Linguaggi di Programmazione

Corso di Laurea in "Informatica"

Strutturare i dati

Valeria Carofiglio

(Questo materiale è una rivisitazione del materiale prodotto da Nicola Fanizzi)

obiettivi

- Tipi e Sistemi di tipo
  - Tipi Scalari
  - Tipi Composti
- Equivalenza
- Compatibilità e conversione
- Polimorfismo
- Safety / sicurezza
- Inferenza
- Recupero risorse

Tipi: definizione

- Un **tipo di dato** è una collezione di valori omogenei ed effettivamente rappresentabili, dotata di un insieme di operazioni che manipolano tali valori
  - Omogeneità
    - I valori condividono alcune proprietà strutturali (simili tra loro)
  - Rappresentabilità
    - Devono potere avere una rappresentazione finita
      - es. non esiste un tipo dei numeri reali veri e propri (espansione decimale infinita → real o float → sottoinsieme dei razionali)
  - Operazioni
    - La stessa collezione di valori ha un set di operazioni con cui puo' essere manipolata
      - es. i tipi interi nei vari linguaggi, assieme ad operazioni di somma, moltiplicazione....

Tipi: a cosa servono ?

- Progetto: Supporto all'organizzazione concettuale
  - Dominare la complessità dei problemi
    - Con soluzioni strutturate che rispecchiano il problema
  - Esplicitare i concetti tipici attraverso nuovi tipi
    - Gestione di struttura alberghiera:
      - Concetti: Clienti, camere, prezzi, date..
      - Tipi: uno per ogni concetto
      - Operazioni: un insieme per ogni tipo
  - Aumento di leggibilità (documentazione) e di sicurezza (controlli)

Tipi: a cosa servono ?

- Programma: Supporto alla correttezza
  - Nei LdP → regole di controllo di tipo
    - Come devono essere usati in un programma
      - x=exp -------> compatibili
  - Evitare errori HW ed errori logici
    - es. chiamata non a una funzione/ somma di interi non corrisponde a stringhe sensate
  - Vincolo di tipo (semantica): Type checker
    - Violazione di vincolo → errore semantico
  - Polimorfismo
    - es. stessa funzione su strutture di tipi diversi (vettore di interi/reali)
  - Sicurezza (safety): problemi in fase d'esecuzione linguaggi sicuri e non

Tipi: a cosa servono ?

- Traduzione: Supporto all'implementazione
  - Costituiscono informazioni per la macchina astratta
    - Dimensione richiesta per l'allocazione dei vari oggetti
      - A tempo di compilazione

    - Ottimizzazioni sulle op. d'accesso
      - Offset rispetto al puntatore al RdA
      - Con dimensione fissata

    - Calcoli statici

sistemi di tipo

- Complesso delle ***informazioni*** e delle ***regole*** che governano i tipi di un linguaggio
  - Insieme dei tipi predefiniti
  - Costrutti per definire nuovi tipi
  - Meccanismi per il controllo dei tipi
    - Regole di equivalenza: due tipi **formalmente diversi** possono essere **equivalenti** livello semantico
    - Regole di compatibilità: un valore di un **tipo diverso** da quello atteso può essere **comunque utilizzato**
    - Regole di inferenza: **attribuzione di un tipo** ad un'espressione complessa, partendo dalle sue componenti
  - se/quali vincoli controllare staticamente/dinamicamente

sistemi di tipi
sicurezza

- Un sistema di tipi è **sicuro** a livello di tipi (type-safe) quando ***nessun programma può ignorare*** le differenze di tipo definite dal sistema
  - Nessun programma può generare errori inattesi generati da violazioni di tipo a run-time
    - es. accesso a memoria non allocata

classificazione di tipi

- I valori di un tipo possono corrispondere a diverse entità sintattiche (costanti, espressioni,...)
- Fissato un LdP, classifichiamo i tipi in base a
  - come i suoi valori possono essere manipolati
  - Che genere di entità sintattiche corrispondono a quei valori

classificazione di tipi

- In base ai valori (... ed alle operazioni):
  - denotabili: possono essere associati ad un nome
  - esprimibili: possono essere il risultato di una espressione complessa (più di un semplice nome)
  - memorizzabili: possono essere memorizzati in una variabile

classificazione di tipi
Esempi

- Tipo delle funzioni (int -> int)
  - Val. denotabile in quasi tutti i linguaggi:
    - possiamo associare un nome ad una funzione
      - » es. int succ (int n) {return x+1;}
  - Val. Esprimibile solo nei ling. funzionali e non negli imperativi
    - Non esistono Funzioni che siano risultato della valutazione di un'espressione
  - Val. memorizzabile
    - Non si puo' assegnare una funzione ad una variabile
  - Idem (ML, Haskell, Scheme: linguaggi funzionali → denotabili/esprimibili/memorizzabili)

sistemi di tipo
controlli

- Ling. a tipizzazione dinamica: controlli a run-time (es. Lisp)
  - Ogni oggetto ha un *descrittore* che ne specifica il tipo
  - La macchina astratta controlla la correttezza degli operandi nelle operazioni
    - Il compilatore ha generato codice di controllo opportuno
  - Caratteristiche
    - Previene errori di tipo (troppo tardi?)
    - Inefficiente in esecuzione

sistemi di tipo
controlli

- Ling. a tipizzazione statica: controlli dei vincoli a compile-time (es. Java)
  - Controlli anticipati
  - Correttezza garantita per ogni sequenza d'esecuzione
    - Controlli a run-time inutili: maggiore efficienza
    - La progettazione del linguaggio è più complessa se il linguaggio deve anche essere type-safe
    - Compilazione lenta e complessa ma facilita debugging/testing

sistemi di tipo: controlli

controlli statici: un prezzo da pagare

- Programmi sbagliati, ma che non causano errori durante l'esecuzione.
  - Controllo statico: più conservatore
    int x;
    if (1==0) x = "errore";
    else x = x + 2;
  - Non si passa mai per il ramo di successo (sequenza non ammessa)
  - Ma un controllo statico segnalerà l'errore

Controllo sui tipi in generale: problema indecidibile

Per prudenza un controllo statico esclude anche casi non pericolosi, come il precedente (escludere piu' programmi di quelli necessari)

sistemi di tipo: controlli

- Controllo sui tipi in generale: problema indecidibile
  - Per prudenza un controllo statico esclude anche casi non pericolosi, come il precedente (escludere piu' programmi di quelli necessari)
- In molti linguaggi (es. Pascal) controllo statico+dinamico
  - es. utilizzo di vettori con controllo degli indici a run-time

tipi scalari

- Tipi **scalari** (o semplici): tipi i cui valori non* sono costituiti da aggregati di altri valori
- Denotati
  - type <nometipo> = <espressione>;
    - Introduce il nuovo tipo <nometipo>
    - Con struttura data da <espressione>
  - typedef <espressione> <nometipo>
    - Alternativa (C)

booleani

- Valori logici (o di verità)
- Valori: uno per il vero uno per il falso
- Operazioni:
  - Congiunzione (and), Disgiunzione (or), Negazione (not),
    or esclusivo, uguaglianza
- Valori denotabili, esprimibili, memorizzabili
  - Memorizzazione nelle minime unità di memoria indirizzabili (un byte)

caratteri

- Valori: un insieme di codici di caratteri fissato alla progettazione del linguaggio
  - Es: ASCII, UNICODE (insiemi piu' comuni)
- Operazioni: forteente dipendenti dal linguaggio.
  - (sempre) Uguaglianza, confronti, carattere successivo (succ) o precedente (prec)
- Valori denotabili, esprimibili, memorizzabili
  - Memorizzazione con 1 (ASCII) o 2 (UNICODE) byte

interi

- Valori: sottoinsieme finito dei numeri interi fissato alla progettazione del linguaggio o della macchina astratta (→ problemi di portabilità)
  - Di solito è un intervallo di valori del tipo $[-2^t,+2^t-1]$
- Operazioni aritmetiche e confronti
  - Somma, differenza, prodotto, divisione (intera), resto, potenza
- Valori denotabili, esprimibili, memorizzabili
  - Memorizzazione con un numero (pari) di byte in complemento a due

Reali virgola fissa

- numeri in virgola fissa
- Valori: sottoinsieme finito dei numeri razionali fissato durante la progettazione del linguaggio o della macchina astratta
  - La struttura (Ampiezza e granularità), ecc. dell'insieme dipendono dalla rappresentazione scelta
- Operazioni aritmetiche e confronti
  - Somma, differenza, prodotto, divisione (intera), resto, esponenziale, radice quadrata
- Valori denotabili, esprimibili, memorizzabili
  - Memorizzazione con un numero di 4 o 8 byte
    - complemento a 2
    - numero fissato di bit per la parte decimale

Reali virgola mobile

- Reali o numeri in virgola mobile (float)
- Valori: sottoinsieme finito dei numeri razionali fissato durante la progettazione del linguaggio o della macchina astratta
  - Struttura (Ampiezza e granularità, ecc). dell'insieme dipendono dalla rappresentazione scelta
- Operazioni aritmetiche e confronti
  - Somma, differenza, prodotto, divisione (intera), resto, esponenziale, radice quadrata
- Valori denotabili, esprimibili, memorizzabili
  - Memorizzazione con un numero di 4,8,10 byte
  - Standard IEEE 754 (post 1995)

complessi

- Valori: sottoinsieme finito dei numeri complessi fissato durante la progettazione del linguaggio
- Ampiezza e granularità, ecc. dell'insieme dipendono dalla rappresentazione scelta
- Operazioni aritmetiche e confronti
  - Somma, differenza, prodotto, divisione (intera), resto, esponenziale, radice quadrata
- Valori denotabili, esprimibili, memorizzabili
  - Memorizzazione con una coppia di reali in virgola mobile

void

- Valori: un solo valore
  - void o unit (insieme singoletto) oppure ()
  - Perché una funzione matematica non può avere codominio vuoto
    - Sono così le funzioni che divergono sempre
- Operazioni: nessuna
- Utile a denotare operazioni che non restituiscono un valore
  - es. assegnamento (in molti linguaggi, non in C o Java)

enumerazioni

- Tipi semplici definiti dall'utente
- Valori: un insieme finito di costanti caratterizzate da un proprio nome
  - es. type giorni = (lu,ma,me,gi,ve,sa,do); → un nuovo tipo di nome giorno costituito da un insieme di 7 elementi
- Operazioni: confronti, operatori per raggiungere il valore precedente o il successivo
- Vantaggi
  - Aumenta la leggibilità
  - Ausilio del controllo dei tipi: valori corretti per una variabile
- Memorizzazione: mappaggio sugli interi (con un byte)
  - In alcuni linguaggi (C/C++, Ada, ...) si possono anche scegliere esplicitamente

intervalli

- Tipi semplici definiti dall'utente
- Valori: sottoinsieme contiguo dei valori di un altro tipo scalare (*tipo base*)
  - es. type NumeriLotto = 1..90; (tipo base: intero)
  - Giorniferiali = lu..ve; (tipo base: giorni)
- Operazioni: confronti, operatori per raggiungere il valore precedente o il successivo
- Vantaggi
  - Aumenta la leggibilità
  - Ausilio del controllo dei tipi: obbligo controllo dinamico
- Memorizzazione: mappaggio sugli interi
  - Come per il tipo base

tipi ordinali

- I tipi **ordinali** (o *discreti*) sono dotati di una relazione d'ordine totale
  - Valori: discreti
    - Booleani, caratteri, interi, enumerazioni ed intervalli
  - Operazioni: precedente e successivo
  - Utili per gli indici

tipi composti

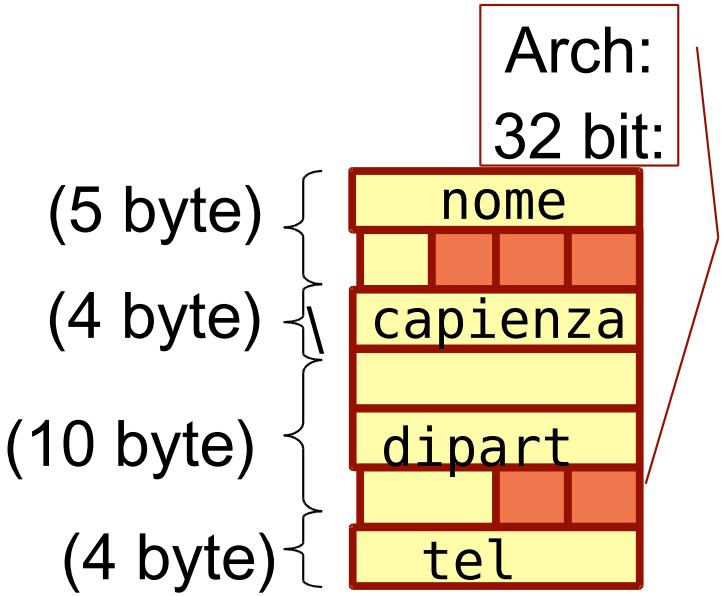
- Tipi non scalari ottenuti per combinazione di tipi più semplici
  - Record (o strutture): collezione di valori eterogenei
  - Array: collezione di valori omogenei
  - Insiemi: sottoinsiemi di un tipo (base, ordinale)
  - Puntatori: l-valori per accedere indirettamente ad altri valori
  - Tipi ricorsivi: definiti per ricorsione (liste, alberi...)

Record (o strutture)

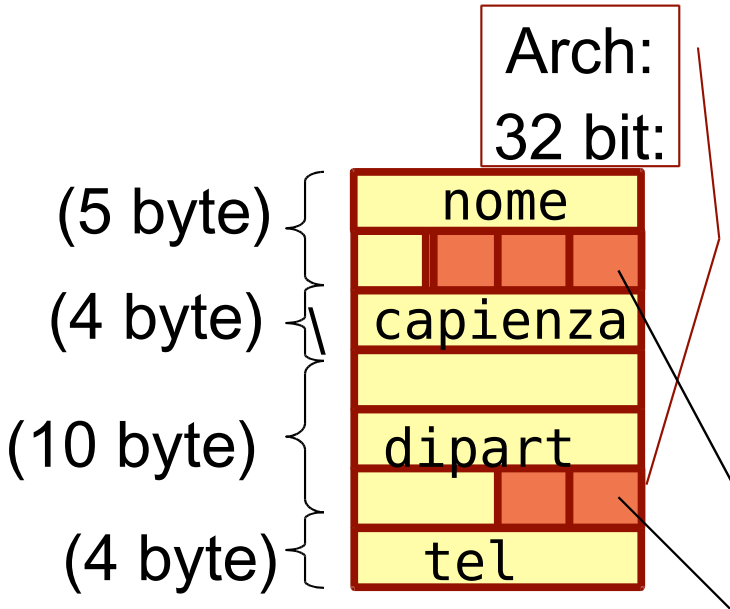
- Collezione di un numero finito (e spesso ordinato) di elementi detti campi
  - Ogni campo è caratterizzato
    - dal suo nome
    - dal suo tipo (anche diverso da quello degli altri campi)
  - Si può assimilare quindi ad una variabile del proprio tipo
    - es.
```
type studente = struct {
                int matricola; float altezza;
        };
```
  - Spesso è possibile annidare record all'interno di record
    - es.
```
type Aula = struct {
                char nome[5]; int capienza;
                struct { char dipart[10]; int tel; };
        };
```

Record: operazioni e implementazione

- Operazione: selezione di un campo, indicata con ".".
  - Es: (s variabile di tipo studente) s.matricola: 339567
  - Alcuni linguaggi (Ada) ammettono l'assegnazione e l'operatore di uguaglianza tra record
    - Se non permesso bisogna procedere campo per campo (C, C++,Pascal)
  - L'ordine dei campi può essere significativo



Record: operazioni e implementazione



- La memorizzazione avviene nell'ordine di definizione in locazioni contigue
  - Diverse organizzazioni possibili dovute all'allineamento
    - Migliora l'efficienza nel reperimento

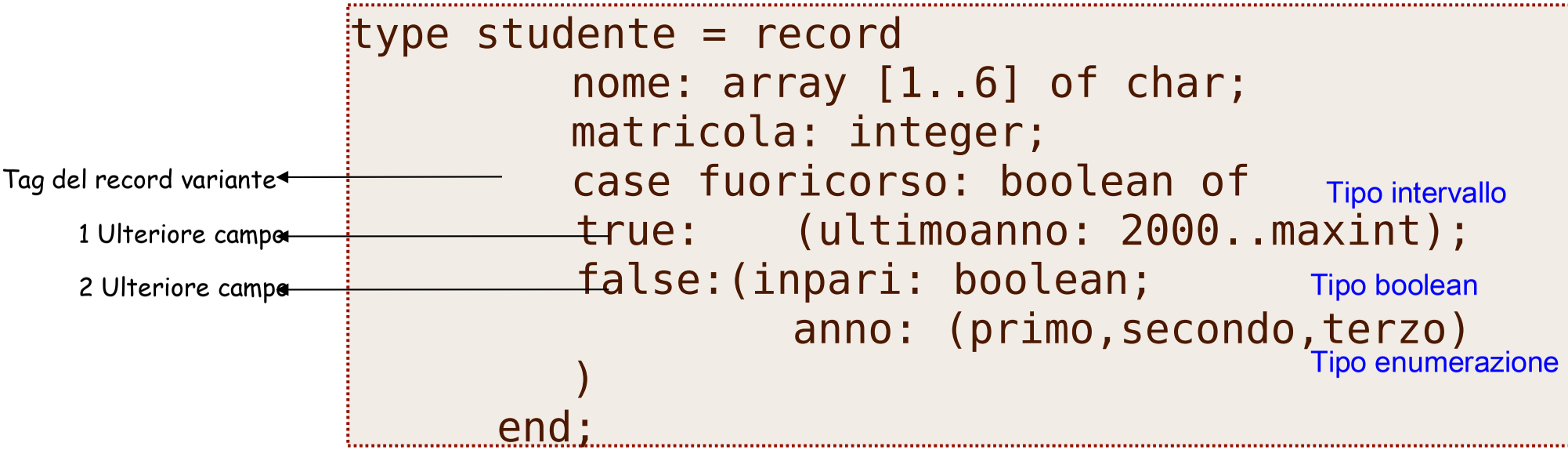
7 parole di 4 byte anche se solo 23 byte sono signficativi

Alcuni LdP non consentono uguaglianza tra record

es.
```
type Aula = struct {
                    char nome[5]; int capienza;
                    struct { char dipart[10]; int tel; };
            };
```

Record varianti

- Record con campi mutuamente esclusivi
  - Nomi e sintassi diverse per ogni linguaggio
    - PASCAL: record con una parte variabile
    - solo una variante è significativa



- Tra parentesi tonde: varianti del record
- Il tag puo' essere di qualsiasi tipo ordinale (con numero di variant pari alla cardinalità del tipo)
- Semantica: solo una delle due varianti è significativa (in base al valore del tag)
-Implementazione: le varianti condividono la stessa zona di memoria

unioni

- Record con campi mutualmente esclusivi
  - Nomi e sintassi diverse per ogni linguaggio
    - C: record ove un solo campo alla volta è valido
    - union per la parte variante

```
struct studente {
        char[6] nome;
        int matricola;
        int fuoricorso;
        union {
           int ultimoanno;
           struct {
           int inpari;
           int anno;
           } studente_in_corso;
        } campi_varianti;
}
```

record varianti e unioni

confronto

- Similarità
  - Le varianti e le union condividono le stesse aree di memoria
- Differenze
  - In C un campo della union è svincolato dal resto (è un campo come tutti gli altri)
    - più flessibile,
    - più oneroso per il programmatore,
    - più rischioso
  - Livelli di nomi:
    - In Pascal: s.inpari
    - In C occorre aggiungere ulteriori livelli di nomi (es: s.campi_varanti.stud_in_corso.inpari)

record varianti e unioni

sicurezza

- Sia in Pascal sia in C accesso al tag discriminante con assegnamento ordinario (no garanzia tra valore del tag e varianti significative)
  - La macchina astratta potrebbe verificare (dinamicamente) il valore del tag per sapere se il record è usato correttamente, prima di usarlo
    - Risolve molti problemi semantici ma non tutti

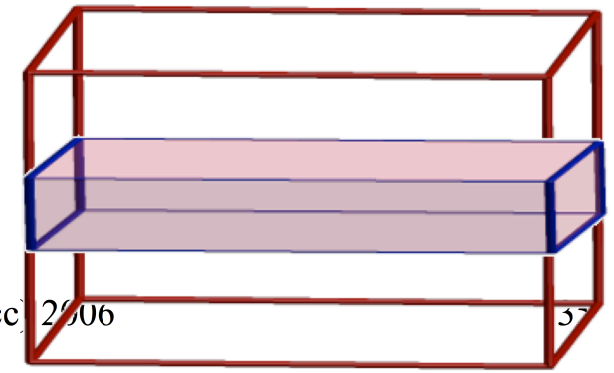
Vale la pena un "costo elevato" in termini di sicurezza?

```
type Tre = 1..3;
var tmp: record
           case quale: Tre of
               1: (a:integer);
               2: (b:boolean);
               3: (c:char);
           end
...
tmp.quale:=1;
tmp.a:=123;
tmp.quale:=3;
write(tmp.c); (* errato *)
```

array

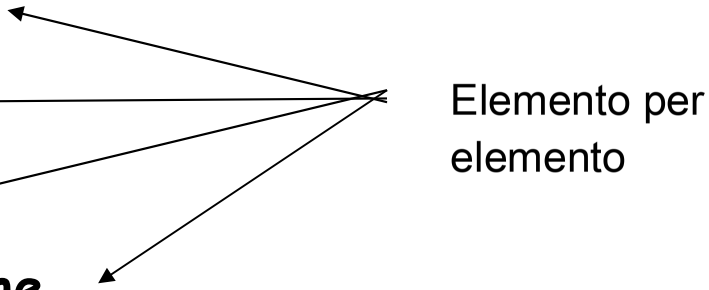
- Collezione finita di elementi dello stesso tipo (tipo base) indicizzata su un intervallo di tipo ordinale (tipo indice)
- Specifica
  - Nome
  - Tipo indice
  - Tipo dei componenti
    - es. C: int v[10]; int w[21,..,30]
    - Type giorno ={lun,mar,mer,gio,ven,sab,dom}; float[lun,..dom]
- Più indici: array multidimensionali
  - Int v [1..10, 1..10] (in pascal)
- A volte array di array
  - Int v[1..10][1..10] (in C, in pascal)
  - Slicing
    - ritagliare una "fetta" fissando una dim
    - Solo su array di array.

Talvolta equivalenza



array
operazioni

- **Selezione**: si indica (spesso tra quadre) un'espressione il cui valore rappresenta l'indice dell'elemento da selezionare
  - es. v[<espr>]
  - es. m[<espr$_1$>][<espr$_2$>]...[<espr$_n$>] (multidimensionalità)
- ***Assegnamento***
- ***Uguaglianza***
- ***Confronti***,
- ***Op. aritmetiche***



  - In ling. orientati al trattamento delle matrici
- **Slicing**
  - Slice: sezione di array costituita da elementi contigui
  - In alcuni linguaggi si possono estrarre anche selezioni diagonali cornici, ecc.

array
controlli

- Selezione entro i limiti del tipo indice
  - A run-time (salvo eccezioni)
  - Per un linguaggio safe:
    il compilatore *genera* controlli *per ogni* selezione
    - generazione spesso disattivabile con un'opzione del compilatore (pascal)
  - Safety & security
    - Attacchi buffer overflow
      - Messaggi via rete da leggere in un baffer destinatario
      - Senza controlli accesso a tipo indice possibile → sovrascittura di area di memoria (se riservata ad indirizzo di ritorno da procedura....)

array
memorizzazione

- Sezioni contigue di memoria
  - Array monodimensionale
    - Allocazione secondo l'ordine degli indici
  - Array multidimensionale
    - Ordine di riga (eccetto primo e ultimo)
      - Elementi contigui differiscono di un'unita nell'indice più a destra nella lista
      - Piu' frequente (slicing di riga)
    - Ordine di colonna (eccetto primo e ultimo)
      - Elementi contigui differiscono di un'unita nell'indice più a sinistra nella lista

array

calcolo dell'indice di un generico elemento

- Dato un array m a n dimensioni di tipo T
  - T $m[L_1,U_1][L_2,U_2]...[L_n,U_n]$
  - S_n unità di mem. Indirizzabili (tipic. byte) per memorizzare il tipo base T
    (Per riga):
    - $S_{n-1} = (U_n - L_n + 1)S_n$ (slice di ordine superiore)

      ...

      $S_1 = (U_2 - L_2 + 1)S_2$

Per cercare l'indirizzo di $m[i_1][i_2]...[i_n]$ ($m[i_1,i_2,..,i_n]$)
si somma all'indirizzo iniziale l'offset:
$(i_2 - L_2)S_1+(i_2 - L_2)S_2+...+(i_n - L_n)S_n$

- Se le dim non sono tutte note, meglio usare:
  $i_2S_1 + ... + i_nS_n - (L_1S_1 +...+ L_nS_n)$
  - n moltiplicazioni, n addizioni, n sottrazioni
    (che non servono se gli indici partono sempre da 0)

Per n=3 S_2
quantità di memoria
per
Memorizzare una riga
S_1 piano

array

forma

- **Shape**: numero delle dim. e intervallo per ogni dim.

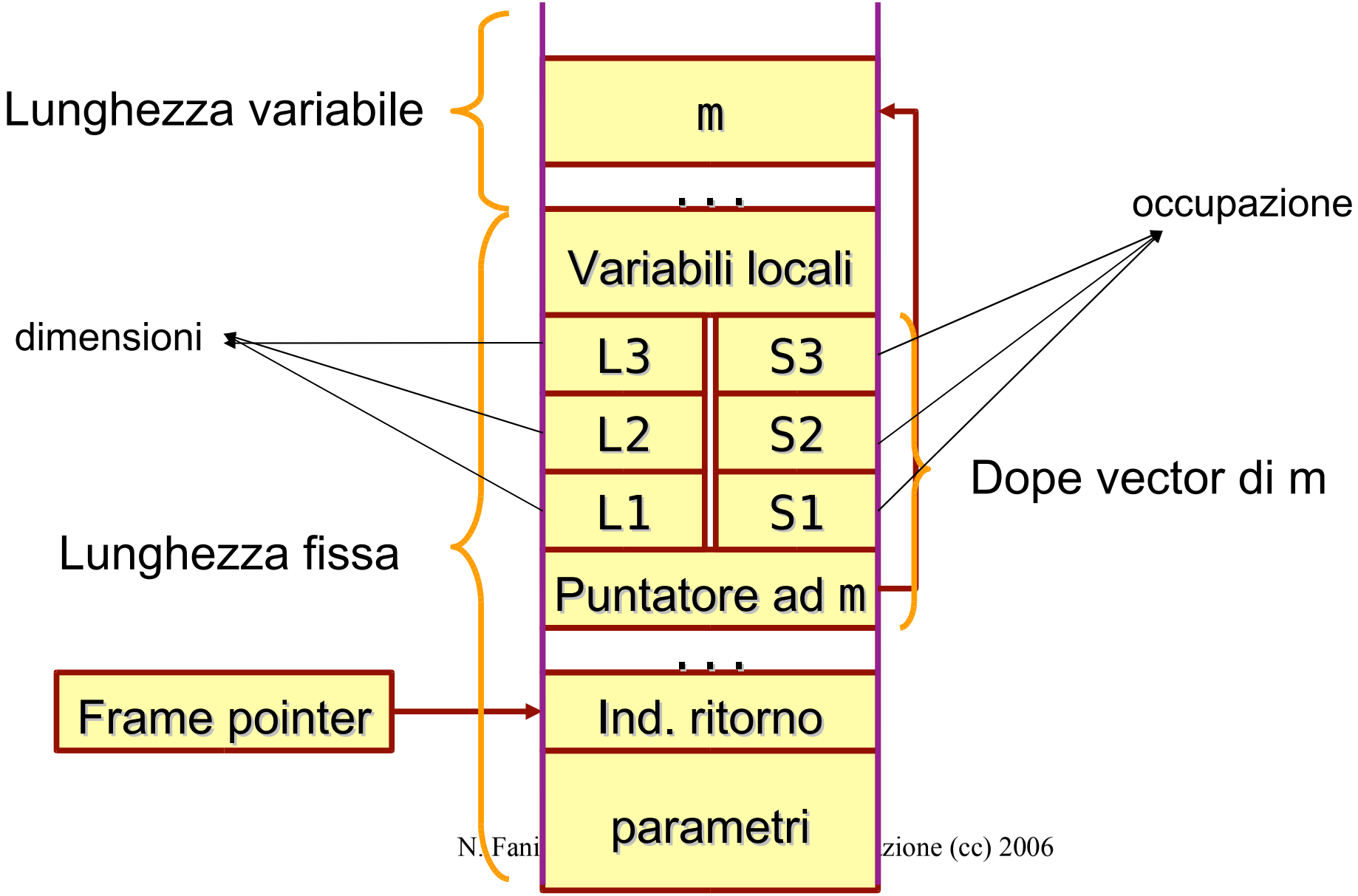
Quando viene fissata ?

- Staticamente: momento della compilazione (dim. Costante)
  - Array nel RdA del blocco in cui è definito (o nella mem. per le var. globali)
    - Accesso tramite la formula precedente / dimensione costante
- Dinamicamente
  - Limiti modificabili
  - Allocazione sull'heap: (non è possibile allocarlo sulla pila)
  - Nel RdA: puntatore all'array
- Al momento della Elaborazione della dichiarazione
  - Intervallo indici dipendente da un'espressione variabile
    - Calcolo a run-time
    - Array nel RdA: ma offset non noto!
    - Divisione del RdA in parte fissa (offset) e parte variabile (ind. Indiretto ← descrittore dei dati, in parte fissa che punta all'inizio della struttura di lunghezza variabile)

array
dope vector

- Descrittore di array di forma non nota staticamente
  - Di solito allocato nella parte a lunghezza fissa del RdA,
  - contiene
    - Puntatore alla prima cella dell'area di mem. riservata all'array
    - Info dinamiche utili
      (non memorizzate se staticamente determinabili)
      - Numero dimensioni
      - Limiti per ogni dim. (L_i)
      - Occupazione per ogni dim. (S_i)
  - Accesso ad un elemento:
    - Accesso per offset tramite frame pointer al dope vector

Array RdA con dope vector: al momento della creazione



insiemi

- Collezioni di valori costituiti da un sottoinsieme di un tipo base (universo)
  - solitamente il tipo base deve essere ordinale (Pascal)
    - es. `set of char S;`
`set of Giorni IG;`
    - es. `WE = (Sa, Do);` (con operazione di assegnazione)
- Operazioni
  - Appartenenza (di elemento ad insieme)
  - Unione (+), intersezione (*), differenza (-)
    - (a volte anche complemento)
- Rappresentazione
  - vettori di bit (*vettore caratteristico, con cardinalità pari alla cardinalità del tipo base*)

    - V_j = 1 sse il j-esimo elemento dell'universo appartiene all'insieme

puntatori

- Costituiti da l-valori manipolabili direttamente
  - In genere è possibile indicare anche il tipo delle variabili puntate
    - es. TipoBase * P;
    - In Pascal o Ada: possono solo puntare a variabili del tipo dato
    - In C/C++: non c'è un vincolo stretto
  - Tra tutti i valori che un puntatore puo' assumere esiste il valore nullo (null, nil..)
  - Ove presenti i puntatori consentono la definizione di tipi ***ricorsivi*** (strutture concatenate) senza primitive apposite
  - Linguaggi con variabili viste come riferimenti
    - non possono essere manipolati direttamente

puntatori

- Costituiti da l-valori manipolabili direttamente
  - In genere è possibile indicare anche il tipo delle variabili puntate
    - es. TipoBase * P;
    - In Pascal o Ada: possono solo puntare a variabili del tipo dato
    - In C/C++: non c'è un vincolo stretto
  - Ove presenti i puntatori consentono la definizione di tipi ***ricorsivi*** (strutture concatenate) senza primitive apposite
  - Linguaggi con variabili viste come riferimenti
    - non possono essere manipolati direttamente
- Implementazione del puntatore a tipo (TipoBase*)
  - Puntatori a Indirizzi Ovvero locazione di memoria (variabili modificabili), Che cntengono valori di tipo TipoBase
  - Locazioni
    - Arbitrarie
    - Sullo heap (pascal, ada)
    - Sulla pila di sistema (c,c++

puntatori

Operazioni

- creazione/assegnamento
- test di uguaglianza
- Dereferenziazione (valore puntato da)

Puntatori operazioni

- Assegnamento di un valore ad un puntatore mediante
  - Allocazione esplicita (new ..)
    - int *p;
      p = (int *) malloc(sizeof (int));

    Alloca sullo HEAP →
    E restituisce riferimento
  - Operatore &:
    - float pigreco = 3.1415, *pp;
      pp = &pigreco;

    Alloca sulla memoria →
    pp punta alla locazione di memoria che contiene la variabile pigreco
- Dereferenziazione
  - "^" (pascal) oppure "*" prefisso (in C/C++)
    - es. precedente:
      float circ = 2 * r * (*pp);

    Valore puntato
  - *prefisso mantiene distinzione 'l-valore/r-valore
    - * a sinistra = → l-valore che si ottiene deferenzando
    - * a destra di = → r-valore della stessa locazione

Puntatori operazioni

- Creazione di strutture ricorsive
  - Quando non previste dal linguaggio
  - es. lista di interi
    (successione ordinata di dim. variabile)
    - typedef nodo* lista_int;
      typedef struct { int val;
      lista_int succ} nodo;

Puntatori aritmetica

- C e alcuni derivati: operazioni aritmetiche su puntatori
  - Incremento/decremento di un puntatore: p++ / p--
    - indirizzo incrementato di sizeof(tipobase)
  - Sottrazione di puntatori: p1-p2
    - Offset tra p1 e p2
  - Somma di un quantità ad un puntatore: p+n
    - punta alla variabile con offset pari a n*sizeof(tipobase)

```
int *p;
int *c;
p = (int*) malloc(siz eof(int));
c = (char*) malloc(sizeof(char));
p = p+1
c++;
```

Nociva per la type-safety del linguaggio

Non c'è garanzia che in tutti i momenti un puntatore punti effettivamente ad una variabile del tipo atteso

Puntatori aritmetica

- Nociva per la type-safety del linguaggio
  - Non c'è garanzia che in tutti i momenti un puntatore punti effettivamente ad una variabile del tipo atteso

Puntatori deallocazione

- Implicita: nessuno strumento per deallocare la memoria
  - Avviene quando non c'è più spazio sullo heap allocabile
    - Possibile recuperare memoria inutilizzata
      - Tecniche di *garbage collection (lett. Raccolta di spazzatura)*
- Esplicita: meccanismo del linguaggio
  - In C: funzione free()
    - es. free(p) libera la memoria sullo heap puntata da p
      e conviene anche assegnare: p=null;
  - Se p vale già null allora si ha un errore semantico
  - ***Dangling reference***: puntatori con valore diverso da null che puntano a zone non più significative
    - Memoria deallocata o ri-allocata
    - Riferimenti non validi alla pila d'esecuzione
      (anche con deallocazione implicita)

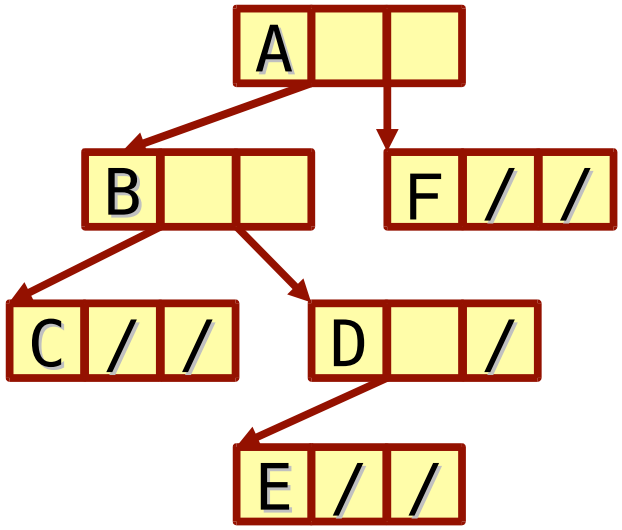
tipi ricorsivi

- Tipo composto in cui un valore può contenere (un riferimento ad) un valore dello stesso tipo (assimilabili ai record)
- Esempi (pseudocodice):
  - Lista di interi. Fino a “null”
    - ```
      type ListaInt = {int val;
                       ListaInt next; }
      ```
      ```

tipi ricorsivi

- Rappresentati con strutture dati su heap
  - Ling. **Imperativi**: Strutture concatenate di elementi RECORD (implementati con riferimenti/puntatori al successivo RECORD)
    - *Allocazione esplicita*

```
type AlberoChar = {char val;
                   AlberoChar sinistro;
                   AlberoChar destro; }
```



No deallocazione esplicita dei valori, per linguaggi che non ammettono tipi ricorsivi primitivi --> garbage collection (macchina astratta)

tipi ricorsivi

- Tipo composto in cui un valore può contenere (un riferimento ad) un valore dello stesso tipo (assimilabili ai record)
- Esempi (pseudocodice):
  - Lista di interi. Fino a “null”
    - `type ListaInt = {int val; ListaInt next; }`
- Operazioni:
  - Selezione
  - Test di uguaglianza sul valore null

tipi di funzioni

- Alcuni linguaggi permettono di denotare tipi di funzioni (dargli un nome nel linguaggio)
  - I cui valori sono funzioni
    - sia
      T f(S_1 s_1, S_2 s_2, ... , S_n s_n) {...}
      f ha tipo denotato con
      S_1 x S_2 x ... x S_n -> T
    - Denotabili, ma raramente esprimibili o memorizzabili
- Operazioni su valori di tipo funzioni
  - Definizione (soprattutto funzionali)
  - Applicazione (chiamata su parametri attuali)
  - Sui linguaggi funzionali: il tipo funzione ha la stessa dignita di altri tipi

regole di correttezza relativamente alla tipizzazione

Relazione di equivalenza

regole di correttezza relativamente alla tipizzazione

- Stabilire quando due tipi, ***formalmente diversi***, sono ***intercambiabili***, ossia non distinguibili nel loro uso

```
type nuovotipo = espressioneTipo
```

- Interpretazione della regola nei LdP:
  - Definizione di tipo opaca: equivalenza per **nome**
  - Definizione di tipo trasparente: equivalenza **strutturale**

Equivalenza per nome

Il LdP usa definizioni di tipo opache

- Ogni nuova definizione introduce un nuovo tipo
- Due tipi si diranno **equivalenti per nome** sse essi hanno lo stesso nome
  - (un tipo è equivalente solo a se stesso)
    - T1 = 1..10;
      T2 = 1..10;
      T3 = int;
      T4 = int;
      sono tutti diversi e non equivalenti
  - Indebolita in alcuni linguaggi (pascal)
    - es. la ridenominazione (T3 e T4) **genera alias** e non nuovi tipi
- Osservazioni
  - Definizione in un solo punto: OK dal punto di vista ingegneristico
  - Equivalenza ***relativa ad un programma***, non in generale

Equivalenza strutturale

Il LdP usa definizioni di tipo trasparenti

- Definizioni trasparenti: nome come abbreviazione del nuovo tipo definito
  - Due tipi sono equivalenti se hanno la stessa struttura
  - Esempio (pseudo-codice): equivalenza di T3 e T4
    - `type T1 = int;`
    - `type T2 = char;`
    - `type T3 = struct {T1 a; T2 b;}`
    - `type T4 = struct {int a; char b;}`
- L'equivalenza strutturale soddisfa le seguenti proprietà:
  - Un nome di tipo è equivalente a se stesso.
  - Se introdotto con `type T1 = espressione;`
    - T1 è equivalente ad `espressione`
  - Se T1 e T2 definiti applicando lo stesso costruttore di tipo a due tipi equivalenti, allora essi sono equivalenti

Equivalenza strutturale

- Esempio: equivalenti?
  - ```
    type S = struct {int a; int b;}
    type T = struct {int n; int m;}
    type U = struct {int m; int n;}
    ```

Nome diverso

- Esempio: equivalenti (ricorsione) ?
  - ```
    type R1 = struct {int a; R2 p;}
    type R2 = struct {int a; R1 p;}
    ```

Mutua ricorione non risolvibile dal controllore

Equivalenza strutturale

- Equivalenza strutturale ***non legata al singolo programma***
  - Sostituzione sempre possibile (**Trasparenza referenziale)**
- I linguaggi spesso adottano regole di equivalenza miste
  - Pascal: eq. Per nome debole
  - Java: eq. Per nome tranne che per array (eq. strutturale)
  - C:
    - eq. Strutturale per array e per tipi definiti con "typedef"
    - eq. per nome per struct e Union
  - C++: eq. Per nome (salvo cio' che erdità da C)
  - ....

compatibilità

Il tipo T è **compatibile** con il tipo S sse

un valore del tipo T è ammesso in qualsiasi contesto in cui sarebbe richiesto un valore di tipo S.

- Compatibilità più debole dell'equivalenza
  - Due **tipi equivalenti sono** sempre comp**atibili** (ma non viceversa)
  - Relazione riflessiva, transitiva ma non simmetrica
    - Es.: int e float
- Adottata da molti linguaggi
  - nella disciplina dell'assegnazione:
    tra il tipo dell'espressione (RHS) e quello della variabile (LHS)
  - nella disciplina del passaggio parametri

gradi di compatibilità

linee: T compatibile con S quando:

- T e S equivalenti
- I valori di T costituiscono un sottoinsieme dei valori di S
  - Tipi intervallo (contenuti nel suo tipo base)
- Tutte le operazioni per i valori di S sono possibili sui valori di T
  - `type S = struct {int a;}`
  - `type T = struct {int a; char b;}`
  - op. di selezione del campo a (ha senso su entrambi i tipi T ed S)
- I valori di T corrispondono *canonicamente* ad alcuni valori di S
  - `int` e `float` (distinti nella implementazione)
- I valori di T corrispondono ad alcuni valori di S, tramite una convenzione per la trasformazione da T a S (procedura)
  - `float` e `int`, tramite arrotondamento, troncamento,...

conversione di tipo

per gestire tutte queste differenze di compatibilità

- Conversione **implicita**:
  operata tacitamente dalla macchina astratta
  - Si chiama anche coercizione, o conversione forzata
  - Poche nei linguaggi a forte controllo di tipo
    - C controllo aggirabile → molte coercizioni
- Conversione **esplicita**:
  indicata mediante strutture linguistiche nel sorgente del programma
  - Si chiama anche cast

Coercizioni

Operata da Macchina Astratta

- Se T compatibile con S (I Valori di tipo T sono ammessi dov'è atteso un valore di tipo S
  - *Sintatticamente: le corcizioni sono* ***annotazione di compatibilità***

Implementazioni (in dipendenza della nozione di compatibilità)

- Stessa rappresentazione in memoria per T ed S:
  - **Coercizione di livello sintattico**, nulla da aggiungere
- Compatibilità canonica per trasformare T in S:
  - **codice di conversione** (a run-time) aggiunto dalla m. astratta
    - Es.: da int a float (complemento a due → virgola mobile)
- Corrispondenza arbitraria tra i valori di T ed S (es. val. di T sovrainsieme dei val. di S)
  - **la m. astratta inserisce codice per la conversione** (e per controllo dinamico per la type safety → il val.di T è in S?)

conversioni esplicite

Annotazioni *nel linguaggio* che specificano che un tipo deve essere convertito in un altro tipo

- ***cast: converte il tipo di t in S***

```
S s = (S)t;
```

Se il ling. Sa come convertire

- Possibilità
  - Indicazione sintattica (documentazione)
  - Indicazione per la m. astratta (per la conversione)
- Vantaggi (favorite rispetto a coercizioni)
  - In genere possibile anche quando basterebbe la compatibilità
  - Maggiore leggibilità
  - Indipendenza dal contesto sintattico (inutile sintatticamente)
  - Utili ad overloading e polimorfismo

polimorfismo

- Sistema di tipi si dice
  - **monomorfo**: ogni oggetto del linguaggio ha un solo tipo
  - **polimorfo**: ogni oggetto può avere più tipi
    - **Casi di polimorfismo in molti linguaggi anche datati**
      - L'operatore "+"
        int x int -> int ma anche float x float -> float
      - Funzione length: non dipende dal tipo di vettore
    - **Funzioni polimorfiche indipendenti da tipi specifici non consentito nei LdP convenzionali**
      - Es.: ordinamento di vettori (di interi, caratteri, ecc.)
        Uso di tipi generici:
        void sort(<T>[] v)

polimorfismo

- Due diverse forme:
  - Polimorfismo ad hoc (overloading)
  - Polimorfismo universale
    - p. parametrico
    - p. di sottotipo o inclusione

polimorfismo ad hoc (overloading)

Stesso nome, oggetti diversi (Nome sovraccaricato)

- L'informazione del contesto aiuta a decidere staticamente
  - Esempi: operatore '+' (per int o per float) / funzioni con ugual nome ma tipo e numero di parametri differente
- Polimorfismo apparente
  - Legato ai nomi più che agli oggetti del linguaggio
    - es. funzioni distinte / codice distinto
  - Può essere gestito con una fase di pre-processing
    - Assegna un nome interno diverso ai nomi sovraccarichi
- Overloading < > Coercizione
  - Esempio: + polimorfico (in dipendenza del LdP)
    - `1 + 2` — + sovraccaricato con quattro diversi significati/
      `1.0 + 2.0` — + sovraccaricato con due significati (int float, con coercizioni)
      `1 + 2.0`
      `1.0 + 2` — + somma real, coercizioni nel resto

polimorfismo universale parametrico

- Istanziazione dell'oggetto polimorfo
  - Automatica: compilatore o m. astratta in base al contesto
  - In base al meccanismo di poliformismo fornito dal LdP

Funzione di `swap()` chiamata dalla `sort()`

```
void swap(reference<T>, reference<T>)
```

Istanziazione (scambio var)

```
int * k = null;
 char v,w;
 int i,j;
 ...
 swap(v,w);
 swap(i,j);
```

Istanziazione su caratteri e su interi

polimorfismo parametrico (universale)

un valore esibisce PPU quando può assumere ***un'infinità di tipi diversi***, ottenuti per ***istanziazione di un unico*** schema universale

- Funzione Polimorfa: ha unico codice che lavora su tutte le istanze del suo tipo generale

Esempi:

Operatore null: per ogni tipo T*

Void Sort(<T> A[ ]) ordinamento di vettore

<T> parametro

polimorfismo parametrico (universale)

due forme

- **p. esplicito**: annotazioni esplicite <T> (visto)
  - Template C++ / Generics Java J2SE 5.0 (2004)
- **p. implicito**: operato dal modulo di inferenza dei tipi: per ogni oggetto del linguaggio, ricerca il suo tipo generale dal quale tutti gli altri tipi si ottengono per instanziazione dei parametri
  - Ling. di scripting
  - Ling. funzionali: l'applicazione comporta l'istanziazione giusta
    - `fun Ide(x){return x;}`
      `funzione identità` è di tipo <T> → <T> **(assegnato dall'inferenza)**
    - Funzioni di ordine superiore
      `fun Comp(f,g,x){return f(g(x));}`
      di tipo (<S> → <T>) x (<R> → <S>) x <R> → <T> **(assegnato di tipo piu' generale possibile)**

polimorfismo universale di sottotipo

- Non tutte le possibili istanziazioni dello schema universale sono ammissibili
  - Limite ad una forma di compatibilità strutturale
    - Nozione di sottotipo indicata con :< (C:<D → C sottotipo di D)
- p. **universale di sottotipo**:
  un valore può assumere un'***infinità di tipi diversi***, ottenuti ***istanziando*** un parametro in uno schema generale ***con i sottotipi di un tipo assegnato***
  - Esprimibile tramite la notazione: Funzione polimorfa di tipo Per ogni S:<T.S->void
    - Istanziabile con qualunque sottotipo S di T

polimorfismo
cenni di implementazione

- gestione **statica**: linking-time
  - istanziazione funzioni polimorfe (in base ai tipi)
    - produzione codice opportuno (più copie del template, una per ogni istanza, nel codice eseguibile)
    - e collegamento dei pezzi di codice
    - Allocazione dei dati in dipendenza del tipo per la funzione su RdA
  - efficiente come le per le funzioni non polimorfich
- gestione **dinamica**:
  - unica versione del codice generata
    - Allocazione dati sullo Heap: sul RdA vengono allocati loro descrittori (dim., struttura, ind.)
  - più flessibile ma meno efficiente a run-time (acceso indietto ai dati)

Controllo di tipo

determina e controlla la compatibilità dei tipi degli oggetti (assegnazioni, dichiarazioni, uso parametri, conversioni, ...)

- tipo di controllo
  - **statico**: modulo del compilatore (semantica statica)
    - segue l'albero sintattico (bottom-up)
    - determina e trasmette le informazioni sui tipi degli oggetti coinvolti (a partire da info del programmatore e info del sistema dei tipi)
  - **dinamico**: modulo del supporto al run-time

Inferenza di tipo

deduzione dei tipi coinvolti in presenza di informazione implicita

- spesso sostituisce il controllo di tipo (es. in Javascript, ML, ...)

esempio:

```
fun f(n) {return n+1;}
```

si può dedurre che f è di tipo int-> int

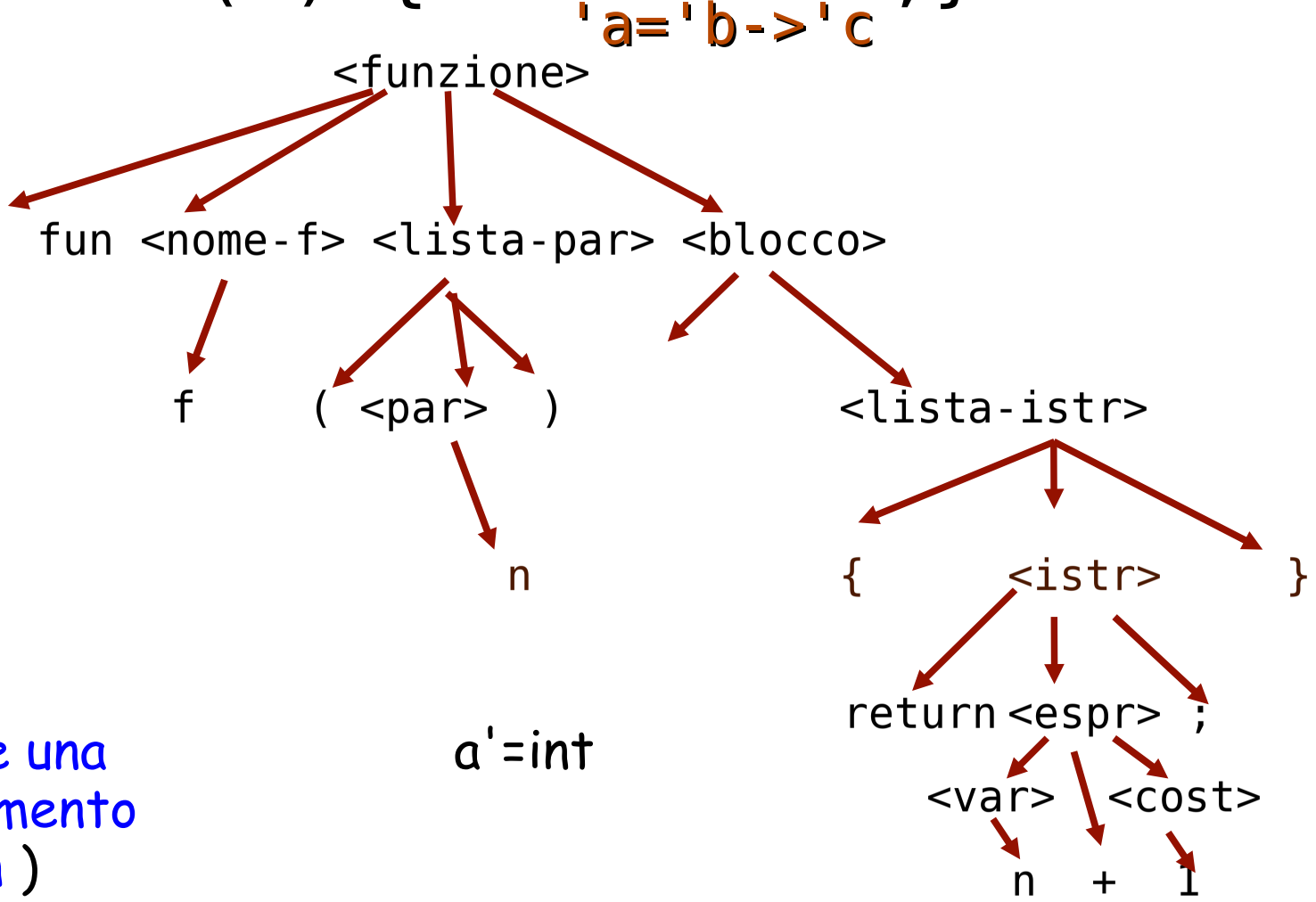
Inferenza di tipo

deduzione dei tipi coinvolti in presenza di informazione implicita

- Algoritmo: dato l'albero di derivazione
  - Assegnare ad ogni nodo:
    - un tipo (nomi predef. E costanti)
    - variabile. di tipo (espressioni complesse)
      - (not. Variabile di tipo 'a)
  - risalire l'albero generando vincoli di tipo (uguaglianze di tipo) tra tipi in corrispondenza di nodi interni
  - usa l'algoritmo di unificazione per risolvere i vincoli

Inferenza di tipo

- esempio: `fun f(n) {return n+1;}`



(applicando f
all'argomento n:

'a= 'b → 'c

ovvero f deve essere una
funzione e il suo argmento
dello stesso tipo di n)

Inferenza di tipo

NON SEMPRE LA RISOLUZIONE DEI VINCOLI RIESCE A RIMUOVERE TUTTE LE VARIABILI

- esempio: fun g(v) {return v;}
  - Polimorfa (il tipo piu' generale contiene una variabile di tipo)
  - Otterremmo 'a → 'a
-

Evitare dereferenziazioni di dangling reference

Conseguenza della

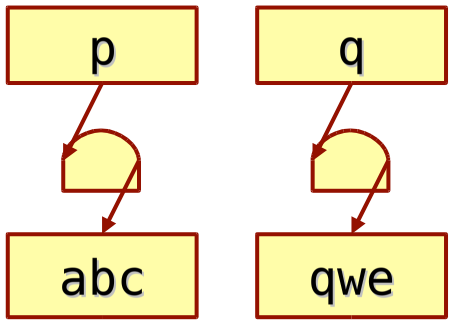
Deallocazione esplicita

Puntatori pendenti che si riferiscono a zone di memoria non sgnificative

(dangling reference)

tombstone (pietre tombali)

```
p=malloc()
q=malloc()
  *p=abc
  *q=qwe
```

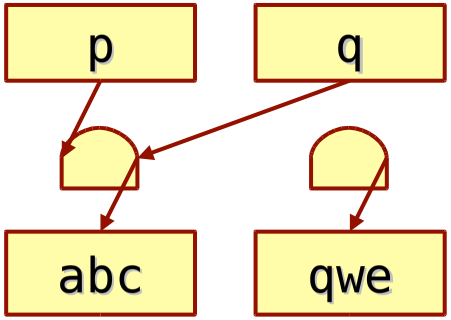


<u>allocazione</u> di nuovo oggetto su heap/pila con accesso tramite puntatore

creazione tombstone (con indirizzo dell'oggetto allocato)

Puntatore contiene indrizzo della tombstone

```
q=p;
```



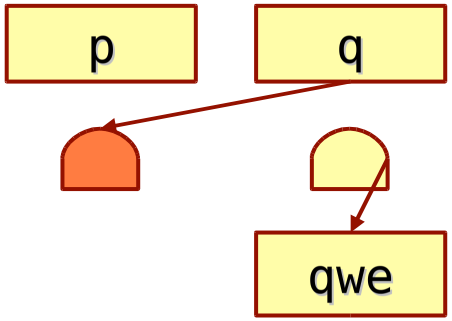
Dereferenziazione (valore puntato da)

Nuovo livello di indirizzamento:

-accesso a tonmstone e poi a oggettp puntato

- in assegnamento: modifica del valore non della tombstore

```
free(p);
```



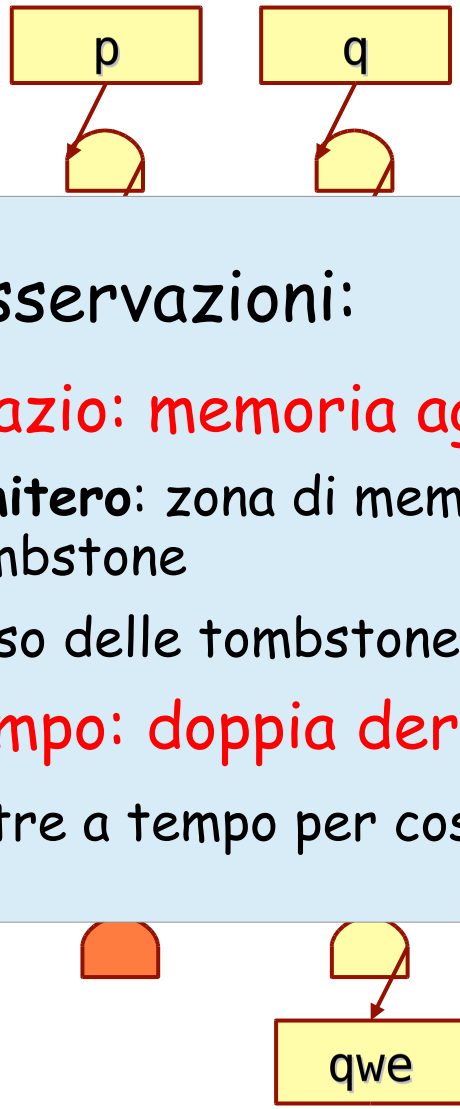
<u>deallocazione</u> (da heap/pila): inserimento valore nullo speciale

Tomstone segnalata (dati morti)

Accesso impossidile

tombstone (pietre tombali)

```
p=malloc()
q=malloc()
  *p=abc
  *q=qwe
```



<u>allocazione</u> di nuovo oggetto su heap/pila con accesso tramite puntatore

creazione tombstone (con indirizzo dell'oggetto allocato)

stone

la)

```
q=p;
```

e non

```
free(p)
```

inserimento valore nullo speciale

Tomstone segnalata (dati morti)

Osservazioni:

spazio: memoria aggiuntiva

cimitero: zona di memoria speciale per le tombstone

riuso delle tombstone (garbage collection)

tempo: doppia dereferenziazione

(oltre a tempo per costruzione e controllo)

Lucchetti e chiavi
(evita l'accumolo delle tombstone)

- lucchetto: parola in memoria inizializzata con valore casuale associata all'oggetto allocato
- chiave: parola in memoria corrispondente ad un lucchetto
- puntatore = indirizzo + chiave
  - allocazione di nuovo oggetto su heap: crea lucchetto
  - copia tra puntatori: copia di indirizzo e chiave
  - dereferenziazione: controllo che la chiave del puntatore “apra” il lucchetto (abbia lo stesso valore)
  - deallocazione: inserimento valore nullo speciale nel lucchetto
-

Lucchetti e chiavi
(evita l'accumolo delle tombstone)

- **lucchetto**: parola in memoria inizializzata con valore casuale associata all'oggetto allocato
- **chiave**: parola in memoria corrispondente ad un lucchetto

puntatore = indirizzo + chiave

- allocazione di nuovo oggetto su heap: crea lucchetto
- **copia tra puntatori**: copia di indirizzo e chiave
- dereferenziazione: controllo che la chiave del puntatore "apra" il lucchetto (abbia lo stesso valore)
- deallocazione: inserimento valore nullo speciale nel lucchetto

Lucchetti e chiavi
(evita l'accumolo delle tombstone)

- **lucchetto**: cella in memoria inizializzata con valore casuale associato
- **chiave**: p

  - allocaz
  - **copia tra puntatori**: copia di indirizzo e chiave
  - dereferenziazione: controllo che la chiave del puntatore "apra" il lucchetto (abbia lo stesso valore)
  - deallocazione: inserimento valore nullo speciale nel lucchetto

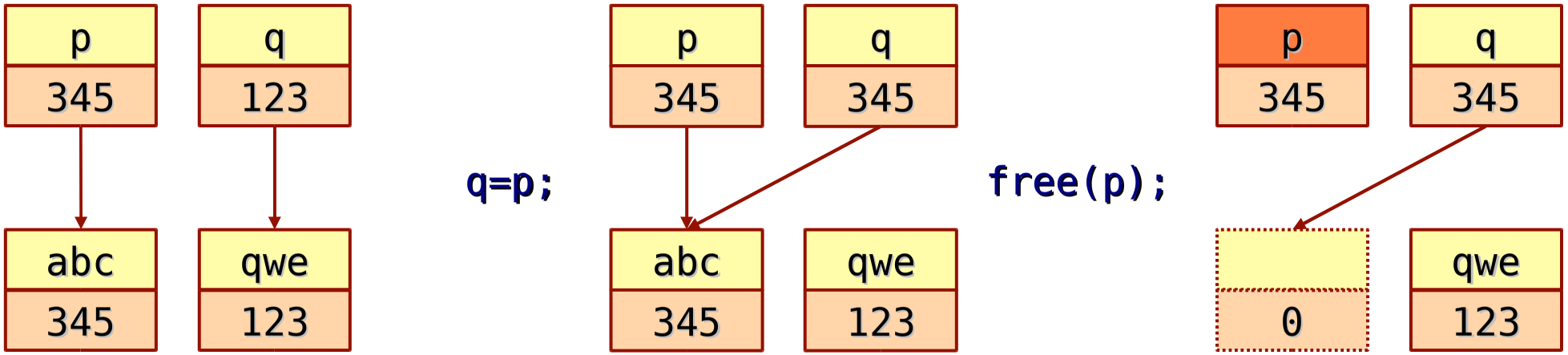
Osservazioni

spazio: memoria aggiuntiva (chiave+ lucchetto)

tempo : op. più efficienti che con il tombstone

deallocazione automatica della mem. aggiuntiva

dangling reference
locks & keys



garbage collection
(in assenza di deallocazione esplicita)

- Recupero della memoria allocata sullo heap
- **Fasi** (non necessariamente separate)
  - distinguere gli oggetti utilizzati dagli altri
    - per sicurezza politica conservativa
  - recuperare la memoria degli oggetti non utilizzati
- classificazione dei *GC* (in base a come determinano oggetti non piu' in uso)
  - Basati su contatori di riferimenti
  - Basati su marcatura
    - mark & sweep
    - mark & compact
  - Basati su copia

garbage collection: contatori

Un contatore (reference count) per ogni oggetto, conta il numero di puntatori attivi su quell'oggetto;

Contatore solo per la m. astratta: inaccessibile al programmatore

- allocazione: inizializza il contatore a 1
- assegnazione p = q;
  (o anche quando si esce da un ambiente locale con puntatori)
  - incremento di 1 del contatore di p
  - decremento di 1 del contatore di q
- deallocazione: quando un contatore raggiunge il valore 0, si può deallocare la memoria e restituirla alla lista libera
  - se l'oggetto da deallocare contiene puntatori
    - applica la procedura ricorsivamente

garbage collection: contatori

Un contatore (reference count) per ogni oggetto, conta il numero di puntatori attivi su quell'oggetto;

Contatore solo per la m. astratta: inaccessibile al programmatore

- allo
- asse
  (o a
  - i
  - d
- dea
  si p
  - se l'oggetto da deallocare contiene puntatori
    - applica la procedura ricorsivamente

atori)

ore 0

libera

Osservazioni

vantaggi: semplicità e incrementalità nel normale flusso di esecuzione

svantaggi: inefficienza (costo del programma); GC non funziona con str. circolari

garbage collection: mark & sweep

- mark
  - marcatura oggetti “non in uso” attraversando l'heap
  - a partire dai puntatori attivi sulla pila (root set), visita in ampiezza/profondità degli oggetti referenziati lungo gli archi rappresentati dai puntatori, marcando gli oggetti attraversati come “in uso”
- sweep
  - deallocazione degli oggetti marcati come “non in uso” (rilascio alla lista libera)

garbage collection: mark & sweep

Osservazioni

- svantaggi:
  - non incrementale: GC parte quando la memoria si sta esaurendo
    - improvviso segrado di performance a tempo di esecuzione
  - frammentazione esterna
  - inefficiente: tempo proporzionale alla dim. Dell'heap
    - Indipendentemente dalla percentuale di spazio usato
  - sfavorisce la località dei riferimenti in memoria
    - Oggetti vicini con età molto differenti
    -

garbage collection: rovesciare i puntatori

- visita agevolmente strutture concatenate
  - es. alberi
  - occorre uno stack (ricorsione)
- rovesciamento dei puntatori
  - basta ricordare il nodo corrente e quello precedente

garbage collection: mark & compact

Riduce il problema della frammentazione causato dal mark & sweep

- Fase di sweep => fase di compattamento
  - Blocchi di mem vivi spostati in zone di mem. Contigue
    - Scansione lineare dello heap
    - Traslazione di ogni blocco (contiguo all'ultimo vivo)

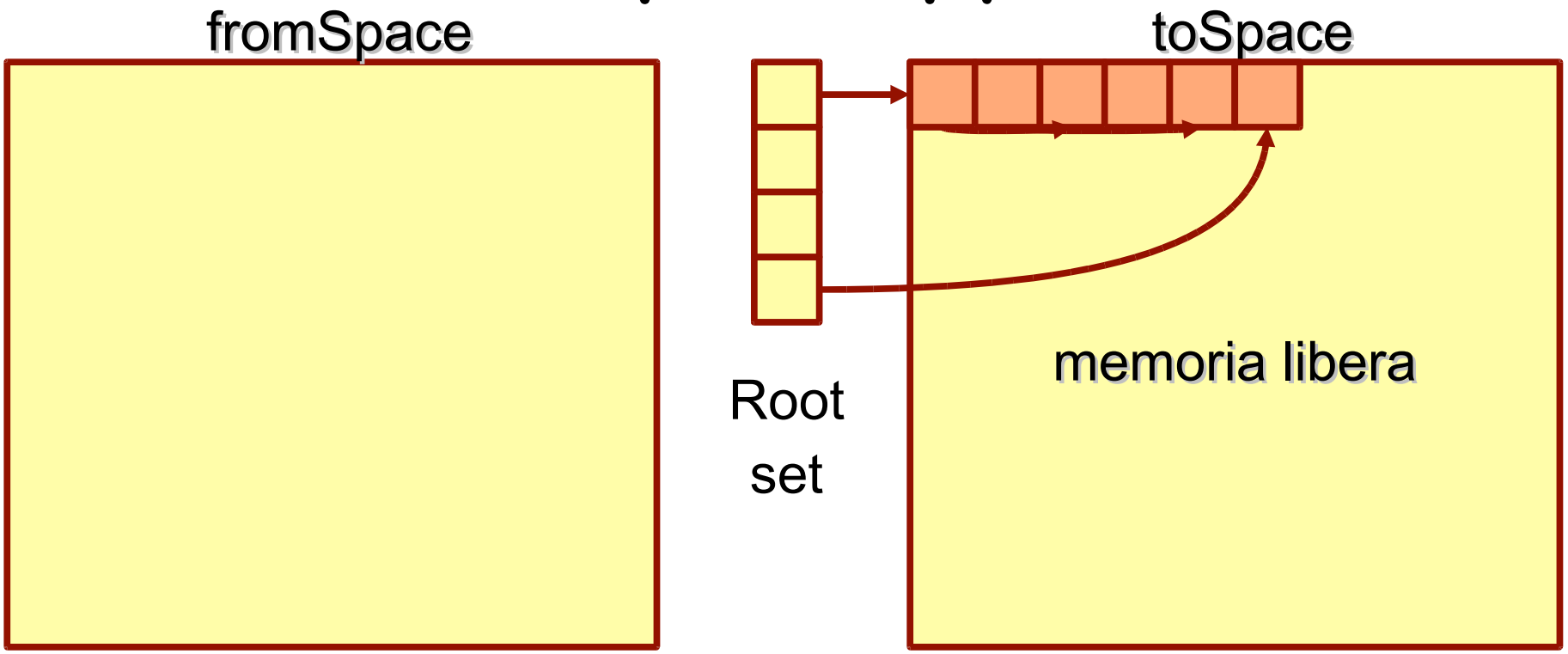
Osservazioni

- svantaggi
  - più passaggi necessari sullo heap (→ tecnica costosa se molti blocchi vivi):
    - Marcatura, calcolo nuova posizione blocco, calcolo puntatori, spostamento, ...
- vantaggi
  - no frammentazione
  - lista libera monoblocco (allocazione facile)

garbage collection: copia

- no fase marcatura: ma copia (e compattazione) dei blocchi vivi
- *GC* **stop & copy**: heap diviso in 2 semispazi (stessa dimensione)
  - A run-time: solo 1 in uso (fromSpace);
    - Memoria allocata all'estremità
    - memoria libera = unico blocco (che si assottiglia con le allocazioni)
  - memoria esaurita: chiamata al *GC*
    - a partire dal root set si copia (alg. di Cheney) nell'altro semispazio (toSpace) compattando
    - quindi toSpace e fromSpace si scambiano i ruoli
- Osservazioni
  - vantaggi
    - allocazione efficiente: soprattutto se e' noto il fabbisogno degli oggetti vivi contemporaneamente
    - no frammentazione

garbage collection
stop & copy



Quando la memoria del semispazio è esaurita:
Visita e copia delle strutture concatenate presenti nel semispzio corrente (fromSpace → toSpace) + compattazione estremità