Materiale ausiliario di supporto
per lo studio dell'esame di

Linguaggi di Programmazione

c.d.l. Informatica e TPS

1 - Macchine Astratte

La nozione di macchina astratta

Un calcolatore è una macchina fisica che permette di eseguire degli algoritmi. Una macchina astratta è un 'astrazione del concetto di calcolatore fisico. Per essere eseguiti gli algoritmi devono essere rappresentati mediante le istruzioni di un opportuno linguaggio di programmazione L. Un programma scritto in L è un insieme finito di istruzioni di L. Una macchina astratta M è un insieme di strutture dati e di algoritmi che permettano di memorizzare ed eseguire programmi scritti in L. Una macchina astratta è composta da una memoria e da in interprete. La memoria immagazzina dati e programmi mentre l'interprete esegue le istruzioni contenute nei programmi.

Interprete

L'interprete compie delle operazioni specifiche che dipendono dal particolare linguaggio che deve essere interpretato. Ci sono quattro categorie di operazioni che l'interprete esegue:

1. Operazioni per elaborare i dati primitivi: i numeri e le operazioni aritmetiche
2. Operazioni e strutture dati per il controllo della sequenza di esecuzione delle operazioni: servono per gestire il flusso di esecuzione delle istruzioni presenti in un programma mediante particolari operazioni diverse da quelle che manipolano i dati.
3. Operazioni e strutture dati per il controllo del trasferimento dei dati: controlla i dati che devono essere trasferiti dalla memoria all'interprete e viceversa.
4. Operazioni e strutture dati per la gestione della memoria: riguarda tutte le operazioni relative all'allocazione di memoria per i dati e per i programmi.

Il ciclo di esecuzione dell'interprete è costituito dalle seguenti fasi:

1. acquisizione della prossima istruzione da eseguire dalla memoria;
2. decodifica dell'istruzione;
3. individuare l'operazione richiesta e quali sono gli operandi;
4. sono prelevai gli operandi ed eseguita l'operazione richiesta;
5. memorizza il risultato e passa all'istruzione successiva.

Un linguaggio macchina è il linguaggio compreso dall'interprete della macchina astratta.

Implementazione di un linguaggio

Una macchina astratta corrisponde univocamente ad un linguaggio, il suo linguaggio macchina. Inversamente, dato un linguaggio, vi sono infinite macchine astratte che hanno L come proprio linguaggio macchina. Tali macchine differiscono tra loro nel modo in cui l'interprete è realizzato e nelle strutture dati che utilizzano. Implementare un linguaggio di programmazione significa realizzare una macchina astratta che abbia L come linguaggio macchina.

Realizzare una macchina astratta

Una macchina astratta deve essere realizzata mediante uno dei seguenti tre metodi:

1. Realizzazione in Hardware: questa realizzazione è sempre possibile e abbastanza semplice, si tratta infatti di realizzare mediante dispositivi fisici una macchina fisica tale che il suo linguaggio macchina coincida con L. Questo ha il vantaggio di avere un esecuzione molto veloce ma solo sui linguaggi a basso livello o dedicati. Invece non è possibile realizzare linguaggi di alto livello. Inoltre tale macchina una volta realizzata non può essere più modificata.
2. Simulazione in Software: è una realizzazione delle strutture dati e di algoritmi di M mediante programmi scritti un un altro linguaggio implementato su un'altra macchina. Quindi possiamo realizzare la macchina M con programmi scritti nell'altro linguaggio che interpretano i costrutti scritti in L simulando le funzionalità di M. In questo caso avremmo la

massima flessibilità, potendo cambiare con facilità i programmi che realizzano i costrutti di M. Tuttavia avremmo prestazioni inferiori, in quanto la realizzazione di M passa attraverso un'altra macchina.

3. Emulazione in Firmware: emulare le strutture dati e gli algoritmi di M mediante microprogrammi, la macchina astratta è simulata mediante opportuni programmi che sono poi eseguiti da una macchina fisica, ma tali programmi sono dei microprogrammi invece che programmi di un linguaggio di alto livello. Questi microprogrammi usano un linguaggio di basso livello e risiedono in un'opportuna memoria di sola lettura per poter essere eseguiti dalla macchina fisica ad alta velocità anche se inferiore alla soluzione hardware. Però la flessibilità è inferiore a quella software perché richiede opportuni dispositivi per riscrivere le memorie sulle quali i microprogrammi sono memorizzati.

Implementazione: il caso ideale

Per implementare un linguaggio L, ovvero di cui si vuole realizzare una macchina astratta M, possiamo supporre di avere a disposizione una macchina astratta ospite, che è già stata realizzata e che quindi ci permette di usare direttamente i costrutti del suo linguaggio macchina ospite. L'implementazione avviene sulla macchina ospite mediante una traduzione del linguaggio L nel linguaggio ospite. Ci sono due modalità di implementazione:

1. interpretativa pura: si realizza l'interprete di M mediante un inseme di istruzioni in L, cioè un programma che è un interprete scritto nel linguaggio ospite che interpreta tutte le possibili istruzioni di L. Una volta che tale interprete sia realizzato, per eseguire un programma scritto nel linguaggio L con un certo dato in input,, dovremo semplicemente eseguire, sulla macchina ospite, il programma con il programma scritto nel linguaggio L e i dati come input. Nell'implementazione interpretativa pura di L non vi è una traduzione esplicita dei programmi scritti in L. Vi è solo un procedimento di decodifica.
2. compilativa pura: l'implementazione di L avviene traducendo esplicitamente i programmi scritti in linguaggio sorgente in programmi scritti in linguaggio oggetto. La traduzione è eseguita da un opportuno programma detto compilatore. Per eseguire un programma scritto il L con un dato input dovremmo prima eseguire il compilatore con il programma come input e dopo eseguire il programma appena compilato sulla macchina ospite.

Vantaggi e svantaggi dei due tipi di implementazione:

- L'implementazione interpretativa pura ha come svantaggio principale una scarsa efficienza dato che per eseguire il programma l'interprete deve effettuare al momento dell'esecuzione una decodifica dei costrutti del linguaggio. Inoltre per ogni nuova occorrenza di uno stesso comando, l'interprete deve effettuare uno nuova decodifica. D'altro canto offre dei vantaggi in termini di flessibilità, infatti, interpretare al momento dell'esecuzione permette di interagire in modo diretto con l'esecuzione del programma. Inoltre lo sviluppo di un interprete è più semplice di quello di un compilatore. Infine l'implementazione interpretativa permette di usare una quantità di memoria molto ridotta , dato che il programma è memorizzato solo nella sua versione sorgente e non viene prodotto nuovo codice.
- Nell'implementazione compilativa pura la traduzione del programma sorgente in un programma oggetto avviene separatamente dall'esecuzione di quest'ultimo. Se trascuriamo il tempo necessario alla compilazione, l'esecuzione risulterà più efficiente. Inoltre la decodifica di una istruzione del linguaggio viene fatta dal compilatore una sola volta. Uno degli svantaggi maggiori risiede nella perdita di informazioni riguardo alla struttura del programma sorgente, perdita che rende più difficile l'interazione con il programma a tempo di esecuzione, infatti se a run-time si verificasse un errore potrebbe essere difficile determinare qual'è il comando che lo ha causato. In questo caso è più difficile realizzare strumenti di debugging, quindi si ha una flessibilità minore.

Implementazione: il caso reale e la macchina intermedia

Nelle implementazioni dei linguaggi reali sono quasi sempre presenti entrabe le componenti. Supponiamo di avere un linguaggio L che deve essere implementato e di disporre di una macchina ospite già realizzata. Fra la macchina che vogliamo realizzare e la macchina ospite esiste un ulteriore livello formato dal linguaggio intermedio e dalla relativa macchina intermedia. Abbiamo sia un compilatore che traduce da L al linguaggio intermedio, sia un interprete che gira sulla macchina ospite e che simula la macchina intermedia. Un generico programma, creato con il linguaggio L, per essere eseguito è prima tradotto dal compilatore in un programma del linguaggio intermedio. Quindi questo programma è eseguito dall'interprete: nel caso in cui il linguaggio intermedio e il linguaggio macchina ospite non siano molto distanti, per simulare la macchina intermedia può bastare l'interprete della macchina ospite, esteso da opportuni programmi detti supporto a run-time.

Gerarchia di macchina astratte

Si usano spesso gerarchie di macchine astratte in cui ogni macchina usa le funzionalità del livello sottostante e offre nuove funzionalità al livello soprastante. La generica macchina M è implementata sfruttando le funzionalità, cioè il linguaggio, della macchina sottostante; contemporaneamente M fornisce il proprio linguaggio alla macchina sovrastante. Spesso tale gerarchia ha lo scopo di mascherare i livelli inferiori: M può solo accedere alle risorse della macchina subito inferiore, e non a tutte le altre. Una tale strutturazione è utile per dominare la complessità del sistema e permette una certa indipendenza fra i vari livelli.

2 – Descrivere un linguaggio di programmazione

Livelli di descrizione

La descrizione di un linguaggio può avvenire in tre livelli:

- Sintassi (o grammatica): “Quali frasi sono corrette?”; una volta definito l'alfabeto, sono individuate le sequenza corrette di simboli che costituiscono le parole (token) del linguaggio. Dopo la sintassi descrive quali sequenze di parole costituiscano rasi legali;
- Semantica: “Cosa significa una frase corretta?”; attribuisce un significato ad ogni frase corretta;
- Pragmatica: “Come usare una frase corretta?”; frasi con lo stesso significato possono essere usate in modo diverso da utenti diversi.
- Implementazione: “Come eseguire una frase corretta, un modo da rispettarne la semantica?”; il processo le frasi “operative” del linguaggio realizzano lo stato di cose di cui parliamo.

Grammatica e sintassi

Fissato un insieme finito A, chiamato alfabeto, indichiamo con A* l'insieme di tutte le stringhe finite su A (* è detto stella di Kleene). Un linguaggio formale su A non è altro che un sottoinsieme di A*. Una grammatica libera da contesto è una quadrupla (NT, T, R, S) dove:

- NT è un insieme finito di simboli non terminali;
- T è un insieme finito di simboli terminali;
- R è un insieme finito di produzioni (o regole);
- S è il simbolo di partenza appartenente ai NT.

La forma normale di Backus e Naur (BNF) è una notazione per descrivere la grammatica, dove si usano astrazioni per rappresentare classi di strutture sintattiche. La grammatica è formata da un insieme di regole: una regola ha una parte sinistra (LHS) e una parte destra (RHS), che consistono di simboli terminali e non terminali. Una regola può avere più di una parte destra separata dal simbolo “|”.

Una derivazione consiste nella ripetuta applicazione di regole di una grammatica. Ogni stringa di simboli presenti nella derivazione si chiama forma di frase; una forma di frase fatta di soli terminali di chiama semplicemente frase. Una singola regola di derivazione mette in relazione di derivazione diretta due forme di frasi: P => bPb. La derivazione, quindi, non è altro che la chiusura transitiva della relazione => e si indica con =>*: P => aPa => abPba => abba, quindi P =>* abba.

Un albero è una struttura costituita da un insieme di nodi interni, da nodi foglie e dal nodo radice. Un grafo connesso è formato dai nodi e dagli archi che li collegano. Un albero di derivazione è una rappresentazione gerarchica di una derivazione, dove:

- Nodo radice: S
- Nodi interni: NT
- Nodi foglia: T

Una derivazione sinistra (destra) è una derivazione dove, in ogni forma di frase intermedia, si espande il non-terminale più a sinistra (destra), tuttavia ci sono derivazioni che non sono né sinistre né destre.

Una grammatica è ambigua se e solo se esiste almeno una forma di frase prodotta attraverso due o più alberi di derivazione distinti. Tuttavia una stringa può avere più derivazioni distinte senza che la grammatica sia ambigua, però non può avere più di due derivazioni canoniche dello stesso tipo.

La forma normale di Backus e Naur estesa (EBNF) introduce delle nuove specifiche quali:

- le parti opzionali sono poste tra parentesi quadre;
- le parti destre alternative sono poste tra parentesi tonde e separate con barre verticali;
- ripetizioni di zero o più elementi sono poste tra parentesi graffe

Vincoli sintattici contestuali

La correttezza sintattica di una frase dipende dal contesto nel quale si trova e non si possono esprimere con una grammatica libera. Degli esempi sono il dichiarare un identificatore prima dell'uso, o nel caso di un assegnamento, il tipo di un'espressione deve essere compatibile con quello della variabile a cui si assegna. La necessità di gestire i vincoli della sintassi contestuale porta alla nozione di semantica statica: ossia verificabile sul testo del programma sorgente.

Compilatori

L'analisi lessicale (scanning o lexing) è implementata dal sottoprogramma scanner o lexer. Lo scopo è quello di leggere sequenzialmente i simboli di ingresso di cui è composto il programma e di raggruppare tali simboli in unità logicamente significative, che chiamiamo token, che sono l'analogo delle parole del dizionario di un linguaggio naturale. Lo strumento tecnico che si usa per l'analisi lessicale è una classe di grammatiche generative: le grammatiche regolari (o lineari).

L'analizzatore sintattico (o parser), costruita la lista di token, cerca di costruire un albero di derivazione per tale lista. Ogni foglia corrisponde ad un token, e tali foglie lette da sinistra verso destra, devono costituire una frase corretta nel linguaggio. Se la stringa in ingresso non è una stringa corretta secondo la grammatica del linguaggio, il parse non sarà in grado di costruire l'albero e riporterà un errore, abortendo la compilazione.

L'analisi semantica sottopone l'albero di derivazione ai relativi controlli contestuali. Man mano che vengono fatti, l'albero viene aumentato con la relativa informazione. Per gli identificatori le informazioni sono anche registrate nella tabella dei simboli.

Le fasi successive si dividono in:

1. generazione della forma intermedia:
    1. visita dell'albero per generare il codice;
    2. codice in forma intermedia;
2. Ottimizzazione del codice:
    1. rimozione codice inutile;
    2. espansioni delle chiamate di sottoprogrammi;
    3. fattorizzazione delle espressioni;
    4. ottimizzazione dei cicli;
3. Generazione del codice:
    1. generazione del codice;
    2. ulteriore fase di ottimizzazione.

Semantica

La semantica è nata dall'esigenza di mediare tra due istanze contrapposte: la ricerca dell'esattezza e della flessibilità. I metodi formali per la semantica si dividono in due categorie: la semantiche denotazionali e quelle operazionali. Quella denotazionale fa uso di funzioni e tecniche logico-matematiche. Quella operazionale fa uso di un formalismo a basso livello usando automi, assiomi, stati e transizioni.

Lo stato è un modello di memoria che mantiene in memoria i valori delle variabili. Nello stato corrente una certa variabile ha un certo valore. Le operazioni possibili sono l'assegnazione di un nuovo valore ad una variabile e il ritrovamento del valore di una variabile.

La transizione esprime un passo di trasformazione sullo stato del programma, provocato dall'esecuzione di un programma. Cioè il passaggio da uno stato ad un altro. Esistono anche transizioni composte e transizioni condizionali.

Le computazioni sono una sequenza di transizioni concatenate non ulteriormente estendibile in cui ogni transizione è permessa da qualche regola. Ci sono due tipi di computazioni: quelle determinanti o finite; e quelle divergenti o infinite (loop).

3 – Fondamenti e Calcolabilità

Il problema della fermata

Dobbiamo capire se esiste un analizzatore statico capace di scoprire se un programma, su un certo dato di ingresso, possa andare un ciclo. Un analizzatore statico è un programma usato come sottoprogramma all'interno di un compilatore. Si dimostra come non esiste alcuna procedura di decisione capace di verificare se un altro programma termina su un input. Questo risultato va sotto il nome di indecidibilità del problema della fermata.

Espressività dei linguaggi di programmazione

Alcune funzioni sono calcolabili, altre non lo sono. Alcuni programmi possono registrare un errore se il risultato del programma è indefinito, però non si possono riportare errori se il programma non termina. Non esiste alcun programma che funzioni per argomenti arbitrari; o che termini sempre; o che discrimini gli argomenti che sono soluzione del problema da quelli che non lo sono.

Esistono funzioni che non sono calcolabili mediante una Macchina di Turing. Secondo la tesi di Church, tutte le funzioni calcolabili lo sono tramite un sistema formale equivalente alla Macchina di Turing o alle funzioni ricorsive.

4 – I nomi e l'ambiente

Un nome non è altro che una sequenza di caratteri usata per rappresentare qualche altra cosa e permette di astrarre sia aspetti relativi ai dati, sia aspetti relativi al controllo. La gestione corretta dei nomi richiede sia regole semantiche precise che meccanismi implementativi adeguati.

Nomi e oggetti denotabili

Quando in un programma dichiariamo una nuova variabile oppure definiamo una nuova funzione introduciamo dei nuovi nomi per rappresentare un oggetto. Un nome non è altro che una sequenza di caratteri usata per rappresentare, o denotare, un altro oggetto. Nella maggior parte dei linguaggi i nomi sono costituiti da identificatori, ossia token alfanumerici, tuttavia possono essere nomi anche altri simboli: ad esempio + e * sono nomi che denotano in genere operazioni primitive. Uno stesso oggetto può avere più nomi (si parla di aliasing), mentre uno stesso nome può denotare oggetti diversi in momenti diversi. Nella pratica programmativa, quando si usa un nome quasi sempre si intende riferirsi all'oggetto da esso denotato. Se usiamo il comando di assegnamento il valore assegnato sarà memorizzato nella locazione che è stata riservata per la variabile e l'uso del nome evita di doversi preoccupare di quale sia questa locazione. La corrispondenza fra nome e locazione di memoria dovrà essere garantita dall'implementazione e chiameremo ambiente quella parte dell'implementazione che è responsabile delle associazioni tra i nomi e gli oggetti che questi denotano. I nomi sono fondamentali anche per realizzare una forma di astrazione sul controllo: una procedura non è altro che un nome associato ad un inseme di comandi, insieme a determinate regole di visibilità che rendono disponibile al programmatore la sola interfaccia.
Gli oggetti ai quali può essere dato un nome si dicono oggetti denotabili e sono di due tipi:

- oggetti i cui nomi sono definiti dall'utente
- oggetti i cui nomi sono definiti dal linguaggio di programmazione

Il legame o binding fra un nome e l'oggetto da esso denotato può avvenire in momenti diversi: alcuni nomi sono associati a degli oggetti al momento della progettazione di un linguaggio, mentre altre associazioni sono introdotte solo al momento dell'esecuzione di un programma. Nel processo che va dalla definizione di un linguaggio all'esecuzione del programma si distinguono varie fasi:

- Progettazione del linguaggio
- Scrittura del programma
- Compilazione del programma
- Collegamento del programma
- Caricamento del programma
- Esecuzione del programma

Nella pratica si distinguono due fasi principali usando i termini “statico” e “dinamico”: con statico ci si riferisce a tutto quello che avviene prima dell'esecuzione, mentre con dinamico si indica tutto quello che avviene al momento dell'esecuzione. La gestione statica della memoria è quella operata dal compilatore, mentre quella dinamica è quella realizzata da opportune operazioni eseguite dalla macchina astratta a tempo d'esecuzione.

Ambiente e blocchi

L'insieme delle associazioni fra nome e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell'esecuzione è detto ambiente. L'ambiente è dunque quella componente della macchina astratta che, per ogni nome introdotto dal programmatore e in ogni punto del programma, permette di determinare quale sia l'associazione corretta. La presenza dell'ambiente costituisce una delle principali caratteristiche dei linguaggi di altro livello.

Una dichiarazione è un costrutto ce permette di introdurre un'associazione nell'ambiente. I linguaggi di alto livello hanno spesso dichiarazioni esplicite, alcuni linguaggi permettono dichiarazioni implicite, che introducono un'associazione in ambiente per un nome al momento del primo uso di tale nome. Il tipo dell'oggetto denotato viene dedotto dal contesto nel quale il nome è usato la prima volta.

Un blocco è una regione testuale del programma, identificata da un segnale di inizio ed uno di fine, che può contenere dichiarazioni locali a quella regione, cioè che compaiono nella regione. Si possono distinguere due casi:

- blocco associato ad una procedura:è il blocco associato alla dichiarazione di una procedura e che testualmente corrisponde al corpo della procedure stessa, esteso con le dichiarazioni relative ai parametri formali;
- blocco in-line (o anonimo): è il blocco che non corrisponde ad una dichiarazione di procedura, e che può pertanto comparire in genere un una qualsiasi posizione dove sia richiesto un comando.

L'ambiente cambia durante l'esecuzione di un programma, all'entrata e all'uscita di un blocco. L'ambiente di un blocco è costituito in primo luogo dalle associazioni relative ai nomi dichiarati localmente al blocco stesso. I blocchi possono essere annidati , ossia la definizione di un blocco può essere interamente inclusa in quella di un altro; non sono invece mai permesse sovrapposizioni di blocchi.

Una dichiarazione locale ad un blocco è visibile in un altro blocco, se l'associazione creata per effetto di quella dichiarazione è presente nell'ambiente del secondo blocco. Sono chiamate regole di visibilità quei meccanismi di un linguaggio che regolano come e quando una dichiarazione è visibile. 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 nasconde la precedente: l'associazione per il nome dichiarato nel blocco esterno sarà disattivata per tutto il blocco interno e sarà riattivata all'uscita da tale blocco. L'ambiente associato ad un blocco è formato dalle tre parti:

- ambiente locale: associazioni per nome dichiarati localmente al blocco.
- ambiente non locale: associazioni relative ai nomi che sono visibili all'interno di un blocco ma che non sono stati dichiarati localmente.
- ambiente globale: associazioni create all'inizio dell'esecuzione del programma.

Nel momento in cui, durante l'esecuzione di un programma, si entra in un nuovo blocco vengono create le associazioni fra i nomi dichiarati localmente al blocco e disattivate le associazioni per quei nomi già esistenti all'esterno del blocco che sia ridefiniti al suo interno. Nel caso dell'uscita dal nuovo blocco, invece, si verifica l'opposto le associazioni precedentemente create vengono distrutte e quelle disattivate vengono riattivate. Possiamo identificare delle operazioni sui nome e sull'ambiente, quali riferimento di oggetto denotato mediante il suo nome, creazione, disattivazione, riattivazione e distruzione e di un'associazione fra il nome e l'oggetto denotato. Per quanto riguarda gli oggetti denotabili sono possibili le operazioni di creazione, accesso, modifica e distruzione.

In molti casi le operazioni di creazione fra nome e oggetto avvengono allo stesso tempo, ma non vale sempre. In generale non è detto che il tempo di vita di un oggetto denotabile, ossia il tempo che intercorre fra la sua creazione e la sua distruzione, coincida con il tempo di vita dell'associazione fra nome e oggetto. Infatti un oggetto può avere un tempo di vita maggiore di quello della sua associazione con un nome. Può accadere anche che il tempo di vita di un 'associazione sia superiore a quello dell'oggetto, cioè può avvenire che un nome permetta di accedere ad un oggetto che non esiste più. Questo può accadere, per esempio, se si passa per riferimento un oggetto creato e quindi si dealloca la memoria per tale oggetto prima che la procedura termini.

Regole di scope

In un linguaggio con scope statico, un qualsiasi ambiente esistente dipende solo dalla strutture sintattica del programma. L a regola dello scope statico è definita dai seguenti tre punti:

1. Le dichiarazioni locali di un blocco definiscono l'ambiente locale di quel blocco e non quelle presenti nei suoi eventuali blocchi annidati.
2. Se si usa un nome all'interno di un blocco, l'associazione valida per tale nome è quella presente nell'ambiente locale del blocco, se esiste. Se non esiste, si considerano le associazione esistenti nell'ambiente locale del blocco immediatamente esterno che contiene il blocco.
3. Un blocco può avere un nome, nel qual caso tale nome fa parte dell'ambiente locale del blocco immediatamente esterno che contiene il blocco a cui abbiamo dato un nome.

Lo scope statico ha due importanti conseguenze positive; innanzitutto in programmatore ha una migliore comprensione del programma osservandone la struttura testuale, inoltre il compilatore può collegare ogni occorrenza di un nome alla sua dichiarazione corretta, questo fa si che siano possibili a tempo di compilazione un maggior numero di controlli di correttezza. D'altro canto risulta meno efficiente in fase di esecuzione.

Lo scope dinamico è stato introdotto principalmente per semplificare la gestione a run-time dell'ambiente. Molti linguaggi lo usano e determinano le associazioni fra nomi e oggetti denotati seguendo a ritroso l'esecuzione del programma. Tali linguaggi per risolvere i nomi non locali usano la sola struttura a pila impiegata per la gestione a run-time dei blocchi. Secondo la regola dello scope dinamico, l'associazione valida per un nome X, in un qualsiasi punto P di un programma, è la più recente, in senso temporale, associazione creata per X che sia ancora attiva quando il flusso di esecuzione arriva a P. Questo metodo ha il vantaggio di avere un'ottima flessibilità, ma ha una minore leggibilità e una complicata gestione a run-time.

Esistono alcuni problemi di scope; le differenza maggiori fra le regole di scope statico sono relative a dove possono essere introdotte le dichiarazioni e a quale sia l'esatta visibilità delle variabili locali. Nel caso del Pascal le dichiarazioni possono comparire solo all'inizio del blocco e prima di essere usati, inoltro lo scope di un nome si estende dall'inizio alla fine del blocco nel quale appare la dichiarazione. Nel caso del C e di ADA lo scope della dichiarazione viene limitato alla porzione di blocco compresa fra il punto in cui la dichiarazione compare e la fine del blocco stesso; anche in questi linguaggi i nomi devono essere dichiarati prima di essere usati. Però la dichiarazione prima dell'uso in alcuni casi è particolarmente gravosa: impedisce infatti la definizione di tipi ricorsivi. In JAVA è consentito che una dichiarazione possa apparire in un punto qualsiasi di un blocco. Se la dichiarazione corrisponde ad una variabile, lo scope del nome dichiarato si estende dal punto della dichiarazione fino alla fine del blocco; se la dichiarazione invece si riferisce al membro di una classe, essa è visibile in tutta la classe nella quale compare.

5 – Gestione della Memoria

Tecniche di gestione della memoria

La gestione della memoria costituisce una delle funzionalità dell'interprete, quali la gestione dell'allocazione di memoria per i programmi e i dati: come disporli in memoria, quanto tempo ci devono rimanere e quali strutture ausiliarie siano necessarie. Nel caso di una macchina stratta di baso livello tipo la macchina hardware, la gestione della memoria è molto semplice e può essere interamente statica. Nel caso di un linguaggio di alto livello l'allocazione statica della memoria non è più sufficiente, per cui dobbiamo usare una gestione dinamica della memoria. Questa gestione può essere realizzata con una pila, ma ci sono casi dove la pila non è sufficiente; in questo caso si usa una struttura detta heap.

Gestione statica della memoria

La memoria gestita staticamente è quella allocata dal compilatore prima dell'esecuzione. Gli oggetti per i quali la memoria è allocata staticamente risiedono in una zona fissa di memoria per tutta la durata dell'esecuzione. Tipici elementi per i quali è possibile allocare staticamente la memoria sono le variabili locali, le istruzioni del codice oggetto prodotte dal compilatore, le costanti e le varie tabelle prodotte dal compilatore necessarie per il supporto a run-time del linguaggio. Nel caso in cui il linguaggio non sopporti la ricorsione, è possibile gestire staticamente anche a memoria per le rimanenti componenti del linguaggio: sostanzialmente si tratta di associare, staticamente, ad ogni procedura una zona di memoria nella quale memorizzare le informazioni della procedura stessa.

Gestione dinamica mediante pila

La maggior parte dei linguaggi di programmazione moderni permette una strutturazione a blocchi dei programmi. I blocchi vengono aperti e chiusi usando la politica LIFO: quando si entra in un blocco A e poi in un blocco B, prima di uscire da A si deve uscire da B. Lo spazio di memoria allocato sulla pila è detto record di attivazione (RdA). Il record di attivazione è associato ad una specifica attivazione di procedura e non ad una sua dichiarazione. La pila sulla quale sono memorizzati i record di attivazione è detta pila a run-time.

La struttura di un record di attivazione per un blocco in-line è composta da vari settori che contengono le seguenti informazioni:

- Risultati intermedi: nel caso in cui si debbano effettuare dei calcoli può essere necessario memorizzare alcuni risultati intermedi.
- Variabili locali: sono dichiarate all'interno di un blocco, devono avere a disposizione uno spazio di memoria la cui dimensione dipenderà dal numero e dal tipo delle variabili; in alcuno casi vi possono essere dichiarazioni che dipendono da valori noti solo al momento dell'esecuzione, come ad esempio gli array dinamici. In questi casi il record di attivazione prevede anche una parte di dimensione variabile che sarà definita al momento dell'esecuzione.
- Puntatore di catena dinamica: questo campo serve per memorizzare il puntatore al precedente record di attivazione sulla pila, essendo gli Rda con dimensioni diverse; l'insieme dei collegamenti realizzati da questi puntatori è detto catena dinamica.

La struttura di un record di attivazione per le procedure e le funzioni è formato dai seguenti campi:

- Risultati intermedi, variabili locali e puntatore di catena dinamica.
- Puntatore di catena statica: serve per gestire le informazioni necessarie a realizzare le regole di scope statico.
- Indirizzo di ritorno: contiene l'indirizzo della prima istruzione da eseguire dopo che la chiamata alla procedura ha terminato l'esecuzione.
- Indirizzo del risultato: serve solo per le funzioni e restituisce l'indirizzo della memoria dove viene depositato il risultato.
- Parametri: contiene i valori dei parametri attuali usati nella chiamata della procedura.

Per gestire la pila viene usato un puntatore esterno alla pila che indica l'ultimo RdA inserivo nella pila stessa, chiamato puntatore ad record di attivazione. Un altro puntatore, chiamato puntatore alla pila, indica la prima posizione di memoria libera nella pila. I record di attivazione vengono inseriti e rimossi dalla pila a tempo di esecuzione: quando si entra in un blocco, o si chiama una procedura, il relativo RdA verrà inserito nella pila, per poi essere eliminato quando si esce dal blocco o quando termina l'esecuzione della procedura. La gestione a run-time della pila di sistema è realizzata da alcuni frammenti di codice che il compilatore inserisce prima e dopo la chiamata di una procedura o prima dell'inizio e dopo la fine di un blocco. La gestione della pila è fatta sia dal programma o procedura chiamante che dal programma o procedura chiamato. Per questo scopo nel chiamante viene aggiunta una parte di codice detta sequenza di chiamata che è eseguita in parte

immediatamente prima della chiamata di procedura, e in parte immediatamente dopo la terminazione della procedura. Nel chiamato invece viene aggiunto un prologo, da eseguirsi subito dopo la chiamata, ed un epilogo, da eseguirsi al termine dell'esecuzione della procedura. Al momento della chiamata di procedura la sequenza di chiamata ed il prologo si devo occupare delle seguenti attività:

- Modifica del valore del contatore programma
- Allocazione dello spazio sulla pila
- Modifica del puntatore RdA
- Passaggio dei parametri
- Salvataggio dei registri
- Esecuzione del codice per l'inizializzazione

Al momento del ritorno del controllo al programma chiamante, quando la procedura chiamata termina la sua esecuzione, l'epilogo e la sequenza di chiamata devono invece gestire le seguenti operazioni:

- Ripristino del valore del contatore programma
- Restituzione dei valori
- Ripristino dei registri
- Esecuzione del codice per la finalizzazione
- Deallocazione dello spazio sulla pila

Gestione dinamica mediante heap

Nel caso in cui il linguaggio includa comandi espliciti di allocazione della memoria, la sola gestione mediante pila non è sufficiente, ma si usa una particolare zona di memoria, detta heap. I metodi gestione dello heap si dividono in due categorie principali, a seconda che i blocchi di memoria siano considerati di dimensione fissa oppure di dimensione variabile.

Nella gestione dei blocchi di dimensione fissa lo heap è diviso in un certo numero di elementi di dimensione fissa abbastanza limitata, collegati in una struttura a lista, detta lista libera. Quando in esecuzione, un'operazione richiede l'allocazione di un blocco, il primo elemento della lista libera viene rimosso dalla lista, il puntatore a tale elemento è restituito all'operazione che aveva richiesto la memoria ed il puntatore alla lista libera punterà all'elemento successivo. Quando il blocco viene deallocato, viene collegato nuovamente alla testa della lista libera.

La gestione con blocchi di dimensione variabile usa tecniche con lo scopo di migliorare l'occupazione di memoria e la velocità di esecuzione. Riguardo all'occupazione di memoria si verificano i fenomeni di frammentazione. Quella interna si verifica quando si alloca un blocco di dimensione maggiore di quella richiesta da programma: la porzione di memoria non utilizzata andrà sprecata finché il blocco non sarà deallocato. La frammentazione esterna, si verifica quando i blocchi sono di dimensione piccola, per cui anche se la somma totale della memoria libera è sufficiente, non si riesce effettivamente ad usare la memoria libera. Le tecniche di allocazione di memoria tendono a ricompattare la memoria libera unendo blocchi liberi contigui; per ottenere questo scopo possono essere richieste operazioni aggiuntive che però riducono l'efficienza.

La prima tecnica considera un'unica lista libera, costituita inizialmente da un unico blocco di memoria, contenente l'intero heap. Quando viene richiesta l'allocazione di un blocco di n parole di memoria, le prime n parole sono allocate e il puntatore avanza di n; mentre i blocchi deallocati vengono collegati ad una lista libera. Quando si raggiunge la fine dello spazio di memoria dedicato allo heap si utilizza lo spazio di memoria deallocato, e questo può essere fatto in due modi:

- Utilizzo diretto della lista libera: si mantiene una lista libera di blocchi di dimensione variabile. Quando si richiede l'allocazione per un blocco di n parole viene cercato nella lista

un blocco di dimensioni maggiori o uguali. La porzione non usata se è superiore ad una soglia prefissata va a costituire un nuovo blocco libero. La ricerca del blocco di dimensioni adeguate è effettuata o con il metodo first-fit o con i metodo best-fit. Quando un blocco viene liberato, si controlla se i blocchi adiacenti sono liberi e in questo caso si compattano.

- Compattazione della memoria libera: quando si raggiunge la fine dello spazio inizialmente dedicato allo heap si spostano tutti i blocchi ancora attivi ad un'estremità dello heap.

La seconda tecnica usa più liste libere per blocchi di dimensione diversa. Quando viene richiesto un blocco di dimensione n viene selezionata la lista che contiene blocchi di dimensione maggiore o uguale a n. Le dimensioni dei blocchi possono essere statiche o dinamiche, e nel caso dinamico sono usati due metodi: Il buddy system ed gli heap di Fibonacci. Nel primo le dimensioni dei blocchi delle varie liste sono potenze di due; nella seconda usa i numeri di Fibonacci invece che le potenze di due, con questo metodo i ha una minore frammentazione interna.

Implementazione delle regole di scope

Se si usa la regola dello scope statico, l'ordine con cui esaminare i record di attivazione per risolvere un riferimento non locale non è quello definito dalla posizione degli stessi nella pila. Per gestire a run-time la regola di scope statico, il record di attivazione del generico blocco B è collegato dal puntatore di catena statica al record del blocco immediatamente esterno a B. Inoltre se B è attivo, ossia il suo RdA è sulla pila, allora anche i blocchi esterni a B che lo contengono devono essere attivi. Oltre alla catena dinamica, costituita dai vari record presenti sulla pila di sistema, esiste anche una catena statica,, costituita dai vari puntatori di catena statica e usata per rappresentare la struttura statica di annidamento dei blocchi nel programma. Al momento in cui si entra in un nuovo blocco il chiamante calcola il puntatore di catena statica del chiamato e quindi passa tale puntatore al chiamato. Tale calcolo è abbastanza semplice e può essere facilmente compreso dividendo due casi:

- Il chiamato si trova all'esterno del chiamante: perché il chiamato sia visibile si deve trovare in un blocco esterno, che includa il blocco del chiamante, dunque il RdA del chiamato si deve trovare già sulla pila. Se il chiamante si trova a livello di annidamento n ed il chiamato si trova a livello m, allora tra loro ci sono k = n – m livelli di annidamento.
- Il chiamato si trova all'interno del chiamante: le regole di visibilità assicurano che il chiamato è dichiarato immediatamente nel blocco in cui avviene la chiamata. Il chiamante può semplicemente passare al chiamato, come puntatore di catena statica, il puntatore al proprio record di attivazione. Una volta che ha ricevuto il puntatore lo memorizza in un'apposita area del proprio record di attivazione. Il compilatore tiene traccia del livello di annidamento usando la tabella dei simboli, nella quale vengono identificati e numerati i vari scope in base al livello di annidamento. In base a tale numerazione possiamo calcolare, a tempo di compilazione, la distanza tra lo scope della chiamata e quello della dichiarazione, necessaria a run-time per gestire la catena statica, oltre a risolvere a run-time i riferimenti non locali. Il compilatore non può risolvere completamente in modo statico un riferimento ad un nome non locale ed occorre sempre seguire, a run-time, i link di catena statica.

La tecnica del display permette di ridurre il numero degli accessi ad una costante (due). Questa tecnica usa un vettore, detto display, contenente tanti elementi quanti sono i livello di annidamento dei blocchi, dove l'elemento k-esimo del vettore contiene il puntatore al RdA di livello di annidamento k attivo. Quando ci si riferisce ad un oggetto non locale, dichiarato in un blocco esterno di livello k, il RdA contenente tale oggetto può essere reperito accedendo alla posizione k del vettore. La gestione del display è molto semplice, anche se più costosa di quella della catena statica in quanto, quando si entra o esce da un ambiente, oltre ad aggiornare il puntatore nel vettore si deve salvare il vecchio valore. Più precisamente, quando viene chiamata una procedura di livello k, la posizione k del display dovrà essere aggiornata con i valore del puntatore RdA del chiamato, in quanto questo è diventato il nuovo blocco attivo di livello k. Prima di questo aggiornamento, però, deve essere salvato il precedente contenuto della posizione k del display.

L'implementazione della regola di scope dinamico è molto più semplice di quella dello scope statico. Infatti gli ambienti non locali si considerano nell'ordine con cui sono arrivati a run-time, per risolvere un riferimento basterà percorrere a ritroso la pila. Le varie associazioni fra nomi e oggetti denotati possono essere memorizzate direttamente nei record di attivazione, ma anche separatamente in una lista di associazioni della A-list, e gestita come una pila: quando l'esecuzione di un programma entra o esce da un nuovo ambiente le associazioni locali vengono inserite o rimosse dalla A-list. L'informazione relativa all'oggetto denotato conterrà la locazione di memoria dove l'oggetto è memorizzato, il suo tipo ed eventuali altre informazioni utili ad effettuare a run-time controlli semantici. Sono presenti anche degli inconvenienti come l'impossibilità di determinare staticamente qual'è il blocco da usare per risolvere un riferimento non locale e non possiamo sapere in quale posizione del RdA andare a cercare l'associazione. L'unica possibilità è di memorizzare il nome e fare una ricerca a tempo di esecuzione. Il secondo inconveniente è è l'inefficienza di questa ricerca a run-time: può essere frequente scandire quasi tutta la lista.

Per limitare questi due inconvenienti, al prezzo di una maggiore inefficienza, si usa un'altra implementazione dello scope dinamico, basata sulla Tabella Centrale dell'Ambiente o CTR (Central Referencing environment Table). Secondo questa tecnica tutti blocchi del programma, fanno riferimento ad un unica tabella centrale. In tale tabella sono presenti tutti i nomi usati nel programma e per ogni nome è associato un flag, che indica se la sua l'associazione è attiva o meno, ed un valore che costituisce un puntatore alle informazioni sull'oggetto associato al nome. Se tutti gli identificatori usati sono noti a tempo di compilazione, ogni nome può avere una posizione fissa nella tabella, e a run-time l'accesso alla tabella può avvenire in tempo costante: si somma all'indirizzo di memoria dell'inizio della tabella un offset relativo alla posizione del nome. Se invece non tutti i nomi sono noti la ricerca può essere fatta a run-time usando tecniche di hashing.
Quando da un blocco A si entra in un blocco B, la CTR deve essere modificata per descrivere il nuovo ambiente locale di A, ma le associazioni deattivate dovranno essere salvate per poter poi essere ripristinate quando si uscirà dal blocco B. La pila è la struttura dati adatta per memorizzare queste associazioni. Le associazioni relative ad un blocco non sono necessariamente memorizzate in posizioni contigue nella CTR. Per effettuare le operazioni all'entrata e all'uscita dei blocchi si dovranno considerare quindi i singoli elementi della tabella. Questo può essere fatto associando ad ogni entrata della tabella una specifica pila che contenga nella prima posizione l'associazione valida e nelle posizioni successive le associazioni disattivate per quel nome. In alternativa si può usare un'unica pila “nascosta”, separata dalla CTR, per memorizzare per tutti i nomi solo le associazioni deattivate. In questo caso la tabella contiene per ogni nome un flag che indica se l'associazione è attiva o meno, e il riferimento all'oggetto. Usando la CTR non serve quindi alcuna ricerca run-time.

6 – Strutturare il Controllo

Le espressioni
Un'espressione è un'entità sintattica la cui valutazione produce un valore oppure non termina, nel qual caso l'espressione è indefinita. La caratteristica essenziale di un'espressione è che la sua valutazione produce un valore. Anche in un caso semplice per poter ottenere il risultato corretto si deve fare un'assunzione implicita sulla precedenza fra gli operatori. In generale un'espressione è composta da un'entità singola oppure da un operatore applicato ad un certo numero di argomenti che sono a loro volta espressioni. La sintassi delle espressioni è espressa in modo preciso da una grammatica libera ed è rappresentata da un albero di derivazione. Per poter usare le espressioni nel testo le varie notazioni si differenziano a seconda di come si rappresenta l'applicazione di un operatore ai suoi operandi.

Secondo la notazione infissa, il simbolo di un operatore binario è posto fra i due operandi. Per evitare delle ambiguità si usano le parentesi.

Nella notazione prefissa o notazione polacca, il simbolo che rappresenta l'operatore precede i simboli che rappresentano gli operandi, nella scrittura da sinistra a destra. Usando questo tipo di notazione non servono parentesi e regole di precedenza fra gli operatori purché vi sia noto il numero di operandi di ogni operatore. La maggior parte dei linguaggi usa la notazione prefissa per gli operatori unari e per le funzioni definite dall'utente.

La notazione postfissa o polacca inversa, è analoga alla precedente con la differenza che il simbolo dell'operatore segue quello degli operandi. Questa notazione è impiegata nel codice usato da alcuni compilatori ed è usata anche da alcuni linguaggi di programmazione. Un vantaggio della notazione polacca è che può essere usata per rappresentare in modo uniforme operatori con un numero qualsiasi di operandi. Un secondo vantaggio è la possibilità di evitare le parentesi.

Usando la notazione infissa si paga la facilità e la naturalezza d'uso con una maggiore complicazione nei meccanismi di valutazione delle espressioni. Innanzitutto se non si usano le parentesi occorre chiarire le precedenza fra i vari operatori, per evitare l'uso delle parentesi, i linguaggi usano delle regole di precedenza che specificano una gerarchia fra gli operatori del linguaggio. Un secondo problema è relativo all'associatività degli operatori che vi compaiono. Non è possibile valutare un'espressione con un'unica scansione da sinistra verso destra, dato che in alcuni casi dobbiamo prima valutare la parte successiva di un'espressione per poi ritornare a quella che stiamo considerando.

Le espressioni scritte usando la notazione polacca prefissa sono valutate in modo semplice scandendo l'espressione ed usando una pila. L'algoritmo per la valutazione è descritto dai seguenti passi, dove usiamo una normale pila ed un contatore C per memorizzare il numero di operandi richiesti dall'ultimo operatore letto:

(1) Leggi il prossimo simbolo dall'espressione e mettilo sulla pila;
(2) Se il simbolo letto è un operatore, inizializza C con il numero di argomenti dell'operatore e torna a (1);
(3) Se il simbolo letto è un operando decrementa C;
(4) Se C è diverso da 0 torna a (1);
(5) Se C è uguale a 0 esegui le seguenti operazioni:
    (a) applica l'ultimo operatore inserito nella pila agli operandi inseriti successivamente, memorizza il risultato in R, elimina operatore ed operandi dalla pila e memorizza il valore di R sulla pila;
    (b) se non vi sono simboli di operatore sulla pila vai a (6);
    (c) inizializza il contatore C con n – m dove n è il numero di argomenti dell'operatore che si trova al top della pila e m è il numero di operandi presenti sulla pila sopra tale operatore;
    (d) torna a (4);
(6) Se la sequenza che rimane da leggere non è vuota torna a (1).

La valutazione delle espressioni nel caso della notazione polacca inversa è ancora più semplice: in questo caso infatti non serve controllare che sulla pila ci siano tutti gli operandi per l'ultimo operatore inserito, dato che gli operandi vengono letti prima degli operatori. L'algoritmo per la valutazione è il seguente:

(1) Leggi il prossimo simbolo dell'espressione e mettilo sulla pila
(2) Se il simbolo letto è un operatore, applicalo agli operandi immediatamente precedenti sulla pila, memorizza il risultato in R, elimina operatore e operandi dalla pila e memorizza il valore di R sulla pila;
(3) Se la sequenza che rimane da leggere non è vuota torna a (1);
(4) Si il simbolo letto è un operando torna a (1).

Le espressioni possono essere rappresentate da alberi nei quali:

- ogni nodo interno è un operatore;
- ogni foglia è un operando semplice;
- ogni sotto-albero radicato in N è un operando per l'operatore di N.

Le rappresentazioni lineari infissa, prefissa e postfissa si ottengono rispettivamente dalle visite dell'albero in ordine simmetrico, anticipato e differito. La rappresentazione ad albero di un'espressione chiarisce precedenza e associatività degli operatori. Per i linguaggi implementati in modo compilativo il parser realizza l'analisi sintattica costruendo un albero di derivazione. Mentre in termini matematici l'ordine di valutazione delle sotto-espressioni non hanno importanza, in termini informatici sono molto rilevanti, per i seguenti cinque motivi:

- Effetti collaterali: la valutazione di un'espressione può modificare il valore di alcune variabili nonché i risultati parziali e finali. La possibilità di effetti collaterali fa sì che l'ordine di valutazione degli operandi sia rilevante ai fini del risultato.
- Aritmetica finita: i riordinamento dei numeri finiti presenti in un calcolatore può causare problemi di overflow; inoltre la precisione limitata dell'aritmetica del calcolatore fa sì che cambiando l'ordine degli operandi si possano ottenere risultati diversi.
- Operandi non definiti: la strategia detta “eager” consiste nel valutare prima tutti gli operandi e poi passare ad applicare l'operatore ai valori ottenuti dalla valutazione degli operandi; però alcune espressioni possono essere definite anche se alcuni operandi che vi compaiono non lo sono. La strategia “lazy” consiste nel passare gli operandi non valutati all'operatore il quale deciderà quali operandi sono necessari, valutando solo questi.
- Valutazione con “corto-circuito”: usa la valutazione lazy, dove l'espressione è “corto-circuitale” nel senso che si arriva al valore finale prima di conoscere il valore di tutti gli operandi.
- Ottimizzazione: l'ordine di valutazione delle sotto-espressioni influenza l'efficienza della valutazione di un'espressione. In alcuni casi, i compilatori possono cambiare l'ordine degli operandi nelle espressioni per ottenere codice più efficiente, ma semanticamente equivalente. Il risultato di questo tipo di approccio è che si hanno degli errori a run-time dei quali non si riesce a capire con facilità l'origine. Per scrivere codice corretto è opportuno eliminare le fonti di ambiguità nella valutazione delle espressioni.

La nozione di comando

Un comando è una entità sintattica la cui valutazione non necessariamente restituisce un valore, ma può avere un effetto collaterale, se esso influenza il risultato della computazione senza che la sua valutazione restituisca alcun valore al contesto nel quale esso si trova. Sono presenti tipicamente nei linguaggi imperativi.

In informatica le variabili non hanno un'unica definizione, ma in base a un diverso paradigma:

- Il paradigma classico usa la variabile modificabile, secondo questo modello la variabile è vista come un contenitore al quale gli si può dare un nome e che contenere dei valori.
- Il paradigma ad oggetti vede le variabili con un riferimento, cioè un meccanismo che permette di accedere a un valore, memorizzato sullo heap. Nozione analoga ai puntatori;
- Il paradigma funzionale usa le variabili come un identificatore che denota un valore.
- Il paradigma logico usa come variabili degli identificatori associati a valore, una volta creato un legame fra un identificatore di variabile ed un valore, tale legame non può essere eliminato.

L'assegnamento è il comando di base che permette di modificare i valore delle variabili modificabili, e quindi dello stato, in un linguaggio imperativo. L'effetto di tale comando è un 'effetto collaterale, dato che la valutazione del comando non ha, di per sé, restituito alcun valore. Gli l-

valori sono quei valori che indicano locazioni e quindi sono i valori di espressioni che possono stare alla sinistra di un comando di assegnamento; gli r-valori sono invece i valori che possono essere contenuti nelle locazioni, e quindi sono i valori di espressioni che possono stare alla destra di un comando di assegnamento.

Comandi per il controllo di sequenza

I comandi servono per definire il controllo di sequenza, cioè l'ordine con cui devono essere effettuate le istruzioni. Tali comandi si possono dividere in tre categorie:

- Sequenza: il comando sequenziale permette di specificare l'esecuzione sequenziale di due comandi; il comando composto permette di raggruppare una sequenza di comandi in un blocco; il goto trasferisce il controllo al punto del programma nel quale è presente l'etichetta associata; il break termina l'esecuzione di un iterazione; il continue termina l'iterazione e forza l'inizio di quella successiva; il return termina la valutazione di una funzione.
- Selezione: esprimono un'alternativa fra due o più possibili prosecuzioni della computazione sulla base di opportune condizioni logiche: comando if; comando case: implementato in assembly con un vettore di celle contigue detto "jump table", più efficiente delle if annidate.
- Iterazione: formato da cicli nei quali si ripetono, o iterano, dei comandi, e sono di due tipi: iterazione indeterminata, si itera fino a verificarsi di una data condizione; iterazione determinata, si itera per un numero prefissato di volte.

Programmazione strutturata

Consiste in una serie di prescrizioni volte a permettere uno sviluppo il più possibile strutturato del codice e simile al flusso di controllo. I punti più importati sono:

- Progettazione top-down o gerarchica;
- Modularizzazione del codice: raggruppare codice che svolge una specifica funzione;
- Uso di nomi significativi;
- Uso di tipi di dato strutturati: raggruppare e strutturare informazioni di ogni tipo;
- Uso di costrutti strutturati per il controllo: hanno un solo punto di ingresso ed uno di uscita.

La ricorsione

Una funzione ricorsiva è una procedura nel cui corpo compare una chiamata a se stessa. Si può avere anche una ricorsione indiretta, o meglio mutua ricorsione, quando una procedura P chiama un'altra procedura Q che, a sua volta, chiama P.

La ricorsione rende necessaria la gestione dinamica della memoria, in quanto non è possibile determinare staticamente il numero massimo di istanza di una stessa funzione. In alcuni casi possiamo evitare l'allocazione di nuovi RdA per le chiamate successive di una stessa funzione, in quanto possiamo riutilizzare sempre lo stesso spazio di memoria.

Sia F una funzione che nel suo corpo contenga la chiamata ad una funzione G. La chiamata di G si dice "chiamata in coda" se la funzione F restituisce il valore restituito da G senza ulteriori computazioni. La funzione F ha la ricorsione in coda se tutte le chiamate ricorsive presenti in F sono chiamate in coda.

In generale è sempre possibile trasformare una funzione che non abbia la ricorsione in coda in una che la abbia, complicando la funzione: tutta la computazione che deve essere fatta dopo la chiamata ricorsiva sia fatta prima della chiamata; la parte restante di lavoro viene passata con dei parametri aggiuntivi alla chiamata ricorsiva stessa. La trasformazione può essere fatta in modo automatico, usando una tecnica detta "continuation passing style" che rappresenta in un punto del programma la parte rimanente del programma mediante una funzione detta continuazione. Questa tecnica non sempre produce delle funzioni che possano essere eseguite con uno spazio di memoria costante, in quanto potrebbe contenere delle variabili che necessitano dell'RdA del nuovo ambiente.

7 – Astrarre sul Controllo

Sottoprogrammi

Ogni programma è composto da molte componenti, ognuna delle quali forma una parte della soluzione globale. La scomposizione permette di gestire meglio la complessità. Il linguaggio fornisce un supporto linguistico che permette di decomporre e ricomporre. Il concetto chiave è quello di sottoprogramma, o procedura, o funzione. Una funzione è una porzione di codice identificata da un nome, dotata di ambiente locale proprio e capace di scambiare informazioni col resto del codice mediante parametri. Questo concetto si traduce in due diversi meccanismi linguistici: la definizione (o dichiarazione) di una funzione, e il suo uso (o chiamata). Una funzione scambia informazioni col resto del programma mediante tre meccanismi principali:

- Parametri: parametri formale che compaiono nella definizione di una funzione; parametri attuali che compaiono nella chiamata.
- Valore di ritorno: certe funzioni restituiscono un valore come risultato della funzione stessa.
- Ambiente non locale: quando una funzione modifica una variabile non locale.

I sottoprogrammi sono meccanismi che permettono al progettista di software di ottenere astrazione funzione. Una componente software è un'entità che fornisce sevizi al suo ambiente. I clienti di tale componente non hanno interesse a conoscere come tali sevizi sono resi, ma solo come richiederli. Si ha una vera astrazione funzionale quando il cliente non dipende dal corpo di una funzione, ma solo dalla sua intestazione.

La modalità con cui i parametri attuali sono accoppiati ai parametri formali sono dette modalità di passaggio dei parametri e sono:

- Passaggio per valore: è una modalità che corrisponde ad un parametro di ingresso. L'ambiente locale è esteso con un'associazione tra il parametro formale ed una nuova variabile. Il parametro attuale viene valutato al momento della chiamato, e lo r-valore così ottenuto viene assegnato al parametro formale. Durante l'esecuzione non c'è nessun legame tra il parametro formale e l'attuale. Al termine della procedura il parametro formale e l'ambiente locale vengono distrutti. Con il modello di allocazione a pila, il formale corrisponde ad una locazione nel RdA della procedura, nella quale viene memorizzato il valore dell'attuale. Si tratta di una modalità costosa nel caso in cui il parametro corrisponde ad una struttura dati di grandi dimensioni, per contro, il costo dell'accesso al parametro formale è minimo.
- Passaggio per riferimento: realizza un meccanismo nel quale il parametro è sia di ingresso che di uscita. Il parametro attuale deve essere un'espressione dotata di l-valore; al momento della chiamata, viene valutato lo l-valore dell'attuale e l'ambiente della procedura è esteso con un'associazione tra il parametro formale e lo l-valore dell'attuale. Al termine della procedura, insieme all'ambiente locale viene distrutto anche il legame tra il formale e lo l-valore. Ogni modifica al formale è una modifica all'attuale. Nel modello a pila, al formale è associata una locazione nel RdA, l'accesso al formale avviene per via indiretta. Si tratta di una modalità di costo molto contenuto: al momento della chiamata si deve solo memorizzare un indirizzo, mentre ogni riferimento al formale viene pagato con un accesso indiretto.
- Passaggio per costante: qualora il parametro formale non riceva alcuna modifica nel corpo della funzione si mantiene la semantica del passaggio per valore, implementandola con il passaggio per riferimento. Stabilisce una comunicazione di ingresso e nella quale sono permesse espressioni generiche come parametro attuale. Per dati di piccole dimensioni, il passaggio per costante potrà essere implementato come il passaggio per valore; per strutture dati più grandi, si sceglierà di implementarlo mediante un riferimento.
- Passaggio per risultato: realizza una comunicazione di sola uscita. L'ambiente locale della procedura è esteso con un'associazione tra il parametro formale e una nuova variabile. Il parametro attuale deve essere un'espressione dotata di l-valore; al momento della chiamata,

non avviene alcuna forma di comunicazione, mentre 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 alcun legame tra il parametro formale e l'attuale. La modalità per risultato rende agevole il progetto di funzioni che devono restituire più valore in variabili distinte.

- Passaggio per valore-risultato: realizza una comunicazione bidirezionale, col formale che è a tutti gli effetti una variabile locale alla procedura. Il parametro attuale deve essere un'espressione dotata di l-valore; 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 alcun legame tra il parametro formale e l'attuale. L'implementazione canonica del passaggio per valore, anche se alcuni linguaggi scelgono di implementarlo come il passaggio per riferimento nel caso di dati di grandi dimensioni.
- Passaggio per nome: definisce la semantica di una chiamata di funzione mediante la regola di copia: sia F una funzione con unico parametro formale X e sia A un'espressione compatibile con il tipo X. Una chiamata a F con parametro attuale A è semanticamente equivalente all'esecuzione di F nel quale tutte le occorrenze del di X sono sostituite con A. Se ci sono variabili con lo stesso nome, quella del parametro formale viene catturata dalla dichiarazione locale. Quindi la sostituzione della regola di copia deve essere senza cattura di variabile: dopo la sostituzione viene valutato nell'ambiente del chiamante. Ciò che viene sostituito non è il parametro solo attuale, ma il parametro attuale insieme al proprio ambiente. La regola di copia impone che il parametro attuale sia valutato ogni volta che il formale viene incontrato durante l'esecuzione. La coppia espressione, ambiente è definita come chiusura. Nel caso del passaggio per nome, il chiamante passa una chiusura. Nel caso di un'allocazione a pila, la chiusura è una coppia di puntatori: uno punta al codice di valutazione dell'espressione e l'altro è un puntatore di catena statica. Il passaggio per nome è una modalità che corrisponde ad un parametro sia di ingresso che di uscita. Si tratta di una modalità molto costosa, sia per la necessità di passare una struttura complessa, sia per a valutazione ripetuta del parametro attuale.

Funzione di ordine superiore

Una funzione è di ordine superiore qualora abbia come parametro, o restituisca come risultato, un'altra funzione. Nel caso di una funzione come parametro, ci potrebbe essere una definizione multipla del nome cosicché è necessario stabilire quale sia l'ambiente non locale in cui la funzione F sarà valutata. Vi sono due possibilità per l'ambiente da utilizzare al momento dell'esecuzione di una funzione F invocata da un parametro formale H:

- utilizzare l'ambiente attivo al momento della creazione de legame tra H e F; in questo caso il linguaggio adotta una politica di deep binding: utilizzato dai linguaggio sia con scope statico che scope dinamico;
- utilizzare l'ambiente attivo al momento della chiamata di F tramite H; in questo caso il linguaggio adotta una politica di shallow binding: utilizzato dai linguaggio scope dinamico.

Lo shallow binding non pone ulteriori problemi implementativi rispetto alle tecniche utilizzate per lo scope dinamico: basta ricercare per ogni nome la sua ultima associazione presente. Il deep binding, invece, che richiede strutture dati ausiliarie rispetto all'ordinaria catena statica o dinamica.

Tutti i linguaggi con scope statico adottano il deep binding. Potrebbe sembrare che deep o shallow binding non facciano alcuna differenza nel caso di scope statico, ma non è così. La ragione è da ricercare nella possibilità che sulla pila siano presenti più RdA corrispondenti alla stessa funzione. Se dall'interno di una di queste attivazioni viene passata per parametro una procedura, si può creare una situazione nella quale le sole regole di scope non sono sufficienti a determinare quale ambiente

non locale va utilizzato. Ma cosa definisce l'ambiente:

- regole di visibilità: garantite dalla struttura a blocchi;
- eccezioni alle regole di visibilità: ridefinire i nomi e usare un nome prima della sua dichiarazione;
- regole di scope;
- regole relative alla modalità di passaggio di parametri;
- politica di binding.

La possibilità di generare funzioni come risultato di altre funzioni permette la creazione dinamica di funzioni a tempo di esecuzione. La funzione restituita, per essere rappresentata a tempo di esecuzione, ha bisogno del suo codice e dell'ambiente nel quale dovrà essere valutata. Quando una funzione restituisce una funzione come risultato, tale risultato è una chiusura; e quindi si deve modificare la macchina astratta per tenere conto della chiusura. Quando viene invocata una funzione il cui valore è stato ottenuto dinamicamente il puntatore di catena statica del suo RdA viene determinato utilizzando la chiusura associata. Se è possibile restituire una funzione dall'interno di un blocco annidato, si può creare la possibilità che il suo ambiente di valutazione faccia riferimento a un nome che sarebbe distrutto. In questi linguaggi il tempo di vita degli ambienti è illimitato. La soluzione è quella di allocare tutti i RdA sullo heap, e lasciare ad un garbage collecton la responsabilità di deallocarli quando si scoprisse che non vi sono riferimenti verso i nome in essi contenuti.

Eccezioni

Un'eccezione è un evento particolare che si verifica durante l'esecuzione di un programma e che non deve essere gestito dal normale flusso del controllo. Si può trattare del verificarsi di un errore di semantica dinamica (divisione per zero, overflow) o di una terminazione esplicita del flusso corrente. Sono presenti dei costrutti che permettono di interrompere una computazione e andare alla ricerca di un gestore per le eccezioni. Per gestire correttamente le eccezioni, un linguaggio deve:

1. specificare quali eccezioni sono gestibili e come possono essere definite: eccezioni della macchina stratta: violato un vincolo di semantica; ed eccezioni definite dall'utente: valori di un tipo speciale, o un valore qualsiasi.
2. specificare come un'eccezione può essere sollevata: un'eccezione può essere sollevata implicitamente se si tratta di un'eccezione della macchina astratta, oppure esplicitamente dal programmatore.
3. specificare come un'eccezione possa essere gestita: si definisce una capsula attorno ad una porzione di codice (blocco protetto), per intercettare le eccezioni che si verificano all'interno della capsula stessa; si definisce un gestore dell'eccezione che trasferisce al blocco protetto, il controllo quando la capsula intercetta l'eccezione.

Un'eccezione ha un nome che viene esplicitamente menzionato nel costrutto throw e che viene usato dai costrutti di tipo try-catch per intrappolare una classe di eccezioni. L'eccezione potrebbe inglobare un valore, che il costrutto passa al gestore come argomento su cui operare per “reagire” all'avvenuta eccezione. Se l'eccezione non è gestita all'interno della procedura in esecuzione, occorre terminarla e risollevare l'eccezione nel punto in cui la procedura era stata invocata. Se non è gestita neppure dal chiamante, l'eccezione viene propagata lungo la catena delle chiamate di procedura, fino a quando si incontra un gestore compatibile o si raggiunge il livello più alto, che fornirà un gestore default (terminazione con errore).

Il modo con il quale una macchina astratta può implementare le eccezioni è quello di sfruttare la pila dei RdA. Ogni volta che, in run-time, si entra in un blocco protetto, nel RdA viene inserito un puntatore al gestore corrispondente. Quando si esce da un blocco protetto viene tolto dalla pila il riferimento al gestore. Infine, quando viene sollevata un'eccezione, la macchina astratta cerca un gestore per quell'eccezione nel RdA corrente. Se non lo trova, ripristina lo stato della macchina,

toglie il record dalla pila e risolleva l'eccezione. Questa soluzione ha un difetto, cioè, ogni volta che si entra in un blocco protetto, o se ne esce, la macchina stratta deve manipolare la pila. Una soluzione più efficiente a tempo di esecuzione la si ottiene anticipando il lavoro a tempo di compilazione.

8 – Strutturare i Dati

Tipi di dato

Un tipo di dato è una collezione di valori omogenei, cioè che condividono delle proprietà strutturali, effettivamente rappresentabili, cioè rappresentabili in modo finito, e dotata di un insieme di operazioni che manipolano tali valori. La presenza di tipo diversi permette al progettista di utilizzare quello più adatto per ogni classe di concetti. Ogni linguaggio ha proprie regole di controllo dei tipi, che disciplinano come i tipi possono o devono essere usati in un programma. Tali vincoli sono presenti nei linguaggi per evitare errori hardware e logici. Molto linguaggi verificano, tramite il type checker, che i vincoli di tipo siano tutti soddisfatti prima di procedere all'esecuzione. La presenza di vari tipi serve a fornire importanti informazioni per la macchina astratta, informazioni che sono disponibili staticamente, e non cambiano durante l'esecuzione.

Sistemi di tipi

Ogni linguaggio ha un proprio sistema di tipi, ossia il complesso delle informazioni e delle regole che governano i tipi in quel linguaggio. Un sistema di tipi è costituito da:

1. insieme dei tipi predefiniti dal linguaggio;
2. meccanismi che permettono di definire nuovi tipi;
3. meccanismi relativi al controllo dei tipi, tra i quali distinguiamo:
   - regole di equivalenza: quando due tipi diversi corrispondono allo stesso tipo;
   - regole di compatibilità: quando un valore di un certo tipo può essere utilizzato in un contesto nel quale sarebbe richiesto un tipo diverso;
   - regole di inferenza: come attribuisce un tipo ad un'espressione complessa;
4. specifica se i vincoli siano da controllare staticamente o dinamicamente

Un sistema di tipi è sicuro ai tipi (type safe) quando nessun programma durante l'esecuzione può generare errore non segnalato che derivi da una violazione di tipo. Possiamo classificare i tipi a seconda di come i valori possono essere manipolati:

- denotabili: se possono essere associati ad un nome;
- esprimibili: se possono essere il risultato di un'espressione complessa;
- memorizzabili: se possono essere memorizzati in una variabile.

Un linguaggio ha tipizzazione statica se i controlli dei vincoli di tipizzazione possono essere condotti a tempo di compilazione. In tal modo i controlli sono anticipati e vengono così rilevati dal programmatore prima dell'esecuzione. In un controllo completamente statico, il mantenimento esplicito dei tipi durante l'esecuzione è inutile, perché la correttezza è stata garantita staticamente. L'esecuzione è così più efficiente, visto che non sono necessari controlli a run-time. Vi sono però dei prezzi da pagare: la progettazione è più complessa, nonché la sua compilazione che risulta anche più lenta, ma quest'ultimo è un prezzo che si paga volentieri, visto che la compilazione viene eseguita poche volte, e perché i controlli abbreviano la fase di testing e debugging. Inoltre i controlli statici possono decretare come sbagliati programmi che, in realtà, non causano un errore di tipo durante l'esecuzione.

Ha tipizzazione dinamica se i controlli avvengono a tempo di esecuzione, dove prevede che ogni oggetto possieda un descrittore a run-time che ne specifica il tipo. La tipizzazione dinamica previene gli errori di tipo, ma è inefficiente in esecuzione.

Tipi scalari

Sono detti tipi scalari tutti quei tipi i cui valori non sono costituiti da aggregazioni di altri valori:

- Booleani: valori [vero o falso]; operazioni [operazioni logiche];
- Caratteri: valori [codici di caratteri]; operazioni [dipendono dal linguaggio];
- Interi: valori [numeri interi]; operazioni [confronti e operazioni aritmetiche];
- Virgola mobile: valori [numeri razionali]; operazioni [confronti e operazioni aritmetiche];
- Virgola fissa: valori [numeri razionali]; operazioni [confronti e operazioni aritmetiche];
- Complessi: valori [numeri complessi]; operazioni [confronti e operazioni aritmetiche];
- Void: valori [solo uno]; operazioni [nessuna];
- Enumerazioni: valori [inseme finito di costanti]; operazioni [confronti e passaggio da un valore al successivo o precedente];
- Intervalli: valori [sottoinsiemi di valori di un altro tipo scalare]; operazioni [confronti e passaggio da un valore al successivo o precedente];

Tipi composti

I tipi composti si ottengono combinando tra loro altri tipi mediante opportuni costruttori e sono:

- record: collezione di un numero finito di campi distinti dal loro nome; ciascun campo può essere diverso dagli altri. L'uguaglianza tra record non è sempre definita; analogamente non è sempre consentito l'assegnamento di record nella loro interezza. Dove queste operazioni non sono consentite si deve confrontare una campo alla volta. Una forma particolare di record è quella in cui alcuni campi sono tra loro mutuamente esclusivi: record varianti. Le varianti condividono la stessa zona di memoria, visto che non possono essere attive contemporaneamente.
- array: è una collezione finita di elementi dello stesso tipo, indicizzata su un intervallo di un tipo ordinale. Gli array multidimensionali sono indicizzati su due o più tipi indice. Le operazione permesse sono la selezione di un elemento mediante l'indice, assegnamento, uguaglianza, confronti, operazioni aritmetiche e selezionare le slice (fette) di un array. C'è un controllo a tempo di esecuzione che serve a verificare che ogni accesso all'array avvenga “entro i limiti”, prevenendo così anche uno dei più seri attacchi alla sicurezza di un sistema, detto buffer overflow. Un array è memorizzato in un porzione contigua di memoria: per gli array monodimensionali l'allocazione segue l'ordine degli indici; nel caso multidimensionale c'è la tecnica di ordine di riga e di ordine di colonna. La forma (shape) di un array è costituita dal numero delle sue dimensioni e dall'intervallo su cui ciascuna di esse può variare, ci sono tre modalità di allocazione: forma statica (tutto deciso a tempo di compilazione); elaborazione della dichiarazione (definisce la forma al momento in cui il controllo raggiunge la dichiarazione dell'array); forma dinamica (cambia forma dopo la sua creazione per effetto dell'esecuzione). Il descrittore di un array di forma non nota staticamente, detto dope vector, allocato nella parte di lunghezza fissa di un RdA, contiene un puntatore alla prima locazione dove è memorizzato e tutte le informazioni dinamiche necessarie al calcolo della quantità. Si accede per offset mediante il frame pointer, si calcola l'espressione e la si somma all'indirizzo di inizio.
- Insiemi: i suoi valori sono costituiti dai sottoinsiemi di un tipo base ristretto ad essere un tipo ordinale. Le operazioni possibili sono il test di appartenenza di un elemento ad un insieme e le usuali operazioni insiemistiche. Un insieme è rappresentato o come un vettore di bit di lunghezza pari alla cardinalità del tipo base, o mediante tabelle hash che permettono di rappresentare in modo più compatto gli insiemi.
- puntatori:servono per manipolare l-valori: ogni variabile è sempre un riferimento, cioè un l-valore. I puntatori possono riferirsi ad un l-valore senza de-referenziarlo automaticamente. Uno degli usi principali dei puntatori è quello di costruire valori di tipi ricorsivi. Le operazioni permesse sono la creazione, il test di uguaglianza, in particolare con null, e la loro de-referenziazione, cioè l'accesso all'oggetto puntato. Il modo per creare un puntatore è

usare un costrutto che alloca sullo heap un oggetto del tipo opportuno e restituisce un riferimento a tale oggetto, oppure un altro modo è applicare opportuni operatori che restituiscono l'indirizzo di memorizzazione di un oggetto allocato in memoria. Si possono effettuare operazioni aritmetiche: incrementare e decrementare un puntatore, sottrarre tra loro due puntatori e sommare una quantità arbitraria ad un puntatore. Le deallocazione della memoria può essere implicita o esplicita. In quella implicita il programmatore continua a chiedere allocazioni, fin quando vi è spazio sullo heap; è possibile recuperare memoria inutilizzata con un garbage collection. Nel caso della esplicita, ci sono dei meccanismi con i quali si può rilasciare la memoria riferita da un puntatore. Il problema di quella esplicita è che si possono creare dei dangling reference (riferimenti pendenti), cioè puntatori che puntano ad informazioni non più significative.

- tipi ricorsivi: è un tipo composto nel quale un valore del tipo può contenere un (riferimento a un) valore dello stesso tipo. Le operazioni permesse sono la selezione di una componente e il test di uguaglianza. Corrispondono ad una struttura concatenata dove ogni elemento è costituito da un record. Nel linguaggi funzionali i valori sono esprimibili con espliciti costrutti. Nel linguaggi imperativi i valori vengono costruiti mediante allocazione esplicita dei loro componenti sullo heap.
- funzioni: alcuni linguaggi permettono di definire tipi di funzioni. I valori sono denotabili in tutti i linguaggi, ma raramente esprimibili. Oltre alla definizione, l'altra operazione ammessa è l'applicazione, cioè l'invocazione di una funzione su alcuni argomenti (parametri attuali).

Equivalenza

Stabilire quando due tipi, formalmente diversi, sono da considerare intercambiabili, cioè non distinguibili all'interno di un programma. I vari linguaggi interpretano una definizione di tipi in due modi diversi:

- opaca: da luogo all'equivalenza per nome, dove ogni nuova definizione introduce un nuovo tipo, diverso dai precedenti. Due tipi sono equivalenti per nome solo se hanno lo stesso nome;
- trasparente: da luogo all'equivalenza strutturale, dove il nome del tipo non è che un'abbreviazione del tipo che viene definito. Due tipi sono equivalenti se hanno la stessa struttura, cioè se, sostituendo tutti i nomi con le relative definizioni, si ottengono tipi identici. L'equivalenza strutturale fra tipi è la relazione d'equivalenza che soddisfa le seguenti proprietà:
  - un nome di un tipo è equivalente a se stesso;
  - se un tipo T è introdotto con una definizione `type T = espressione`, T è equivalente a espressione;
  - se due tipi sono costruiti applicando lo stesso costruttore a tipi equivalenti, allora sono equivalenti.

Compatibilità e conversione

La relazione di compatibilità tra tipi, più debole dell'equivalenza, permette di usare un tipo in un contesto nel quale sarebbe richiesto un altro tipo. Il tipo T è compatibile con il tipo S, se un valore di tipo T è ammesso in un qualsiasi contesto in cui sarebbe richiesto un valore di tipo S. In molti linguaggi è la compatibilità a regolare la correttezza di un assegnamento, quella del passaggio dei parametri. Due tipi equivalenti sono anche compatibili uno con l'altro, ma la relazione di compatibilità non è simmetrica. La conversione di tipo è fatta in due modi:

- conversione implicita (o coercizione): da un punto di vista sintattico, la coercizione annota la presenza di una situazione di compatibilità. Da un punto di vista implementativo può corrispondere a cose distinte a seconda nella nozione di compatibilità adottata:
  1. T è compatibile con S e condividono la rappresentazione in memoria: la coercizione rimane al livello sintattico e non genera alcun codice;
  2. T è compatibile con S perché esiste un modo canonico di trasformare i valori di T in S:

la coercizione viene eseguita dalla macchina astratta;
3. la compatibilità di T con S è stabilita in virtù di una corrispondenza arbitraria tra i valori di T e quelli di S: la macchina astratta inserisce il codice che corrisponde alla trasformazione.

- conversione esplicita (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. La conversione esplicita è inutile sintatticamente, i linguaggi con poche compatibilità rendono disponibile molte conversioni esplicite con le quali il programmatore annota le situazioni dove è necessario un cambio di tipo. I cast sono più espressivi da un punto di vista della documentazione, non dipendo dal contesto sintattico in cui compaiono, e si comportano meglio in presenza di overloading e polimorfismo.

Polimorfismo

Un sistema di tipi nel quale ogni oggetto del linguaggio ha un unico tipo si dice monomorfo. Se invece un oggetto può avere più di un tipo è detto polimorfo. Per esempio un linguaggio polimorfo ci permette di creare un'unica funzione di ordinamento per tutti i tipi, invece, in altri linguaggi dovremmo creare una funzione per ogni tipo.

L'overloading (sovraccaricamento) o polimorfismo ad hoc, è in realtà un polimorfismo solo in apparenza. Un nome è sovraccaricato quando ad esso corrispondono più oggetti; gli esempi più comuni sono l'uso del nome + per indicare sia la somma di interi che la somma reale, e la possibilità di definire più funzioni con lo stesso nome, ma distinte dal numero o dal tipo di parametri. La situazione di ambiguità viene risolta staticamente utilizzando l'informazione di tipo presente nel contesto. Può essere gestito da una fase di pre-analisi, che risolve i casi di overloading e sostituisce ogni simbolo sovraccaricato con un nome non ambiguo che denota univocamente un solo oggetto.

Un valore esibisce un polimorfismo universale parametrico quando ha un'infinità di tipi diversi, che si ottengono per un istanziazione da un unico schema di tipo universale. Una funzione polimorfa universale è costituita da un unico codice, che lavora uniformemente su tutte le istanze del suo tipo generale. Tra gli esempi la funzione di ordinamento valida per tutti i tipi. Questa forma di polimorfismo è più generale e flessibile, e ha due forme di notazione:

- Polimorfismo esplicito: è quello fino a ora discusso: sono presenti esplicite annotazioni che indicano quali tipi devono essere considerati parametri;
- Polimorfismo implicito: il programma non può riportare alcuna indicazione di tipo, ed è invece il modulo che si occupa dell'inferenza dei tipi a cercare di ottenere per ogni oggetto il suo tipo più generale.

Il polimorfismo di sottotipo, presente nei linguaggi orientati agli oggetti, è una forma più limitata di polimorfismo universale. Un valore può assumere 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.

Un primo modo di gestire il polimorfismo è quello di risolverlo in modo statico a tempo di link. Vengono identificate le funzioni polimorfe, il loro codice viene opportunamente modificato (istanziato) per tener conto dl tipo con cui sono chiamate e il codice risultante viene collegato tutto assieme. Esistono nel codice eseguibile più copie dello stesso template, una per ogni istanza. Un altro modo è quello dinamico, dove viene generata un'unica versione del codice di una funzione polimorfa, ed è proprio quel codice ad essere eseguito. Per gestire le informazioni che vengono dal tipo, occorre cambiare la rappresentazione dei dati: invece di allocare direttamente il dato, nel RdA si mette un puntatore al dato stesso, che comprende anche un suo descrittore. La flessibilità è pagata, però, da un po' di efficienza.

Controllo e inferenza di tipo

La responsabilità di verificare che un programma rispetti le regole imposte dal sistema di tipi è del controllore dei tipi. Nel tipo di controllo statico, il controllore è un modulo del compilatore; nel caso dinamico, il controllore è un modulo del supporto a run-time. Per ottenere il suo scopo il controllore deve determinare il tipo delle espressioni presenti nel programma, sfruttando le informazioni di tipo date dal programmatore, e quelle implicite nella definizione del programma. Per determinare il tipo delle espressioni complesse, il controllore esegue una vista dell'albero sintattico del programma: parte dalle foglie, risale l'albero verso la radice, calcola il tipo delle espressioni composte a partire dalle informazioni fornite dal programmatore e da quelle che derivano dal sistema di tipi.

Al posto del controllo dei tipi, alcuni linguaggi adottano l'inferenza di tipo (deduzione di tipo). L'inferenza è un processo di attribuzione di un tipo ad un'espressione nella quale non appaiono esplicite dichiarazioni di tipo per i suoi componenti; è in grado di scoprire il tipo più generale di una funzione, cioè di esplicitare tutto il polimorfismo in un'espressione, procedendo così:

1. Assegna un tipo ad ogni nodo dell'albero sintattico;
2. Risale l'albero sintattico, generando un vincolo di uguaglianza tra tipi in corrispondenza ad ogni nodo interno;
3. Risolve i vincoli raccolti, usando l'algoritmo di unificazione.

Evitare i dangling reference

Mediante le tombstone una macchina astratta è in grado di segnalare ogni tentativo di dereferenziare un dangling reference. Ogni volta che viene allocato un oggetto sullo heap, la macchina astratta alloca anche un'ulteriore parola di memoria: la tombstone. Questa viene inizializzata con l'indirizzo dell'oggetto allocato, mentre il puntatore riceve l'indirizzo della tombstone. Quando si dereferenzia un puntatore, la macchina astratta inserisce un secondo livello di indirizzamento, per accedere prima alla tombstone e poi da questa all'oggetto puntato. Quando un puntatore viene assegnato ad un altro, è il contenuto del puntatore ad essere modificato. Alla deallocazione di un oggetto, la tombstone relativa viene invalidata, memorizzandovi un valore particolare che segnala che i dati a cui si riferiva sono “morti”. Le tombstone allocate in un'area particolare della memoria, della il cimitero, che può essere gestita dallo heap. Il meccanismo delle tombstone pecca in termini di efficienza che di spazio. Per quanto riguarda il tempo, bisogna considerare il tempo necessario alla creazione e al controllo delle tombstone, e il doppio accesso indiretto. Per quanto riguarda lo spazio richiedono una parola di memoria per ogni oggetto allocato sullo heap, e le tombstone invalidate rimangono per sempre allocato.

Un'alternativa alle tombstone è la tecnica nota come “lucchetti e chiavi”: ogni volta ce 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. Un puntatore è costituito in quest'approccio da una coppia: l'indirizzo e una chiave, cioè una parola di memoria che viene inizializzata al valore del lucchetto corrispondente all'oggetto puntato. Quando un puntatore viene assegnato ad un altro, viene assegnata tutta la coppia; tutte le volte che si dereferenzia un puntatore, la macchina astratta controlla ce la chiave altra il lucchetto, in caso contrario viene segnalato un errore. Nel momento in cui un oggetto viene deallocato, il suo lucchetto viene annullato, così tutte le chiavi che prima lo aprivano, ora causino un errore. Lucchetti e chiavi in termini di spazio costano più delle tombstone. D'altra parte sono deallocati insieme all'oggetto o al puntatore di cui fanno parte.

Garbage collection

Un meccanismo con il quale si possa automaticamente recuperare la memoria allocata sullo heap e non più utilizzata è il garbage collector (spazzino). Il suo funzionamento si compone di due fasi:

1. distinguere gli oggetti vivi da quelli non più utilizzati (garbage detection);
2. recuperare gli oggetti non più utilizzati, così da poterli riutilizzare.

Le due fasi non sono sempre temporalmente separate: le tecniche per il recupero degli oggetti dipendono da quelle usate per la determinazione degli oggetti non più in uso. Possiamo classificare i garbage collector a seconda di come determinano gli oggetti non più in uso: avremo i collector basati su contatori dei riferimenti, su marcatura e su copia.

La tecnica dei contatori dei riferimenti, al momento della creazione sullo heap di un oggetto, alloca insieme ad esso anche un intero (il contatore), inaccessibile al programmatore; questo contatore viene inizializzato a 1. Al momento di un assegnamento tra puntatori P = Q, il contatore dell'oggetto puntato da Q viene incrementato di uno, mentre l'altro viene decrementato di uno. Quando si esce da un ambiente, tutti i contatori di quell'ambiente sono decrementati di uno. Se il contatore raggiunge il valore 0, l'oggetto è deallocato, e se contiene puntatori al suo interno, vengono recuperati ricorsivamente. Un vantaggio è la semplicità e la sua incrementalità: controllo e recupero sono mescolati con il funzionamento del programma. Il difetto maggiore è nella sua incapacità di deallocare tutte le strutture circolari.

La tecnica di mark and sweep prende il nome dalle due modalità con il quale vengono realizzate le due fasi astratte di distinzione e recupero degli oggetti:

- mark: marca ciascuno degli oggetti inutilizzati come “inutilizzato”; partendo poi dai puntatori attivi sulla pila, si attraversano ricorsivamente tutte le strutture dati presenti sullo heap, marcando come “in uso” ogni oggetto attraversato;
- sweep: lo heap viene spazzato: tutti i blocchi marcati come “in uso” sono lasciati immutati, mentre quelli inutilizzati sono restituiti alla lista libera.

Gli svantaggi sono la non incrementalità, la frammentazione esterna e l'inefficienza

La tecnica di rovesciamento dei puntatori (pointer reversal) usa lo spazio già presente per i puntatori per poter essere utilizzabile in un garbage collector. Per visitare una struttura concatenata, occorre marca un nodo e visitare ricorsivamente le sottostrutture i cui puntatori siano parte del nodo. In questo schema ricorsivo, bisogna memorizzare su una pila l'indirizzo del blocco appena visitato, così da poter tornare indietro quando si raggiunga il termine di una sottostruttura. Mediante il rovesciamento dei puntatori, questa pila viene memorizzata al posto dei puntatori che costituiscono la strutture. Quando si arriva al termine della sottostruttura, si ripristina il puntatore al suo valore originale, cosicché al termine della visita la struttura è nella stessa situazione iniziale. Sono necessari solo due puntatori per gestire la visita.

La tecnica di mark and compact è utilizzata per ovviare alla frammentazione causata dalla tecnica mark and sweep, possiamo modificare la fase di sweep e convertirla in una fase di compattamento: gli oggetti vivi sono spostati in modo da renderli contigui e lasciare tutta la memoria libera in un nico blocco contiguo. La compattazione su può eseguire scandendo linearmente lo heap e traslando ogni blocco vivo che si incontra. Si tratta di una tecnica più costosa, ma la compattazione ha ottime ripercussioni sulla frammentazione e sulla località e permette la gestione della lista libera come un unico blocco.

Nei garbage collector basati su copia c'è una fase di copia e compattazione dei blocchi vivi. Lo heap è diviso in due semispazi di uguali dimensioni. Durante l'esecuzione solo uno di questi è in uso: la memoria è allocata ad un'estremità del semispazio, mentre la memoria libera consiste in un unico blocco contiguo. L'allocazione è estremamente efficiente e non c'è frammentazione. Quando la memoria del semispazio è esaurita, viene invocato il garbage collector. Questi, a partire dai puntatori presenti sulla pila, inizia una visita delle strutture concatenate nel semispazio corrente, copiandole nell'altro semispazio. Al termine di questo processo il ruolo dei due semispazi viene invertito. La visita e la copia sono eseguiti dall'algoritmo di Cheney. Questo tipo di garbage collector può essere reso molto efficiente a patto di avere molta memoria per i due semispazi.

9 – Astrarre sui Dati

La macchina fisica ha dati di un solo tipo: le stringhe di bit. I tipi di un linguaggio di alto livello dotano ogni valore di un rivestimento: il valore è racchiuso in una capsula (il suo tipo) che fornisce le operazioni che possono manipolarlo. Il sistema di tipi stabilisce quanto questa capsula di rivestimento è trasparente: nei linguaggi sicuri è opaca, cioè non permette l'accesso alla rappresentazione.

Tipi di dato astratti

I comuni meccanismi di costruzione di tipo, non consentono la definizione di operazioni sui valori che si rappresentano. In caso vengano consentite, il linguaggio non garantisce che questi siano i soli modi con cui i tipi possano essere manipolati. Nel caso vogliamo rappresentare una pila di interi con un array, nulla impedisce che si possa accedere direttamente alla sua rappresentazione come array, cioè di manipolarlo direttamente. Mentre il linguaggio fornisce delle astrazioni sui dati (i tipi) che nascondo l'implementazione, lo stesso non è possibile al programmatore. Per ovviare q questo problema, alcuni linguaggi permettono di definire delle astrazioni sui dati che si comportano come i tipi predefiniti, per permettere l'inaccessibilità della rappresentazione. Questo meccanismo, chiamato tipo di dato astrato, è caratterizzato da:

1. un nome per il tipo;
2. un'implementazione per il tipo;
3. un insieme di nomi di operazioni per la manipolazione;
4. per ogni operazione, un'implementazione che usi la rappresentazione data dal punto 2;
5. una capsula che separi i nomi del tipo e delle operazioni dalle loro realizzazioni.

Information Hiding

L'incapsulamento è la tecnica di nascondere il funzionamento interno di una parte di un programma, in modo da proteggere le altre parti del programma dai cambiamenti che si produrrebbero in esse nel caso che questo funzionamento fosse difettoso. Per avere una protezione completa è necessario disporre di una robusta interfaccia che protegga il resto del programma dalla modifica delle funzionalità soggette a più frequenti cambiamenti. L'information hiding è il principio teorico su cui si basa la tecnica dell'incapsulamento. Una parte di un programma può nascondere informazioni incapsulandole in un modulo, o in un costrutto di altro tipo, dotato di interfaccia.

L'uso più frequente dell'incapsulamento è quello di nascondere lo strato fisico in cui vengono memorizzati i dati; in tal modo, se la rappresentazione interna dei dati cambia, le modifiche si propagano soltanto ad una piccola parte del programma. Supponiamo, ad esempio, che un punto di uno spazio tridimensionale sia rappresentato dalle coordinate x,y,z espresse da tre valori scalari in virgola mobile, e che, successivamente, si passi ad una rappresentazione delle coordinate del punto mediante un singolo array a tre dimensioni: un modulo di programma progettato secondo la tecnica dell'incapsulamento proteggerà il resto del programma da questo cambiamento di rappresentazione.

Moduli

I tipi di dato astratti sono meccanismi pensati per incapsulare un tipo con le relative operazioni. Più comune è che un'astrazione sia composta da più tipo tra loro correlati, dei quali si vuole dare una visione limitata. I meccanismi che realizzano questo tipo di incapsulamento sono chiamati moduli o package e appartengono ai quei linguaggi che si preoccupano delle realizzazione di un sistema complesso. Questo meccanismo permette di partizionare staticamente un programma in parti distinte: un modulo raggruppa più dichiarazioni e insieme definisce delle regole di visibilità mediante le quali si può realizzare una forma di incapsulamento e di occultamento dell'informazione. Il meccanismo dei moduli fornisce molta flessibilità, sia nella definizione della capsula, sia nella possibilità di definire moduli generici, cioè polimorfi.