Macchine astratte, interpreti e compilatori

Ciclo Fetch-Execute

Fetch: l'istruzione viene presa dalla memoria e trasferita nella CPU.
Decode: l'istruzione viene interpretata.
Data Fetch: sono prelevati dalla memoria i dati sui quali eseguire l'operazione prevista dall'istruzione.
Execute: viene portata a termine l'esecuzione
Store: viene memorizzato il risultato

Interprete

Programma che prende in ingresso il programma da eseguire lo esegue ispezionandone la struttura per vedere cosa fare.

Compilatore

Programma che traduce una serie di istruzioni scritte in un determinato linguaggio, in istruzioni di un altro linguaggio.
Una volta ricevuto il programma sorgente, il compilatore deve superare due fasi: Front end (analisi) e Back end (sintesi).

- Front end → lettura del programma e generazione della struttura sintattica e semantica
- Back end → generazione del codice nel linguaggio macchina

Scanner

Trasforma il programma sorgente nel lessico (token)

Parser

Legge i token e genera il codice intermedio.

Albero di sintassi astratta (AST)

Strutture sintattiche essenziali che mostrano la struttura semantica significativa dei programmi.
Si parla di ambiguità quando per lo stesso programma esistono almeno due AST diversi.

Parser a discesa ricorsiva

Parser di natura top-down, che parte dal simbolo iniziale della grammatica e applicando le regole cerca di riscrivere il simbolo iniziale fino ad ottenere il programma in ingresso

Semantica statica

É il sistema dei tipi di un linguaggio di programmazione, ovvero l'insieme delle regole che consentono di dare un tipo ad espressioni, comandi ed altri costrutti.

Lambda calcolo

Currying

Applicazione parziale di una funzione, ovvero l'applicazione di una funzione a parte dei suoi argomenti

λ- calcolo

Sistema formale sviluppato per analizzare formalmente le funzioni ed il loro calcolo.

$e ::= x \longrightarrow$ variabile
$\lambda x.e \longrightarrow$ astrazione funzionale (fun dec)
$e\ e \longrightarrow$ applicazione (fun call)

Alpha equivalenza

Due espressioni sono α- equivalenti se sono uguali eccetto per le variabili

Valutazione espressione

Standard: presa $\lambda x.e_1$ con e_2 parametro attuale si esegue il corpo di e_1 e ogni occorrenza del parametro formale x viene sostituita con e_2.
Non Call by value: presa $\lambda x.e_1$ con e_2 parametro attuale, non viene valutato e_2 e ogni occorrenza del parametro formale x viene sostituito con e_2.
Lazy: presa $\lambda x.e_1$ con e_2 parametro attuale, non viene valutato e_2 e consiste nel ritardare i parametri finché il valore del parametro non è richiesto.

Valutazione applicazione

1) Valutazione espressione
2) Passaggio dei parametri:

- Valutazione eagere (Call by value): tutte le occorrenze della variabile legata alla funzione sono sostituite dal valore dell'espressione.
- Valutazione lazy (Call by name): tutte le occorrenze della variabile legata alla funzione sono sostituite dall'espressione non valutata.

Beta riduzione

Riduzione alla forma normale beta di una λ-espressione.

Beta equivalenza

Due espressioni sono β-equivalenti se:

- Sono identiche
- $e_1 \Rightarrow e_2$ o $e_2 \Rightarrow e_1$
- $e_1 \Rightarrow e$, $e_2 \Rightarrow e$

Controllo dei tipi

Sistema dei tipi

Metodo sintattico, effettivo per dimostrare l'assenza di comportamenti anomali del programma.

Type safety

É la correttezza dell'uso dei tipi, senza di esso sarebbe possibile scrivere programmi pieni di bug. Ne esistono due tipi

- Statico (compilazione)
- Dinamico (esecuzione)

Ambiente dei tipi

É una funzione che associa nomi ai tipi.

$$\Gamma = x_1 : \tau_1, x_2 : \tau_2, \dots, x_K : \tau_K$$

Progresso

Se $\emptyset \vdash e : \tau$ allora e é un valore oppure $e \to e'$

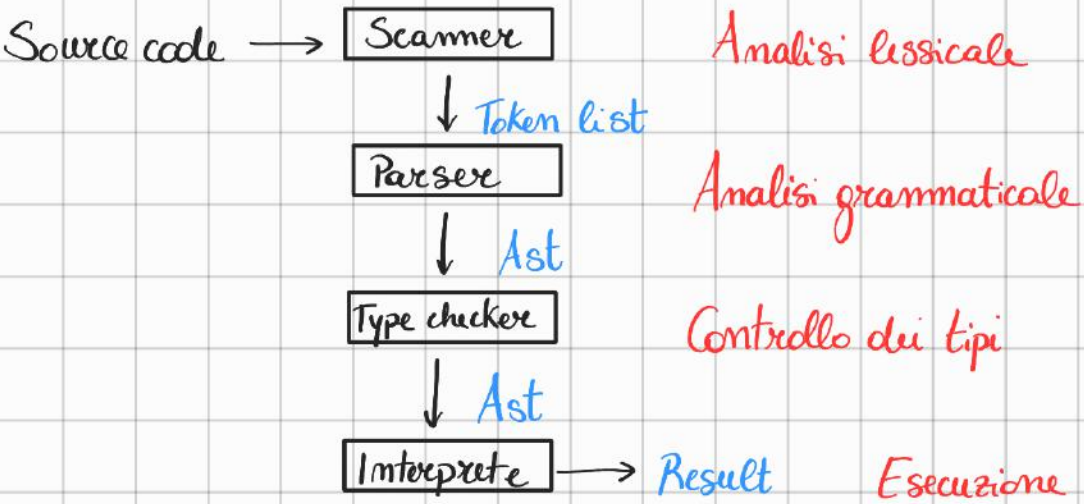
Conservazione

Se $\Gamma \vdash e : \tau$ e $e \to e'$ allora $e' : \tau$

Substitution lemma

$\Gamma, x : \tau_1 \vdash e : \tau \longrightarrow \Gamma \vdash e_1 : \tau_1$ allora $\Gamma \vdash e\{x ::= e_1\} : \tau$

Interprete



Structural Operational Semantics (SOS)

- Semantics: è il significato del linguaggio.
- Operational: descrive il comportamento dei programmi tramite una relazione di transizione.
- Structural: segue delle regole di inferenza basate sulla sintattica.

Viene divisa in due parti:

- Small - Step: ogni passo della relazione di transizione si esegue una singola operazione. (Sequenza di passi)
- Big - Step: la relazione di transizione descrive in un solo passo l'intera computazione. (Singolo passo)

Interprete (eval Big-Step)

```
let rec eval e =
    match e with
    | Val n -> Val n
    | Op(op, e1, e2) -> match (eval e1, eval e2) with
                        | (Val n1, Val n2) -> (match op with
                                               | Add -> Val(n1+n2)
                                               | Sub -> Val(n1-n2)
                                               | Mul -> Val(n1*n2)
                                               | Div -> Val(n1/n2)
                                               )
    | _ -> failwith "Errore impossibile";;
```

Interprete (eval Small-Step)

```
let rec eval e =
    match e with
    | Val n -> Val n
    | Op(op, e1, e2) -> (match (e1, e2) with
                         | (Val n1, Val n2) -> Val (match op with
                                                    | Add -> n1+n2
                                                    | Sub -> n1-n2
                                                    | Mul -> n1*n2
                                                    | Div -> n1/n2)
// e1 può fare un passo     | (Op(_,_,_), _) -> Op(op, (eval e1), e2)
// e2 può fare un passo     | (Val _, Op(_,_,_)) -> Op(op, e1, (eval e2))
                         );;
```

Minicaml

Entità denotabili

Elementi di un linguaggio di programmazione a cui posso assegnare un nome.

- Entità i cui nomi sono definiti dal linguaggio di programmazione (tipi primitivi, operazioni primitive, ecc...)
- Entità i cui nomi sono definiti dall'utente (variabili, parametri, procedure, tipi, costanti simboliche, classi, oggetti, ecc...)

Binding e scope

Un binding è un'associazione tra un nome e un'entità del linguaggio (funzione, struttura dati, oggetto, ecc...)

Lo scope di un binding definisce quella parte di programma nella quale il binding è attivo.

Il termine static (dynamic) binding è utilizzato per fare riferimento ad un'associazione attivata prima di mandare (dopo avere mandato) il programma in esecuzione.
I linguaggi interpretati devono risolvere il binding dinamicamente.

Ambiente (Ide → Value + Unbound)

→ rende la funzione totale

Insieme delle associazioni nome-entità esistenti a run time in uno specifico punto del programma e in uno specifico momento dell'esecuzione.

- Ambiente locale: insieme delle associazioni dichiarate localmente al blocco, comprese le eventuali associazioni relative ai parametri.
- Ambiente non locale: associazioni dei nomi visibili all'interno del blocco ma non dichiarati nel blocco stesso.
- Ambiente globale: associazioni per i nomi usabili da tutte le componenti che costituiscono il programma.

Operazioni su ambienti

Naming: creazione di associazione nome-oggetto
Referencing: riferimento ad un oggetto mediante il suo nome
Disattivazione di associazione nome-oggetto
Riattivazione di associazione nome-oggetto
Unnaming: distruzione di associazione nome-oggetto

Dati

Un tipo di dato è una collezione di valori rappresentati da opportune strutture dati e un insieme di operazioni per manipolarli.
Ne esistono due classi:

- di sistema: definiscono lo stato e le strutture dati utilizzate nella simulazione/interpretazione del comportamento delle primitive.
- di programma: domini semantici corrispondenti ai tipi primitivi del linguaggio e ai tipi che l'utente può definire.

Denotabili: se possono essere associati a un nome
Esprimibili: se possono essere il risultato della valutazione di un'espressione complessa.
Memorizzabili: se possono essere memorizzati in una variabile

Type safety

La mancanza del type safety é la fonte principale di errori di programmazione e permette agli attaccanti di sfruttare bug per alterare maliziosamente il comportamento del programma.

Polimorfismo

Una funzione é polimorfa se ha la caratteristica di essere applicabile ad argomenti di tipo diverso.

Subsumption

Grazie alla relazione di sottotipo $S <: T$ (S sottotipo di T), la regola di subsumption afferma che se $S <: T$, allora ogni valore di tipo S puó essere considerato di tipo T.
Top é supertipo di ogni tipo $\longrightarrow$ $S <: Top$.

Tipo di dato astratto (ADT)

Consiste in un insieme di dati e una collezione di operazioni per operare sui dati di quel tipo.

- Estendibili: esistono meccanismi per costruire nuovi tipi di dati astratti
- Astratti: la rappresentazione é nascosta agli utenti
- Incapsulati: si opera su di essi solo attraverso le operazioni fornite dai ADT.

La specifica di un ADT descrive il tipo di dati e le operazioni senza fornire dettagli di implementazione (semantica).

L'implementazione di un ADT definisce come viene implementato l'ADT ma non è accessibile all'utente tramite forme di controllo dell'accesso.

Paradigma Object - Oriented

Gli oggetti sono caratterizzati da uno stato e da un insieme di funzionalità.
Lo stato di un oggetto è rappresentato da un gruppo di attributi / proprietà / variabili.
Le funzionalità di un oggetto sono rappresentate da un gruppo di metodi / funzioni che l'oggetto mette a disposizione degli altri oggetti.

Incapsulamento

Lo stato di un oggetto non dovrebbe essere accessibile agli altri oggetti (Information hiding).

Object - based

Consente di lavorare con oggetti in modo flessibile e non è necessario scrivere il codice della classe prima di creare un oggetto.
È però difficile predire quale sarà il tipo di un oggetto e ostacola i controlli di tipo statici.
Usano una prototype - based inheritance quindi per ogni oggetto mantengono una lista di prototipi che sono tutti gli oggetti da cui esso eredita funzionalità.

Class-based

Richiede maggiore disciplina in quanto bisogna implementare le classi prima di creare un oggetto ma consente di fare controlli di tipo statici sugli oggetti.
Permettono di definire una classe come estensione di un'altra. La nuova classe eredita tutti i membri della precedente con la possibilità di aggiungerne altri (overriding).

Structural Subtyping

Nei linguaggi object-based un oggetto B è sottotipo di un oggetto A se contiene almeno tutti i suoi membri pubblici.
È più flessibile, non è necessario dire esplicitamente chi estende chi e favorisce il polimorfismo.

Nominal Subtyping

Nei linguaggi class-based il tipo di un oggetto corrisponde alla classe da cui è stato istanziato. Un tipo-classe B è sottotipo di un tipo-classe A se la classe B è stata definita come estensione della classe A.
È più rigoroso, mette in relazione di sottotipo solo le classi che il programmatore ha esplicitamente dichiarato con "extends" ed è più semplice da verificare per l'interprete.

Type coercion (conversione di tipo)

e :> t forza il typechecker a trattare l'espressione e come se fosse di tipo t, t deve essere generale (un supertipo).

Teorema (Subtyping)

Per ogni coppia di classi S e T t.c. $S <: T$, ogni membro pubblico di T é anche membro pubblico di S.

Structural subtyping $<:_s$ é una relazione più debole del Nominal subtyping $<:_n$.

$$S <:_n T \Rightarrow S <:_s T \quad \text{ma} \quad S <:_s T \not\Rightarrow S <:_n T$$

Overloading

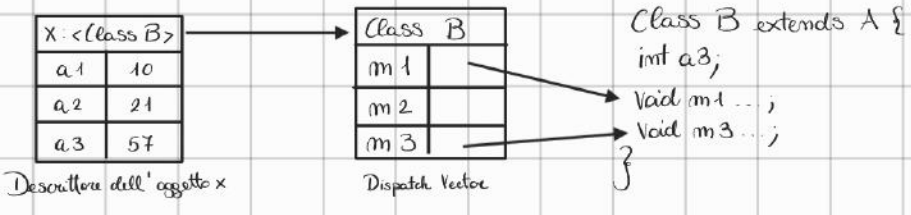
Definire più metodi con lo stesso nome, ma con diverse firme (informazioni di tipo e parametri).

Dynamic Dispatch

Ricerca di un metodo partendo dalla classe corrente, che corrisponde al tipo effettivo dell'oggetto, e risalendo la gerarchia. Può essere inefficiente in quanto si potrebbero incontrare più metodi con la stessa firma.
JVM si basa su Dispatch vector e sharing strutturale.

Dispatch Vector

Tabelle dei metodi con puntatori al codice dei metodi



Sharing strutturale

La tabella della sottoclasse riprende la struttura della tabella della superclasse aggiungendo righe per i nuovi metodi.

Principio di sostituzione di Liskov (LSP)

Un oggetto di un sottotipo può sostituire un oggetto del supertipo senza influire sul comportamento dei programmi che usano il supertipo.
La sottoclasse deve soddisfare le specifiche della superclasse.
Problemi: i contesti (programmi) possibili sono infiniti quindi verificare che LSP vale tra due classi è un problema indecidibile.
Il principio di Liskov si traduce in un insieme di regole:

- Regola della segnatura: gli oggetti del sotto-tipo devono avere tutti i metodi del super-tipo. Le segnature (firme) dei metodi del sotto-tipo devono essere compatibili con le segnature dei corrispondenti metodi del super-tipo.
  In caso di overriding, il metodo della sottoclasse deve avere la stessa firma del metodo della superclasse, può sollevare meno eccezioni e avere un return type più specifico.
- Regola dei metodi: le chiamate dei metodi del sotto-tipo devono comportarsi come le chiamate dei corrispondenti metodi del super-tipo. Per avere compatibilità della specifica il metodo del sotto-tipo e del super-tipo devono soddisfare:
  - regola della pre-condizione: $pre_{super} \Rightarrow pre_{sub}$
  - regola della post-condizione: $(pre_{super} \;\&\&\; post_{sub}) \Rightarrow post_{super}$
- Regola della proprietà: il sottotipo deve preservare tutte le proprietà che possono essere provate sugli oggetti del supertipo.

Varianza per tipi

Supponiamo che A(T) sia un opportuno tipo definito usando T:

A é covariante se $T <: S \Rightarrow A(T) <: A(S)$
A é controvariante se $T <: S \Rightarrow A(S) <: A(T)$
A é bivariante se é entrambe.
A é invariante se non é nessuno.

Diamond problem

Ereditare da due superclassi che a loro volta possono avere una superclasse in comune puó portare a variabili d'istanza e metodi duplicati.

Ereditarietá multipla

C++

Supportata senza restrizioni ma il programmatore deve essere consapevole di quello che fa. L'ereditarietá con replicazione (senza virtual) consente un accesso ai metodi in tempo costante ma richiede di fare attenzione quando le gerarchie sono complicate. L'ereditarietá con condivisione risolve il diamond problem ma ha un overhead.

Java

Non supportata ma usa ereditarietá singola + implementazione multipla di interfacce, ció é trasparente al programmatore ma consente un uso limitato dei meccanismi di ereditarietá. L'accesso ai metodi tramite

itable è inefficiente se la classe implementa varie interfacce.

Python

Non supportata ma usa un metodo di linearizzazione della gerarchia di classi per risolvere il problema del dispatching dei metodi con ereditarietà multipla e senza ricompilazione. Questo metodo prende il nome di Method resolution order (MRO) e si base sull'algoritmo C3 che garantisce determinismo, conservazione dell'ordinamento e monotonia. Esistono gerarchie non linearizzabili e ciò va riconosciuto dal programmatore.

Mixin

Anziché ereditare da altre classi, una classe potrebbe essere il risultato della composizione di altre classi. Si può estendere una sola classe e poi comporla con quanti mixin si voglia aggiornando la definizione della classe.

Garbage Collection

Gestione della memoria

- Static area: dimensione fissa, contenuti determinati e allocati a tempo di compilazione
- Run time stack: dimensione variabile, gestione sottoprogrammi
- Heap: dimensione fisse o variabile, supporto alla allocazione di oggetti e strutture dati dinamiche.

Heap

Ad ogni richiesta di allocazione cerca un blocco di dimensione opportuna:

- first fit: primo blocco grande abbastanza
- best fit: quello di dimensione più piccola, grande abbastanza

Se il blocco scelto è molto più grande di quello che serve, viene diviso in due e la parte inutilizzata è aggiunta alla LL (lista libera). Nella LL vanno inseriti tutti i blocchi da de-allocare, individuati grazie alla de-allocazione esplicita (free) o senza, infatti una porzione di memoria è recuperabile se non è più raggiungibile.

Frammentazione interna

Lo spazio richiesto è X, viene allocato un blocco di dimensione Y > X, lo spazio Y - X è sprecato.

Frammentazione esterna

Lo spazio necessario ci sarebbe ma è inusabile perché suddiviso in "pezzi" troppo piccoli.

Dangling reference

Quando una cella o porzione di memoria è de-allocata ma non tutti i puntatori ad esse vengono rimossi, questi prendono il nome di dangling pointers.

Garbage Collector

É un processo di gestione della memoria dinamica che identifica e "marchia" le celle garbage e le rende riutilizzabili per esempio inserendole nella LL.
Si basa su:

- Reference counting
- Tracing (mark & sweep e copy collection)
- Generational GC

Root set

Insieme dei dati attivi (variabili statiche + variabili allocate sul run-time stack)

Reachable active data

Insieme dei dati raggiungibili anche indirettamente a partire dal root set seguendo i puntatori.
Una cella si dice live se appartiene ai reachable active data, altrimenti é detta garbage.

Reference counting

Aggiunge un contatore di riferimenti a ogni cella (numero di cammini di accesso attivi verso la cella). Facile da implementare e coesiste con la gestione della memoria esplicita del programma (malloc e free) e c'é un riuso delle celle libere immediato. Tra le limitazioni: overhead sparso tempo, mancata esecuzione di un operazione sul valore di RC può

generare garbage e non permettere di gestire strutture dati con cicli (memory leak).

Mark - Sweep

Si parte dal root set e si marcano tutte le celle live (mark), tutte le celle non marcate sono garbage e sono restituite alla lista libera, successivamente viene effettuato un reset del bit di marcatura sulle celle live (sweep).
Opera correttamente su strutture circolari senza overhead di spazio ma si ha sospensione dell'esecuzione e non interviene sulla frammentazione dello heap.

Copy collection

L'algoritmo di Cheney opera suddividendo l'heap in due parti ("from - space" e "to - space"). Solitamente una delle due parti è attiva e quando viene attivato il garbage, le celle live vengono copiate nella parte non attiva. Una volta finita l'operazione di copia, i ruoli tra le due parti vengono invertiti e le celle nella parte non attiva vengono restituite alla lista libera in un unico blocco evitando problemi di frammentazione.

Generational Garbage collector

Dato che la maggior parte delle celle che "muoiono" sono giovani, si divide lo heap in un insieme di generazioni. Si copiano i blocchi vivi in Old e si ripulisce Young continuando ad allocare nuovi blocchi in Young.

Programmazione concorrente

Interleaving

La concorrenza è un'astrazione che consente di fare molte analisi dei programmi che vorremo anche in situazioni di parallelismo. Possibilità di mantenere attivi più programmi contemporaneamente.

Shared memory

I processi (o thread) possono accedere alle stesse aree di memoria, si sincronizzano e comunicano tra loro scrivendo e leggendo variabili condivise.

Message passing

I processi accedono ad aree diverse della memoria, ma possono inviarsi messaggi e sincronizzarsi usando i servizi di inter-process comunication (ICP) messi a disposizione dal sistema operativo.

Mutex

Entità astratta associata ad ogni area di memoria da condividere e da accedere in mutua esclusione.
Prima di accedere all'area di memoria, il thread T1 deve acquisire il mutex m eseguendo lock m, se nessun altro thread T2 sta accedendo a quell'area di memoria, T1 procede. In caso contrario, T1 si blocca e attende che T2 finisca eseguendo unlock m.

Coarse - grained

Associa un unico mutex per tutte le locazioni di memoria quindi richiede meno lavoro al singolo thread ma riduce la concorrenza (dato che lock in blocca tutti gli altri thread)

Fine - grained

Associa un mutex diverso per ogni locazione di memoria, ma possono provocare i deadlock.

Deadlock

É dato da situazioni di attesa circolare, quando T1 aspetta T2 intanto che T2 aspetta T1.

Deadlock prevention

Si stabiliscono regole controllabili staticamente sull'uso dei lock che garantiscono che i deadlock non si verifichino.

Deadlock avoidance

L'esecuzione del programma é monitorata dall' RTS che si accorge quando il programma sta andando in deadlock e interviene cambiando l'ordine di esecuzione dei thread.

Deadlock recovery

Il programma é libero di andare in deadlock ma se accade l'RTS se ne accorge e ripristina a uno stato senza deadlock.

Synchronized

É possibile eseguire un metodo in mutua esclusione acquisendo il lock sull'oggetto tramite il modificatore synchronized. Fa acquisire il lock per tutta la durata del metodo, quindi applica una strategia coarse-grained.
Le strutture dati synchronized sono thread-safe, avero possono essere condivise e usate concorrentemente tra thread diversi senza rischi di interferenze. In un contesto non concorrente meglio non avere strutture dati con synchronized dato che sarebbero meno efficienti perdendo tempo con i lock.