UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

05 – Type Systems and Type Inference

Programmazione Avanzata

Anno di corso: 1

Anno accademico di offerta: 2024/2025

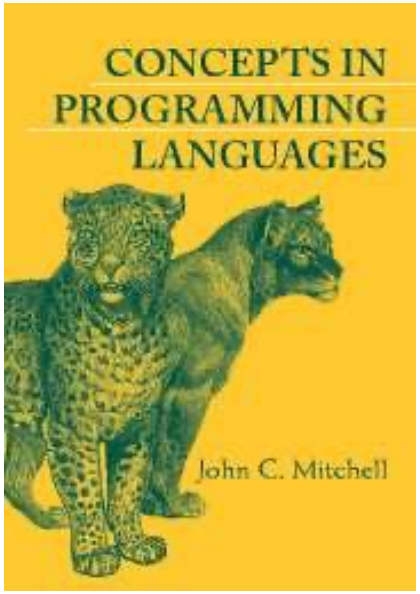
Crediti: 6

INGEGNERIA INFORMATICA

Prof. Claudio MENGHI

Dalmine

02 Ottobre 2024



Capitolo 6 - Type Systems and Type Inference

La sicurezza dei tipi nei linguaggi di programmazione

La sicurezza dei tipi in un programma è molto importante

- se l'esecutore (il PC o la macchina virtuale) non riesce a distinguere i tipi di un certo programma può facilmente causare errori
- molti attacchi sfruttano proprio ≈ nel controllo dei tipi di linguaggi diffusi come il C

> "*Well-typed programs never go wrong*."
>
> *Robert Milner*

Tipo

Tipo: Insieme di valori omogenei + operazioni che si possono fare

Esempi:

- tipi semplici: Integers, String,
- tipi strutturati come classi, ...
- funzioni: int -> bool
  - Funzione che da un intero mi dà un boolean
  - Anche le funzioni e i metodi definisco un tipo

Esempi di non tipi:

- numeri dispari
- array contenenti String e Integer

Dipende però dal linguaggio di programmazione

A cosa servono i tipi

Per organizzare e dare un nome ai concetti (documentazione)

- Spesso corrispondenti ai concetti nel dominio del problema che si vuole risolvere
- Indicare l'uso che si vorrà fare di certi identificatori (così il compilatore può controllare)

Per assicurarsi che sequenze di bit in memoria siano interpretate correttamente

- Per evitare errori come: 3 + true + "Angelo"

Per istruire il compilatore come rappresentare i dati

- Esempio short richiedono meno bit di int

Errori di tipi a livello Hardware

Confondere dati con programmi

- Caricando quindi nei registri della CPU possibilmente codici non corretti

> *Esempio*: cerco di eseguire un dato chiamando **x()** dove **x** non è una procedura ma un intero

Confondere tipi di dati semplici

> *Esempio*: eseguo **float_add(3,4.5)** con 3 int
> `float_add`: operazione della CPU che chiama una routine della FPU, se la CPU prende 3 come sequenza di bit float, potrebbe generare un errore hardware

Errori semantici

Il programma fa qualcosa che non è quello che dovrebbe fare

- Esempio con tipi primitivi: `int_add(3,4.5)`
- `01000000100100000000000000000000 = 4.5`
- `01000000100100000000000000000000 = 1083179008`

In questo caso la sequenza di bit che rappresenta 4.5 può essere interpretato come int ma non sarà uguale come valore

- Esempio con oggetti ed ereditarietà in Java

Sia Quadrato sottoclasse di Figura:

```
class Quadrato extends Figura
```

Se non riesco a distinguere istanze di Qu. e Fig.:

`Figura a1 = new Quadrato()` **OK**

`Quadrato b1 = new Figura()` **NO**: Quadrato potrebbe avere dei metodi in più che potrei invocare ma non trovare perché b1 è una Figura

Type safety: sicurezza dei tipi

Un linguaggio di programmazione L si dice type safe se non esiste programma scritto in L che possa violare la distinzione di tipi in L

Esempi di violazioni dei tipi:

- confondere interi e float
- chiamare una funzione attraverso un intero
- accedere ad una zona di memoria sbagliata (non memory safe)

Sicurezza di alcuni linguaggi

Ecco una tabella che riporta la sicurezza di alcuni linguaggi di programmazione molto diffusi

Safety	Linguaggio	Motivo
Non safe	C e C++	Type cast, aritmetica dei puntatori
Quasi safe	Pascal	Deallocazione esplicita e dangling pointers
Safe	Java, Lisp,Python	Controllo completo dei tipi

Problemi del C/C++

Il C/C++ ha un sistema dei tipi non sicuro (posso facilmente violare la distinzione di tipi)

Alcuni tipi errori

- Type cast
- Dereferenziazione del null, …
- Pointer arithmetic
- Accesso alla memoria non valida
  - Violazione spaziale come out of bound
  - Violazione temporale come dangling pointer

Quando si fa il type checking?

Tra i linguaggi type safe distinguiamo due categorie a seconda del momento in cui avviene il controllo dei tipi

run-time type checking

- Il controllo avviene durante l'esecuzione
- Esempio LISP: quando esegue l'istruzione (car x) - che applica car a x e car restituisce il primo elemento di una lista - controlla prima che x sia una lista

compile time type checking

- Il controllo avviene durante la compilazione
- Esempio ML: se compila f(x) controlla che se f sia A -> B e x : A

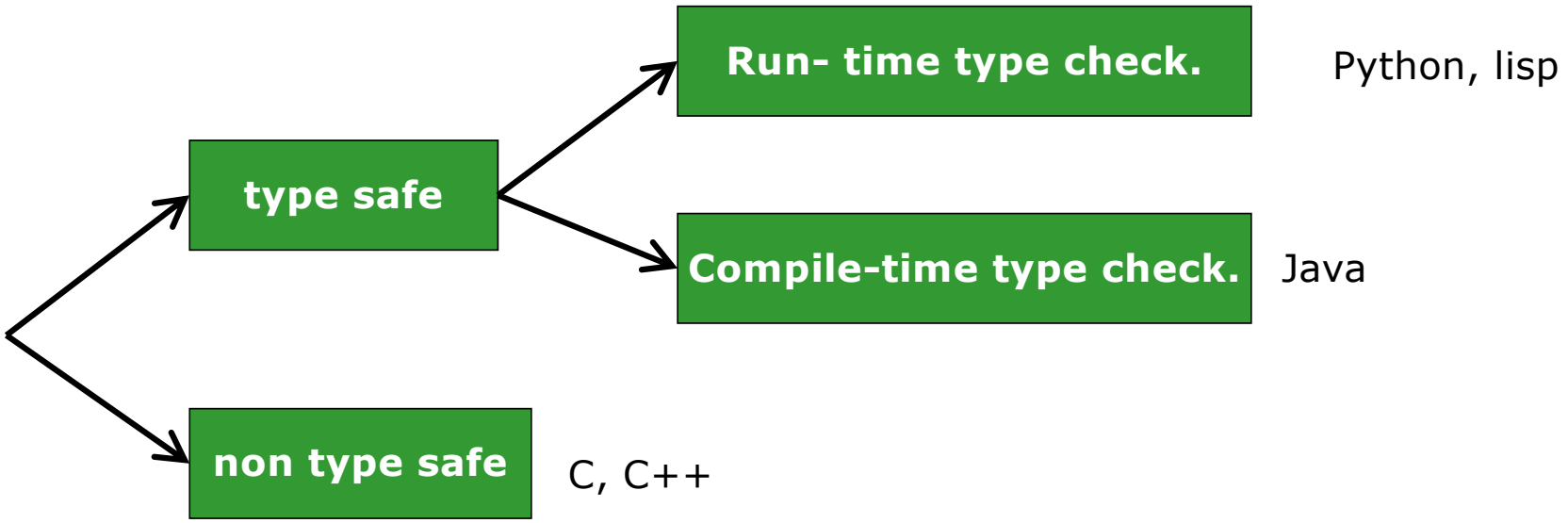
Conservativity of Compile-Time checking

I compilatori che eseguono controlli di tipo a compile-time sono conservativi: Trovano problem a comile-time anche se è possible che a run-time i programmi non diano errori.

I compilatori sono sound e conservative.

Form of Type Checking	Advantages	Disadvantages
Run-time	Prevents type errors	Slows program execution
Compile-time	Prevents type errors Eliminates run-time tests Finds type errors *before* execution and run-time tests	May restrict programming because tests are *conservative*.

Classificazione dei linguaggi



Vedi syllabus per approfondimento

Java

Java usa compile time, però dove il compilatore non è sicuro della sicurezza dei tipi, introduce un controllo run-time (conversioni dei tipi controllate)

considera la seguente istruzione

```
Quadrato a = (Quadrato) b
```

- con b dichiarato di classe Figura (padre di Quad.)
- la conversione al sottotipo `Quadrato` è corretta solo se b è effettivamente una istanza di `Quadrato` (o di una sottoclasse)
- tale controllo non si può fare in compilazione
- il compilatore introduce un controllo da fare durante l'esecuzione che b sia convertibile a `Quadrato`

Pro e contro

Entrambi gli approcci (run-time e compile time) prevengono errori di tipo, però:

- run-time checking rallenta l'esecuzione
  - controlla le conversioni di tipo ogni volta
- compile-time checking limita la flessibilità dei programmi
  - tutte le istruzioni anche non eseguite devono essere corrette
  - Il controllo è **conservativo**

alcuni programmi che non sono corretti compile time sono invece run time corretti

Pro e contro

Entrambi gli approcci (run-time e compile time) prevengono errori di tipo, però:

- run-time checking rallenta l'esecuzione
  - controlla le conversioni di tipo ogni volta
- compile-time checking limita la flessibilità dei programmi
  - tutte le istruzioni anche non eseguite devono essere corrette
  - Il controllo è **conservativo**

alcuni programmi che non sono corretti compile time sono invece run time corretti

[**Perché 1bpt**]

Dynamic Type checking in Python

In python ogni variabile ha un tipo e viene controllata (type safe) ma:

-Non è necessario dichiarare il tipo di una variabile

```
x = 5
print(type(x))
```

-Posso comunque specificare il tipo (con ancge conversioni esplicite)

```
x = float(20)
x = bool(5)
z = float("3")
```

-Controlla comunque che un'operazione sia type safe:

y = x / "pippo" -> errore non esiste l'operazione (in C?)

problemi

I tipi si possono ridefinire

-x = int(5)

-print(type(x))

-x = "pippo"

-print(type(x))

Problemi

-Se il mio codice contiene un errore di tipo e non lo eseguo non me ne accorgo

```
x = int(5)
if x < 5:
        y = x / "pippo"
print(x)
```

"duck" typing

If it walks like a duck and it quacks like a duck, then it must be a duck

With duck typing, an object is of a given type if it has all methods and properties required by that type.

Per le funzioni anche peggio

```
# define a sum function (intended to be used for integers)
def sum(x,y):
    return x+y

print( sum(8,'hello'))
```

Puoi usare mypy:

```
def sum(x:int,y:int) -> int:
    return x+y
print( sum(8,4))
```

Static typing e annotazioni

In alcuni linguaggi (es Rust, Xpand) non è necessario indicare il tipo di una variabile

ma poi non si può cambiare

```
let mut sum = 5 + 10;
println!("{}", sum);
sum = "pippo" → ERRORE
```

Anche in java dalla 10 in poi **local variable type inference**,

```
var id=0;// At this moment, compiler interprets
//variable id as integer.
id="34"; // This will result in compilation error
```

Flessibilità del run time chkng

In Lisp/python, possiamo scrivere

```
(cond ((< x 10) x) (else (car x)))
```
OK

alcune volte ci sara' errore (catturato dal lisp stesso) altre no - se x non è < 10 valuto car che si aspetta una lista

In Java, non posso scrivere

```
int x;
if (0>-1) { x++;} else { x = "ciao";}
```
NO

perchè assegna ad x int una String

eppure questo programma è type safe, perchè nessuna esecuzione causa errori di tipo (0 è sempre > -1)

Type Inference

E’ il processo di capire il tipo dei dati basandosi sul tipo delle espressioni nei quali appaiono

- type checking: il compilatore controlla che I tipi dichiarati dal programmatore sono in linea con le espressioni

- type inference: il tipo non e’ specificato e un’inferenza logica e’ richiesta per capire il tipo degli identificatory da usare

Type Inference: Esempio 1

Esempio

```
- fun f1(x) = x+2;
val f1 = fn : int → int
```

+ potrebbe essere applicato a vari tipi di parametri, tuttavia

Considerato che 2 è un intero, x deve essere a sua volta un intero

- ne segue che f1 va da interi ad interi

Type Inference: Esempio 2

Esempio

```
- fun f2(g,h) = g(h(0));
val f2 = fn : ('a → 'b) * (int → 'a) → 'b
```

f2 è parsato come ((‘a->’b)*(int->a’))->’b

- considerate che h e’ applicato a un intero, h va da interi ad un altro tipo (chiamiamolo ‘a) h: int->a’
- g prende un ‘a e lo trasforma in qualche cosa d’altro (chiamiamolo ‘b) g: a’->’b
- ne segue che f2 va da ‘a ad ’b (‘a->’b)
- f2 prende le due funzioni come argomento ((a’->’b)*(int->a’))->b’

Type Inference Algorithm

- Step 1: Assegna un tipo ad ogni espressione e sotto espressione. Utilizza il tipo conosciuto per ogni variabile conosciuta (per esempio, a 3 assegna il tipo int)

- Step 2: Genera un insieme di vincoli sui tipi utilizzando l'albero sintattico dell'espressione. Per esempio, se una funzione e' applicate ad un argomento, il tipo dell'argomento deve corrispondere al tipo del dominio della funzione

- Step 3: Risolvi questi vincoli per mezzo dell'unificazione (un metodo basato sulle sostituzioni per risolvere I sistemi di equazioni)

In sintesi

- Abbiamo visto:
  - l'importanza della sicurezza dei tipi
  - la definizione di linguaggio sicuro nei tipi
  - alcuni linguaggi sono safe altri no
- Ricordate che:
  - il C non è type safe – vedremo alcuni errori tipici
- Inoltre,
  - i linguaggi safe possono effettuare il controllo dei tipi o durante l'esecuzione (run-time) come il LISP o durante la compilazione (compile-time) come Java
  - i pro e contro dei due approcci sono: flessibilità (maggiore con runtime) e efficienza (maggiore con compile time)

“C is not Safe”

Alcune caratteristiche del linguaggio C e C++ che possono dare errori:

1. dereferenziazione del null
2. type cast non controllato
3. pointer arithmetic
4. accesso alla memoria non valida
   - violazione spaziale come out of bound
   - violazione temporale come dangling pointers

Queste caratteristiche rendono il C molto flessibile e veloce a discapito della sua sicurezza

- è responsabilità del programmatore stare attento a non introdurre difetti

Type Cast non Safe

Il C permette la conversione non controllata da un tipo ad un altro:

- da un tipo ad un sottotipo con possibile perdita di informazioni.
  - Esempio da double a int
- da intero ad una funzione per cercare di eseguire una certa locazione di memoria che potrebbe non essere un'istruzione corretta o fare qualcosa di non voluto

Programma corretto in C ma con type cast non safe:

```
double d;
int i;
…
i = d;
```

➔ possibile perdita di informazioni

Dereferenziazione di null

- La dereferenziazione di un puntatore in C non viene controllata
- Se accedo ad una cella puntata da un puntatore nullo ho “segmentation fault”, cioè un errore del sistema operativo

Programma con accesso tramite puntatore null

```
int main(){
int * ptr; ...
   ptr = NULL;
*ptr = 2;
}
```

Pointer arithmetic

Mediante l'aritmetica dei puntatori possiamo puntare a zone di memoria con tipi diversi

Esempio:

- se il puntatore `p` è definito di tipo `A*`
- l'espressione `*(p+i)` ha tipo `A`
- poiché il valore memorizzato a `p+i` potrebbe avere qualsiasi tipo
- l'assegnamento `x = *(p+i)` con `x` di tipo `A`, permette di memorizzare un valore di qualsiasi tipo in `x`

C non è memory safe

Inoltre mediante i puntatori si può facilmente accedere a memoria in modo scorretto

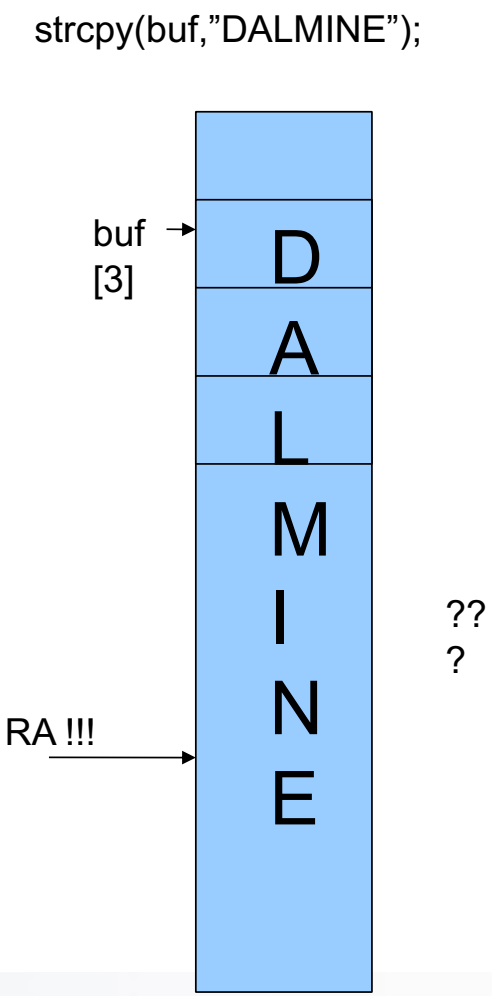
```
void f(int* p, int i, int v) {
p[i] = v;
}
```

Accedo in questo modo all'indirizzo p+i e p+i potrebbe contenere dati importanti o altro codice

- Posso modificare il return address di una chiamata di una procedura ed eseguire altro codice,posso modificare dei diritti o leggere informazioni riservate
- Tipico "buffer overflow" /buffer overrun

Buffer overrun

- A stack-based buffer overrun occurs when a buffer declared on the stack is overwritten by copying data larger than the buffer.
- For example copying the user input directly in the buffer using a strcpy,
- Variables declared on the stack are located next to the return address for the function's caller.
  - the result is that the return address for the function gets overwritten by an address chosen by the attacker.



https://secgroup.dais.unive.it/teaching/security-course/overwriting-return-address/

Esempio semplice

```
#include <string.h>

void foo (char *bar){
   char  c[12];
   strcpy(c, bar);  // no bounds checking...
}

int main (int argc, char **argv){
   foo(argv[1]);
}
```

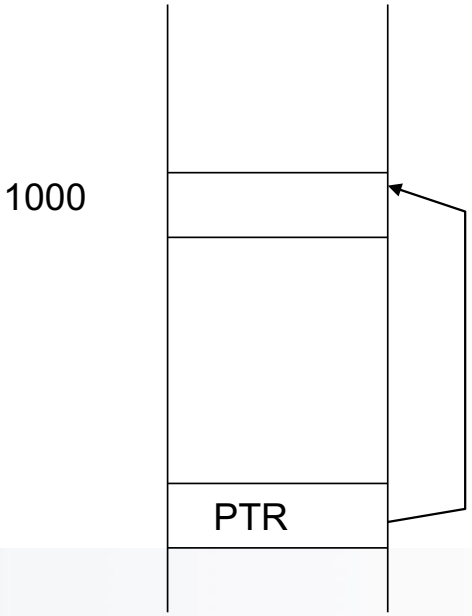
type cast e violazione memoria

I puntatori in C sono assimilati a interi

Tramite cast di dati interi a puntatori, posso accedere ad una zona di memoria a piacere

*Programma (OK in compilazione) con conversione da int a char**

```
int main(){
  char * PTR;
  PTR = 1000;
  *PTR = 'a';
}
```



Deallocazione esplicita e Dangling Pointers

In Pascal, C, ... una locazione puntata da un puntatore *p* può essere deallocata (liberata) dal programmatore: p è un “dangling pointer”

Ad esempio in C, faccio il **free** di un puntatore poi continuo ad usarlo

Un puntatore è dangling se punta ad una zona di memoria che è stata liberata per essere riutilizzata

- Il sistema operativo potrebbe allocare la stessa memoria nuovamente per memorizzare un altro tipo di valore
- Posso continuare ad usare p per accedere a questa memoria e rompere la type safety

Uso di free

- Posso usare anche un puntatore dopo averne fatto il free
- Vedi esempio

Memory leak

- In informatica, un memory leak ("perdita o fuoriuscita di memoria") è un particolare tipo di consumo non voluto di memoria dovuto alla mancata deallocazione dalla stessa, di variabili/dati non più utilizzati da parte dei processi. [wikipedia]

Esempio

Funzione che converte un intero in stringa corrispondente, restituendo il puntatore alla stringa ottenuta:

```
char * itoa(int i){
   char buf[20];
   sprintf(buf,"%d",i);
   return buf;
}
```

A cosa punta buf ? buf viene restituito ma punta ad un array locale che viene deallocato

Ricorda sprintf

Dangling Pointers sullo stack

Un esempio frequente di errore dovuto a dangling pointers è quando si usano puntatori a celle dello stack

Si verifica quando:

- si crea un puntatore p ad una zona A di memoria che è locale ad un metodo (ad esempio variabili locali)
- A è quindi allocata sullo stack
- A viene liberata all'uscita del metodo
- p è a questo punto un dangling pointer

Esempio in C++

Esempio in C++:

```
struct Point {int x; int y;};
struct Point *newPoint(int x,int y) {
   struct Point result = {x,y};
   return &result;
}
void bar() {
   struct Point *p = newPoint(1,2);
   p -> y = 1234;
}
```

newPoint restituisce un puntatore ad un oggetto (*result*) locale: in bar p è un dangling pointer

Soluzione

Come si possono evitare dangling pointers?

1. Evitare di puntare zone di memoria sullo stack ed usare la malloc:

```
struct Point * result =
   (struct Point*)malloc(sizeof(struct Point))
```

   La malloc crea puntatori a zone sicure

   Però attenzione che la sua gestione non è automatica come le variabili sullo stack

2. Uso del garbage collector (gc) invece che della deallocazione esplicita
   - Il gcc marca lui le zone da liberare e che si possono riutilizzare
   - Non usando free, il gc recura la memoria

Cosa fare per avere evitare tali errori?

Se vogliamo scrivere codice safe cosa possiamo fare?

- Scrivere attentamente, progettare prima, documentare, etc.

Se vogliamo essere sicuri che il nostro codice è safe?

- Due soluzioni possibili
  - usare linguaggi type safe (Java, lisp;..) e linguaggi + astratti
  - usare linguaggi come C e dei tools che ci aiutano a rendere i programmi C safe

In sintesi

- Abbiamo visto alcune fonti di violazioni di sicurezza del C:
  - dereferenziazione non controllata
  - typecast non controllato
  - aritmetica dei puntatori
    - Violazione “spaziale” della memoria, Buffer overflow, ...
  - deallocazione esplicita e dangling pointers
    - Violazione “temporale” della memoria, puntatori allo stack
- Le soluzioni proposte sono:
  - non usare C e passare a Java/C#, ...
  - usare C con tool e librerie che vedremo la prossima lezione

UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Domande?