Capitolo 1 — Macchine astratte

Calcolatore:

Macchina fisica che permette di eseguire degli algoritmi, opportunamente formalizzati per poter essere comprensibili all'esecutore.

Programma di *L* (o programma scritto in *L*):

Insieme finito di istruzioni del linguaggio *L*.

Macchina astratta:

Supponiamo che sia dato un linguaggio di programmazione *L*. Una macchina astratta per *L* (M_L) è un qualsiasi insieme di algoritmi e di strutture dati che permettano di memorizzare ed eseguire programmi scritti in *L*.

Memoria di una macchina astratta:

Componente della macchina astratta che serve ad immagazzinare dati e programmi.

Interprete di una macchina astratta:

Componente della macchina astratta che esegue le istruzioni contenute nei programmi. L'interprete si occupa:

- del controllo della sequenza di esecuzione delle operazioni;
- del controllo del trasferimento dei dati e di gestione dei dati primitivi (dati che sono rappresentabili in modo diretto nella macchina);
- della gestione della memoria.

Ciclo di esecuzione dell'interprete:

- acquisizione dalla memoria della prossima istruzione da eseguire;
- decodifica dell'istruzione (qual'è l'operazione e quali sono gli operandi);
- acquisizione dalla memoria degli operandi;
- esecuzione dell'operazione richiesta (un'operazione primitiva della macchina);

- memorizzazione del risultato;
- esecuzione dell'istruzione successiva oppure arresto.

Linguaggio macchina:

Data una macchina astratta M_L, il linguaggio L "compreso" dall'interprete di M_L è detto linguaggio macchina di M_L.

Linguaggi di basso livello:

Linguaggi di macchine astratte molto vicine alla macchina hardware o che coincidano con essa.

Linguaggio assembly:

Un particolare linguaggio di basso livello che è una versione simbolica del linguaggio di una macchina hardware (cioè che usa simboli come ADD, MUL, ecc invece dei codici binari corrispondenti) i programmi in linguaggio assembly sono tradotti in codice macchina da un opportuno programma detto assemblatore.

Linguaggi di alto livello:

Linguaggi che, mediante opportuni meccanismi di astrazione, permettono di usare costrutti che prescindono dalle caratteristiche fisiche del calcolatore, ciò che rende il loro uso più facile per l'utente umano.

Implementare un linguaggio di programmazione:

Realizzare una macchina astratta che abbia L come linguaggio macchina.

Microprogrammazione:

Tecnica che permette a calcolatori diversi, con una CPU diversa, di avere lo stesso linguaggio assembly: la macchina astratta assembly non corrisponde direttamente alla macchina hardware, anzi, il suo interprete è realizzato (implementato) in un linguaggio di livello più basso (microistruzioni molto semplici) il quale è interpretato direttamente dalle strutture fisiche della macchina microprogrammata.

Realizzazione di una macchina astratta in hardware:

Realizzare in hardware, mediante dispositivi fisici quali memorie, reti aritmetico-logiche, bus, ecc, le strutture dati e algoritmi che costituiscono M_L, costruire una macchina fisica tale che il suo linguaggio macchina coincida con L.

Simulazione di una macchina astratta mediante software:

Realizzazione delle strutture dati e degli algoritmi di M_L mediante programmi scritti in un altro linguaggio L' già implementato.

Emulazione di una macchina astratta mediante firmware:

Simulazione delle strutture dati e degli algoritmi di ML mediante microprogrammi, invece che programmi di un linguaggio di alto livello, come nella simulazione mediante software.

Vantaggi e svantaggi dei tre modi per realizzare una macchina astratta:

- realizzazione in hardware: massima velocità, ma flessibilità nulla;
- realizzazione mediante software: massima flessibilità e minima velocità;
- realizzazione mediante firmware: soluzione intermedia tra le due.

Pedice *L*:

Indica che un particolare costrutto (macchina, interprete, programma, ecc) si riferisce al linguaggio L.

Apice *L*:

Indica che un programma è scritto nel linguaggio L.

***Prog*^{L}:**

L'insieme di tutti i possibili programmi che si possono scrivere nel linguaggio L.

Descrizione di un programma scritto in L come una funzione parziale:

$P^L : D \rightarrow D$ (D denota l'insieme dei dati di input e output) tale che

- P^L *(Input) = Output* se l'esecuzione di P^L sul dato di ingresso *Input* termina e produce come risultato *Output*;

- la funzione non è definita se l'esecuzione sul dato di ingresso *Input* non termina.

Interprete (definizione formale):

Un interprete per un linguaggio L, scritto nel linguaggio L_0 (già implementato), è un programma che realizza la funzione parziale: $I^{L0}{}_L : (Prog^L \times D) \to D$ tale che $I^{L0}{}_L(P^L, Input) = P^L(Input)$

Compilatore (definizione formale):

Un compilatore da L a L_0 è un programma che realizza una funzione $C_{L,L0} : Prog^L \to Prog^{L0}$ tale che, dato un programma P^L, se $C_{L,L0}(P^L) = Pc^{L0}$ allora per ogni *Input* appartenenti a D, $P^L(Input) = Pc^{L0}(Input)$.

Linguaggio sorgente di un compilatore:

Linguaggio del programma che il compilatore prende in input.

Linguaggio oggetto di un compilatore:

Linguaggio del programma che il compilatore produce in output, linguaggio nel quale traduce i programmi.

Gerarchia di macchine astratte:

$M_{L0}, M_{L1}, ..., M_{Ln}$ dove M_{Li} è implementata sfruttando le funzionalità (cioè il linguaggio) della macchina sottostante M_{Li-1}; contemporaneamente, M_{Li} fornisce il proprio linguaggio L_i alla macchina sovrastante M_{Li+1} che, sfruttando tale linguaggio, utilizza le nuove funzionalità che M_{Li} offre rispetto ai livelli inferiori.

Macchina intermedia:

Fra la macchina M_L che vogliamo realizzare e la macchina ospite Mo_{Lo} introduciamo un livello intermediario con il suo stesso linguaggio L_i (il linguaggio intermedio) e la sua stessa macchina astratta Mi_{Li} (la macchina intermedia).

Implementazione interpretativa pura:

Per implementare un linguaggio L, cioè realizzare una macchina astratta M_L, usando una macchina ospite Mo_{Lo}, una soluzione possibile è quella di scrivere un interprete per L in L_o che sia eseguibile su Mo_{Lo}.

Questa soluzione è flessibile, più facile da realizzare e occupa meno memoria (non genera nuovo codice da memorizzare). Inoltre, questa soluzione favorisce la portabilità tra macchine fisiche e sistemi operativi diversi.

Implementazione compilativa pura:

Per implementare un linguaggio L, cioè realizzare una macchina astratta M_L, usando una macchina ospite Mo_{Lo}, una soluzione possibile è quella di scrivere un programma ${C^{La}}_{L,Lo}$ (un compilatore) in grado di tradurre, preservandone la semantica, programmi in L in programmi in L_0, che saranno eseguiti su Mo_{Lo}.

Questa soluzione è più efficiente (in tempo di esecuzione), ma meno flessibile e più difficile da implementare.

Capitolo 6 — I nomi e l'ambiente

Nome:

Una sequenza di caratteri usata per rappresentare qualche altra cosa e che permette di astrarre sia aspetti relativi ai dati (ad esempio denotando con un nome una locazione di memoria), sia aspetti relativi al controllo (ad esempio rappresentando un insieme di comandi con un nome).

Procedura:

Funzioni, metodi e sottoprogrammi.

Binding:

Legame o associazione fra un nome e l'oggetto da esso denotato.

Statico:

Si riferisce a tutto quello che avviene prima dell'esecuzione.

Dinamico:

Indica tutto quello che avviene al momento dell'esecuzione.

Ambiente (referencing environment):

L'insieme delle associazioni tra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell'esecuzione. I cambiamenti nell'ambiente avvengono generalmente all'entrata e all'uscita di un blocco.

Aliasing:

Situazione nella quale uno stesso oggetto è visibile mediante nomi diversi nello stesso ambiente (es: il passaggio per riferimento è una frequente causa di aliasing).

Blocco:

Regione testuale del programma, identificata da un segnale di inizio ed uno di fine, che può contenere dichiarazioni locali a quella regione.

Blocco associato ad una procedura:

Blocco associato alla dichiarazione di una procedura e che testualmente corrisponde al corpo della procedura stessa, esteso con le dichiarazioni relative ai parametri formali.

Blocco in-line (o anonimo):

Blocco che non corrisponde ad una dichiarazione di procedura e che può pertanto comparire (in genere) in una qualsiasi posizione dove sia richiesto un commando.

Regola di visibilità canonica per i linguaggi con blocchi:

Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome. In tal caso, nel blocco in cui compare la ridefinizione, la nuova dichiarazione maschera (nasconde) la precedente. Una variabile locale ad un blocco non è visibile nei blocchi che sono esterni ad esso o allo stesso livello.

Ambiente locale:

Ambiente costituito dall'insieme delle associazioni per nomi dichiarati localmente al blocco; nel caso in cui il blocco sia relativo ad una procedura. L'ambiente locale contiene anche le associazioni relative ai parametri formali.

Ambiente non locale:

Ambiente costituito dalle associazioni relative ai nomi che sono visibili all'interno di un blocco ma che non sono stati dichiarati localmente.

Ambiente globale:

Ambiente che contiene le associazioni per i nomi che sono usabili in tutti i blocchi che compongono il programma (l'ambiente globale fa parte dell'ambiente non locale).

Scope statico:

Regola, detta anche dello scope annidato più vicino, definita da tre punti:

- le dichiarazioni locali di un blocco includono solo quelle presenti nel blocco e non quelle eventualmente presenti in blocchi annidatti all'interno di esso;

- l'associazione valida per un nome usato all'interno di un blocco è quella presente nell'ambiente locale del blocco, se esiste; se non esiste, si considerano le associazioni esistenti nell'ambiente locale del blocco immediatamente esterno al blocco di partenza e così via; se nessuna associazione per il nome è presente nel programma si cerca nell'ambiente predefinito del linguaggio, se anche qui essa non esiste, si ha un errore;

- se un blocco ha un nome (p.e. blocchi associati a procedure) tale nome fa parte dell'ambiente locale del blocco immediatamente esterno, che contiene questo blocco.

Nello scope statico, le regole di visibilità dipendono solo dalla struttura sintattica del programma, questa è la regola più usata nei linguaggi moderni.

Scope dinamico:

Secondo questa regola, detta anche regola dell'associazione più recente, l'associazione valida per un nome X, in un qualsiasi punto P di un programma, è la più recente associazione creata per X che sia ancora attiva quando il flusso di esecuzione arriva a P.

Nello scope dinamico, le regole di visibilità sono influenzate dal flusso del controllo a run-time, questa regola è più semplice da implementare ma è oggi poco utilizzata.

Scope di una dichiarazione:

Porzione del programma nella quale la dichiarazione è visibile secondo la regola di scope statico.

Capitolo 7 — La gestione della memoria

Gestione statica della memoria:

Quando la memoria è allocata dal compilatore, prima dell'esecuzione.

Indirizzo di ritorno:

L'indirizzo al quale deve tornare il controllo quando termina la procedura.

Record di attivazione (RdA):

Lo spazio di memoria, allocato sulla pila, dedicato ad un blocco in-line o ad una chiamata di procedura. L'RdA di un blocco in-line contiene:

- il puntatore di catena dinamica;
- le variabili locali;
- eventuali risultati intermedi.

Nel RdA di una chiamata di procedura o funzione ad essi si aggiungono:

- il puntatore di catena statica (per realizzare le regole di scope statico);
- l'indirizzo di ritorno;
- l'indirizzo del risultato (una locazione di memoria all'interno del RdA del chiamante);
- i parametri.

Pila a run-time (o pila di sistema):

Pila sulla quale sono memorizzati gli RdA.

Puntatore di catena dinamica:

Campo del record di attivazione che serve per memorizzare il puntatore al precedente record di attivazione sulla pila (questa informazione è necessaria perché gli RdA in generale hanno dimensioni diverse).

Catena dinamica:

Insieme dei collegamenti realizzati dai puntatori di catena dinamica.

Offset (per gestire i campi del RdA):

Valore positivo o negativo che si aggiunge al valore del puntatore di catena dinamica per ottenere l'indirizzo di un campo specifico in un RdA.

Stack pointer:

Puntatore che indica la prima posizione di memoria libera in una pila.

Heap:

Zona di memoria nella quale blocchi di memoria possono essere allocati e deallocati in modo relativamente libero. È usata per gestire allocazioni esplicite di memoria, che possono avvenire in momenti di tempo arbitrari.

Heap con blocchi di dimensione fissa:

Lo heap è costituito da blocchi di dimensione fissa, collegati in una struttura a lista, detta *lista libera*. Quando un blocco è allocato, il primo elemento viene rimosso dalla lista ed il puntatore alla lista libera diventa il puntatore all'elemento successivo. Quando della memoria viene deallocata, il blocco liberato viene collegato nuovamente alla testa della lista.

Frammentazione interna:

Quando si alloca un blocco di dimensione maggiore di quella richiesta dal programma la memoria non utilizzata andrà sprecata (questo avviene ad esempio quando i blocchi allocati devono essere di una dimensione fissa).

Frammentazione esterna:

Quando la lista libera è composta di blocchi di dimensione relativamente piccola può accadere che, anche se la somma della memoria libera totale è sufficiente, non si riesce ad usare effettivamente la memoria libera perché non è contigua.

First fit:

Modo di gestione di un heap con blocchi di dimensione variabile in cui per allocare un blocco di dimensione k si cerca nella lista libera il primo

blocco di dimensione sufficiente (questa tecnica privilegia il tempo di gestione).

Best fit:

Modo di gestione di un heap con blocchi di dimensione variabile in cui per allocare un blocco di dimensione *k* si cerca nella lista libera il blocco di dimensione minima fra tutti quelli di dimensione sufficiente (questa tecnica privilegia la riduzione dell'occupazione di memoria).

Liste libere multiple:

Metodo di gestione dello heap che usa più liste libere, ognuna con blocchi di dimensione diversa.

Due esempi con blocchi di dimensione dinamica:

- *buddy system*: le dimensioni dei blocchi delle varie liste libere sono potenze di due (se non c'è un blocco disponibile nella lista adeguata se ne prende uno della lista successiva e lo si divide in due, dopo, quando un blocco risultante da una divisione viene restituito alla lista libera, cerca il suo 'compagno' per riunirsi a lui);

- *heap di Fibonacci*: simile, ma usa i numeri di Fibonacci invece che le potenze di due (la dimensione dei blocchi cresce più lentamente).

Implementazione dello scope statico con la catena statica:

Per gestire a run-time la regola di scope statico, il record di attivazione di un blocco è collegato dal puntatore di catena statica al record del blocco che gli è immediatamente esterno. Per trovare l'RdA giusto a cui puntare, se il chiamato si trova all'esterno del chiamante, si usa il livello di annidamento nella struttura del programma del chiamante e del chiamato per sapere di quanto risalire sulla catena statica (il livello di annidamento dei blocchi è calcolato alla compilazione); se il chiamato si trova all'interno del chiamante, allora il puntatore di catena statica del chiamato deve puntare al RdA del chiamante.

Tabella dei simboli:

La tabella dei simboli, creata durante la compilazione, memorizza i nomi e tutte le informazioni necessarie per gestire gli oggetti che denotano (il

tipo, la distanza fra lo scope della chiamata e quello della dichiarazione, necessaria a run-time per gestire la catena statica, l'offset, ecc).

Implementazione dello scope statico tramite display:

Questa tecnica usa un vettore, detto display, contenente *n* elementi (*n* = il livello massimo di annidamento presente nel programma). L'elemento *i* del display punta verso l'RdA del blocco di livello *i* correntemente attivo (se ci sono più blocchi attivi di questo livello, bisogna che il blocco più recente ricordi il blocco precedente che è stato attivato in questo livello, così da potere ripristinare questo elemento del display quando si esce da questo blocco). Per trovare un oggetto non locale, dichiarato in un blocco esterno di livello *k*, si accede direttamente al RdA contenente questo oggetto accedendo all'elemento *k* del display e seguendo il puntatore presente in tale posizione.

Implementazione dello scope dinamico con memorizzazione diretta nei RdA:

Per risolvere un riferimento non locale al nome X basta percorrere a ritroso la pila a run-time (o pila di sistema) fino a trovare l'RdA nel quale il nome X è stato dichiarato. In questo caso, bisogna memorizzare le associazioni fra nomi e oggetti denotati direttamente nei RdA (a differenza dello scope statico, in cui basta l'offset per identificare l'oggetto denotato all'interno di un RdA).

Implementazione dello scope dinamico mediante una lista di associazione:

La lista di associazione (detta *A-list*) è una struttura LIFO dove si inseriscono le nuove associazioni locali quando si entra in un ambiente nuovo e si rimuovono le associazioni locali quando si esce da un ambiente. Ogni elemento della lista contiene il nome dell'oggetto, la locazione di memoria dove esso è memorizzato, il suo tipo, un flag che indica se questa associazione è attiva oppure no ed eventuali altre informazioni utili.

Implementazione dello scope dinamico mediante CRT (Central Referencing environment Table):

La CRT è una tabella centrale che, ad ogni nome usato nel programma, associa un flag che indica se l'associazione per quel nome è attiva o no ed un

puntatore alle informazioni sull'oggetto (locazione in memoria, tipo, ecc). La tabella viene modificata ogni volta che si entra o si esce da un blocco; le associazione temporaneamente disattivate (sostituite da quelle del nuovo blocco chiamato) devono allora essere memorizzate da qualche parte (ad esempio associando ad ogni entrata della tabella una pila con l'associazione attiva e poi quelle disattivate, oppure usando una pila separata dalla tabella, contenente solo le associazioni disattivate).

Capitolo 8 — Strutturare il controllo

Controllo di sequenza:

La gestione del flusso di esecuzione delle istruzioni di un programma (al livello fisico, corrisponde alla gestione del PC, dei salti, ecc).

Espressione:

Entità sintattica la cui valutazione produce un valore (≠ un comando) oppure non termina, nel qual caso l'espressione è indefinita.

Notazione infissa:

Il simbolo di un operatore binario è posto fra le espressioni che rappresentano i due operandi.

es. x+y

NB: per evitare ambiguità nell'applicazione degli operatori agli operandi, sono necessarie le parentesi e opportune regole di precedenza.

Notazione prefissa (o notazione polacca prefissa):

Il simbolo che rappresenta l'operatore precede i simboli che rappresentano gli operandi.

es. + x y oppure + (x y)

NB: usando questo tipo di notazione non servono parentesi e regole di precedenza tra gli operatori, purché sia nota l'arietà di ogni operatore.

La valutazione di queste espressioni è semplice: si scade l'espressione di sinistra a destra e si usa una pila. Serve però controllare che sulla pila ci siano tutti gli operandi per l'ultimo operatore letto.

Effetto collaterale:

Azione che influenza i risultati (parziali o finali) di una computazione senza però restituire esplicitamente un valore al contesto nel quale essa è presente, ad es. una funzione che modifica il valore di un suo parametro, senza che questo operando sia il valore restituito.

Valutazione eager:

Si valutano prima tutti gli operandi per poi applicare l'operatore ai valori ottenuti.

Ad esempio, per valutare (a==0) || (b>a) si valutano prima (a==0) e (b>a) e poi tutta l'espressione.

Valutazione lazy:

Si passano gli operandi non valutati all'operatore, il quale, al momento della sua valutazione, decide quali operandi sono effettivamente necessari, valutando solo questi.

Ad esempio, per valutare (a==0) || (b<a) se (a==0) è vero, l'espressione viene valutata come vera, senza aver bisogno di valutare il secondo operando.

Comando:

Una entità sintattica la cui valutazione non necessariamente restituisce un valore, ma può avere un effetto collaterale.

Lo scopo di un comando è quello di modificare lo stato di partenza.

Variabile modificabile:

Secondo questo modello (tipico del paradigma imperativo classico), la variabile è vista come una sorta di contenitore o locazione, al quale si può dare un nome e che può contenere dei valori (solitamente di un tipo determinato). Questi valori possono cambiare nel tempo.

Modello a riferimento (reference model):

Nel contesto di alcuni linguaggi imperativi (soprattutto orientati agli oggetti), una variabile non è costituita da un contenitore per un valore, ma è un riferimento per un valore, tipicamente memorizzato sullo heap. Questi riferimenti sono analoghi alla nozione di puntatore, ma senza la possibilità di manipolarli direttamente come è permesso dall'aritmetica dei puntatori.

Variabile (in un linguaggio funzionale):

Nei linguaggi funzionali puri la nozione di variabile è simile a quella della notazione matematica: una variabile non è altro che un identificatore che denota un valore, che non può essere modificato.

Assegnamento:

Il comando di base che permette di modificare il valore delle variabili modificabili.

l-valori (left values):

Valori che sostanzialmente indicano locazioni e quindi sono i valori di espressioni che possono stare alla sinistra di un comando di assegnamento.

r-valori (right values):

Valori che possono essere contenuti in locazioni, quindi sono i valori di espressioni che possono stare alla destra di un comando di assegnamento.

Programmazione strutturata:

Una serie di "prescrizioni" elaborate negli anni '70 volte a permettere uno sviluppo il più possibile strutturato del codice. Alcuni dei suoi punti salienti sono:

- l'uso di costrutti strutturati per il controllo (costrutti che abbiano un solo punto di ingresso ed un solo punto di uscita, tali if, while, for, funzioni, ecc., ma non goto), in modo da evitare il "codice spaghetti";
- la progettazione top-down (dal più astratto al più preciso): il programma è sviluppato per raffinamenti successivi;
- la modularizzazione del codice (raggruppare in moduli più piccoli i comandi che corrispondono ad ogni specifica funzione dell'algoritmo che si vuole implementare);
- l'uso di tipi di dati strutturati (meglio usare, per esempio, un record studente che tante variabili diverse per le informazioni relative ad uno stesso studente);
- l'uso di nomi significativi;
- l'uso estensivo di commenti.

Break:

Comando strutturato che salta alla fine di un ciclo (è sostituibile con goto fine, ma, a differenza di questo, è strutturato, perché il punto di uscita del costrutto rimane unico: si può saltare alla fine, ma non altrove).

Ricorsione in coda:

Sia f una funzione che nel suo corpo contenga la chiamata ad una funzione g (diversa da f o anche eguale ad f stessa). La chiamata di g si dice *chiamata in coda* se la funzione f restituisce il valore restituito da g senza dover fare alcuna ulteriore computazione. La funzione f ha la ricorsione in coda (è *tail recursive*) se tutte le chiamate ricorsive presenti in f sono chiamate in coda.

NB: la ricorsione in coda, a differenza della ricorsione generica, può essere implementata usando un solo record di attivazione e quindi uno spazio di memoria costante.

Capitolo 9 — Astrarre sul controllo

Parametri formali:

Parametri che compaiono nella definizione della funzione. Sono nomi che si comportano come dichiarazioni locali alla funzione stessa.

Parametri attuali:

Parametri che compaiono nella chiamata della funzione.

Variabili *static*:

In C, variabili locali ad una funzione che mantengono il proprio valore tra un'invocazione della funzione e quella successiva.

Passaggio per valore:

Al momento della chiamata, il parametro attuale (che può essere un'espressione) viene valutato e lo r-valore ottenuto viene assegnato al parametro formale. Al termine della procedura, il parametro formale viene distrutto come tutto l'ambiente locale della procedura stessa.

Passaggio per riferimento (detto anche per variabile):

Al momento della chiamata, viene valutato lo l-value del parametro attuale e l'ambiente locale della procedura è esteso con un'associazione tra il parametro formale e lo l-value dell'attuale (creando così una situazione di aliasing) se il parametro attuale è una variabile, i parametri attuale e formale sono due nomi per la stessa variabile.

Passaggio per costante:

I parametri formali passati con questa modalità sono soggetti al vincolo statico di non poter essere modificati nel corpo della funzione. Da un punto di vista semantico, il passaggio per costante coincide col passaggio per valore, ma può essere implementato come il passaggio per valore (per dati di piccole dimensioni) o mediante riferimento (senza copia, per strutture dati più grandi).

Passaggio per risultato:

L'ambiente locale della procedura è esteso con un'associazione tra il parametro formale e una nuova variabile. Al momento della terminazione (normale) della procedura, subito prima della distruzione dell'ambiente locale, il valore corrente del parametro formale viene assegnato alla locazione ottenuta mediante lo l-valore dell'attuale.

Passaggio per valore-risultato:

Al momento della chiamata, il parametro attuale viene valutato e lo r-valore così ottenuto viene assegnato al parametro formale. Al termine della procedura, subito prima della distruzione dell'ambiente locale, il valore corrente del parametro formale viene assegnato alla locazione corrispondente al parametro attuale durante l'esecuzione del corpo. Non c'è un legame tra il parametro formale e l'attuale.

Passaggio per nome:

La semantica della chiamata per nome è stabilita dalla regola di copia:

Sia f una funzione con un parametro formale x e sia a un'espressione compatibile con col tipo di x. Una chiamata a f con parametro attuale a è semanticamente equivalente all'esecuzione del corpo di f nel quale tutte le occorrenze del parametro formale x sono state sostituite con a.

Questa regola implica di mantenere durante l'esecuzione un legame costante tra parametro formale e parametro attuale.

Iteration count:

$$\left|\frac{(fine - inizio + passo)}{passo}\right|$$

Determina il numero di volte che verrà eseguito un ciclo for prima dell'inizio delle iterazioni.

Funzione di ordine superiore:

Funzione che ha come parametro o restituisce come risultato un'altra funzione.

NB: le funzioni che restituiscono un'altra funzione come risultato forzano il linguaggio ad abbandonare la disciplina a pila per i record di attivazione.

Deep binding:

L'ambiente non locale in cui valutare la funzione passata in parametro è l'ambiente attivo al momento della creazione del legame tra le due funzioni (al momento della chiamata della funzione che prende l'altra come parametro). Con scope statico, è sempre usato il deep biding.

Shallow binding:

L'ambiente non locale in cui valutare la funzione passata in parametro è l'ambiente attivo al momento della chiamata della funzione passata come argomento, all'interno dell'altra funzione. Il shallow biding è usato a volte con lo scope dinamico.

Eccezione:

Un evento particolare che si verifica durante l'esecuzione di un programma e che non deve (o non può) essere gestito dal normale flusso del controllo (ad es. divisione per zero, overflow, ecc).

Implementazione canonica del passaggio per nome:

L'ambiente locale della procedura è esteso con un'associazione tra il parametro formale e una chiusura, costituita dal parametro attuale e dall'ambiente in cui occorre la chiamata. Ogni chiamata al formale viene risolta mediante una valutazione *ex novo* del parametro attuale nell'ambiente fornito dalla chiusura.

Chiusura: Struttura dati costituita da una porzione di codice e un ambiente di valutazione, rappresentata a livello implementativo da una coppia (puntatore al codice, puntatore ad un RdA) costituisce la modalità canonica di implementazione del passaggio per nome e di tutte le situazioni in cui una funzione debba essere passata per parametro o restituita come risultato.

Implementazione del deep binding (con scope statico):

Al parametro formale che è una funzione è associata una chiusura. Al momento in cui il formale è usato per invocare una funzione, la macchina astratta trova il codice al quale trasferire il controllo nella prima componente

della chiusura e assegna il contenuto della seconda componente della chiusura al puntatore di catena statica del RdA della nuova invocazione.

Puntatore di catena dinamica:

Campo dei RdA che serve a memorizzare il puntatore al precedente RdA sulla pila (l'ultimo RdA creato in precedenza).

Implementazione delle eccezioni mediante RdA:

Ogni volta che, durante l'esecuzione, si entra in un blocco protetto, nel RdA viene inserito un puntatore al gestore corrispondente (insieme al tipo delle eccezioni a cui si riferisce) quando si esce normalmente dal blocco, questo riferimento viene tolto dalla pila. Quando viene sollevata un'eccezione, la macchina astratta cerca un gestore nel RdA corrente, se non lo trova, toglie l'RdA dalla pila e risolleva l'eccezione.

Vantaggio: concettualmente semplice.

Svantaggio: la macchina astratta deve manipolare la pila quando entra e esce da un blocco protetto anche quando non si verifica l'eccezione.

Implementazione mediamente più efficiente delle eccezioni:

Ad ogni procedura, il compilatore associa un blocco nascosto costituito dal corpo della procedura e di un gestore nascosto che ripristina lo stato e risolleva l'eccezione. Si usa una tabella EH in cui, per ogni blocco protetto (nascosto o no), ci sono due indirizzi i_p (indirizzo (blocco) protetto) e i_g (indirizzo gestore).

Quando si entra e si esce normalmente da un blocco protetto non si deve fare nulla (vantaggio di questa implementazione). Quando si solleva (o si risolleva) un'eccezione, si cerca nella tabella la linea con il massimo i_p tale che $ip \leq PC \leq ig$ (PC = program counter al momento del (ri)sollevamento). Lo svantaggio di questa soluzione è che l'implementazione è più costosa caso si verifichi un'eccezione.

Capitolo 10 — Strutturare i dati

Tipo di dato:

Una *collezione di valori* omogenei (che condividono alcune proprietà strutturali, che li rendono simili tra loro) ed effettivamente presentati, dotata di un *insieme di operazioni* che manipolano tali valori.

Sistema di tipi di un linguaggio:

Il complesso delle informazioni e delle regole che governano i tipi in quel linguaggio, costituito da:

- insieme tipi predefiniti del linguaggio;
- meccanismi per definire nuovi tipi;
- meccanismi relativi al controllo dei tipi (regole di equivalenza, compatibilità e inferenza dei tipi);
- se i vincoli siano da controllare staticamente o dinamicamente.

Sistema di tipi type safe:

Quando nessun programma, durante l'esecuzione, può generare un errore non segnalato che derivi da una violazione di tipo (accesso a memoria non allocata, chiamata di un valore che non sia una funzione, ecc.).

Valori denotabili:

Valori che possono essere associati ad un nome (ad esempio, nei comuni linguaggi imperativi i valori del tipo delle funzioni da int a int sono denotabili perché gli possiamo dare un nome con una dichiarazione).

Valori esprimibili:

Valori che possono essere il risultato di un'espressione complessa, cioè diversa da un semplice nome (ad esempio, nei comuni linguaggi imperativi i valori del tipo delle funzioni da int a int non sono esprimibili perché non ci sono espressioni che restituiscono una funzione come risultato della loro valutazione, nei linguaggi funzionali invece, di solito sono esprimibili).

Valori memorizzabili:

Valori che possono essere memorizzati in una variabile (ad esempio, nei comuni linguaggi imperativi i valori del tipo delle funzioni da int a int non sono memorizzabili, perché non possiamo assegnare una funzione ad una variabile, mentre nei linguaggi funzionale spesso lo si può fare).

Controllo statico dei tipi (tipizzazione statica):

Caratteristica di un linguaggio dove i controlli dei vincoli di tipizzazione avvengono a tempo di compilazione, sul testo del programma.

Vantaggi: gli errori possono essere rilevati dal programmatore prima della consegna del programma all'utente; l'esecuzione è più efficiente; abbrevia la fase di testing e debugging.

Svantaggi: la progettazione di un tale linguaggio è più complessa; la compilazione è più lenta; il controllo può decretare sbagliati programmi che non causano errori di tipo durante l'esecuzione.

Controllo dinamico dei tipi (tipizzazione dinamica):

Caratteristica di un linguaggio dove i controlli dei vincoli di tipizzazione avvengono a tempo di esecuzione.

Vantaggi: previene gli errori di tipi.

Svantaggi: è inefficiente in esecuzione, dato che le operazioni sono intercalate con i controlli di tipo; l'eventuale errore è rilevato solo durante l'esecuzione.

Tipi scalari (o tipi semplici):

Tipi i cui valori non sono costituiti da da aggregazioni di altri valori.

Tipi ordinali:

Tipi dotati da una ben definita nozione di ordine totale, che possiedono una nozione di predecessore e di successore (ad esempio i booleani, i caratteri, gli interi, gli intervalli, ecc.).

Tipi composti:

Tipi non scalari, si ottengono combinando tra loro altri tipi mediante l'utilizzo di opportuni costruttori (ad esempio i record o strutture, gli array o vettori, gli insiemi, i puntatori ed i tipi ricorsivi come le liste o gli alberi).

Dope vector:

Il descrittore di un array di dimensioni non note staticamente (siano esse fissate al momento della dichiarazione dell'array che completamente dinamiche). Un dope vector, usualmente allocato nella parte di lunghezza fissa di una RdA, contiene:

- un puntatore verso l'inizio della zona di memoria dove è memorizzato l'array;
- le informazioni necessarie al calcolo dell'offset per accedere a alla locazione in memoria di uno specifico elemento dell'array in funzione dei suoi indici.

Aritmetica dei puntatori:

Possibilità in un linguaggio di incrementare un puntatore, sottrarre tra loro due puntatori, sommare una quantità arbitraria ad un puntatore, ecc. Un tale linguaggio non può essere *type safe*: non c'è garanzia che in un generico momento dell'esecuzione di un programma una variabile dichiarata come puntatore ad un oggetto di tipo T punti ancora ad un'area di memoria in cui è memorizzato un tale oggetto.

Dangling reference:

Situazione nella quale si può accedere ad un oggetto la cui memoria è stata deallocata. Un linguaggio che permette il verificarsi di dangling reference non può essere *type safe*. In linguaggi che permettono ai puntatori di riferirsi ad oggetti sulla pila si possono sempre generare dangling reference memorizzando in un ambiente non locale l'indirizzo di una variabile locale. In linguaggi che non lo permettono, le dangling reference possono essere evitate impedendo la deallocazione esplicita.

Equivalenza per nome:

Due tipi sono equivalenti per nome solo se hanno lo stesso nome (cioè un tipo è equivalente solo a se stesso). In questo caso si dice che le definizioni di tipo in questo linguaggio sono opache.

Equivalenza strutturale:

Due tipi sono equivalenti se hanno la stessa struttura, cioè, se sostituendo tutti i nomi con le relative definizioni, si ottengono tipi identici. In questo caso, si dice che le definizioni di tipo in questo linguaggio sono trasparenti (il nome del tipo non è che un'abbreviazione del tipo che viene definito). Due tipi equivalenti possono sempre essere sostituiti l'uno al posto dell'altro senza alterare il significato del programma.

Compatibilità:

Il tipo T è compatibile con il tipo S se un valore di tipo S è ammesso in un qualsiasi contesto in cui sarebbe richiesto un valore di tipo S.

NB: la relazione di compatibilità è più debole dell'equivalenza.

Coercizioni (conversione implicita):

In presenza di compatibilità tra il tipo T e il tipo S, il linguaggio permette la presenza di un valore di T laddove sarebbe richiesto un valore di S. Il compilatore e/o la macchina astratta inseriscono allora una conversione implicita da T a S (che può corrispondere a codice eseguito o no), che chiamiamo coercizione di tipo.

Conversioni esplicite (o cast):

Sono annotazioni nel linguaggio che specificano che un valore di un tipo deve essere convertito in un altro tipo. Tale conversione può essere solo un'indicazione sintattica o può corrispondere a codice eseguito dalla macchina astratta.

Polimorfismo:

Un sistema di tipi nel quale uno stesso oggetto (funzione, operatore…?) può avere più di un tipo è detto polimorfo. Per analogia, diciamo che un oggetto è polimorfo quando il sistema di tipi gli assegna più di un tipo.

Overloading (o polimorfismo ad hoc):

Un nome è sovraccaricato (overloaded) quando ad esso corrispondono più oggetti e viene usata l'informazione fornita dal contesto per decidere quale oggetto è denotato da una specifica istanza di quel nome. Per esempio, il nome + è spesso usato per indicare sia la somma intera che la somma reale; è possibile definire più funzioni con lo stesso nome ma distinte dal numero o dal tipo dei parametri. La situazione di ambiguità viene risolta staticamente, utilizzando l'informazione di tipo presente nel contesto. L'overloading in realtà è polimorfismo solo in apparenza.

Polimorfismo universale parametrico:

Un oggetto esibisce polimorfismo universale parametrico quando ha una infinità di tipi diversi, che si ottengono per istanziazione da un unico schema di tipo generale, ad esempio le funzioni polimorfe universali (come una funzione che ordina un array di un qualunque tipo) o il valore null che appartiene ad ogni tipo puntatore verso oggetto di un certo tipo. Il polimorfismo può essere esplicito come in C, Java o implicito come in ML (il programma può non riportare alcuna indicazione di tipo, invece, si inferisce, per ogni oggetto, il suo tipo più generale).

Polimorfismo universale di sottotipo:

Un oggetto esibisce polimorfismo universale di sottotipo quando ha un'infinità di tipi diversi, che si ottengono per istanziazione da uno schema di tipo generale, sostituendo ad un opportuno parametro i sottotipi di un tipo assegnato. È una forma più limitata di polimorfismo universale rispetto al polimorfismo parametrico perché la situazione di polimorfismo non è generale ma limitata ai sottotipi di un dato tipo. Questo tipo di polimorfismo è presente tipicamente nei linguaggi orientati agli oggetti.

Controllore dei tipi (type checker):

Modulo responsabile di verificare che un programma rispetti le regole imposte dal sistema dei tipi (in particolare la compatibilità). Nel caso di un linguaggio con controlli statici, il controllore è un modulo del compilatore; nel caso di controlli dinamici, il controllore è un modulo del supporto a tempo di esecuzione.

Linguaggi non sicuri:

Linguaggi il cui sistema di tipi è nella sostanza un suggerimento metodologico per il programmatore, nel senso che il linguaggio permette di aggirare o rilassare i controlli di tipo. Ogni linguaggio che permette di accedere alla rappresentazione di un tipo di dato o che permette l'aritmetica dei puntatori appartiene a questa categoria (C e C++, ad esempio).

Linguaggi localmente non sicuri:

Linguaggi nei quali il sistema di tipi è ben regolato e i tipi controllati, ma che contengono alcuni, limitati, costrutti che permettono di scrivere programmi non sicuri (record varianti e deallocazione esplicita della memoria, ad esempio). ALGOL, Pascal e Ada fanno parte di questa categoria.

Linguaggi sicuri:

Linguaggi per i quali un teorema garantisce che l'esecuzione di un teorema garantisce che l'esecuzione di un programma tipizzato non può mai generare un errore non rilevato, indotto dalla violazione di un tipo. LISP, Scheme, ML e Java fanno parte di questa categoria.

Tombstone:

Meccanismo per evitare le dangling reference. Ogni volta che viene allocato un oggetto sullo heap o creato un puntatore che si riferisce alla pila, la macchina astratta alloca un ulteriore parola di memoria, detta tombstone, che viene inizializzata con l'indirizzo di dell'oggetto, mentre il puntatore riceve l'indirizzo della tombstone.

Alla deallocazione di un oggetto (o quando un indirizzo sulla pila diventa non più valido perché perché l'RdA di cui fa parte viene tolto dalla pila), la tombstone viene invalidata. Ogni tentativo di accedere all'indirizzo contenuto in una tombstone invalidata è catturato dal meccanismo di protezione degli indirizzi della macchina fisica sottostante.

Le tombstone, per quanto semplici, sono costose in termini di efficienza (tempo di creazione e controllo delle tombstone, doppio accesso indiretto) e di spazio (spazio occupato dalle tombstone create, che rimane occupato anche quando sono invalidate).

Lucchetti e chiavi:

Meccanismo per evitare le dangling reference verso lo heap. Ogni volta che viene creato un oggetto sullo heap, viene associato all'oggetto anche un "lucchetto", cioè una parola di memoria nella quale viene memorizzato un valore casuale. In questo sistema, un puntatore è costituito da una coppia: l'indirizzo vero e proprio e una "chiave" di valore identico al lucchetto dell'oggetto puntato.

Tutte le volte che si dereferenzia un puntatore, se la chiave non coincide con il lucchetto la macchina astratta segnala un errore. Nel momento in cui un oggetto viene deallocato, il suo lucchetto viene annullato, in modo che le chiavi che prima lo aprivano ora causino un errore.

Questo meccanismo non soffre del problema dell'accumulo delle tombstone, ma è anch'esso costoso in termini di efficienza e di spazio (per quest'ultimo è ancora più costoso delle tombstone).

Garbage collector:

Meccanismo della macchina astratta dei linguaggi senza deallocazione esplicita della memoria recupera automaticamente la memoria allocata sullo heap e non più utilizzata. È stato introdotto prima nei linguaggi funzionali, poi anche imperativi, Java ne ha uno.

Garbage collector basati su contatori dei riferimenti:

Al momento della creazione di un oggetto sullo heap, viene allocato insieme ad esso un intero, il contatore dei riferimenti, inaccessibile al programmatore. La macchina astratta si incarica di far sì che a run-time tale contatore contenga il numero dei puntatori attivi a quel oggetto. Quando un contatore raggiunge il valore 0, l'oggetto relativo può essere deallocato e restituito alla lista libera.

Un difetto di questa tecnica è quello di non essere capace di deallocare tutte le strutture circolari. Un altro é di essere anche abbastanza inefficienti, in quanto hanno un costo proporzionale al lavoro complessivo compiuto dal programma. Un suo vantaggio è che in questo caso controllo e recupero sono mescolati con il funzionamento normale del programma.

Garbage collector basati sulla tecnica *mark and sweep*:

1) Si marcano tutti gli oggetti sullo heap come inutilizzati.
2) Partendo dai puntatori attivi presenti sulla pila, si attraversano ricorsivamente tutte le strutture dati presenti sullo heap, marcando come in uso ogni oggetto che viene attraversato.
3) Lo heap viene spazzato (*swept*): tutti i blocchi marcati come inutilizzati sono restituiti alla lista libera.

Un collector *mark and sweep* non è incrementale: viene invocato solo quando la memoria disponibile sullo heap sta per finire e, mentre è attivo, l'utente può sperimentare un significativo degrado della performance.

Inoltre, è causa di frammentazione esterna e fa sì che gli oggetti sullo heap non siano ordinati cronologicamente e ciò ostacola il funzionamento delle cache (?). Per ovviare a questi problemi, durante la fase di sweep si possono traslare i blocchi vivi in modo da renderli contigui. Questa tecnica, detta *mark and compact*, è più costosa di *mark and sweep* se vi sono molti oggetti vivi da spostare.

Garbage collector basati su copia:

Un esempio di essi è il garbage collector detto *stop and copy*, in cui lo heap è diviso in due *semispazi* di uguali dimensioni. Durante l'esecuzione normale, solo uno dei due semispazi è in uso. Quando la memoria del semispazio è esaurita, viene invocato il garbage collector. Questo, a partire dai puntatori presenti sulla pila, inizia una visita delle strutture attive presenti nel semispazio corrente e copiandole nell'altro semispazio, in modo contiguo.

Aumentando la memoria disponibile per i semispazi, diminuiremo la frequenza con la quale è chiamato il garbage collector e dunque il costo totale della garbage collection (supponendo che la quantità di oggetti vivi sia più o meno costante nel tempo, il tempo richiesto per dal garbage collector, quando viene chiamato, è sempre più o meno lo stesso). A patto di avere abbastanza memoria, questo garbage collector può dunque essere molto efficiente.

Capitolo 11 — Astrarre sui dati

Tipo di dato astratto:

Astrazioni sui dati, permesse da alcuni linguaggi tali ML, che si comportano come i tipi predefiniti rispetto all'inaccessibilità della rappresentazione. Un tipo di dato astratto è caratterizzato dal suo nome, della sua implementazione (rappresentazione) e da un insieme di operazione per la manipolazione di valori di questo tipo. L'implementazione del tipo e delle sue operazioni sono inaccessibili al programmatore.

Capitolo 12 — Il paradigma orientato agli oggetti

Oggetto:

Una capsula contenente sia dati (*variabili di istanza*, o *campi*) che operazioni (*metodi*) per manipolarli e che fornisce all'esterno un'interfaccia attraverso la quale l'oggetto è accessibile.

Classe:

Un modello per un insieme di oggetti, che stabilisce quali sono i suoi dati (quanti, di quale tipo, con quale visibilità) e fissa nome, segnatura, visibilità e implementazione dei suoi metodi. In un linguaggio con classi, ogni oggetto appartiene ad almeno una classe, nel senso che la struttura dell'oggetto corrisponde alla struttura fissata dalla classe.

Sottotipo:

Se ci rappresentiamo un oggetto come un record che contiene dati e funzioni (cioè come una chiusura), T è un sottotipo di S quando T è un tipo record che contiene tutti i campi di S, più eventualmente altri.

Shadowing:

Quando una sottoclasse ridefinisce una variabile d'istanza definita in una superclasse. A differenza dell'overriding (ridefinizione di un metodo), lo shadowing è un meccanismo del tutto statico.

Ereditarietà:

Quando non li ridefinisce, una sottoclasse *eredita* i metodi e i dati della superclasse — l'implementazione del metodo e dei dati è resa disponibile alla sottoclasse. Questo permette il riuso del codice in un contesto estendibile (le modifiche vengono propagate automaticamente).

Ereditarietà singola:

Proprietà dei linguaggi in cui una classe può ereditare da una sola superclasse immediata: la gerarchia delle classi è dunque un albero.

Ereditarietà multipla:

Proprietà dei linguaggi che permettono che una classe erediti metodi di da più di una superclasse immediata: la gerarchie delle classi è in tal caso un grafo orientato aciclico. Questo può causare conflitti di nome quando una classe C eredita contemporaneamente da A e B, le quali forniscono entrambe l'implementazione di un metodo con la stessa segnatura. Un'altro problema (implementativo questa) causato dall'eredità multipla è il cosiddetto problema del diamante, che si verifica quando le diverse superclassi da cui si eredita ereditano esse stesse da una stessa superclasse.

Selezione dinamica dei metodi:

Quando un metodo viene ridefinito (una o più volte nella catena di eredità), la selezione di quale delle sue implementazioni venga usata in un'invocazione avviene a tempo d'esecuzione, sempre in funzione del tipo dell'oggetto vero e proprio e non in funzione del tipo del riferimento (o nome) di quell'oggetto (se così fosse, la selezione potrebbe essere statica).

Virtual table function (vtable):

Struttura dati contenente l'insieme di tutti i metodi di una classe, non solo quelli esplicitamente definiti o ridefiniti nella classe, ma anche tutti quelli ereditati dalle superclassi. Quando viene definita una sottoclasse B della classe A, la vtable di B si ottiene facendo una copia della vtable di A, sostituendo tutti i metodi ridefiniti in B e aggiungendo poi in fondo alla vtable i nuovi metodi definiti in B. L'uso delle vtable implica che il linguaggio abbia un sistema di tipi statico, perché sia noto a tempo di compilazione l'insieme dei metodi che possono essere inviati ad ogni oggetto.

Implementazione dell'eredità multipla:

In un linguaggio che permette l'eredità multipla, prima è creata una vtable per la sottoclasse (chiamiamola C) come se essa ereditasse da una sola superclasse A (nel modo che abbiamo descritto in precedenza). Poi si aggiunge in fondo a questa vtable i valori e i metodi dell'altra superclasse B di cui eredita — questi inizieranno ad una distanza d dal inizio del record. Quando

un'istanza di C è vista come un oggetto di tipo B, occorre sommare al riferimento alla vtable l'offset *d* per accedere ai campi e metodi.

Nel caso A e B ereditassero entrambe da una stessa superclasse T (problema del diamante), oppure i metodi e valori di questa superclasse sono copiati due volte in C, l'una per A e l'una per B, e si parla di **ereditarietà multipla con replicazione**, oppure C possiede una sola copia di T, nonostante sia A che B ne possiedano una, e si parla di **ereditarietà multipla con condivisione**.

Polimorfismo di sottotipo nei linguaggi orientati agli oggetti:

Una funzione che prende in argomento un oggetto di tipo A può prendere in argomento un valore di una qualsiasi sottoclasse di A.

Una funzione che restituisce un oggetto di tipo A può restituire un valore di una qualsiasi sottoclasse di A, però per poter usare l'oggetto restituito come un'istanza della sottoclasse a cui effettivamente appartiene e non come un'istanza di A occorre fare un cast all'ingiù esplicito.

Generici in Java:

In Java, le dichiarazioni — sia quelle di tipo (dunque classi e interfacce), che quelle di metodo — possono essere generiche. Al posto di un tipo specifico, viene allora scritto tra parentesi angolate un *parametro formale di tipo* che sarà istanziato successivamente, al momento della creazione dell'oggetto o della chiamata del metodo. Ad esempio, `Class Elem<A> {...}`.

Wildcard:

Il carattere `?` (che si legge "sconosciuto") sta per "un qualsiasi tipo" e può essere usato nelle definizioni generiche. Ad esempio, un valore di tipo `List<?>` è una lista di elementi di cui non si conosce il tipo.

Capitolo 15 — La programmazione concorrente

Thread:

La sequenza di comandi di un programma eseguiti in una specifica computazione. In un programma *sequenziale* in ogni momento dell'esecuzione è attivo un unico thread, mentre in un programma *concorrente* vi sono più thread attivi.

Processo:

Un generico insieme di istruzioni in esecuzione, con il proprio spazio di indirizzamento, come di solito avviene nell'ambito dei sistemi operativi.

Programmazione concorrente parallela:

La programmazione che usa *concorrenza fisica*, cioè in cui c'è una reale esecuzione simultanea, o parallela, di due thread (o processi, o anche programmi), come ad esempio nel caso di macchine con più processori. In questo ambito possiamo distinguere il *parallelismo sui dati*, nel quale la principale fonte di parallelismo deriva dall'applicazione di una stessa operazione a dati diversi, ed il *parallelismo sulle operazioni*, nel quale invece si tende ad applicare in parallelo più operazioni diverse ad uno stesso insieme di dati.

Programmazione concorrente multithread:

Consiste nella presenza di più thread o processi, ossia di più contesti d'esecuzione, attivi contemporaneamente nell'ambito di un'applicazione che "gira" su di una macchina che tipicamente, o ha un'architettura tradizionale con un solo processore oppure è un multiprocessore a memoria condivisa.

Programmazione concorrente distribuita:

Con questo termine si intende la realizzazione di programmi concorrenti pensati per essere eseguiti su macchine realizzate da strutture fisiche distribuite. Queste possono essere multicomputer con memoria distribuita ma anche vere e proprie reti di calcolatori con varie architetture e

topologie. I vari nodi del sistema sono distribuiti fisicamente e quindi non si può assumere la presenza di memoria condivisa.

Comunicazione a memoria condivisa:

I due partner, ad esempio due processi diversi, hanno accesso ad uno stesso spazio di memoria nel quale possono scrivere e leggere dei valori.

Comunicazione a scambio di messaggi:

I partner della comunicazione non condividono alcuno spazio di memoria: per comunicare il mittente deve eseguire esplicitamente una operazione di `send` e il destinatario dovrà eseguire un'operazione esplicita di `receive`. Un'opportuna struttura di comunicazione, usualmente detta canale, dovrà essere accessibile sia al mittente che al destinatario per fornire un percorso che colleghi i due partner.

Blackboard:

Un modello di comunicazione intermedio fra la memoria condivisa e lo scambio di messaggi in cui vari processi separati, che possono avere un proprio spazio di memoria privato, condividono una stessa zona di memoria detta blackboard (o store) e usata per la comunicazione.

Comunicazione asincrona:

In questo tipo di comunicazione l'invio e la ricezione di un messaggio avvengono in momenti diversi, come nel caso della posta elettronica. L'operazione di `send` è non bloccante: il mittente la fa e poi procede nella computazione senza aspettare che il destinatario abbia effettuato la corrispondente `receive`. La `receive` invece è bloccante: dopo averla effettuato il processo destinatario legge il primo messaggio in attesa sul canale (nel buffer), se il canale è vuoto, il processo destinatario viene bloccato in attesa di un `send` del mittente.

Comunicazione sincrona:

In questo caso possiamo vedere la `send` e la `receive` come operazioni che permettono ai processi di sincronizzarsi nel momento in cui viene effettuata la comunicazione, come nel caso di una telefonata. Il mittente, dopo

aver effettuato un'operazione di `send`, prima di proseguire nella sua computazione, aspetta che il destinatario effettui il `receive`, inviandogli un messaggio di conferma (nella comunicazione sincrona anche la `send` è bloccante). Il destinatario si comporta come nel caso della comunicazione asincrona.

Il principale vantaggio della comunicazione sincrona è quello di limitare la lunghezza massima della coda dei messaggi associata ad un canale — essa conterrà al più un messaggio per ogni processo mittente. Uno svantaggio è che ad ogni comunicazione asincrona uno dei processi sicuramente verrà bloccato (il primo a fare l'operazione di `send` o `receive`), causando ritardo. Un'altro è la possibilità di creare delle situazioni di deadlock (per esempio se due processi che comunicano tra di loro rimarranno entrambi bloccati se fanno `send` allo stesso tempo).

Mutua esclusione:

Tecnica che permette di escludere che le sezioni del codice che consentono l'accesso a risorse condivise, le cosiddette *sezioni critiche*, siano eseguite contemporaneamente da più processi.

Sincronizzazione su condizione:

Tecnica che permette di sospendere l'esecuzione di un processo fino al verificarsi di un'opportuna condizione, quale ad esempio un dato valore per una variabile.

Attesa attiva:

Nell'attesa attiva il processo che deve attendere il proprio turno (per entrare in una sezione critica oppure che sia verificata una data condizione) esegue un ciclo nel quale valuta continuamente una condizione fino a quando questa non diventi vera. Nell'attesa viene impiegata attivamente la CPU e quindi questa tecnica non ha molto senso se impiegata su una macchina uniprocessore per sincronizzare processi diversi; si tratta invece di una tecnica efficiente se usata su macchine multiprocessore.

Lock:

Tecnica di attesa attiva per implementare la mutua esclusione nella quale prima di accedere ad una sezione critica il processo deve acquisire un lock. Questa procedura consiste in un ciclo nel quale si testa la variabile condivisa `B` fino a quando questa non ha il valore `false`, allora si accede alla sezione e lo si mette a `true`. Quando avrà finito di eseguire la sezione critica, il processo rilascerà il lock, mettendo il valore di `B` a `false`.

Un problema di questa tecnica è che non è garantita la *fairness*, ossia non è garantito che un processo che voglia accedere alla sezione critica prima o poi riesca a farlo (può, in teoria, rimanere in attesa per sempre). Una soluzione per garantire la fairness è quella di introdurre un algoritmo che distribuisce dei *ticket* ai processi, come quelli usati per gestire l'accesso dei clienti ad uno sportello.

Barriere:

Meccanismo di sincronizzazione su condizione tipicamente usato nella programmazione parallela nel quale si introduce nel codice di ogni processo un punto di sincronizzazione, detto *barriera*, nel quale ogni processo attende il verificarsi di una condizione globale (ad esempio che ogni processo abbia finito la sua computazione) prima di procedere.

Sincronizzazione basata sullo scheduler:

Meccanismi nei quali il processo che deve essere posto in attesa rilascia la CPU, perché questa possa essere usata da altri, associando però all'evento che il processo sta attendendo un'informazione che identifichi il processo stesso. Quando l'evento si verificherà lo scheduler potrà sapere che quel particolare processo è pronto per essere eseguito.

Semafori:

Un semaforo può essere visto come un tipo di dato che ha un valore intero superiore o uguale a zero e le operazioni `P` (accesso al semaforo) e `V` (rilascio del semaforo). Quando un processo fa `P(s)`, se `s` vale 0 allora il processo è bloccato fino a quando il valore di `s` non cambia. Se `s` ha un valore strettamente positivo, il processo lo mette a 0 e accede alla sezione critica. Quando ha finito, rilascia il semaforo con `V(s)`.

Siccome il blocco è di solito gestito a livello di nucleo del sistema operativo, il processo in attesa rilascia l'uso della CPU, che può quindi essere usata da altri. Quando il semaforo viene rilasciato, il sistema operativo "risveglia" il processo in attesa.

La fairness dipende dalle ipotesi dello scheduler. Di solito, la coda dei processi bloccati è gestita in modo FIFO.

Monitor:

Un monitor è un tipo di dato astratto che contiene delle variabili permanenti, che rappresentano lo stato dell'oggetto e non sono accessibili fuori di esso, delle procedure per agire su tali variabili dall'esterno e dei comandi di inizializzazione, usati quando il monitor è creato. Questo permette che la programmazione del monitor sia relativamente indipendente da quella dei processi, che hanno solo bisogno di conoscere la segnatura delle procedure del monitor per poter usarlo.

La mutua esclusione è garantita implicitamente dal monitor stesso dato che le sue procedure sono eseguite in mutua esclusione per definizione. Il programmatore deve invece, manipolando, attraverso l'uso delle opportune procedure a disposizione, le variabili condizionali del monitor (che permettono di sospendere un processo nell'attesa di una condizione), programmare esplicitamente la sincronizzazione condizionale.

Remote procedure call:

Tipo di comunicazione sincrona usato per invocare una procedura che è eseguita in remoto, tipicamente da un processo che "gira" su un'altra macchina, anche geograficamente distante. Il mittente, dopo aver fatto la `send`, aspetta un messaggio di risposta contenente il risultato dell'invocazione della procedura e non solo una conferma della ricezione. Nel caso della RPC nel destinatario non esiste un processo attivo che aspetta la chiamata, invece, al momento della chiamata, viene creato nel destinatario un nuovo apposito processo per gestirla, che termina dopo avere restituito il risultato al mittente.

Rendez-vous:

Similmente alla RPC, si invoca una procedura che è eseguita da remoto, ma in questo caso il processo chiamante effettua un rendez-vous con un

processo già attivo nel destinatario. Quando arriva la chiamata, tale processo esegue la procedura richiesta (previo passaggio dei parametri, al solito), trasmette i risultati al chiamante e poi continua nella propria attività.

Modelli di computazione non deterministica:

Quando ci possono essere momenti della computazione in cui, per un dato stato corrente, la prossima istruzione da eseguire non è univocamente determinata (cioè diverse istruzione potrebbero essere scelte e l'esito non può essere noto in anticipo).

Comandi con guardia:

Comandi composti da una serie di condizioni (dette anche guardie), seguita ognuna da un comando. Se la condizione è vera allora il ramo corrispondente può essere scelto e può essere eseguito il comando associato. Se vi sono più condizioni vere, allora uno dei rami associati è scelto in modo non deterministico.

Se i comandi fossero ordinati ed eseguiti in modo deterministico, alcuni comandi di comunicazione (`send` e `receive`) potrebbero bloccare il programma mentre altri comandi sarebbero pronti per essere eseguiti: i comandi con guardia permettono di ovviare a questo problema.