Linguaggi di programmazione

Modulo 2

Giacomo Fantoni
Telegram: @GiacomoFantoni
Github: https://github.com/giacThePhantom/AlgoritmiStruttureDati

6 agosto 2022

Indice

Capitolo 1

Machines

A computer is composed of at least the following:

- A processor (CPU) which executes the machine instructions, and which can move data from and into memory.
- A main memory (RAM) that stores data and program (as sequences of machine insructions), it is fast but volatile.
- Mass storage, slower than the RAM but persistant.
- Peripheral for I/O.

The different components are connected by a bus, a series of electrical connections that is used to transmit machine instructiones and data between the CPU and the RAM and for input and ouput of data through mass storage devices. This represent the basis of the Von Neumann architecture.

1.1 The CPU

The processor obtains the machine instructions from the memory and executes them. It is composed by:

- a control part which obtains and executes instructions
- an operative part (ALU), which executes arithmetic and logic instructions. Modern CPUs also jave a floating arithmetic part (FPU).
- registers.

1.1.1 Registers

Registers can be divided in two main categories.

1.1.1.1 Invisible registers

Cannot be accessed directly by machine instructions, they are:

- Address Register (AR), which is the address to access the bus,
- Data Register (DR), which is the data to read or write.

1.1.1.2 Visible registers

They are mentioned by machine instructions, and they are:

- Program Counter (PC) or Instruction Pointer (IP), which is the address of the next machine instruction to execute,
- Status Register (SR) or Flag register (F), which are the flags describing the result of an operation of the ALU and the state of the machine.
- Several registers for data and addresses.

1.1.2 Execution of instructions

When the CPU needs to execute an instruction it read it (fetch), it copies the program counter into the address register, then it transer the data addressed in the addres register from the RAM to the data register (DR), it saves the data register in an invisible register and increment the program counter. After that the CPU decode the instruction and execute it. The instruction can modify the address register, read the data register, modify the program counter, etc. The CPU cycles round always this way, usualli executing machine instructions sequentially, which are specified in the machine language (LM).

1.2 Main memory

According to Von Neumann model the same memory may contain both data and instruction. It is accessed via bus, it is composed by a set of cells (or locations) 8 bits long (longer in modern machines). To access the memory the address to be accessed is loaded in the address register, then is signaled if the operation is of read or write. For the former the data that is read will be in the data register, for the latter the data is loaded to be written in the data register.

1.3 Physical machine

A computer is a physical machine designed for executing programs, written in a language that it can understand and execute. The language can be the same for different machines. The execution of a program is a continuous cycle of fetch, decode, load, execute, save implemented in hardware by the CPU, for this to happen is necessary to have an algorithm that can understand and execute the machine language.

1.4 Abstract machines

The algorithm that executes a program can also be implemented via software by an abstract machine, a collection of algorithms and data structures. In the same way of a physical machine each abstract one is associated to a language.

1.4.1 Operation of an abstract machine

If we name $\mathcal{M}_{\mathcal{L}}$ an abstract machine that can undestand and execute the language $\mathcal{L}$, to exexute a program this machine must:

- execute elementary operation (like the ALU),
- control the sequence of execution such as non sequential operations like jumps and cycles (control the program counter),
- transer data through memory,
- organize memory.

The execution cycle is similar to the one of a CPU but implemented in software.

Capitolo 2

Implementation of a language

Suppose to have an abstract machine $\mathcal{M}_{\mathcal{L}}$ that understands the machine language L. Every abstract machine can understand only one language, but a language can be understood by different machines. To implement the language $\mathcal{L}$ means to realize an abstract machine $\mathcal{M}_{\mathcal{L}}$ that can execute programs written in this language. The implementation can be done via hardware, software or firmware.

2.1 Implementation in software

There are two implement $\mathcal{M}_{\mathcal{L}}$ in software, meaning you need it to run on a host machine ($\mathcal{MO}_{\mathcal{LO}}$): interpreted or compiled.

2.1.1 Interpreted

By this approach there is a program written in $\mathcal{LO}$ that understands and executes $\mathcal{L}$ and implements the cycle fetch, decode, load, execute and save. This program called the interpreter translates from $\mathcal{LO}$ to $\mathcal{L}$ instruction by instruction.

2.1.2 Compiled

By this approach there is a program that can translate other programs from $\mathcal{L}$ to $\mathcal{LO}$. This program is called the compiler which translate the entire program before execution and can be executed from a different machine.

2.1.3 Hybrid implementation

By this approach there is a compiler that translates the program into an intermediate language ($\mathcal{LI}$). This new program is then interpreted by a $\mathcal{MO}_{\mathcal{LO}}$ program written in $\mathcal{LI}$. Depending on the differences between $\mathcal{LI}$ and $\mathcal{LO}$ it is said that an implementation is mainly compilative (like C) or mainly interpretive (like Java).

2.1.4 Comparison between the approaches

The purely interpretive approach has a less efficient implementation of $\mathcal{M}_{\mathcal{L}}$ but it is more flexible and portable and debugging is simpler with interpretation at run-time. The purely compilative has a

more efficient implementation of $\mathcal{M}_\mathcal{L}$ but more complex due to the difference between the languages and debugging is usually harder.

2.2 Implementation in firmware

It is the intermediate implementation between hardware and software, in which the abstract machine is inplemented as a microinterpreter and the program cycle is implemented using microinstruction invisible to normal users. The data structures and algorithms are realized as microprograms. This is more flexible than a pure hardware implementation.

2.3 Formal definition

A program can be seen as a function written in the language $\mathcal{L}$: $P^\mathcal{L} : \mathcal{D} \to \mathcal{D}$. If the program is $P^\mathcal{L}(i) = o$, $i \in \mathcal{D}$ is the input, $o \in \mathcal{D}$ is the output. Interpreters and compilers are program whose input and ouput are other programs.

2.3.1 Definition of an interpreter

An interpreter for a language $\mathcal{L}$ written in language $\mathcal{LO}$ implements an abstract machine $\mathcal{M}_\mathcal{L}$ and is the function:

$$\mathcal{I}_\mathcal{L}^{\mathcal{LO}} : (\mathcal{PR}^\mathcal{L} \times \mathcal{D}) \to \mathcal{D} \tag{2.1}$$

Where $\mathcal{PR}^\mathcal{L}$ is the set of programs $\mathcal{P}^\mathcal{L}$ written in the language $\mathcal{L}$. So $\forall i \in \mathcal{D}, \mathcal{I}_\mathcal{L}^{\mathcal{LO}}(\mathcal{P}^\mathcal{L}, i) = \mathcal{P}^\mathcal{L}(i)$ meaning that for any input the interpreter applied to the program and the input returns the same result as the program applied to the input. If $\mathcal{P}^\mathcal{L}(i)$ is calculated by the abstract machine $\mathcal{M}_\mathcal{L}$, $\mathcal{I}_\mathcal{L}^{\mathcal{LO}}(\mathcal{P}^\mathcal{L}, i)$ is calculated by $\mathcal{MO}_{\mathcal{LO}}$.

2.3.2 Definition of compiler

A compiler from the language $\mathcal{L}$ to the language $\mathcal{LO}$ written in language $\mathcal{LA}$, which implements $\mathcal{M}_\mathcal{L}$ on $\mathcal{MO}_{\mathcal{LO}}$ transforms a program $\mathcal{P}^\mathcal{L} \in \mathcal{PR}^\mathcal{L}$ into a program $\mathcal{P}^{\mathcal{L}=} \in \mathcal{PR}^{\mathcal{LO}}$. It is described by the function:

$$\mathcal{C}_{\mathcal{L},\mathcal{LO}}^{\mathcal{LA}} : \mathcal{PR}^\mathcal{L} \to \mathcal{PR}^{\mathcal{LO}} \tag{2.2}$$

Meaning that $\forall i \in D : \mathcal{P}_\mathcal{C}^{\mathcal{LO}} = \mathcal{C}_{\mathcal{L},\mathcal{LO}}^{\mathcal{LA}}(\mathcal{P}^\mathcal{L}) \Rightarrow \mathcal{P}_\mathcal{C}^{\mathcal{LO}}(i) = \mathcal{P}^\mathcal{L}(i)$, that describes the fact that for any input i, the compiled program applied to i returns the same result as the non compiled program applied to i. $\mathcal{P}^\mathcal{L}(i)$ is calculated by $\mathcal{M}_\mathcal{L}$, $\mathcal{P}_\mathcal{C}^{\mathcal{LO}}(i)$ is calculated by $\mathcal{MO}_{\mathcal{LO}}$ and the compiled program $\mathcal{C}_{\mathcal{L},\mathcal{LO}}^{\mathcal{LA}}(\mathcal{P}^\mathcal{L})$ is calculated by $\mathcal{MA}$

Capitolo 3

Nomi e ambiente

3.1 Nomi

Un nome è una sequenza di caratteri che denota qualcos altro, nei linguaggi di programmazione sono identificatori e il loro uso permette di indicare all'oggetto denotato. In un linguaggio di programmazione i nomi denotabili si dividono in:

- Definiti dall'utente, come variabili, parametri formali, procedure, tipi, tags, costanti e exxezioni.
- Primitivi: costanti, operazioni e tipi primitivi.

L'associazione di un nome all'oggetto denotato è chiamata binding. Nomi e oggetti sono differenti: il nome è una stringa di caratteri, mentre l'oggetto può essere qualsiasi cosa denotabile, a cui possono essere assegnati più nomi: "aliasing". Durante l'esecuzione di un programma lo stesso nome può denotare oggetti divesi.

3.1.1 Binding

I binding tra nomi e oggetti possono avvenire a diversi momenti:

- Fatti per design del linguaggio (i tipi primitivi).
- Durante la scrittura del programma, dove vengono definiti, ma non creati fino a che lo spazio in memoria per tale oggetto viene creato.
- Compile time: il compilatore alloca la memoria per tutte quelle strutture dati che possono essere staticamente processate.
- Runtime: tutte le associazioni precedentemente non avvenute, come il binding per variabili locali o puntatori a variabili allocate dinamicamente.

3.1.2 Gestione dei nomi in memoria

Esistono due tipi di gestioni della memoria nel salvataggio dei nomi:

- Statica: tutto avviene prima dell'esecuzione, fatta dal compilatore.
- Dinamica: avviene durante l'esecuzione, grazie ad opportune operazioni eseguite a runtime.

3.2 Ambienti

Un ambiente è l'insieme delle associazioni tra nomi e ogetti denotabili che esistono a runtime in un punto specifico del programma, in uno specifico momento dell'esecuzione. Il processo che crea un associazione in un ambiente è chiamato dichiarazione. Come già detto, in momenti diversi del programma lo stesso nome può essere associato a oggetti diversi e diversi nomi possono essere associati allo stesso oggetto.

3.2.1 Blocchi

Nei linguaggi di programmazione moderni l'ambiente è strutturato in blocchi, ovvero sezioni del programma che contiene dichiarazioni locali a quella regione. L'uso di nomi locali permette migliore chiarezza, ottimizza l'uso di memoria e permette la ricorsione. Blocchi intersecati devono essere per forza essere annidati. Una dichiarazione in un blocco è visibile nel blocco e nei blocchi annidati a meno che in questi ultimi ce ne sia un'altra con lo stesso nome che la mascheri.

3.2.2 Suddivisione dell'ambiente

L'ambiente in un blocco può essere diviso in:

- Ambiente locale, che contiene tutte le associazioni che avvengono all'entrata del blocco, come paramtri formali e variabili locali
- Ambiente non locale, ovvero tutte le associazioni ereditate da altri blocchi.
- Ambiente globale, quella parte dell'ambiente non locale comune a tutti i blocchi come variabili locali esplicitamente dichiarate, dichiarazioni nei blocchi esterni o ereditata da altri moduli.

3.2.3 Operazioni nell'ambiente

Nell'ambiente possono essere eseguite le seguenti operazioni:

- Binding;
- Utilizzo di un nome;
- Disattivazione di un binding a causa della creazione di un altro che lo maschera;
- Riattivazionie di un binding all'uscita dal blocco che conteneva la maschera;
- Distruzione di un associazione all'uscita dal blocco che conteneva la variabile locale;

Il periodo di vita di un oggetto può non corrisponde al periodo di vita di una sua associazione: può essere più lunga (passaggio per reference) o più corta, aliasing e distruzione attraverso uno dei due nomi.

3.3 Scopes

Le regole di visibilità sono considerabili come segue: ogni dichiarazione locale è visibile in tutti i blocchi annidati a meno che non sia mascherata. Sorge un problema con le funzioni in quanto viene eseguito codice presente in altre aree rispetto a dove è stato definito. Per questo motivo si deve fare una scelta tra:

- Static scoping: viene scelta la variabile che include sintatticamente la funzione, ovvero nel blocco che la include testualmente.
- Dinamic scoping: viene scelta la variabile nel blocco che viene eseguito immediatamente prima della funzione, ovvero nel blocco più recentemente attivato e non ancora disattivato.

3.3.1 Confronto tra static e dinamic scoping

Nello static scoping tutte le informazioni sono incluse nello stesso testo, le associazioni sono derivabili a compile time, sono indipendenti, è più facile da implementare e più efficiente. Nel dinamic scoping le informazioni sono derivate a runtime, risulta spesso in programmi difficili da leggere, più difficile da implementare e meno efficiente.

3.3.2 Determinare l'ambiente

L'ambiente è determinato dalle regole di scoping, dalle visibilità di variabili: in molti linguaggi una variabile esiste solo dopo che è stata dichiarata, regola che va rilassata sulle funzioni e sui tipi per permettere la mutua ricorsione, o grazie a definizioni incomplete.

Capitolo 4

Gestione della memoria

Ci sono tre tipi principali per gestire la memoria:

- Statico, in cui la memoria viene allocata durante la compilazione.
- Dinamico, in cui la memoria viene allocata a runtime, si divide in:
  - Stack, in cui gli oggetti sono allocati secondo una FIFO.
  - Heap, in cui ogni oggetto può essere allocato e disallocato ad ogni momento.

4.1 Allocazione statica

Utilizzando l'allocazione statica un oggetto ha un indirizzo assoluto che mantiene per tutta la durata dell'esecuzione.Viene solitamente utilizzata per salvare variabili
globali, variabili locali in assenza di ricorsione, costanti determinate a runtime e tabelle per garbage collection e type-checking. Presenta un problema per la ricorsione, in quanto salvando sempre nello stesso indirizzo i parametri attuali per una funzione questi vengono persi dopo la chiamata ricorsiva.

4.2 Allocazione dinamica

4.2.1 Stack

Per ogni istanza a runtime di un sottoprogramma, come in ogni blocco, esiste un record di attivazione (frame) che contiene le informazioni dell'istanza. La struttura dati naturale per gestire i record è una coda LIFO, uno stack può essere utilizzato anche in linguaggi senza ricorsione per ridurre il consumo di memoria.

4.2.1.1 Record di attivazione

Il record di attivazione consiste di tre parti:

1. Puntatori alla catena dinamica.
2. Variabili locali.
3. Risultati intermedi.

L'implementazione dell'allocazione dinamica di uno stack utilizza una sequenza di chiamate, un prologo, un epilogo e una sequenza di ritorno. L'indirizzo del record di attivazione non è conosciuto a compile time, e il suo puntatore punta al record di attivazione del blocco attivo. Le informazioni nel record di attivazione sono accessibili grazie ad un offset determinato staticamente, grazie alla somma con l'indirizzo del record di attivazione, operazione compiuta attraverso una sola operazione della macchina.

4.2.1.2 Push e Pop

All'entrata in un blocco, viene creato un nuovo linck dinamico (puntatore al record precedente sullo stack) dal nuovo blocco al blocco precedente, il record di attivazione viene settato a puntare al nuovo record. All'uscita dal blocco viene elimitato il puntatore al record e settato un nuovo record di attivazione alla cima dello stack.

4.2.1.3 Necessità dello stack

Per la gestione di blocchi annidati non è uno stack, il compilatore può analizzare tutti i blocchi e allocare memoria per tutte le variabili, causando un maggior consumo di memoria, ma rimanendo più efficiente non perdendo risorse per la gestione dello stack.

4.2.2 Heap

Lo heap è una regione di memoria in cui blocchi e sottoblocchi possono essere allocati e disallocati in momenti arbitrari. Questo processo si rende necessario quando si trova un'allocazione di memoria esplicita a runtime, ci sono ogetti di dimensione variabile e il loro tempo di vita non è LIFO. La gestione dello heap presenta dei problemi: si necessita di allocare efficientemente lo spazio (bisogna evitare la frammentazione) e tenere conto della velocità di accesso.

4.2.2.1 Blocchi di grandezza fissa

In questo caso lo heap è diviso in blocchi di grandezza fissa all'inizio linkati in una free list. Durante l'allocazione uno o più blocchi vengono eliminati dalla free list e nella deallocazione vi sono reinseriti.

4.2.2.2 Blocchi a grandezza variabile

Lo heap contiene inizialmente un blocco singolo. Durante l'allocazione viene ricercato un blocco della dimensione appropriata che viene poi ripristinato al blocco libero durante la deallocazione.

4.2.2.3 Gestione dello heap

Frammentazione interna: se si necessita di allocare uno spazio di dimensione X ma la dimensione del blocco è $Y > X$, viene sprecato $Y - X$ spazio.
Frammentazione esterna: lo spazio necessario è disponibile ma inaccessibile in quanto diviso in blocchi di dimensione troppo piccola.

4.2.2.4 Gestione della free list

Lista singola: all'inizio esiste un blocco che occupa tutto lo heap ed ad ogni richiesta di allocazione viene ricercato un blocco della dimensione appropriata secondo uno dei due possibili paradigmi:

- Best fit: ricercato il blocco più piccolo che può contenere l'oggetto. Buon utilizzo della memoria ma più lento.
- First fit: viene ricercato il primo blocco libero. Veloce con pessimo utilizzo della memoria.

Se il blocco è molto più largo di quanto si necessita è diviso in due pezzi e il blocco inutilizzato è riagiunto alla free list. L'allocazione è lineare nel numero dei blocchi liberi.
Liste multiple: che possono essere implementate staticamente; dinamicamente in due modi:

- Buddy system: con k liste con la k-esima lista con blocchi di dimensione 2^k:
  - Se la dimensione di 2^k è richiesta ma non è disponibile si alloca un blocco di dimensione 2^{k+1} diviso in due.
  - Quando un blocco di dimensione 2^k viene deallocato, se esiste un'altra metà libera viengono uniti.
- Sistema di Fibonacci: analogo al Buddy System ma con l'utilizzo della sequenza di Fibonacci.

4.3 Implementazione delle regole di scoping

4.3.1 Scoping statico

4.3.1.1 Catena statica

Viene determinato il primo activation record in cui la variabile si trova, l'accesso viene eseguito tramite l'offset con questo record di attivazione. Questo può essere determinato dal link dinamico visto sopra o attraverso un link statico al blocco che contiene il testo del blocco corrente. Se il link dinamico dipende dalla sequenza di esecuzione, il link statico dipende dall'annidamento statico della dichiarazione della funzione. Il link statico è determinato dal chiamante che possiede le informazioni riguardante l'annidamento statico determinato dal compilatore e il proprio record di attivazione. Se il chiamato è automaticamente incluso nel chiamante, questo passa il proprio static pointer al chiamato, altrimenti il puntatore statico viene trovato dopo k passaggi nella catena statica.

4.3.1.2 Display

Viene utilizzato per ridurre il costo di far scorrere la catena ad una costante: è un array contenente la catena in cui l'i-esimo elemento è un puntatore al record di attivazione al sottoprogramma a livello di annidamento i. Se il sottoprogramma si trova a livello i e l'oggetto in uno scope esterno a livello h, questo può essere trovato nel display al livello $i - h$. Quando il chiamante chiama una funzione a livello di nesting i salva il valore di $display[i]$ nel proprio record di attivazione e setta il puntatore ad una copia del suo nuovo record di attivazione. Implementazioni moderne raramente utilizzano questo metodo in quanto non si trovano spesso catene statiche più lunghe di 3.

4.3.2 Scoping dinamico

L'associazione tra nomi e oggetti dipende dal flusso di controllo a runtime e dall'ordine in cui i sottoprogrammi sono chiamati. Di base l'associazione è determinata dall'ultima associazione creata che non è stata ancora distrutta. I nomi vengono salvati nel record di attivazione e i nomi vengono chiamati attraverso uno stack.

4.3.2.1 A-list

Le associazioni sono salvate in una struttura apposita gestita come uno stack. È semplice da implementare, tutti i nomi sono listati esplicitamente. I costi di gestione sono rappresentati dall'accesso uscita di un blocco e il conseguimento inserimento o rimozione da uno stack. Il costo di accesso è linare con la profondità della lista.

4.3.2.2 Central table of reference (CRT)

La tavola salva tutti i nomi distinti del programma staticamente con accesso in tempo costante o grazie all'utilizzo di funzioni di hash. Ogni nome ha una lista di associazioni: con come primo il più recente, seguito dagli inattivi. È più complesso di una A-list ma utilizza meno memoria in quanto tutti i nomi sono salvati un'unica volta e se sono utilizzati staticamente non sono necessari, il costo di gestione è rappresentato dall'entrata e uscita di un blocco che rende necessario la gestione della lista di tutti i nomi presenti nel blocco. Il tempo di accesso è costante.

Capitolo 5

Struttura di controllo

5.1 Espressioni

Le espressioni sono entità sintattiche la cui valutazione produce un valore o non termina.

5.1.0.1 Notazione infissa

Necessita la conoscenza delle regole di precedenza e dell'associatività, necessario l'uso di parentesi spesso la sua valutazione non è immediata.

5.1.0.2 Notazione postfissa

Non necessita regole di precedenza, associatività o di parentesi. La valutazione avviene come uno stack: si legge un valore e lo si pone sullo stack, se tale valore è un operatore lo si applica ai simboli sulla cima dello stack, li si poppano e li si sostituisce con il risultato. Se l'input non è vuoto si ripete.

5.1.0.3 Notazione prefissa

Simile a quella postfissa ma necessita di un contatore del numero di operandi.

5.1.1 Ordine di valutazione

Le valutazioni sono rappresentate internamente da alberi che il compilatore utilizza per produrre codice che valuta l'espressione. L'ordine di valutazione è importante a causa di effetti collaterali, finite arithmetic, operatori indefiniti e ottimizzazione.

5.1.1.1 Operatori indefiniti

Short circuit evaluation: o lazy evaluation in cui gli operandi sono valutati solo se non è già stato possibile definire univocamente il risultato dell'espresisone.
Eager evaluation: un tipo di valutazione in cui tutti gli operandi sono valutati prima di essere passati all'operatore.

5.2 Comandi

Sono entità sintattiche che non hanno necessariamente un risultato: possono avere dei side effects.

5.3 Variabili

Le variabili nei linguaggi imperativi sono dei contenitori di valori con un nome che possono essere cambiati attraverso un comando come l'assegnamento.

5.3.1 Modelli diversi

Nei liguaggi funzionali una variabile denota un valore che non può essere modificato, mentre nei linguaggi logici può essere modificata solo attraverso istantiazione, nei linguaggi ad oggetti una variabile è una reference ad un oggetto, simile ad un puntatore di cui non si può accedere direttamente al valore.

5.4 Iterazione

L'iterazione può essere indeterminata con cicli con controlli logici o determinata con controllo numerico.

5.5 Ricorsione

Una funzione ricorsiva è definita in termini di sè stessa, del tutto simile all'induzione matematica. La ricorsione può esistere in ogni linguaggio che permette alle funzioni di chiamare sè stesse e une gestione dinamica della memoria. Ogni funzione ricorsiva può essere espressa in termini di iterazione e viceversa. La ricorsione può essere ottimizzata per essere efficiente come l'iterazione con la tail recursion, in cui la chiamata ricorsiva è l'ultimo elelmento della funzione, in quanto in questo modo è necessario un solo record di attivazione.

Capitolo 6

Astrazione dei controlli

6.1 Funzione di ordine superiore

Dei linguaggi permettono il passaggio di funzioni come parametro o il ritorno di funzioni per un'altra funzione. Per gestire l'ambiente di questi casi si può usare un puntatore al record di attivazione nello stack nel caso di funzioni come parametri. Nel caso di funzioni come risultato si deve mantenere un record di attivazione della funzione esterno allo stack.

6.1.1 Regole di binding

Quando una funzione è passata come parametro questo crea una reference tra il nome del parametro formale e il parametro attuale. L'ambiente non locale utlilzzato dipende dal tipo di binding utilizzato.

6.1.1.1 Deep binding

In questo caso viene utilizzato l'ambiente esistente al momento della creazione del link, utilizzato sempre in caso di scoping statico.

6.1.1.2 Shallow binding

In questo caso viene l'ambiente esistente al momento della chiamata, può essere utilizzato con lo scoping dinamico.

6.1.1.3 Chiusura

Sia il link al codice della funzione che del suo ambiente sono passati come parametro, la procedura passata come parametro alloca il suo record di attivazione e considera il puntatore sulla catena statica della chiusura. In caso di utiilizzo di chiusura si perdono le proprietà LIFO dello stack, pertanto non esiste deallocazione automatica, i record di attivazione vengono salvati nello heap con catene statiche o dinamiche che li connettono e quando necessario viene chiamato un garbage collector.

6.2 Eccezioni

Vengono utilizzate quando si riscontra un problema che rende impossibile continuare la computazione, si esce pertanto dal blocco, si passano i dati con un jump e si ritorna il controllo al punto di

attivazione più recente. I record di attivazione non più necessari sono deallocati e le risorse non più necessarie sono liberate. Le eccezioni si basano su due costrutti: la dichiarazione del gestore delle eccezionie il comando che genera l'eccezione. Se l'eccezione non è gestita dalla funzione corrente questa termina e l'eccezione è generata ancora alla funzione chiamante, alla fine si trova il gestore dell'eccezione o uno di default. I frame corrispondenti sono rimossi dallo stack e i valori sono ripristinati. Ogni routine contiene un handler nascosto che ripristina i valori e propaga l'eccezione.

6.2.1 Implementazione delle eccezioni

All'inizio di ogni blocco protetto viene inserita sullo stack del chiamante, quando viene generata si rimuove il chiamante dallo stack e si controlla se è giusto, altrimenti si ripete. È inefficiente se non si trova un handler, una soluzione migliore è la tavola degli indirizzi.

Capitolo 7

Strutture dati e tipi

7.1 Tipi

Un tipo è una collezione di valori omogenei e univocamente definiti collegati a un insieme di operatori su quei valori. Sono determinati dagli specifici linguaggi di programmazione. Sono utilizzati per organizzare le informazioni, riconoscere certi errori e permettere delle ottimizzazioni.

7.1.1 Type system

È l'insieme dei tipi predefiniti, dei meccanismi che definiscono nuovi tipi e meccanismi di controllo di equivalenza, compatibilità e inferenza, specifica se un tipo è statico o dinamico. Un sistema è type safe se a runtime non ci possono essere errori non rilevati causati da errori di tipo.

7.1.2 Classificazione

Tipi ordinali o discreti: sono booleani, iteri e caratteri, intervalli e enumerazioni, ogni valore ha un predecessore e un successore, possono essere utilizzati come iterazione o come indici di un array. Tipi scalari: i tipi ordinali e i reali, hanno una diretta rappresentazione nell'implementazione, non sono composti di aggregazione di altri valori.

7.1.2.1 Enumerazioni

Sono ordinazioni di nuovi valori, rendono i programmi più facili da leggere.

7.1.2.2 Intervalli

Sono intervalli di valori di un tipo ordinabile, sono rappresentati come il tipo base.

7.2 Tipi strutturati, non scalari

- Sono una collezione di campi di tipi diversi, selezionati in base al nome. Questo record può essere variabile in base al numero di campi attivi.
- Array, funzione da un indice scalare a un altro tipo, array di caratteri sono stringhe con particolari operazioni.

- Sets, sottinsiemi di un tipo base.
- Puntatori, reference a oggetti di altro tipo.

7.2.1 Gestione in memoria dei record

Possono essere un insieme di campi salvati sequenzialmente, allinati a parola di confine o non allineati, con maggiore costo di accesso. In un record variabile ci sono possibilità esclusive per ogni record.

7.2.2 Array

Gli array sono collezioni di dati omogenei, possono essere multidimensionali. L'operazione principale è la selezione attraverso l'indice di un elemento. Alcuni linguaggi permettono lo slicing, ovvero la divisione dell'array in sottoarray. Gli array multidimensionali sono comunemente salvati in ordine di riga, e gli elementi sono tutti contigui.

7.2.2.1 Localizzare un elemento

Sia $A[S_1][S_2][S_3]$ un array multidimensionale, la dimensione di S_3 è la dimensione del tipo, la dimensione di $S_2 = numero\ elementi\ S_3 \cdot S_3$, $S_1 = numero\ elementi\ S_2 \cdot S_2$. Ora un elemento $A[i][j][k]$ in un array che comincia ad α è data da $i \cdot S_1 + j \cdot S_2 + k \cdot S_3 + \alpha$.

7.2.2.2 Equivalenza e compatibilità

Due tipi sono equivalenti se ogni oggetto di T è anche oggetto di P e viceversa, sono compatibili se ogni oggetto T può essere utilizzato nello stesso contesto di un oggetto di tipo P.

7.3 Polimorfismo

Un valore singolo può avere più tipi.

7.3.1 Overloading

In questo caso lo stesso simbolo ha differenti significati in base al contesto in cui è utilizzato, viene risolto a compile time dopo l'inferenza dei tipi.

7.3.2 Polimorfismo parametrico

Un valore ha polimorfismo parametrico se il valore ha un infinito possiile numero di tipi, ottenuti grazie all'istanziazione di uno schema generale. È una singola definizione applicata a tutte le istanze di un tipo generale. Implicito in ML.

7.3.3 Polimorfismo di sottotipo

Simile a quello esplicito, ma non tutti i tipi possono essere utilizzati: un valore lo possiede se ha un numero infinito di possibili tipi ottenibili sostituendo per un parametro tutti i sottotipi del tipo dato.

Capitolo 8

Astrazione dei dati

Le compomenti di un programma sono funzioni, strutture dati e moduli. L'interfaccia è composta dai tipi e operandi definiti in una componente visibili dall'utilizzatore. La specificazione è la funzionalità di una componente, espressa dalle funzionalità visibilia attraverso l'interfaccia, l'implementazione sono strutture dati interni alla componente, non necesariamente visibilia all'utilizzatore. L'implementazione è spesso nascosta dall'utente che non può ne vedere nè utilizzare per permettere cambi ad essa senza dover necessariamente modificare in alcun modo il codice creato dall'utilizzatore.

Capitolo 9

Lambda calcolo

Il lambda calcolo è la formalizzazione matematica del paradigma di programmazione funzionale, viene espresso come $\lambda x.e$, dove x è una variabile legata e e un espressione. Un espressione del lambda calcolo è un nome, una funzione o un applicazione di espressione a espressioni.

9.1 Sintassi

Siano x un identificatore, e un'espressione, allora si può dire: $e = x$, o $\lambda x.e$ o $e\ e$.

9.1.1 Astrazione

Il processo di astrazione $\lambda x.e$ lega la variabile x nell'espressione e. Sostituiendo con λy e tutte le occorrenze di x con y in e la formalizzazione non subisce alcuna variazione.

9.1.1.1 Applicazione

L'applicazione è il processo di sostituzione di un espressione con un altra senza cattura (si intende cattura il processo di cambio di stato di una variabile da libera a legata).

9.2 Variabili libere e legate

Una variabile x è detta legata da $\lambda x.$, è libera se non è legata a nessun λ. Ovvero Siano $F_v(e)$ le variabili libere e $B_v(e)$ le variabili legate nell'espressione e. Si consideri x un identificatore, $F_v(x) = \{x\}$ e $B_v(x) = \emptyset$, ovvero se un'espressione è composta da una sola variabile questa è libera. $F_v(e_1e_2) = F_v(e_1) \cup F_v(e_2)$ e $B_v(e_1e_2) = B_v(e_1) \cup B_v(e_2)$, ovvero la composizione di due espressioni non cambia lo stato delle loro variabili.

9.2.1 Definizione di variabile legata

$F_v(\lambda x.e) = F_v(e) \backslash \{x\}$ e $B_v(\lambda x.e) = B_v(e) \cup \{x\}$, ovvero l'operatore λ di astrazione rimuove una variabile dell'espressione dalle sue variabili libere per aggiungerle alle variabili legate.

9.3 Sostituzione

Si scrive come $e[\frac{e'}{x}]$, che vuole dire: sostituisci x con e'.

9.3.1 Definizione per valore

- Se x è un identificatore $x[\frac{e'}{x}] = e'$.
- Se $x \neq y$, $y[\frac{e'}{x}] = y$

9.3.2 Definizione per applicazione

$(e_1 e_2)[\frac{e'}{x}] = (e_1[\frac{e'}{x}]e_2[\frac{e'}{x}])$.

9.3.3 Definizione per astrazione

- Se $x \neq y$ e $y \notin F_v(e')$, $(\lambda y.e)[\frac{e'}{x}] = (\lambda y.e[\frac{e'}{x}])$, metodo semplice per evitare la cattura di y (e sostituita con un espressione che non dipende da y).
- Se $x = y$, $(\lambda y.e)[\frac{z}{x}] = (\lambda y.e)$, in quanto sostituire la variabile legata da λ non cambia nulla.
- Se $x \neq y$ e $y \in F_v(e')$, per evitare cattura si deve cambiare il nome della variabile legata da λ, in quanto una funzione non dipende dal nome del parametro formale, in questo modo non ci sono fenomeni di aliasing tra variabili libere in e' e variabili legate in λy..

9.4 Equivalenza tra espressioni

9.4.1 Alfa equivalenza

Due espressioni e_1 ed e_2 si dicono equivalenti quando differiscono solo per il nome di variabili legate, ovvero data una variabile y non presente in e, $\lambda x.e \equiv \lambda x.e[\frac{y}{x}]$ in quanto cambia λx in λy e tutte le occorrenze di x in e con y. Due espressioni sono α-equivalenti se una si ottiene dall'altra sostituendo una sua parte con una α-equivalente.

9.4.2 Beta riduzione

Si intende con beta riduzione il processo di semplificazione delle funzioni, ovvero $(\lambda x.e)e' = e[\frac{e'}{x}]$ in presenza di più espressioni riducibili c'è confluenza. La beta riduzione introduce una relazione tra espressioni non simmetrica. Pertanto non è una relazione di equivalenza.

9.4.2.1 Beta equivalenza

È la chiusura transitiva e riflessiva della beta riduzione. Ovvero $e_1 =_\beta e_2$ allora esiste una catena di riduzioni che da e_1 porta a e_2.

9.4.2.2 Forme normali

È un espressione che non contiene redex e alla quale pertanto non si possono avere beta riduzioni. Una beta riduzione può terminare in una forma normale o continuare all' infinito.