heap
- la definizione di una variabile vettore non alloca spazio
- spazio allocato con una chiamata a `new`

```
int[] vettore = new int[4];
```

- array multidimensionali: memorizzati come vettori di puntatori

Array e sicurezza

Controllo degli indici di un array

- controllo necessariamente dinamico
- svolto solo da alcuni linguaggi
- importante per:
  - type safety: accedo a zone arbitrarie della memoria con tipo sbagliato
  - sicurezza del codice, buffer overflow:
    - istruzioni di scrittura in un vettore con indici fuori range
    - possono scrivere in zone arbitrarie del RdA
    - cambiando gli indirizzi di ritorno della procedura
    - si può di eseguire codice arbitrario, es. codice scritto nel vettore

Esercizi su matrici

- In una matrice `bool A [8][5]`, memorizzata per righe, colonne, con indici che iniziano da 1, in che posizione di memoria, rispetto all'indirizzo base, `b`, si trova l'elemento `A[2][4]`. Si supponga che ogni intero richieda 4 locazioni di memoria.
- In una matrice `int A [4][8][16]`, memorizzata per piani, righe, colonne, con indici che iniziano da 0, in che posizione di memoria, rispetto all'indirizzo base, b, si trova l'elemento `A[2][4][8]`. Si supponga che ogni intero richieda 4 locazioni di memoria.
- In una matrice quadrata 5x5, quali elementi sono memorizzati sempre nella stessa posizione, sia che la matrice sia memorizzata per righe che per colonne?
  - In una matrice rettangolare 2x4.
  - 3 x 5
  - 4 x 7
  - Generalizzare

Esercizi Matrici

- Gli array multidimensionali in Java sono memorizzati come righe di puntatori (row pointer). Determinare le istruzioni assembly ARM per accedere all'elemento:
  - `A[2][4]` di un vettore `int A [4][8]`.
  - `A[2][4][8]` di un vettore `int A [4][8][16]`.

Esercizi Dimensioni dati

Nelle ipotesi di parole di memoria di 4 byte, memorizzazione a parole allineate, codice caratteri ASCII (UNICODE), quanti byte occupano i seguenti record:

```
struct {
int i;
char a[5];
int j;
char b;
}
```

```
struct {
int i;
char a[5];
char b;
int j;
}
```

Insiemi

Disponibili in diversi linguaggi
direttamente: Python, Javascript, Pascal, R, ...
tramite librerie standard: Java, Ruby, Haskell, Scala, ...

- `set of char S`
  `set of [0..99] I`
- operazioni insiemistiche: unione, intersezione, differenza, differenza simmetrica (xor), appartenenza

Implementazione degli insiemi

- mediante vettori (compatti) di booleani
  - definiscono la funzione caratteristica
  - le operazioni insiemistiche ottenute applicando point-wise le corrispondenti operazioni booleane
    unione $\rightarrow$ or, intersezione $\rightarrow$ and, complemento $\rightarrow$ not
- per insiemi con un universo di grosse dimensioni conveniente l'implementazione tramite tabelle hash
  chiave: l'elemento, contenuto: bit di presenza
- altenativa: linguaggi funzionali e funzioni di appartenenza
  esempio: `char -> bool`

Puntatori

identificano: l-value, indirizzi di memoria

- Valori : riferimenti (l-valori) e la costante null (o nil)
  - tutti i puntatori codificati allo stesso modo: indirizzo di memoria tipizzati in base all'oggetto puntato
- Tipi: nel tipo del puntatore specifico il tipo dell'oggetto puntato

```
int  *i  // int* i
char *a  // char* a
```

- Operazioni:
  - creazione di un oggetto puntato
    - funzioni di libreria che allocano e restituiscono l'indirizzo (es., malloc)
  - dereferenziazione
    - accesso al dato "puntato": *p
  - test di uguaglianza

Puntatori, scelte nei diversi linguaggi

L'uso dei puntatori non ha molto senso in linguaggi con modello a riferimento, vedi Java

- ogni variabile contiene un riferimento in memoria al dato associato (puntatore)

In alcuni linguaggi, i puntatori fanno riferimento solo all'heap

- in C, i puntatori posso far riferimento allo stack ra

```
int i = 5;
p = &i;
```

Puntatori e array in C

- Array e puntatori sono intercambiabili in C
  una variabile array codificata come un puntatore al primo elemento dell'array

```
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
```

- Aliasing: `a[3] += 1;`
  modificherà anche `b[3]` stessa locazione di memoria.

Aritmetica sui puntatori

- In C, sui puntatori sono possibili alcune operazioni aritmetiche, con significato diverso
  - `a+3`, incrementa il valore di `a` di `12` (`3*` dimensione intero)
    `a+3` fa riferimento a 3 interi più in là nello spazio di memoria
  - in generale gli incrementi sono moltiplicati per la dimensione dell'elemento del vettore
- Nessuna garanzia di correttezza, posso accedere a zone arbitrarie di memoria
- Array multidimensionali, sono equivalenti le espressioni:

```
b[i][j]
(*(b+i))[j]
*(b[i]+j)
*(*(b+i)+j)
```

Puntatori e tipi di dato ricorsivi

Uso principale dei puntatori:

- definire strutture dati ricorsive
  - liste
  - code
  - alberi binari
  - alberi

Nei linguaggi con puntatori, implementate con record e puntatori

```
struct int_list {
    int info;
    struct int_list *next;
}

struct char_tree {
    char info;
    struct char_tree *left, *right;
}
```

Assenza di puntatori in Java

In Java definisco direttamente il tipo ricorsivo,
nascondendo l'uso degli indirizzi di memoria
la definizione di `char_tree` diventa:

```
class Char_tree {
    char info;
    Char_tree left;
    Char_tree right;
    ...
```

Dangling reference

- Problema con l'uso dei puntatori: si fa riferimento a una zona della memoria che contiene dati di un tipo non compatibile, arbitrari
- Possibili cause
  - aritmetica sui puntatori:
    modifico arbitrariamente il valore di un puntatore
  - deallocazione dello spazio sull'heap `free(p)`
    nel caso di aliasing
  - deallocazione di RdA (vedremo esempi)
- Soluzioni
  - restringere (impedire) l'uso dei puntatori
  - nessuna deallocazione esplicita e meccanismi di garbage collection
  - oggetti puntati solo nell'heap
  - introdurre dei meccanismi di controllo nella macchina astratta

Dangling reference con deallocazione di RdA

A che zona della memoria fanno riferimento i puntatori.

- per sicurezza: l'heap
- in C non necessariamente

Copiare un vettore in un altro, come riferimento, può portare a riferimenti pendenti

```
int *ref
int vett1[2];
void proc (){
    int i = 5;
    int vett2[2];
    ref = &i;
    vett1 = vett2;
    }
```

dopo una chiamata a `proc`, `vett1` e `ref` puntano a zone deallocate della memoria

Garbage collection

Senza deallocazione esplicita necessario un programma garbage collector, per il recupero di spazi di memoria heap con dati non più utilizzati, accedibili, non riferiti da alcun puntatore

- più sicuro della deallocazione esplicita
- più pesante da implementare
  - algoritmi complessi
  - necessità di gestire informazioni sull'uso della memoria
  - l'attivazione del garbage collector rallenta l'esecuzione

Rust

Garantisce sicurezza della memoria e una garbage collection semplice ed efficiente

Nozione di ownership:

- un dato in heap appartiene a una variabile,
- che può “prestarlo” ad un’altra
- non ci sono dati in heap condivisi

```
let mascot = String::from("stringaEsempio");
let ferris = mascot;

println!("{}", mascot) // Error, mascot has without ownership
```

eventuale duplicazione esplicita:

```
let mascot = String::from("stringaEsempio");
let ferris = mascot.clone();

println!("{}", mascot) // Works
```

All'uscita di una procedura, si recuperano i dati riferiti dalle variabili locali della procedura

Inferenza di tipo

- Le dichiarazioni definiscono un tipo per ogni identificatore di variabile o costante
- Il compilatore associa un tipo alle altre parti del codice:
  - espressioni e sottoespressioni
  - blocchi di comandi
  - funzioni o procedure
- e si verifica che i dati siano usati coerentemente con il codice
- Algoritmi di type inference possono essere più o meno sofisticati:
  - tipo per le funzioni presente o meno (se le funzioni possono essere argomento)
  - in ML, Haskell non serve definire il tipo delle variabili, dedotti dal contesto
    si simula la concisione di linguaggi con type checking dinamico (es. Python)

Equivalenza e compatibilità tra tipi

Due relazioni tra tipi:

- Equivalenza: determina se due variabili hanno lo stesso tipo
  - se le espressioni di tipo `T` e `S` sono equivalenti
  - allora ogni variabile di tipo `T` è anche una variabile di tipo `S` e viceversa
    i due tipi sono perfettamente intercambiabili
    è equivalente dichiarare una variabili di tipo `T` o di tipo `S`
  - passaggio dei parametri, per riferimento, richiede equivanleza di tipo tra formale e attuale
- Compatibilità: `T` è compatibile con `S` quando
  - valori di tipo `T` possono essere usati in contesti dove ci si attende valori di tipo `S`
  - assegnamento richiede la solo compatibilita tra tipi
  - non necessariamente il viceversa

Equivalenza tra tipi: per nome

Due espressioni di tipo sono equivalenti solo se sono lo stesso identificatore di tipo

- essere espressioni di tipo uguali non è sufficiente
- esempio, con equivalenza per nome dopo

```
x : array [0..4] of integer;
y : array [0..4] of integer;
```

- le variabili `x` e `y` hanno tipi differenti
- usata in Java, Kotlin, Rust, Pascal, Ada

Equivalenza per nome lasca (loose) o stretta

Data la seguente dichiarazione, `x` e `y` hanno steso tipo?

```
type A =  ... ;
type B = A;  {* alias  *}
x : A;
y : B;
```

- equivalenza per nome stretta: no
- equivalenza per nome lasca: si
  (Pascal, Modula-2)
  - spiegazione: una dichiarazione di un tipo alias di tipo (`B = A`) non genera un nuovo tipo (`B`) ma solo un nuovo nome per `A`

Passaggio dei parametri

- il passaggio dei parametri, in genere, richiede l'equivalenza di tipo tra parametro formale e attuale
- codice come:

```
void ordina (int[4] vett);
int[4] sequenza = {0, 1, 2, 3};
...
ordina(sequenza)
```

lecito in C ma non in Pascal, Ada,
si usano equivalenze di tipo diverse

Con equivalenza per nome devo scrivere:

```
typedef int[4]  vettore
void ordina (vettore vett);
vettore sequenza = {0, 1, 2, 3};
...
ordina(sequenza)
```

Equivalenza strutturale

Due tipi sono equivalenti se hanno la stessa struttura:

Definizione

L'equivalenza strutturale tra tipi è la (minima) relazione di equivalenza tale che:

- se un tipo `T` è definito come `type T = espressione` allora `T` è equivalente a `espressione`
- due tipi, costruiti applicando lo stesso costruttore a tipi equivalenti, sono equivalenti

Equivalenza controllata per riscrittura, definizione alternativa:

- due tipi `A` e `B` sono equivalenti strutturalmente se il loro unfolding, riscritti nelle loro componenti elementari, espandendo le definizioni, generano la stessa espressione‘

Equivalenza strutturale

Esempio: i tipi `struct A` e `struct B` sono strutturalmente equivalenti

```
struct coppiaA {int i; float f};
struct A {int j; struct coppiaA c};
struct B {int j; struct coppiaB {int i; float f} c};
```

Diverse interpretazione dell'equivalenza strutturale

- Vanno considerate equivalenti?

```
struct coppia1 {int i; float f};
struct coppia2 {float f; int i};
```

Generalmente no, si per ML o Haskell.

o

```
typedef int[0..9] vettore1
typedef int[1..10] vettore2
```

Generalmente no, sì per Ada e Fortran.

Casi di equivalenze strutturali accidentali

- Equivalenza strutturale: a basso livello
  non rispetta l'astrazione che il programmatore inserisce col nome

```
type student = {
    name: string,
    address: string
}
type school = {
    name: string,
    address: string
}
type distance = float;
type weight = float;
```

- Con l'equivalenza strutturale, possiamo assegnare un valore `school` ad una variabile `student` o un valore `distance` ad una variabile `weight`.

Nei linguaggi,

- spesso si usa in miscuglio delle due equivalenze
  - a seconda del costruttore di tipo usato, si usa o meno l'equivalenza strutturale
- linguaggi recenti tendono a preferire l'equivalenza per nome

Esempi di equivalenza tra tipo

- C: equivalenza per nomi sui tipi `struct` equivalenza strutturale sul resto (array)
- equivalenza per nomi forte in Ada:

```
x1, x2: array (1 .. 10) of boolean;
```

`x1` e `x2` hanno tipi diversi

- Equivalenza strutturale in ML, Haskell:

```
type t1 = { a: int, b: real };
type t2 = { b: real, a: int };
```

`t1` e `t2` sono tipi equivalenti

Compatibilità

Un tipo `T` è compatibile con un tipo `S` quando oggetti di tipo `T` possono essere usati in contesti dove ci si attende valori di tipo `S`

- Esempio: `int` compatibile con `float`

```
int n = 5
float r = 5.2;
r = r + n;
```

- Oggetto: `n`
- Contesto: `r = r + _ ;`

Esempi di compatibilità

Quali tipi siano compatibili dipende dal linguaggio:

Esempi di casi di compatibilità, via via più laschi
`T` è compatibile con `S` se:

- `T` e `S` sono equivalenti;
- I valori di `T` sono un sottoinsieme dei valori di `S`
  es: tipi intervallo
- tutte le operazioni sui valori di `S` sono possibili anche sui valori di `T`
  es: “estensione” di record, sottoclasse
- i valori di `T` corrispondono in modo canonico a valori di `S`
  es: int e float; int long
- I valori di `T` possono essere fatti corrispondere a valori di `S`
  es: float e int con troncamento; long e int;

Conversione di tipo

Se `T` compatibile con `S` avviene una conversione di tipo.
un'espressione di tipo `T` diventa di tipo `S`

Questa conversione di tipo, nel caso di tipi compatibili, è una

- Conversione implicita coercizione, (coercion): il compilatore, inserisce la conversione, nessuna traccia nel testo del programma

Vedremo anche meccanismi di:

- Conversione esplicita, o cast, quando la conversione è implementata da una funzione, inserita esplicitamente nel testo programma.

Coercizione di tipo

L'implementazione deve fare qualcosa.

Tre possibilità, i tipi sono diversi ma:

- con stessi valori e stessa rappresentazione.

  esempio: tipi strutturalmente uguali, nomi diversi

  - il compilatore controlla i tipi, non genera codice di conversione
- valori diversi, ma stessa rappresentazione sui valori comuni.

  Esempio: intervalli e interi

  - codice per controllo di tipo (type checking dinamico)
    non sempre inserito
- valori e rappresentazione diversi. Esempio: interi e floating-point, oppure `int` e `long`

  - codice per la conversione

Cast

Il programmatore inserisce esplicite funzioni che operano conversioni di tipo

- sintassi ad hoc per specificare che un valore di un tipo deve essere convertito in un altro tipo.

```
s = (S) t
r = (float) n;
n = (int) r;
```

- anche qui, a seconda dei casi, necessario codice macchina di conversione,
- si può sempre inserire esplicitamente un cast laddove esiste una compatibilità (utile per documentazione)
- linguaggi moderni tendono a favorire i cast rispetto coercizioni (comportamento più prevedibile)
  la relazione di compatibilità più ristretta
- non ogni conversione esplicita consentita
  - solo i casi in cui linguaggio dispone funzione conversione

Polimorfismo

Una stessa espressione può assumere tipi diversi:

- distinguiamo tra varie forme di polimorfismo:
- polimorfismo ad hoc, o overloading
- polimorfismo universale:
  - polimorfismo parametrico (esplicito e implicito)
  - polimorfismo di sottotipo

Polimorfismo ad hoc: overloading

Uno stesso simbolo denota significati diversi a seconda del contesto:

```
3 + 5
4.5 + 5.3
4.5 + 5
```

- Il compilatore traduce + in modi diversi
- spesso risolto a tempo di compilazione, quando dipende dal contesto
  - dopo l'inferenza dei tipi
- Java supporta questa forma di polimorfismo
  - nelle sottoclassi ridefinisco i metodi definiti nella classe principale,
  - in base al tipo dell'oggetto si decide il metodo da applicare
  - interfaces
- Supportato in Haskell attraverso il meccanismo delle type classes
- Rust con i traits

Overloading

In molti linguaggi posso definire una stessa funzione (metodo)

- più volte
- con tipi diversi

Il linguaggio sceglie quale definizione usare in base a:

- numero di argomenti (Erlang)
- tipo e numero degli argomenti (C++, Java)
- tipo del risultato (Ada)
- tipo degli argomenti e del risultato (Haskell)

Polimorfismo parametrico

Un valore funzione ha polimorfismo parametrico quando

- ha un'infinità di tipi diversi,
- ottenuti per istanziazione da un unico schema di tipo generale

Una funzione polimorfa è definita da un unico codice
applicabile sulle diverse istanze del parametro di tipo

```
identity(x) = x; :  <T> -> <T>          ∀ T. T -> T

reverse(v) = ....; :  <T>[] -> void       ∀ T. T[] -> void
```

`T` è una variabile di tipo

Utile per il riuso del codice

Polimorfismo in Scheme.

Grazie al type-checking dinamico, Scheme è un linguaggio naturalmente polimorfo

- Classiche funzioni polimorfe:
  - (map f list) applica la funzione f a tutti gli elementi di una lista

    ```
    (T -> S) -> (list T) -> (list S)
    ```

  - (select test list) seleziona gli elementi di list su cui test ritorna true

    ```
    (T -> Bool) -> (list T) -> (list T)
    ```

- sono naturalmente definibili
  nessuna inferenza di tipo
  controllo a run-time che non ci siano errori di tipo
- Problema: definire un sistema di tipi statico che permetta il polimorfismo parametrico implicito

Polimorfismo parametrico esplicito

In C++: function template (simile ai generics di Java)

- si vuol generalizzare ad altri tipi una funzione swap che scambia due interi

```
void swap (int& x, int& y){
int tmp = x; x=y; y=tmp;}
```

- metodo: un template swap che scambia due dati qualunque

```
template <typename T>          //T è una sorta di parametro
void swap (T& x, T& y){
T tmp = x; x=y; y=tmp;}
```

- istanziazione automatica

```
int i,j;     swap(i,j); //T diventa int a link-time
float r,s;   swap(r,s); //T diventa float a link time
String v,w; swap(v,w); //T diventa String a link time
```

Generics in Java

```
    public static < E > void printArray( E[] inputArray ) {
        // Display array elements
        for(E element : inputArray) {
            System.out.printf("%s ", element);
        }
    }
    public static void main(String args[]) {
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        printArray(intArray);   // pass an Integer array
        printArray(charArray);   // pass a Character array
    }
}
```

l'uso di `Integer` e `Character`, al posto di `int`, `char` permette di non dover replicare il codice.

Implementazione

- Istanziazione automatica (C++)
  più istanze del codice generico
  una per ogni particolare tipo su cui viene chiamato
- Un unica istanza del codice generico (Java, ML, Haskell . . .) stesso codice macchina funziona su più tipi diversi
  possibile se
  - tutte le variabili memorizzate allo stesso modo:
    modello per riferimento (ogni variabile è un puntatore al dato)
    accesso più lento ma codice universale

Polimorfismo parametrico in ML, Haskell (implicito)

La funzione swap in ML:

```
swap(x,y) = let val tmp = !x in
            x = !y; y = tmp end;
```

- ML, Haskell non necessitano definizione di tipo:
  - inferenza automatica di tipo a tempo di compilazione
  - si determina, per ogni funzione, il tipo più generale che la descrive

```
val swap = fn : 'a ref * 'a ref -> unit
```

- ML, come Java e Haskell, non necessita la replicazione del codice
  - si accede ai valori tramite riferimenti (indirizzi di memoria)
  - il codice manipola riferimenti, indipendenti dal tipo di oggetto puntato.

Polimorfismo di sottotipo

- Tipico dei linguaggi ad oggetti
- può assumere diverse forme a seconda del linguaggio di programmazione
- si basa su una relazione di sottotipo: `T < S` (`T` sottotipo di `S`)
  - un oggetto di tipo `T`, ha tutte le proprietà (campi, metodi) di un oggetto di tipo `S`
  - `T` compatibile con `S`
- in ogni contesto, funzione, che accetta un oggetto di tipo `S` posso inserire, passare, un oggetto di tipo `T`

Polimorfismo di sottotipo e polimorfismo parametrico

- Per una maggiore espressività posso combinare polimorfismo di sottotipo con quello parametrico
- esempio: funzione `select` dato un vettore di oggetti restituisce l'oggetto massimo
  - con solo polimorfismo di sottotipo:
    `select: D[] -> D`
    posso applicare `select` a un vettore sottotipo `E[]` di `D[]`
    ma all'elemento restituito viene assegnato tipo `D` e non `E`
  - combinando i polimorfismi: `select: ∀ T < D. <T>[ ] -> <T>`
    descrive meglio il comportamento di select,
    informazioni in più sul tipo risultato

In Java

- Definisco un metodo che opera uniformemente su tutte le estensioni di una classe `D`

```
public <T extends D> T select (T[] vector) {

}
```

- Java usa anche la nozione di `interface` e `implements` come meccanismo per estendere il polimorfismo

```
public <T implements I> T select (T[] vector) {

}
```

Ulteriore esempio: funzione Quick Sort

Es.

```
void quickSort( E[] inputArray)
```

- non posso applicare `quickSort` a un vettore di elementi non confrontabili
- devo chiedere l'esistenza di un'operazione confronto

Diverse soluzioni:

- passo la funzione di confronto come parametro
  prolisso, devo esplicitamente passare tutte le funzioni ausiliarie

```
void quickSort( E[] inputArray, (E*E)->Bool compare)
```

Soluzioni

- linguaggi ad oggetti: chiedo che `E` contenga un metodo di confronto (istanza di una Interface)

```
class S{
   bool compare (x, y : S)
   }
void quickSort(S[] inputArray,)
```

- linguaggi funzionali, chiedo che sugli elementi di `E` possa essere applicato una funzione di confronto (<) (istanza di una Type Class)

Polimorfismo

Problema complesso, definire un sistema di tipi:

- generale: permette di dare tipo a molti programmi
- sicuro: individua gli errori
- type checking efficiente
- semplice: comprensibile dal programmatore

Conseguenze:

- vasta letteratura
- tante implementazioni diverse
- in evoluzione

Duck Typing

Alcuni linguaggi (Ruby, JavaScript)
non associano un tipo ad ogni oggetto ma si basano sul

- duck test — “If it walks like a duck and it quacks like a duck, then it must be a duck”
- l’uso di oggetto in un contesto è lecito se l’oggetto possiede tutti i metodi necessari
  - indipendentemente dal suo tipo
  - usato nei linguaggi con un sistema di tipo dinamico
  - controllo a tempo di esecuzione che se un oggetto riceve un messaggio `m`,
    `m` appartenga ai suoi metodi
- tipi di dati dinamici
  - maggiore espressività rispetto a tipi statici
  - minore efficienza, maggior numero di controlli a runtime
    altrimenti minor controllo degli errori, possibili errori di tipo nascosti

Esercizi: relazioni tra tipi.

L'equivalenza tra tipi in C è strutturale, in generale, ma per nome sul costrutto `struct`.

Determinare, nelle definizioni seguenti, quali variabili sono equivalenti per tipo rispetto a

- equivalenza per nome
- equivalenza per nome lasca
- equivalenza tra tipi in C
- equivalenza strutturale

Definizioni A

```
typedef int integer;
typedef struct{int a; integer b} pair;
typedef struct coppia {integer a; int b} coppia2;
typedef pair[10] array;
typedef struct coppia[10] vettore;
typedef coppia2[10] vettore2
int a;
integer b;
pair c;
struct coppia d;
coppia2 e;
vettore f;
vettore2 g;
coppia[10] i, j;
array k;
```

Definizioni B

```
typedef bool  boolean;
typedef boolean[32] parola
typedef struct{parola a; parola b} coppia
typedef coppia pair

parola a;
boolean[32] b;
bool[32] c;
coppia d
pair e;
struct{parola a; parola b} f
struct{parola b; bool[32] a} g
struct{parola a; bool[32] b} h
```

Esercizi: conversioni di tipo

Che valore assume la variabile i al termine dell'esecuzione del seguente codice:

```
int i = 2;
float f = i;
union Data {
 int i;
 float f;
} data;
data.f = f;
i = data.i;
```

Esercizi: tipi polimorfi

Trovare il tipo polimorfo più generale per i seguenti funzionali:

```
(define G (lambda (f x)((f(fx)))))
(define (G f x)(f(f x)))
(define (G f g x)(f(g x)))
(define (G f g x)(f(g (g x))))
(define (H t x y)(if (t x y) x y))
(define (H t x y)(if (t x) x (t y)))
(define (H t x y)(if (t x) y (x y)))
(define (H t x y)(if (t x y) x (t y)))
```