LAMBDA CALCOLO E TIPI

Il **lambda calcolo** e' un formalismo matematico fondamentale per la programmazione funzionale. Descrive la definizione e l'applicazione delle funzioni e forma la base teorica di molti linguaggi funzionali (OCaml, Haskell). E' utile per comprendere funzioni anonime, funzioni di ordine superiore e concetti come la ricorsione e la valutazione delle espressioni.

Confluenza nel lambda calcolo (teorema di Church-Rosser)

Il **teorema di Church-Rosser** garantisce la **confluenza**: se un'espressione puo' essere ridotta in piu' modi, tutti i percorsi di riduzione porteranno (se terminano) alla stessa forma normale. Questo garantisce che il risultato finale non dipende dall'ordine di applicazione delle riduzioni. Esempio di espressione non terminante: **Omega = (x.xx)(x.xx)**.

Passaggio dei parametri: Call by name vs Call by value

- **Call by name:** l'argomento e' valutato solo quando e se e' necessario. Vantaggio: evita valutazioni inutili. Svantaggio: puo' portare a valutazioni ripetute.
- **Call by value:** l'argomento e' valutato subito e una sola volta. Vantaggio: piu' efficiente quando l'argomento e' usato piu' volte. Svantaggio: puo' eseguire valutazioni inutili se il valore non viene mai usato.

Nota: La maggior parte dei linguaggi moderni (es. Java, C) usano il **Call by value**.

Proprieta' di progresso e conservazione

- **Progresso:** un'espressione ben tipata o e' un valore o puo' fare almeno un passo di esecuzione.
- **Conservazione:** se un'espressione ben tipata fa un passo, il risultato ha lo stesso tipo.

Queste proprieta' garantiscono che un programma ben tipato **non si blocchi mai per errori di tipo** a run-time, assicurando la **type safety**.

Type-checking statico e dinamico

- **Type-checking statico:** verifica i tipi a compile-time. Vantaggi: rileva errori prima di eseguire il programma, nessun overhead a run-time. Esempi: Java, OCaml.
- **Type-checking dinamico:** verifica i tipi a run-time. Vantaggi: maggiore flessibilita'. Svantaggi: errori di tipo possono emergere solo durante l'esecuzione e ha overhead. Esempi: Python, JavaScript.

Ruolo di evt e exp

In un interprete, l'**evt (environment)** rappresenta l'insieme delle associazioni tra nomi (variabili) e valori, memorizzate durante l'esecuzione. Serve per sapere quali valori sono legati a quali variabili in ogni punto dell'esecuzione.

L'**exp (expression)** e' l'espressione da valutare, rappresentata tipicamente come un albero di sintassi astratta (AST).

L'interprete utilizza l'**env (evt)** per recuperare i valori durante la valutazione di exp. L'env viene aggiornato dinamicamente durante l'ingresso e l'uscita da blocchi o funzioni, seguendo le regole dello scoping (statico o dinamico).

Esempio pratico: valutando `let x = 5 in x + 1`, l'env memorizza temporaneamente `x -> 5`.

Covarianza in Java (e perche' non e' supportata di default nei generics)

In Java, i **generics** non sono covarianti per default. Ad esempio, anche se Integer <: Number, **List<Integer> non e' sottotipo di List<Number>**. Questo per evitare problemi legati alla mutabilita': se fosse permesso, si potrebbe inserire un Double in una List<Integer> attraverso un riferimento List<Number>.

Java consente la covarianza con le **wildcard**:

```
List<? extends Number> numbers;
```

Subtyping nominale e strutturale

- **Nominale:** la relazione di sottotipo e' basata sul nome delle classi/interfacce (Java, C++). Una classe e' sottotipo solo se lo dichiara esplicitamente.
- **Strutturale:** la compatibilita' e' basata sulla struttura degli oggetti (TypeScript, OCaml). Due tipi sono compatibili se hanno la stessa struttura, anche se non e' dichiarato.

Esempio nominale (Java):

```
interface A { void m(); }
class B implements A { public void m() {} }
```

Esempio strutturale (TypeScript):

```
type A = { m(): void; }
type B = { m(): void; n(): void; }
let a: A = { m() {} };
let b: B = { m() {}, n() {} };
a = b;  // valido perche' B ha almeno i metodi di A
```

GARBAGE COLLECTION

La **Garbage Collection (GC)** e' una tecnica automatica di gestione della memoria. Serve a liberare la memoria occupata da oggetti che non sono piu' raggiungibili dal programma, prevenendo memory leak e migliorando la stabilita' e sicurezza delle applicazioni. La GC opera principalmente sull'**heap** ed e' fondamentale in linguaggi come Java e Python, dove l'allocazione e la deallocazione della memoria non sono gestite direttamente dal programmatore.

Cos'e' la Garbage Collection?

La **garbage collection** e' un processo che identifica gli oggetti non piu' utilizzati e li elimina automaticamente. Questo permette al programmatore di non preoccuparsi di liberare esplicitamente la memoria, riducendo il rischio di errori come dangling pointer o memory leak.

Garbage Collection Mark & Sweep

Un algoritmo classico della GC:

- **Mark:** a partire dal **Root-Set** (registri, variabili globali, stack), marca tutti gli oggetti raggiungibili.
- **Sweep:** scansiona l'heap e dealloca gli oggetti non marcati.

Vantaggi: gestisce anche strutture cicliche.

Svantaggi: puo' fermare l'esecuzione del programma durante la pulizia (stop-the-world).

Root-Set: ruolo e struttura

Il **Root-Set** e' l'insieme di riferimenti da cui parte la fase di marcatura. Comprende:

- Variabili globali e statiche;
- Variabili locali nei record di attivazione (stack);
- Registri CPU.

Tutti gli oggetti raggiungibili direttamente o indirettamente dal Root-Set sono considerati vivi.

Garbage collector Copying collection

Questo metodo divide l'heap in due parti:

- **From-space** e **To-space**. Durante la raccolta, gli oggetti vivi vengono copiati da from-space a to-space.
- Una volta terminato, i ruoli delle due aree si invertono.

Vantaggi: previene la frammentazione.

Svantaggi: raddoppia il consumo di memoria perche' necessita di due aree uguali.

Reference counting (e dove fallisce)

Ogni oggetto ha un **contatore di riferimenti**. Quando il contatore scende a zero, l'oggetto puo' essere eliminato immediatamente.

Problema principale: non gestisce i cicli di riferimento. Esempio:

```
class Node { Node next;}

Node a = new Node();
Node b = new Node();

a.next = b;
b.next = a;  // ciclo che non verra' mai eliminato da reference counting
```

Garbage collector generazionale

Sfrutta l'osservazione che la maggior parte degli oggetti ha vita breve. L'heap e' suddiviso in generazioni:

- **Young generation:** oggetti appena creati;
- **Old generation:** oggetti longevi.

La GC agisce piu' frequentemente sulla young generation, migliorando le prestazioni globali.

Tipologie di garbage collector, pregi e difetti

- **Reference Counting:** veloce e semplice, ma non gestisce i cicli.
- **Mark & Sweep:** gestisce tutto, anche i cicli, ma ha pause lunghe (stop-the-world).
- **Copying:** evita frammentazione, necessita di spazio doppio.
- **Generazionale:** ottimizza le prestazioni agendo piu' spesso su oggetti giovani.

SINCRONIZZAZIONE, LOCKING E THREAD

La **sincronizzazione** e' fondamentale nella programmazione concorrente per coordinare l'accesso a risorse condivise tra piu' thread. Senza una corretta sincronizzazione, si possono verificare problemi di **race condition**, inconsistenza dei dati e deadlock. Java fornisce strumenti come `synchronized` e `ReentrantLock` per facilitare la sincronizzazione.

Locking multithread e implementazione in Java

Java offre diverse modalita' di sincronizzazione:

- **synchronized:** parola chiave che permette di bloccare metodi o blocchi di codice. Ogni oggetto ha un **monitor** (lock intrinseco) utilizzato per la sincronizzazione.
- **ReentrantLock:** una classe piu' flessibile che permette di avere timeout, tentativi di lock e sblocco piu' controllato.

Coarse-grained vs Fine-grained locking

- **Coarse-grained locking:** usa un singolo lock per proteggere grandi sezioni di codice o intere strutture dati. Vantaggi: piu' semplice da implementare. Svantaggi: riduce la concorrenza, creando colli di bottiglia.
- **Fine-grained locking:** utilizza piu' lock per proteggere parti piu' piccole delle strutture dati. Vantaggi: maggiore parallelismo. Svantaggi: piu' complesso e aumenta il rischio di deadlock.

Deadlock: definizione e condizioni

Un **deadlock** si verifica quando due o piu' thread rimangono bloccati indefinitamente, ognuno in attesa di una risorsa detenuta da un altro. Le **quattro condizioni** necessarie per un deadlock sono:

1. **Mutua esclusione:** una risorsa puo' essere detenuta da un solo thread alla volta.
2. **Attesa circolare:** esiste un ciclo di thread ognuno in attesa di una risorsa posseduta dal prossimo.
3. **Nessuna prelazione:** le risorse non possono essere sottratte forzatamente ai thread.
4. **Attesa bloccante:** i thread attendono in modo bloccante le risorse.

Come risolvere il deadlock

Strategie per prevenire o gestire i deadlock:

- **Timeout:** impostare un limite massimo di tempo per tentare di acquisire un lock.
- **Ordinamento globale dei lock:** acquisire sempre i lock in un ordine predefinito.
- **Deadlock detection:** rilevare i deadlock e interrompere uno dei thread coinvolti per sbloccare la situazione.
- **Evita attesa circolare:** progettare l'applicazione in modo da non creare cicli di attesa.

Modifica lista durante iterazione

Quando si utilizza un iteratore per attraversare una lista in Java, modificare la lista direttamente (dall'esterno dell'iteratore) durante l'iterazione causa una **ConcurrentModificationException**.

Esempio problematico:

```
List<String> list = new ArrayList<>();

list.add("A");
list.add("B");

for (String s : list) {
        if (s.equals("A")) {
                list.remove(s);  // Eccezione!
        }
}
```

Soluzioni:

- Usare l'**iteratore** stesso per rimuovere elementi in modo sicuro:

```
Iterator<String> it = list.iterator();
while (it.hasNext()) {
        String s = it.next();
        if (s.equals("A")) {
                it.remove();  // OK
        }
}
```

- Usare strutture sicure come **CopyOnWriteArrayList**, che permette modifiche durante l'iterazione senza eccezioni.

2-phase locking

Il **2-phase locking (2PL)** e' un protocollo utilizzato nei database per garantire la **serializzabilita'** delle transazioni. Funziona cosi':

- **Fase crescente:** il processo puo' solo acquisire lock (nessun rilascio).
- **Fase decrescente:** il processo puo' solo rilasciare lock (nessuna nuova acquisizione).

Questo metodo assicura che nessun altro thread possa interferire durante la transazione, mantenendo l'integrita' dei dati.

Importanza di lock e unlock. Differenza coarse vs fine-grained

I **lock** sono cruciali per impedire l'accesso simultaneo non controllato alle risorse condivise. Tuttavia, bloccare troppo codice (coarse-grained) riduce la concorrenza, mentre bloccare troppo poco (fine-grained) aumenta il rischio di deadlock. E' importante trovare un equilibrio tra sicurezza e performance.

Differenza tra coarse e fine-grained locking (approfondimento)

Coarse-grained locking: adatto a strutture dati semplici o quando il rischio di contention e' basso. Si blocca l'intera struttura anche se solo una piccola parte e' modificata.

Fine-grained locking: utile per strutture complesse dove molte operazioni possono essere eseguite in parallelo. Ad esempio, in una hashmap suddivisa in bucket, ogni bucket puo' avere il proprio lock.

OOP: EREDITARIETA', POLIMORFISMO, GENERICS

La **programmazione orientata agli oggetti (OOP)** e' un paradigma che organizza il software in oggetti che incapsulano dati e comportamento. Concetti chiave sono **ereditarieta'**, **polimorfismo**, **incapsulamento** e **astrazione**. Java, C++ e Python sono tra i linguaggi piu' noti che utilizzano OOP.

Dynamic dispatch

Il **dynamic dispatch** permette di selezionare a run-time la versione corretta di un metodo in base al tipo effettivo dell'oggetto. In Java, questo avviene tramite la **dispatch table (vtable)**, che mappa i metodi alle loro implementazioni. Esempio:

```
class A { void m() { System.out.println("A"); } }
class B extends A { void m() { System.out.println("B"); } }

A obj = new B();
obj.m();  // stampa "B"
```

Sostituzione di Liskov (LSP)

Il **principio di sostituzione di Liskov (LSP)** afferma che un oggetto di una sottoclasse deve poter essere utilizzato ovunque sia previsto un oggetto della superclasse **senza alterare la correttezza del programma**. Richiede:

- Firma dei metodi compatibile;
- Stesse (o piu' deboli) precondizioni;
- Stesse (o piu' forti) postcondizioni.

Tipo apparente vs Tipo effettivo

- **Tipo apparente (statico):** il tipo dichiarato della variabile.
- **Tipo effettivo (dinamico):** il tipo reale dell'oggetto referenziato a run-time.

Il compilatore utilizza il tipo apparente per verificare la correttezza, mentre il dynamic dispatch usa il tipo effettivo per invocare il metodo corretto.

Casting e rischi

Il **downcast** consente di convertire un riferimento di tipo superclasse a un tipo sottoclasse. Tuttavia, se non corretto, puo' causare una **ClassCastException**. E' buona pratica verificare con `instanceof` prima di eseguire un cast.

Mixin: cosa sono e come funzionano

Un **mixin** e' un modulo riutilizzabile che fornisce funzionalita' a piu' classi senza richiedere ereditarieta' multipla. In **Python**, i mixin sono supportati direttamente; in **Java**, possono essere simulati tramite interfacce con metodi `default`.

Ereditarieta' multipla (problemi e soluzioni)

L'ereditarieta' multipla consente a una classe di estendere piu' classi. Il **problema del diamante (diamond problem)** si verifica quando una classe eredita da due classi che condividono una stessa superclasse.

- **Java:** evita l'ereditarieta' multipla di classi ma supporta quella di interfacce.
- **C++:** supporta l'ereditarieta' multipla e risolve i conflitti usando la keyword `virtual`.
- **Python:** gestisce l'ereditarieta' multipla con il metodo di risoluzione MRO (**Method Resolution Order**) basato sulla linearizzazione **C3**.

Uso del costrutto virtual in C++

In **C++**, la keyword `virtual` permette di **sovrascrivere** metodi e di abilitare il **dynamic dispatch**. Senza `virtual`, la risoluzione del metodo avviene staticamente (binding statico). Esempio:

```
class A { public: virtual void m() { cout << "A"; } };
class B : public A { public: void m() override { cout << "B"; } };

A* obj = new B();
obj->m();  // stampa "B" grazie al dynamic dispatch
```

Generics: caratteristiche e limiti

I **generics** in Java consentono di creare classi e metodi parametrizzati sui tipi, migliorando la riusabilita' e la sicurezza dei tipi a compile-time. Tuttavia, Java implementa i generics tramite **type erasure**, che elimina le informazioni sui tipi generici a run-time. Di conseguenza:

- Non e' possibile creare array di tipi generici.
- Non e' possibile usare instanceof con parametri di tipo.

Problema della covarianza nei generics

Java **non consente** la covarianza standard nei generics mutabili. Esempio:

```
List<Integer> intList = new ArrayList<>();
List<Number> numList = intList;  // Errore di compilazione
```

Per gestire la covarianza in modo sicuro, usa le **wildcard**:

```
List<? extends Number> numList = intList;
```

JVM E RUNTIME

La **Java Virtual Machine (JVM)** e' l'ambiente di esecuzione che interpreta ed esegue il bytecode Java. La JVM e' responsabile della gestione della memoria, del caricamento delle classi, della sicurezza e della gestione delle chiamate ai metodi. Garantisce la portabilita' del codice Java, permettendo l'esecuzione su qualsiasi macchina che abbia una JVM compatibile.

Struttura della JVM

La memoria della JVM e' suddivisa in diverse aree principali:

- **Heap:** area principale in cui vengono allocati tutti gli oggetti e i dati di istanza.
- **Stack:** contiene i frame di attivazione (record di attivazione) di ciascun thread. Ogni frame memorizza variabili locali, l'operando stack e altre informazioni di controllo.
- **Area delle classi (Metaspace in Java 8+):** memorizza le informazioni delle classi caricate (definizioni di classi, metodi, campi statici).

Questa suddivisione consente una gestione efficiente e separata delle risorse.

Dispatch vector e gestione interfacce (itable)

In Java, le chiamate ai metodi sono gestite tramite:

- **Dispatch table (vtable):** utilizzata per i metodi delle classi. Ogni classe ha una tabella che punta alle implementazioni dei metodi.
- **Interface table (itable):** utilizzata quando un metodo e' chiamato tramite un riferimento a un'interfaccia. La itable permette di mappare i metodi dichiarati nell'interfaccia alla loro implementazione concreta nella classe.

Esempio: se una classe `C` implementa l'interfaccia `I`, la sua itable conterra' i riferimenti ai metodi che realizzano quelli dell'interfaccia.

Caching nelle itable

Alla **prima chiamata** di un metodo tramite un'interfaccia, la JVM effettua la ricerca del metodo corretto nella itable in modo lineare. Per ottimizzare le prestazioni, la JVM **memorizza** (cache) il risultato della ricerca. Le successive chiamate sono quindi molto piu' veloci grazie a questo caching.

Come funziona il run-time support

Il **runtime support** comprende:

- **Gestione dello stack e dell'heap;**
- **Garbage collection;**
- **Gestione delle eccezioni;**
- **Monitoraggio e gestione dei thread.**

Il run-time support e' parte integrante della JVM e fornisce tutto cio' che e' necessario per l'esecuzione corretta e sicura del programma Java.

Importanza della JVM per la portabilita'

La JVM permette di eseguire lo stesso bytecode Java su architetture diverse senza modifiche al codice sorgente. Questo realizza il famoso slogan di Java: **"Write Once, Run Anywhere" (WORA)**.

Just-In-Time (JIT) Compiler

Il **JIT Compiler** migliora le prestazioni della JVM compilando parti frequentemente eseguite del bytecode in codice macchina nativo, riducendo l'overhead dell'interpretazione. Questo consente di ottenere prestazioni simili a quelle dei linguaggi compilati.

Struttura compilatore (Front-end / Back-end)

Il compilatore e' suddiviso in due sezioni principali:

- **Front-end:** si occupa dell'analisi del programma. Comprende:
    - Scanner (lexer): trasforma il codice sorgente in token.
    - Parser: costruisce l'AST (Abstract Syntax Tree).
    - Type-checking: verifica la correttezza dei tipi e altre analisi statiche. Obiettivo: assicurarsi che il programma sia corretto prima della traduzione.
- **Back-end:** si occupa della generazione del codice eseguibile e ottimizzazioni.
    - Ottimizzazione del codice: migliora l'efficienza.
    - Generazione del codice macchina o bytecode.

Obiettivo: creare un programma eseguibile efficiente e compatibile con l'architettura di destinazione.