



UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Corso 2° anno - 9 CFU

Paradigmi di programmazione

Professore:
Prof. Paolo Milazzo

Autore:
Filippo Ghirardini

Anno Accademico 2023/2024

Contents

1	Lambda calcolo	3
2	Sistemi di tipi	4
2.1	Perché?	4
2.2	Cosa sono i tipi?	4
2.3	Come funziona?	4
2.4	Come si progetta?	5
2.4.1	Specifiche del linguaggio	5
2.4.2	Regole di valutazione	5
2.4.3	Type checking	5
2.4.4	Composizionalità	5
2.5	Dimostrazione	6
2.5.1	Progresso	6
2.5.2	Conservazione	6
2.6	Lambda calcolo tipato	7
2.6.1	Sintassi	7
2.6.2	Semantica	7
2.6.3	Ambiente dei tipi	7
2.6.4	Type checker	8
2.6.5	Correttezza	8
2.6.6	Estensioni	9
2.6.7	Ricorsione	9
3	Object Oriented Programming	11
3.1	Modularità	11
3.2	Oggetti	11
3.2.1	Stato	11
3.2.2	Funzionalità	11
3.3	Strutture linguistiche	12
3.3.1	Object-based	12
3.3.2	Class-based	12
3.3.3	Confronto	13
3.4	Ereditarietà	13
3.5	Subtyping	13
3.5.1	Confronto	14
3.6	OOP in OCaml	14
3.6.1	Type weakening	15
3.6.2	Polimorfismo	15
3.6.3	Coercion	15
3.6.4	Classi	15
3.7	OOP in Java	17
3.7.1	Java Generics	17
3.7.2	Java Collection Framework	18
4	Garbage collection	19
4.1	Allocazione heap	19
4.2	Frammentazione	19
4.3	Algoritmi	19
4.3.1	Reference counting	19
4.3.2	Tracing	20
4.3.3	Generational GC	20

5	Programmazione concorrente	21
5.1	Comunicazione tra processi	22
5.1.1	Shared memory	22
5.1.2	Message passing	22

1 Lambda calcolo

2 Sistemi di tipi

2.1 Perché?

Dato che nel lambda calcolo i programmi e i valori sono funzioni possiamo facilmente scrivere programmi che non sono corretti rispetto all'uso inteso dei valori. Ad esempio:

Esempio 2.1 (Errore tipi). Nella seguente espressione si può applicare 0 a *False*, ottenendo quindi un risultato che però non ha alcun senso:

$$False\ 0 = (\lambda t.\lambda f.f)(\lambda z.\lambda s.z) \rightarrow \lambda f.f \quad (1)$$

Analogamente per una macchina tutto è un bit: istruzioni, dati e operazioni. Un esempio più pratico è il seguente:

Esempio 2.2. L'istruzione nel corpo dell'*if* contiene un errore di tipo (stringa divisa per un intero). Se non avessimo il controllo dei tipi l'unico modo per scoprire l'errore sarebbe eseguire numerosi test per riuscire a coprire tutte le possibilità, fino ad entrare nel corpo dell'*if*. Richiederebbe tempo e risorse e non ci garantisce neanche la certezza di aver provato tutti i casi possibili.

```
if (condizione_complicata) {
  return "hello"/10;
}
```

Se in certi linguaggi di programmazione ci troveremmo davanti ad errori di esecuzione, in altri (come ad esempio JavaScript) otterremmo un errore nel risultato in quanto l'interprete proverebbe a fare un cast manuale.

Concludendo, la mancanza di **type safety** aumenta il numero di bug, rendendo così un software meno funzionale e più vulnerabile.

2.2 Cosa sono i tipi?

I **sistemi di tipo** sono meccanismi che permettono di rilevare in anticipo errori di programmazione.

Definizione 2.1 (Tipo). *Il tipo è un **attributo** di un dato che descrive come il linguaggio di programmazione permetta di usare quel particolare dato.*

Un tipo serve quindi a limitare i valori che un'espressione può assumere, che operazioni possono essere effettuate sui dati e in che modo questi ultimi possono essere salvati.

Definizione 2.2 (Sistema dei tipi). *Un sistema dei tipi è un metodo **sintattico**, **effettivo** per dimostrare l'assenza di comportamenti anomali del programma **strutturando** le operazioni del programma in base ai tipi di valori che calcolano.*

Analizziamo i tre aspetti:

- *Sintattico*: l'analisi viene effettuata dal punto di vista sintattico
- *Effettivo*: si può definire un algoritmo che effettui questa analisi
- *Strutturale*: i tipi assegnati si ottengono in maniera **composizionale** dalle sottoespressioni.

2.3 Come funziona?

Un sistema dei tipi associa dei tipi ai valori calcolati. Esaminando il flusso dei valori calcolati prova a dimostrare che non avvengano errori (di tipo, non in generale)facendo un controllo, che può avvenire in due modi:

- *Statico*: avviene in fase di compilazione, non degradando le prestazioni
- *Dinamico*: avviene in fase di esecuzione e aumenta il tempo di esecuzione

2.4 Come si progetta?

2.4.1 Specifiche del linguaggio

Prendiamo come esempio il seguente **linguaggio**:

Espressioni	Valori	Valori numerici	Tipi
$E ::=$	$V ::=$	$NV ::=$	$T ::=$
true	true	$0 1 2 \dots$	Bool
false	false		Nat
NV	NV		
if E then E else E			
succ E			
pred E			
isZero E			

2.4.2 Regole di valutazione

Avremo le seguenti **regole di valutazione**:

$$\text{if true then } E1 \text{ else } E2 \rightarrow E1 \quad (2)$$

$$\text{if false then } E1 \text{ else } E2 \rightarrow E2 \quad (3)$$

$$\frac{E \rightarrow E'}{\text{if } E \text{ then } E1 \text{ else } E2 \rightarrow \text{if } E' \text{ then } E1 \text{ else } E2 \rightarrow E1} \quad (4)$$

$$\frac{E \rightarrow E'}{\text{succ } E \rightarrow \text{succ } E'} \quad \frac{m = n + 1}{\text{succ } E \rightarrow \text{succ } E'} \quad (5)$$

$$\frac{E \rightarrow E'}{\text{pred } E \rightarrow \text{pred } E'} \quad \frac{n > 0, m = n - 1}{\text{pred } n \rightarrow m} \quad \text{pred } 0 \rightarrow 0 \quad (6)$$

$$\frac{E \rightarrow E'}{\text{isZero } E \rightarrow \text{isZero } E'} \quad \text{isZero } 0 \rightarrow \text{true} \quad \frac{n > 0}{\text{isZero } n \rightarrow \text{false}} \quad (7)$$

2.4.3 Type checking

Il **controllo di tipo** definisce una relazione binaria (E, T) che associa il tipo T all'espressione E . Questo ha due caratteristiche principali:

- Utilizza il *metodo sintattico*
- Le regole sono definite per *induzione strutturale* sul programma

Le regole sono le seguenti:

$$\text{true} : \text{Bool} \quad \text{false} : \text{Bool} \quad n : \text{Nat} \quad (8)$$

$$\frac{E : \text{Nat}}{\text{succ } E : \text{Nat}} \quad \frac{E : \text{Nat}}{\text{pred } E : \text{Nat}} \quad \frac{E : \text{Nat}}{\text{isZero } E : \text{Bool}} \quad (9)$$

$$\frac{E : \text{Bool}, E1 : T, E2 : T}{\text{if } E \text{ then } E1 \text{ else } E2} \quad (10)$$

2.4.4 Composizionalità

I sistemi di tipo sono **imprecisi**: non definiscono esattamente quale tipo di valore sarà restituito da ogni programma, ma solo un'**approssimazione conservativa**.

Esempio 2.3. La seguente espressione:

$$\text{if } E \text{ then } 0 \text{ else false} \quad (11)$$

potrebbe restituire come risultato sia un *Bool* che un *Nat* a seconda del valore di E . Il controllo dei tipi quindi non permetterà che possano esserci due risultati diversi, riducendo la precisione ma mantenendo la sicurezza.

Questo avviene proprio per garantire la **composizionalità**, infatti ad esempio la regola dell'equazione 10 necessita che $E1$ ed $E2$ abbiano lo stesso tipo.

2.5 Dimostrazione

La **correttezza** del sistema di tipo è espressa da due proprietà:

- Progresso
- Conservazione

2.5.1 Progresso

Definizione 2.3 (Progresso). *Se $E : T$ allora E è un valore oppure $E \rightarrow E'$ per una qualche espressione E' .*

In pratica, un'espressione ben tipata non si blocca a run-time. Può fare sempre un passo a meno che non sia un valore.

Proof. Utilizziamo l'induzione sulla struttura di derivazione di $E : T$.

I *casi base* sono i seguenti:

- $true : Bool$
- $false : Bool$
- $0|1|2|\dots : Nat$

I *casi induttivi* sono tutti molto simili, vediamo quello per la formula 10.

Per *ipotesi induttiva* abbiamo due casi:

- $E1$ è un valore. In questo caso deve essere $true$ o $false$ e le regole della semantica fanno fare un **passo** del tipo $E \rightarrow E1$ o $E \rightarrow E2$
- Esiste $E4$ tale che $E1 \rightarrow E4$. In questo caso si applica la regola 4 e si esegue un **passo**.

□

2.5.2 Conservazione

Definizione 2.4 (Conservazione). *Se $E : T$ e $E \rightarrow E'$ allora $E' : T$.*

In pratica i tipi sono preservati dalle regole di esecuzione.

Proof. Utilizziamo l'induzione come nella precedente dimostrazione.

I *casi base* sono immediati: $true$, $false$ e $0|1|2|\dots$ sono valori e di conseguenza non fanno nessun passo. Anche qui per i *casi induttivi* vediamo quello per la formula 4. Per l'ipotesi induttiva abbiamo due casi:

- $E1$ è un valore:
 - $true$: in questo caso $E \rightarrow E2$ e sappiamo già per ipotesi induttiva che $E2 : T$ (sappiamo che il passo ha successo)
 - $false$: in questo caso $E \rightarrow E3$ e $E3 : T$
- Esiste $E4$ tale che $E1 \rightarrow E4$. Questo implica:

$$E = \text{if } E1 \text{ then } E2 \text{ else } E3 \rightarrow \text{if } E4 \text{ then } E2 \text{ else } E3$$

Dato che per ipotesi induttiva $E1 : Bool$ abbiamo che $E4 : Bool$ e, grazie alle derivazioni che valgono per ipotesi $E2 : T$ e $E3 : T$, possiamo derivare applicando la regola 10.

□

2.6 Lambda calcolo tipato

Nel nostro caso descriveremo il lambda calcolo tipato con valori di tipo *booleano* e *funzione*.

2.6.1 Sintassi

Tipi	Linguaggio	Valori
$\tau ::=$	$e ::=$	$V ::=$
$Bool$	x	$fun\ x : \tau = e$
$\tau \rightarrow \tau$ ¹	$fun\ x : \tau = e$	$true$
	$Apply(e, e)$	$false$
	$true$	
	$false$	
	$if\ e\ then\ e\ else\ e$	

2.6.2 Semantica

Di seguito le regole della semantica del linguaggio:

$$Apply(fun\ x : \tau = e_1, v) \rightarrow e_1\{x := v\} \quad (12)$$

$$\frac{e_1 \rightarrow e'}{Apply(e_1, e_2) \rightarrow Apply(e', e_2)} \quad (13)$$

$$\frac{e_2 \rightarrow e'}{Apply(v, e_2) \rightarrow Apply(v, e')} \quad (14)$$

A questi corrisponde la β -riduzione con strategia call-by-value vista nel lambda calcolo.

$$\frac{e_1 \rightarrow e_4}{if\ e_1\ then\ e_2\ else\ e_3 \rightarrow if\ e_4\ then\ e_2\ else\ e_3} \quad (15)$$

$$if\ true\ then\ e_2\ else\ e_3 \rightarrow e_2 \quad (16)$$

$$if\ false\ then\ e_2\ else\ e_3 \rightarrow e_3 \quad (17)$$

Queste regole invece sono per le espressioni condizionali.

2.6.3 Ambiente dei tipi

L'ambiente dei tipi ci serve per tenere conto delle associazioni variabile-tipo in modo da poter eseguire correttamente le regole di semantica.

L'ambiente è una **funzione** di dominio finito e la indichiamo come:

$$\Gamma = x_1 : \tau_1, x_2 : \tau_2 \dots x_k : \tau_k$$

Per ottenere il tipo τ_i dal valore x_i si usa la notazione

$$\Gamma(i) = \tau_i$$

Per **estendere** l'ambiente aggiungendo associazioni si usa la notazione

$$\Gamma, x : \tau$$

Per applicare l'ambiente nelle regole di semantica si usa la notazione

$$\Gamma \vdash e : \tau$$

Le regole sono applicate dal compilatore in fase di *analisi statica*. L'ambiente dei tipi è chiamato **symbol table**.

¹Nota bene: l'associazione avviene a destra. Quindi $Bool \rightarrow Bool \rightarrow Bool = Bool \rightarrow (Bool \rightarrow Bool)$

2.6.4 Type checker

$$\Gamma \vdash \text{true} : \text{Bool} \quad \Gamma \vdash \text{false} : \text{Bool} \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad (18)$$

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau} \quad (19)$$

Poi abbiamo il tipo delle funzioni, ovvero $\tau_1 \rightarrow \tau_2$, ovvero prende in ingresso un argomento di tipo τ_1 e restituisce un risultato di tipo τ_2 .

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun } x : \tau_1 = e : \tau_1 \rightarrow \tau_2} \quad (20)$$

In pratica, estendiamo l'ambiente con il tipo dell'argomento in ingresso (che è noto), e poi troviamo il tipo dell'espressione e che sarà anche il tipo del risultato della funzione. Questa estensione "annulla" tutte le dichiarazioni precedenti della stessa funzione per questo scope.

Infine per l'applicazione delle funzioni:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \text{Apply}(e_1, e_2) : \tau_2} \quad (21)$$

2.6.5 Correttezza

Dimostrazione del progresso. Utilizziamo l'induzione sulle derivazioni di tipo.

I casi base sono identici alla dimostrazione 2.5.1.

Il caso delle variabili è banale.

Il caso dell'astrazione di funzione è immediato dato che le funzioni stesse sono valori.

Per il caso dell'applicazione di funzione

$$e = \text{Apply}(e_1, e_2), \emptyset \vdash e_1 : \tau_1 \rightarrow \tau_2, \emptyset \vdash \tau_1$$

Per *ipotesi induttiva* abbiamo due casi:

- Le due espressioni possono fare un passo: applichiamo le regole di riduzione dell'applicazione e terminiamo
- Le due espressioni sono entrambi valori: dovremo avere e_1 nella forma del tipo $\text{fun } x : \tau_1 = e' : \tau_1 \rightarrow \tau_2$ e possiamo applicare la regola 12

□

Dimostrazione della conservazione. Si dimostra per induzione strutturale sulle regole di tipo. Anche qui il caso difficile è quello per l'applicazione

$$e = \text{Apply}(e_1, e_2)$$

Quindi vale che:

$$\begin{aligned} &\Gamma \vdash e : \tau \\ &\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \\ &\Gamma \vdash e_2 : \tau_2 \\ &\tau = \tau_2 \\ &e \rightarrow e' \end{aligned}$$

Dobbiamo dimostrare che:

$$\Gamma \vdash e' : \tau_2$$

La seconda affermazione comporta che

$$e_1 = \text{fun } x : \tau_1 = e_3 \quad \Gamma, x : \tau_1 \vdash e_3 : \tau_2$$

che però non è ciò che ci serve: noi vogliamo ottenere

$$e' = e_3\{x := e_2\}$$

Dobbiamo in qualche modo eseguire la sostituzione, e per questo usiamo un lemma ad hoc:

Lemma 2.6.5.1 (Substitution lemma). I tipi sono preservati dall'operazione di sostituzione.

$$\begin{aligned} \Gamma, x : \tau_1 \vdash e : \tau \\ \Gamma \vdash e_1 : \tau_1 \\ \Gamma \vdash e\{x = e_1\} : \tau \end{aligned}$$

Per dimostrarlo si usa l'induzione sulla derivazione del primo punto.

□

2.6.6 Estensioni

Cambiamo la sintassi del linguaggio per implementare le *operazioni numeriche* e le *dichiarazioni locali*:

$e ::=$	Espressioni
x	Variabili
$\text{fun } x : \tau = e$	Funzioni
$\text{Apply}(e, e)$	Applicazioni
true	Costante true
false	Costante false
n	Costanti numeriche
$e \oplus e$	Operazioni binarie
$\text{if } e \text{ then } e \text{ else } e$	Condizionale
$\text{let } x = e_1 \text{ in } e_2 : \tau_2$	Dichiarazioni locali

Aggiungiamo le seguenti regole di valutazione:

$$\frac{e_1 \rightarrow e'}{\text{let } x = e_1 \text{ in } e_2 : \tau_2 \rightarrow \text{let } x = e' \text{ in } e_2 : \tau_2} \quad (22)$$

$$\text{let } x = v \text{ in } e_2 : \tau_2 \rightarrow e_2\{x = v\} : \tau_2 \quad (23)$$

E le seguenti regole di tipo:

$$\Gamma \vdash n : \text{Nat} \quad (24)$$

$$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \frac{\oplus : \tau_1 \times \tau_2 \rightarrow \tau}{\Gamma \vdash e_1 \oplus e_2 : \tau} \quad (25)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad (26)$$

2.6.7 Ricorsione

Nel lambda calcolo tipato non possiamo utilizzare il combinatore Ω .

Dobbiamo quindi introdurre un generatore **aux** che, applicato ad una funzione iE approssima il comportamento di una ipotetica funzione (ad esempio $isEven$) fino a n . $iE k$ con $k \leq n$ restituisce il valore calcolato da $isEven k$. Allora $aux iE k$ restituisce una migliore approssimazione di $isEven$ fino a $k + 2$.

```
let aux = fun f:Nat -> Bool =
  fun x:Nat=
    if (isZero x) then true else
    if (isZero(pred x)) then false else
      f(pred(pred x))
aux: (Nat -> Bool) -> Nat -> Bool
```

Aggiungiamo al linguaggio un costrutto fix tale che $isEven = fix aux$. Applicando fix ad aux si ottiene il limite delle approssimazioni, ovvero il **punto fisso**.

Estendiamo la sintassi aggiungendo $fix e$, le seguenti regole di valutazione

$$\frac{e \rightarrow e'}{fix e \rightarrow fix e'} \quad (27)$$

$$fix(\text{fun } f : \tau = e) \rightarrow e[f = (fix(\text{fun } f : \tau = e))] \quad (28)$$

e le regole di tipo

$$\frac{\Gamma \vdash e : \tau \rightarrow \tau}{\Gamma \vdash \text{fix } e : \tau} \quad (29)$$

La versione semplificata della sintassi, che poi viene usata anche in Ocaml, è la seguente:

```
let rec x: <Type> = e in e'
```

che corrisponde a

```
let x = fix (fun x: <Type> = e) in e'
```

Esempio 2.4 (Esempio ricorsione semplificata).

```
let rec isEven: Nat -> Bool =
  fun x:Nat =
    if(isZero x) then true else
    if (isZero (pred x)) then false
    else isEven(pred (pred x))
in isEven 7;
```

3 Object Oriented Programming

3.1 Modularità

La **modularizzazione** è una tecnica utilizzata nella programmazione per sviluppare un programma per livelli di astrazione. Ad esempio:

- Sotto il profilo **linguistico** c'è l'astrazione procedurale, che permette di scomporre il problema in sottoproblemi
- Sotto il profilo dei **tipi di dato** abbiamo i tipi di dato astratti
- Sotto il profilo delle **tecniche di compilazione** troviamo la compilazione separata ed il linking

La combinazione di **astrazione** e **problem solving** consentono di trovare soluzioni efficienti a problemi complessi.

Nell'ambito dell'*ingegneria del software* la modularità permette di suddividere il lavoro tra sviluppatori diversi:

- Si definiscono le **specifiche** delle diverse parti (anche grazie ai diagrammi UML)
- Ognuno sviluppa la sua

3.2 Oggetti

Un software, nell'ambito della OOP, è un insieme di oggetti cooperanti, dove ogni oggetto è caratterizzato da:

- Uno **stato**
- Un insieme di **funzionalità**
- Un'**identità**, ovvero il nome che individua l'oggetto
- Il **ciclo di vita**, quindi la creazione, il riferimento e la disattivazione
- La **locazione** di memoria

I vari oggetti comunicano tra di loro scambiandosi messaggi.

3.2.1 Stato

Lo stato di un oggetto è rappresentato da un gruppo di attributi. Deve rispettare la proprietà di **incapsulamento**:

Definizione 3.1 (Incapsulamento). *Lo stato di un oggetto non dovrebbe essere accessibile dagli altri oggetti. Bisogna fare **information hiding**.*

Lo stato di un programma è invece l'insieme degli stati degli oggetti che lo compongono.

3.2.2 Funzionalità

Le funzionalità di un oggetto sono rappresentate da un gruppo di **metodi** che l'oggetto mette a disposizione degli altri oggetti. Questi descrivono il comportamento dell'oggetto, ovvero come questo risponde ad un messaggio ricevuto (modificando il proprio stato o interagendo con altri oggetti). Solitamente:

- L'**invio** di un messaggio corrisponde ad una chiamata di metodo
- La **risposta** di un messaggio corrisponde alla restituzione del risultato

3.3 Strutture linguistiche

I linguaggi di programmazione supportano l'OOP seguendo due approcci principali:

- **Object-based**, come JS, Self, Lua
- **Class-based**, come C++,Java,Scala

3.3.1 Object-based

In questo approccio gli oggetti vengono trattati come **record**. I campi possono essere associati a funzioni (**metodi**) e questi possono accedere ai campi stessi tramite il riferimento *this*. In alcuni casi è anche possibile **modificare** la struttura dell'oggetto dinamicamente.

Esempio 3.1 (JS Object-based).

```
let mario = {
  nome: "Mario",
  cognome: "Rossi",
  eta: 35,
  compleanno: function() {
    this.eta += 1;
  }
}

console.log(mario.nome); // Mario
console.log(mario.eta); // 35
mario.compleanno();
console.log(mario.eta); // 36

// Aggiungo dinamicamente un metodo
mario.nomeCompleto = function() {
  return this.nome + " " + this.cognome;
}
console.log(mario.nomeCompleto()); // Mario Rossi
```

È anche possibile creare un oggetto definendo una **funzione costruttore** che può essere richiamata con new:

Esempio 3.2 (JS Costruttore).

```
function Persona(n, c, e) {
  this.nome = n; this.cognome = c; this.eta = e;
  this.compleanno = function() { this.eta++; }
  this.nomeCompleto = function() {
    return this.nome + " " + this.cognome;
  }
}

anna = new Persona("Anna","Rossi",33) ;;
console.log(anna.nomeCompleto()); // Anna Rossi
anna.compleanno();
console.log(anna.eta); // 34
```

3.3.2 Class-based

L'approccio Class-based prevede un concetto di **classe** che definisce il contenuto (variabili e metodi) di oggetti di un certo **tipo**. Gli oggetti vengono creati successivamente come **istanze** di una certa classe.

Esempio 3.3 (JS Class-based).

```
class Persona {
  constructor(n,c,e) {
    this.nome=n; this.cognome=c; this.eta=e;
  }
}
```

```

    compleanno() { this.eta++; }
    nomeCompleto() {
        return this.nome + " " + this.cognome;
    }
}
rosa = new Persona("Rosa", "Bianchi", 25);
console.log(rosa.nomeCompleto()); // Rosa Bianchi
rosa.compleanno();
console.log(rosa.eta); // 26

```

3.3.3 Confronto

Se l'approccio **object-based** dà maggiore flessibilità al programmatore (non serve prima definire una classe) rende anche la vita difficile ai controlli di tipo statico in quanto, visto che la struttura dell'oggetto può cambiare a tempo di esecuzione, sarà difficile predire il tipo.

Al contrario l'approccio **class-based** necessita di più disciplina da parte del programmatore (deve implementare le classi) ma consente di fare controlli statici sugli oggetti, dove il tipo sarà legato alla classe da cui è stato istanziato (**nominal typing**).

3.4 Ereditarietà

L'ereditarietà è una funzionalità che consente di definire una classe sulla base di un'altra esistente.

I linguaggi **object-based** mantengono per ogni oggetto una lista di **prototipi**, che sono tutti gli oggetti da cui esso eredita funzionalità. La gestione è molto complessa.

Esempio 3.4 (Prototipi).

```

// Costruttore di Studente che eredita da Persona
function Studente(m, n, c, e) {
    this.matricola = m;
    this.__proto__ = new Persona (n,c,e); // prototipo
}
let luigi = new Studente ("1231", "Luigi", "Verdi", 22);
console.log(luigi.matricola); // 1231
console.log(luigi.nomeCompleto()); // Luigi Verdi

```

I linguaggi **class-based** consentono di definire una classe come **estensione** di un'altra. La nuova classe eredita tutti i membri (valori e metodi) della precedente con la possibilità di aggiungerne di nuovi o ridefinirne alcuni (**overriding**).

Esempio 3.5 (Estensione).

```

// Classe Studente che estende la classe Persona
class Studente extends Persona {
    constructor(m,n,c,e) {
        super (n,c,e); // Studente acquisisce i campi di Persona
        this.matricola = m; // NO lista prototipi
    }
}
let giada = new Studente("7212", "Giada", "Neri", 21);
console.log(giada.matricola); // 7212
console.log(giada.nomeCompleto()); // Giada Neri

```

3.5 Subtyping

L'ereditarietà introduce il concetto di **sottotipo**. Idealmente un oggetto B estensione di un oggetto A dovrebbe poter essere usato ovunque si possa usare A.

I linguaggi **object-based** usano il **subtyping strutturale**:

Definizione 3.2 (Subtyping strutturale). *Un oggetto B è sottotipo di un oggetto A se contiene almeno tutti i membri pubblici che sono presenti anche in A.*

I linguaggi **class-based** usano il **subtyping nominale**:

Definizione 3.3 (Subtyping nominale). *Il tipo di un oggetto corrisponde alla classe da cui è stato istanziato: il nome della classe diventa il tipo.*

Un tipo B è sottotipo di un tipo A se la classe B è stata definita sintatticamente come estensione della classe A . Vale la proprietà transitiva.

3.5.1 Confronto

Vediamo il confronto tra i due approcci:

- **Structural subtyping**, più flessibile:
 - Non è necessario dire esplicitamente chi estende chi
 - Favorisce il **polimorfismo**: la relazione di sottotipo è più debole e ci sono più oggetti l'unio sottotipo dell'altro
- **Nominal subtyping**, più rigoroso:
 - Mette in relazione di sottotipo solo classi estese in maniera esplicita dal programmatore
 - È più semplice da verificare per l'interprete

3.6 OOP in OCaml

Un **oggetto** in OCaml è un valore costituito da **campi** e **metodi**. Possono essere creati direttamente senza specificare una classe (**object-based**). Il tipo di un oggetto è dato dai metodi che esso contiene e non dai campi.

Esempio 3.6 (Oggetto in OCaml).

```
(* oggetto che realizza uno stack *)
let s = object
  (* campo mutabile che contiene la rappresentazione dello stack *)
  val mutable v = [0; 2] (* Assumiamo per ora inizializzato non vuoto *)

  (* metodo pop *)
  method pop =
    match v with
    | hd :: tl ->
      v <- tl;
      Some hd
    | [] -> None

  (* metodo push *)
  method push hd =
    v <- hd :: v
end ;;

s#pop ;;
- : int option = Some 0
s#pop ;;
- : int option = Some 2
s#pop ;;
- : int option = None
s#push 9 ;;
- : unit = ()
s#pop ;;
- : int option = Some 9
```

3.6.1 Type weakening

Prendiamo ad esempio il seguente oggetto:

Esempio 3.7.

```
let s = object
  val mutable v = [] (*Lista vuota*)
  method pop = ...
  method push hd = ...
end ;;
```

In questo caso il tipo inferito sarà

$$vals :< pop : ' _weak option; push : ' _weak \rightarrow unit > = < obj >$$

che contiene variabili di tipo, ma non è veramente **polimorfo**. Benchè sia mutabile, la variabile *v* non potrà avere tipi diversi in momenti diversi dell'esecuzione. Il tipo concreto che sarà usato non è noto al momento della dichiarazione, quindi il **type checker** indebolisce temporaneamente il tipo inferito e lo ricalcherà appena possibile istnaziandolo con il tipo concreto.

Esempio 3.8.

```
s;;
val s : < pop : ' _weak option; push : ' _weak -> unit > = <obj>
s#pop ;;
- : ' _weak option = None
s#push 5 ;;
- : unit = ()
s ;;
val s : < pop : int option; push : int -> unit > = <obj>
```

3.6.2 Polimorfismo

Quando si definisce una funzione che prende un oggetto come parametro, il **tipo** dell'oggetto viene inferito dai metodi che sono invocati a partire dall'oggetto:

Esempio 3.9. TODO

Si noti come i **puntini** enfatizzano lo **structural subtyping**: la funzione accetta un qualunque sottotipo di $< width : int >$, ossia qualunque oggetto che contiene almeno il metodo *width*. Si applica la regola di **subsumption** che consente di tipare un'espressione con un suo supertipo:

$$\frac{\Gamma \vdash e : S \quad S <: T}{\Gamma \vdash e : T}$$

3.6.3 Coercion

In alcuni casi può essere necessaria una **type coercion** esplicita che in OCaml si fa tramite l'operatore $:>$. Questo forza il type checker a trattare una certa espressione *e* come se fosse di un tipo *t*.

Il tipo a cui si fa la conversione deve essere un **supertipo** dell'espressione, quindi avere meno metodi.

Esempio 3.10 (Type coercion).

```
let (x:shape) = ((square 2) :> shape) ;; (* OK *)
let (y:square) = ((shape 4.0) :> square) ;; (* ERRORE!! *)
```

3.6.4 Classi

Nonostante OCaml sia **object-based** fornisce dei costrutti di classe in modo da permettere l'**ereditarietà**.

Esempio 3.11 (Classi in OCaml).

```

class istack = object (* classe per stack di interi *)
  val mutable v = [0; 2] (* inizializzato non vuoto *)
  method pop =
    match v with
    | hd :: tl ->
      v <- tl;
      Some hd
    | [] -> None

  method push hd =
    v <- hd :: v
end ;;

let s = new istack ;;
val s : istack = <obj>
s#pop ;;
- : int option = Some 0

```

Una classe può prevedere parametri:

- Di **costruzione**, che vanno passati al momento dell'istanziamento
- Di **tipo**, che la rendono **polimorfa**

Esempio 3.12 (Parametri classi OCaml).

```

class ['a] stack init = object (* classe polimorfa per stack *)
  val mutable v : 'a list = init (* init parametro costruttore *)
  method pop =
    match v with
    | hd :: tl ->
      v <- tl;
      Some hd
    | [] -> None

  method push hd =
    v <- hd :: v
end ;;

let s = new stack ["pippo"] ;;
val s : string stack = <obj>
s#pop ;;
- : string option = Some "pippo"

```

Una classe può ereditare da un'altra:

Esempio 3.13 (Ereditarietà classi OCaml).

```

class sstack init = object (* classe per stack di stringhe *)
  inherit [string] stack init (* eredita da stack *)

  method concat = (* aggiunge un nuovo metodo *)
    List.fold_left (^) v
end ;;

let b = new sstack [" "; "world!"] ;;
val b : sstack = <obj>
b#push "Hello" ;;
- : unit = ()
b#concat ;;
- : string = "Hello world!"

```

e permette di fare overriding:

Esempio 3.14 (Overriding metodi OCaml).

```
(* classe per stack di int che raddoppia i valori inseriti *)
class double_stack init = object
  (* super: l'oggetto da estendere in fase di istanziamento *)
  inherit [int] stack init as super

  method push hd = (* ridefinisce un metodo *)
    super#push ( hd * 2 )
end ;;

let ds = new double_stack [] ;;
val ds : double_stack = <obj>
ds#push 5 ;;
- : unit = ()
ds#pop ;;
- : int option = Some 10
```

3.7 OOP in Java

Java utilizza un approccio di compilazione + interpretazione, con un **bytecode** intermedio. Il bytecode è utilizzabile in ogni architettura per cui sia disponibile un interprete (Java Virtual Machin) senza bisogno di ricompilare. Consente di fare controlli statici (compilatore) e dinamici (interprete).

3.7.1 Classi

Una classe in Java consiste di **variabili** e **metodi** e può prevedere uno o più **costruttori** per inizializzare le variabili.

Definizione 3.4 (Principio di astrazione). *Ogni tipologia di entità identificabile nel programma come caratterizzata da un comportamento autonomo o un insieme di comportamenti correlati dovrebbe corrispondere ad una diversa classe.*

3.7.2 Java Generics

Quando dobbiamo scrivere più strutture dati che si differiscono tra loro, possiamo generalizzare usando i *generics*, ovvero definendo le classi con una variabile di tipo che verrà istanziata quando creo la struttura.

Possiamo quindi usare i generics (più di uno) sia nella dichiarazione di classi che di interfacce.

Esempio 3.15.

```
class NewSet<T> implements Set<T> {
  List<T> theRep;
  T lastItemInserted;
}
```

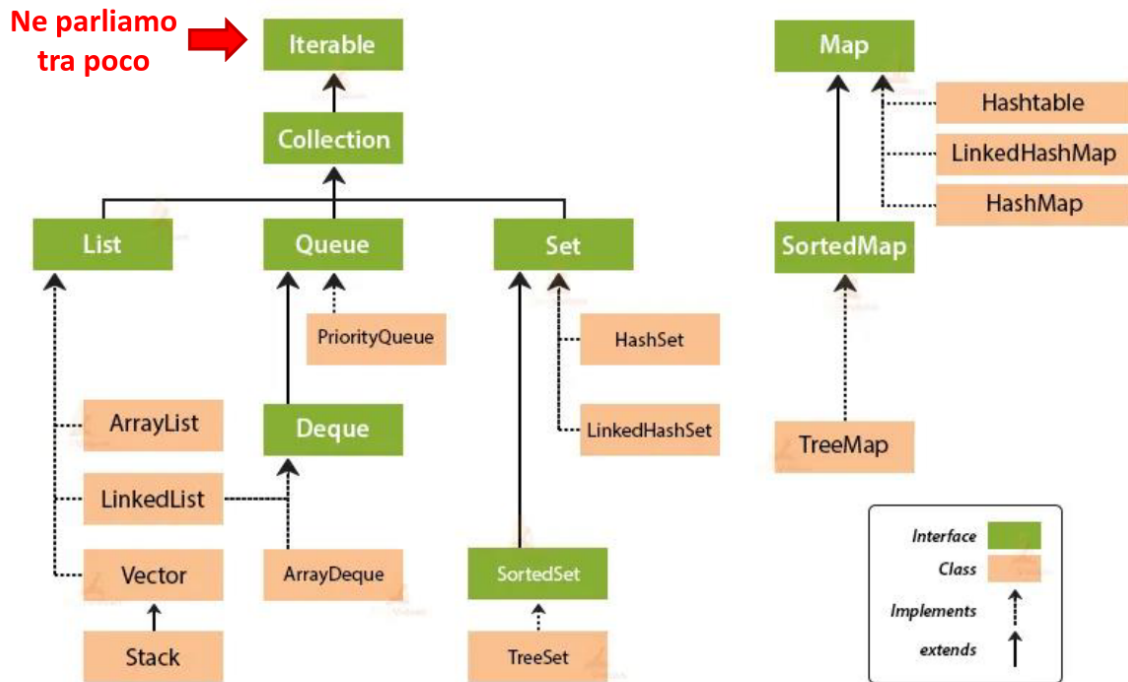
Di seguito alcuni token specifici per i generics di Java:

- **extends** usato dopo un generics ne definisce l'*upper bound*
- **super** usato dopo un generics ne definisce il *lowe bound*
- **?** indica un tipo di cui non si è a conoscenza

Se usiamo come generico un tipo che è **sottotipo** di un altro, le strutture composte da questi tipi **NON** sono l'uno sottotipo dell'altro. Si dice quindi che Java è **invariante**:

$$T2 <: T3 \not\Rightarrow T1 < T2 > <: T1 < T3 >$$

Note 3.7.1 (Varianza per tipi). *Sia $A(T)$ un tipo definito usando il tipo T :*



- A è **covariante** se $T <: S \implies A(T) <: A(S)$
- A è **contravariante** se $T <: S \implies A(S) <: A(T)$
- A è **bivariante** se è sia covariante e contravariante
- A è **invariante** se non è covariante e contravariante

Avendo Java delle strutture modificabili e la possibilità di creare alias con tipi diversi, è necessario che sia invariante. Al contrario OCaml è **covariante** poichè non ha queste caratteristiche.

Note 3.7.2 (Type erasure). Se invece di guardare alle strutture che usano i generics guardiamo gli array di tipi diversi, in quest'ultimo caso abbiamo la covarianza. Questa scelta implementativa c'è per due motivi:

- I generics sono stati implementati per permettere di controllare i tipi mentre gli array no
- I generics sono stati implementati in Java 5 e per garantire la compatibilità del bytecode (per evitare di riscriverlo), nella fase di compilazione viene rimossa l'informazione sui tipi, non permettendo di fare controlli a runtime. Gli array al contrario, esistendo da sempre, contengono già nativamente l'informazione del tipo e possono sollevare eccezioni a runtime.

3.7.3 Java Collection Framework

Contiene classi messe a disposizione da Java per gestire le strutture dati (le collezioni):

Un **iteratore** è un'astrazione che permette di estrarre uno alla volta gli elementi di una collezione senza esporne la rappresentazione. Generalizza la scansione lineare di array e liste a collezioni generiche.

```

public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}

```

Quindi posso creare un iteratore su una certa collezione e poi usare quell'oggetto per scorrerla. Il metodo **remove** esiste perché non si può rimuovere un oggetto dalla collezione mentre si sta visitando. Tramite il **foreach** si può scorrere una collezione: Java sotto il cofano creerà un iteratore.

4 Garbage collection

Il sistema che si occupa di ripulire la memoria quando degli oggetti/variabili non sono più utilizzati. Gestisce quindi l'heap della memoria, che è organizzata come segue:

- *Static area*: ha dimensione fissa e i suoi contenuti sono determinati ed allocati a compilazione
- *Run-time stack*: ha dimensione variabile e gestisce i sottoprogrammi
- **Heap**: può avere dimensione fissa o variabile e gestisce gli oggetti e le strutture dati dinamiche

Evita di saturare la memoria, evita problemi alla deallocazione (memory leak, dandling problem) e risolve problemi di sicurezza (elimina dati potenzialmente sensibili rimasti in memoria).

4.1 Allocazione heap

Può avvenire in due modi:

- Allocazione con blocchi di memoria **fissa** predifiniti organizzati come una lista. Questo porta a frammentazione quando poi avvengono de-allocazioni e nuovi allocazioni (non c'è una sequenza di blocchi contigua abbastanza grande). Esistono due tecniche:
 - *First fit*: si prende il primo blocco grande abbastanza. È più veloce ma spreca più memoria.
 - *Best fit*: si prende il blocco di dimensione più piccola che sia grande abbastanza. Più lento ma più efficiente in termini di memoria.
- Allocazione con blocchi di memoria di dimensione **variabile**: i blocchi vengono creati strada facendo a seconda della dimensione necessaria e uniti quando sono contigui e di nuovo liberi. Si può comunque presentare frammentazione.

4.2 Frammentazione

Ne esistono di due tipi:

- **Interna**: quando c'è memoria in eccesso nei blocchi che vengono allocati
- **Esterna**: quando vengono lasciati blocchi di memoria non allocati tra blocchi allocati che sono troppo piccoli per essere utilizzati

4.3 Algoritmi

Il garbage collector perfetto dovrebbe avere le seguenti caratteristiche:

1. Nessun impatto visibile sulle performance
2. Opera su ogni tipo di programma e struttura dati (anche cicliche)
3. Individua il garbage in modo efficiente e veloce
4. Nessun overhead sulla gestione della memoria complessiva (caching e paginazione)
5. Gestione efficiente dell'heap (nessuna frammentazione)

4.3.1 Reference counting

Si tiene traccia del numero dei riferimenti ai blocchi allocati tramite un **contatore**. Viene inizializzato ad 1 e ad ogni copia del puntatore aumenta di 1 mentre ad ogni deallocazione decrementa. Quando arriva a 0 viene liberata la memoria.

Il problema principale che si pone con questa tecnica è il **memory leak**: ovvero quando alcuni oggetti hanno riferimenti circolari e potrebbero rimanere isolati dal resto dei puntatori ma comunque allocati. Inoltre, essendo necessario spazio per i contatori, si ha un **overhead di memoria**.

4.3.2 Tracing

Sfrutta un **grafo** di raggiungibilità. Ogni tanto interrompono l'esecuzione del programma e verificano quali riferimenti sono ancora attivi e quali no. Si risolve il problema del *memory leak* ma si presenta un **overhead di tempo**. Esistono due principali tecniche:

Definizione 4.1 (mark-sweep). *Ogni cella prevede un **bit di marcatura**. Si interrompe periodicamente il programma e si scorre il grafo: partendo dal root set si marcano le celle live (marking), poi tutte le celle non marcate sono deallocate e vengono resettati i bit (sweep).*

```

// Mark
For each root v:
    DFS(v)

function DFS(x):
    if x is a pointer into heap
        if record x is not marked
            mark x
        for each field fi of record x
            DFS(x.fi)

// Sweep
p <- first address in heap
while p < last address in heap
    if record p is marked
        unmark p
    else let f1 be the first field in p
        p.f1 <- freelist
        freelist <- p
    p <- p + (size of record p)

```

Funziona sulle strutture circolari, non ha overhead di spazio ma necessita della sospensione dell'esecuzione e non interviene sul problema della frammentazione.

Definizione 4.2 (Copy collection - Cheney algorithm). *Suddividiamo l'heap in due parti:*

- **from-space**: viene utilizzato per allocare inizialmente
- **to-space**

Ad ogni chiamata del garbage collector viene fatta una visita al grafo e tutti i riferimenti ancora attivi vengono spostati nell'altra area, che poi diventa quella attiva. Ciò che rimane nella prima viene deallocato.

Il problema principale è l'overhead di memoria in quanto usa solo metà dell'heap. Migliora il problema della frammentazione perché copia in maniera allineata.

4.3.3 Generational GC

Si basa sull'idea "*most cells that die, die young*". Viene quindi fatta una classificazione in base alla lunghezza della vita. L'heap viene diviso in **generazioni**, dove le più "giovani" vengono visitate più frequentemente. Ad ogni iterazione se qualcosa è attivo viene spostato in una generazione successiva che viene visitata meno frequentemente. Su varie generazioni vengono utilizzati algoritmi diversi di quelli elencati precedentemente.

5 Programmazione concorrente

Un **programma concorrente** contiene due o più processi che lavorano assieme per eseguire una determinata applicazione. Ogni sottoprocesso è un programma sequenziale e comunicano tra loro tramite **shared memory** o **message passing**.

Note 5.0.1. *In realtà la programmazione concorrente è solo un'astrazione in quanto non è detto che i processi siano davvero in esecuzione contemporanea.*

Abbiamo due tipi di esecuzione non sequenziale:

- **Preemptive multitasking:** i processi si alternano e sono in **interleaving**.
- **Parallelismo:** ci sono più processori che eseguono più processi separatamente. Ognuno di questi può anche fare multitasking.

Entrambi sono accomunati dall'*esecuzione non sequenziale* e dal fatto che ogni sottoprocesso ha idealmente un proprio *program counter* che avanza autonomamente.

Quando due o più processi provano ad accedere alla stessa cella di memoria, il sistema operativo gestisce la cosa facendoli lavorare uno alla volta, anche nel caso del parallelismo. Per questo alla fine si può considerare l'interleaving come astrazione generale.

Per capire il risultato del sistema di transizioni di due programmi concorrenti è necessario analizzare il codice macchina e non quello ad alto livello, poiché è necessario lavorare su operazioni **atomiche**.

Esempio 5.1 (Analisi ad alto e basso livello). Dato il seguente programma in pseudocodice

integer n ← 0	
p	q
p1: n ← n+1	q1: n ← n+1

avremo due possibili scenari di esecuzione entrambe con lo stesso risultato: 2.

Processo p	Processo q	n
p1: n ← n+1	q1: n ← n+1	0
END	q1: n ← n+1	1
END	END	2

Processo p	Processo q	n
p1: n ← n+1	q1: n ← n+1	0
p1: n ← n+1	END	1
END	END	2

Analizzando invece il codice a basso livello:

integer n ← 0	
p	q
p1: load R1,n	q1: load R1,n
p2: add R1,0x1	q2: add R1,0x1
p3: store R1,n	q3: store R1,n

abbiamo molti più possibili scenari, come ad esempio il seguente in cui il risultato è 1:

Processo p	Processo q	n	p.R1	q.R1
p1: load R1,n	q1: load R1,n	0	?	?
p2: add R1,0x1	q1: load R1,n	0	0	?
p2: add R1,0x1	q2: add R1,0x1	0	0	0
p3: store R1,n	q2: add R1,0x1	0	1	0
p3: store R1,n	q3: store R1,n	0	1	1
END	q3: store R1,n	1	1	1
END	END	1	1	1

Un modo per poter eseguire correttamente l'analisi anche su codice ad alto livello è quello di simulare manualmente il comportamento che avrebbe la macchina a basso livello oppure considerare la lettura come un'istruzione a se che richiede la sua elaborazione.

5.1 Comunicazione tra processi

I processi possono interagire tra loro per scambiarsi i dati o per sincronizzarsi. Può avvenire in due modi.

5.1.1 Shared memory

I threads possono accedere alla stessa area di **memoria** e la usano (in scrittura e lettura) per comunicare o sincronizzarsi.

Esempio 5.2 (Busy waiting). Questo è un modo inefficiente di sincronizzare i thread poiché il 2 perde tempo solamente per testare il valore.

```
// Variabili globali (memoria condivisa)
int x = 0;

// Thread 1
producer() {
    int k = 6;
    x = fattoriale(k);
}

// Thread 2
consumer() {
    while(x==0) sleep(10);
    print(x);
}
```

Un'alternativa più efficiente è tramite servizi messi a disposizione dal **runtime del linguaggio**:

```
// Thread 1
producer() {
    int k = 6;
    x = fattoriale(k);
    wakeup();
}

// Thread 2
consumer() {
    if(x==0) sleep();
    print(x);
}
```

5.1.2 Message passing

I processi sono isolati tra di loro a livello di memoria ma possono inviarsi messaggi e sincronizzarsi tramite servizi di **inter-process communication** messi a disposizione dal sistema operativo.

Esempio 5.3 (IPC).

```
// Processo 1
producer() {
    int k=6;
    int x=fattoriale(k);
    send(x);
}

// Processo 2
consumer() {
    int y = receive();
    print(y);
}
```
