Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

Ultima modifica: Wednesday 2nd August, 2017- 15:57

Linguaggi di Programmazione

Riassunto dei principali argomenti

Candidati:
Davide Bianchi
Matteo Danzi

Anno Accademico 2016/2017

Indice

1 Introduzione

Questa dispensa è scritta sulla base del programma di Linguaggi dell'anno accademico 2016/2017 riguardante i seguenti argomenti: poset, reticoli, punti fissi, macchine astratte, cenni di semantica operazionale e denotazionale. Sono omesse le seguenti sezioni: language of commands e le parti di semantica statica e dinamica dei vari costrutti.
Il codice della dispensa, scritta in LaTeX, è reperibile seguendo il link:
`https://github.com/alx79/dispense-univr.git`.

Chiunque volesse contribuire, segnalare errori (sicuramente presenti) o modificare il testo è ben accetto, è sufficiente scrivere ai gestori del repository GitHub (vedi fine documento).

2 Preliminari matematici

2.1 Posets

Un poset (*partially ordered set*, indicato con $\mathcal{P}$) è un insieme dotato di ordine parziale su tutti i suoi elementi. Un poset possiede alcuni elementi particolari, quali:

- upper bound: un elemento $M \in \mathcal{P}$ tale che $\forall x \in \mathcal{S}$, dove $\mathcal{S} \subseteq \mathcal{P}$, $x \leqslant M$;
- lower bound: un elemento $m \in \mathcal{P}$ tale che $\forall x \in \mathcal{S}$, dove $\mathcal{S} \subseteq \mathcal{P}$, $x \geqslant m$;

NOTA: non è necessario che m e M appartengano a $\mathcal{S}$.
Inoltre, dato un poset $\mathcal{P}$ e un $\mathcal{X} \subseteq \mathcal{P}$, sono definiti:

- least upper bound (notazioni: $\mathrm{lub}X, \mathrm{sup}X, \bigvee X, \sqcup X$): un elemento $x \in \mathcal{X}$ tale che:
  - x è upper bound di $\mathcal{X}$
  - x è il più piccolo degli upper bound
- greatest lower bound (notazioni: $\mathrm{glb}X, \mathrm{inf}X, \bigwedge X, \sqcap X$): un elemento $x \in \mathcal{X}$ tale che:
  - x è lower bound di $\mathcal{X}$
  - x è il più grande dei lower bound

2.2 Reticoli

Prima di definire un reticolo completo, è necessario definire:

- Join-semi lattice: un poset in cui due elementi qualsiasi hanno un least upper bound
- Meet-semi lattice: un poset in cui due elementi qualsiasi hanno un greatest lower bound

Un reticolo è quindi un reticolo che è sia join-semi lattice sia meet-semi lattice. Un caso particolare di reticolo è il reticolo completo, in cui ogni sottoinsieme $\mathcal{X} \subseteq \mathcal{P}$ ha un least upper bound.

Un reticolo completo è definito come un poset $\langle P, \sqsubseteq \rangle$ con la differenza che un qualsiasi sottoinsieme $X \subseteq P$ ha un least upper bound $\sqcup X$ in P. Inoltre un reticolo completo ha un elemento *infimo* $\bot = \sqcup\emptyset$ e un elemento *supremo* $\top = \sqcup\mathcal{P}$

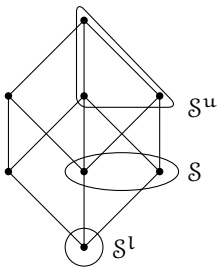
2.3 Esempi

Esempi di posets:

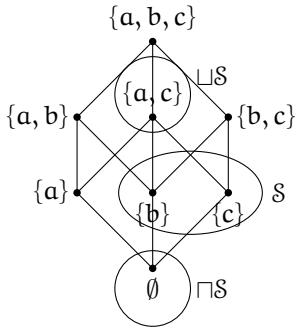
es.1: $\langle \mathbb{N}, \leqslant \rangle$ è un poset $(\forall x, y \in \mathbb{N} : x \leqslant y \Leftrightarrow \exists z \in \mathbb{N} : x + z = y)$

es.2: $\langle \mathbb{N}, \geqslant \rangle$ è un poset $(\forall x, y \in \mathbb{N} : x \geqslant y \Leftrightarrow x \leqslant y)$

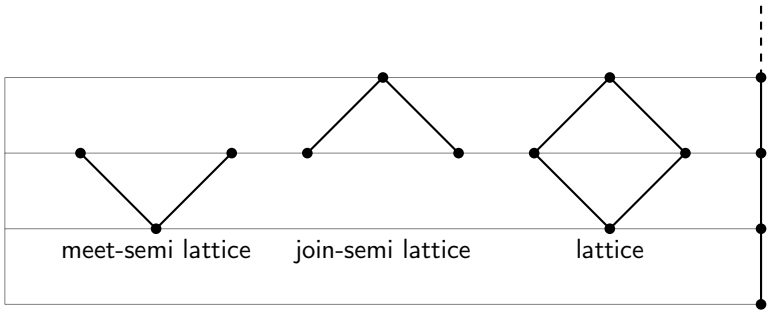
Esempi di upper e lower bound:



Esempi di greatest lower bound:



Esempi di reticoli:



2.4 Teoria di punto fisso

Un punto fisso di un operatore f su un insieme L è un elemento $x \in L$ tale che $f(x) = x$.
Definito come **trace semantics** l'insieme delle tracce di esecuzione di un programma, finite o infinite, un prefisso della traccia è un sequenza di stati attraverso i quali la traccia passa di lunghezza finita.

Applicando la definizione di punto fisso all'insieme dei prefissi di una traccia, otteniamo l'equazione $\mathrm{Pref} = f(\mathrm{Pref})$ che in generale ammette soluzioni multiple. Si deduce che i punti fissi di un operatore possono essere un numero variabile, anche infiniti ($\lambda x \cdot x$).

Esistono ovviamente anche punti fissi particolari, quali:

- least fixpoint: se esiste, $\forall x \in L : (f(x) = x) \Rightarrow (\mathrm{lfp}(f) \sqsubseteq x)$
- greatest fixpoint: se esiste, $\forall x \in L : (f(x) = x) \Rightarrow (\mathrm{gfp}(f) \sqsupseteq x)$

Tramite la definzione di punto fisso sono definiti anche i seguenti insiemi:

- insieme dei fixpoints: $\mathrm{fp}(f) = \{x \in L | f(x) = x\}$
- insieme dei pre-fixpoints: $\mathrm{prefp}(f) = \{x \in L | x \sqsubseteq f(x)\}$
- insieme dei post-fixpoint: $\mathrm{postfp}(f) = \{x \in L | f(x) \sqsubseteq x\}$

3 Macchine astratte

3.1 Introduzione

Il modello fondamentale di una macchina astratta fa riferimento al modello della macchina di Von Neumann. Tale macchina da un quadro generale di una qualsiasi macchina, che svolge fondamentalmente sempre lo stesso ciclo di fasi:

1. Lettura delle istruzioni
   (a) Lettura dell'istruzione da eseguire
   (b) Decodifica dell'istruzione

2. Fase di esecuzione
   (a) Lettura valore degli operandi
   (b) Esecuzione dell'istruzione
   (c) Memorizzazione del risultato

Una macchina astratta è suddivisa in vari livelli, quali:

- **Livello applicativo:** contiene gli applicativi che si interfacciano direttamente con l'utente ed è strutturato con linguaggi ad alto livello, basi di dati e interfacce grafiche;
- **Livello OS:** gestisce processi, programmi, memoria, operazioni di I/O e si interfaccia direttamente col kernel;
- **Livello ASM:** contiene i codici mnemonici per gestire i registri della CPU, è implementato generalmente in Assembly;
- **Livello HW:** livello fisico con i componenti hardware.

Un **Linguaggio L** è definito da una sintassi e da una semantica. La sintassi è un insieme finito di costrutti che permette di creare dei programmi. Il linguaggio è supportato da una determinata **macchina astratta** M_L, ovvero un certo insieme di strutture dati e algoritmi che permettono di memorizzare ed eseguire programmi di L.

Una macchina astratta è costruita su una certa quantità di memoria, che dovrà memorizzare dati e programmi, e un interprete, che esegue le istruzioni dei programmi. Ogni macchina astratta ha tre possibili realizzazioni:

- **Hardware:** usata per macchine a basso livello o sistemi embedded, è dotato della velocità massima ma essendo una macchina specifica ha flessibilità nulla;
- **Firmware:** struttura basata su un insieme di microprogrammi, è molto veloce e poco più flessibile di una macchina puramente hardware;
- **Software:** strutture dati e algoritmi realizzati tramite programmi veri e propri, le performance calano, ma vengono compensate da una flessibilità maggiore e il fatto che può essere realizzata su una qualsiasi macchina ospite.

3.2 Interprete

Ogni interprete ha delle tipologie di operazioni e modalità di lavoro che sono indipendenti dal linguaggio che riconosce, quali:

- **Elaborazione dei dati primitivi:** dati rappresentabili direttamente dalla memoria
- **Control Flow:** gestione del flusso di esecuzione delle istruzioni, che non è sempre sequenziale
- **Data flow:** recupero dei dati necessari all'esecuzione delle istruzioni
- **Gestione della memoria:** gestione dell'allocazione della memoria per dati e programmi da memorizzare

L'interprete interpreta un **linguaggio macchina**, ovvero il linguaggio L_M che ha come stringhe legali tutte le stringhe interpretabili da M. Dal momento che una macchina astratta ha come unico linguaggio legale M_L, implementare un linguaggio significa realizzare una macchina astratta M per L.
NB: esistono infinite macchine astratte per uno stesso linguaggio, la differenza sta nel come l'interprete viene realizzato e nelle strutture dati utilizzate.

3.3 Implementazione del linguaggio

Fondamentalmente per l'implementazione di un linguaggio sono dati:

- Il linguaggio da implementare
- Una macchina astratta $\mathrm{Mo}_{\mathrm{Lo}}$, ovvero la macchina ospite con il suo linguaggio Lo

Quello che bisogna implementare è L su $\mathrm{Mo}_{\mathrm{Lo}}$, traducendo quindi L in Lo. Per realizzare ciò esistono due modalità:

- Una traduzione implicita (*interpretativa*), basata sulla simulazione di M_{L} mediante programmi in Lo
- una traduzione esplicita (*compilativa*), realizzata traducendo programmi in L direttamente in programmi in Lo.

Un po' di notazione. Siano:

- $\mathrm{Prog}^{\mathrm{L}}$ l'insieme dei programmi scritti in L;
- D l'insieme dei dati in I/O (indifferente quale dei due);
- $\mathrm{P}^{\mathrm{L}} : \mathrm{D} \to \mathrm{D}$ una funzione parziale ricorsiva tale che $\mathrm{P}^{\mathrm{L}}(\mathrm{input}) = \mathrm{output}$, con $\mathrm{P}^{\mathrm{L}} \in \mathrm{Prog}^{\mathrm{L}}$.

3.4 Soluzione interpretativa e compilativa

La soluzione interpretativa prevede che si realizzi l'interprete di M_{L} tramite un insieme di istruzioni di Lo, ovvero realizzando un $\mathrm{I}^{\mathrm{Lo,L}}$, scritto in Lo, che interpreta la istruzioni di L (I viene quindi eseguito su Mo). In fin dei conti, un interprete I è definibile come segue:

$$\mathrm{I}^{\mathrm{Lo,L}} : (\mathrm{Prog}^{\mathrm{L}}) \times \mathrm{D} \to \mathrm{D}$$

tale che

$$\mathrm{I}^{\mathrm{Lo,L}}(\mathrm{P}^{\mathrm{L}}, \mathrm{D}) = \mathrm{P}^{\mathrm{L}}(\mathrm{D})$$

La soluzione compilativa invece traduce esplicitamente un programma in L in uno scritto in Lo. La traduzione è eseguita da un **compilatore**. Nel caso della compilazione la fase di esecuzione è completamente separata da quella di traduzione. Dal punto di vista matematico, un compilatore realizza la funzione:

$$\mathrm{C}^{\mathrm{L,Lo}} : \mathrm{Prog}^{\mathrm{L}} \to \mathrm{Prog}^{\mathrm{L}}$$

tale che, dato un $\mathrm{P}^{\mathrm{L}} \in \mathrm{Prog}^{\mathrm{L}}$,

$$\mathrm{C}^{\mathrm{L,Lo}}(\mathrm{P}^{\mathrm{L}}) = \mathrm{P}^{\mathrm{Lo}}$$

e

$$\mathrm{P}^{\mathrm{L}}(\mathrm{input}) = \mathrm{P}^{\mathrm{Lo}}(\mathrm{input})$$

Tra le due opzioni viene solitamente scelta una soluzione ibrida (es. Java), in quanto compilazione e interpretazione pura presentano svantaggi che si possono in parte ridurre usando questo tipo di metodo.

In alcuni casi è necessario caricare strutture dati e sottoprogrammi sulla macchina ospite per poter permettere l'esecuzione del codice prodotto dal compilatore (in particolare quando l'interprete della macchina intermedia e quello della macchina ospite coincidono). Questo insieme di dati è detto supporto a tempo di esecuzione (*runtime support*). Nei linguaggi più vecchi era più che altro una rarità, nei moderni linguaggi ad alto livello questo scenario si presenta con quasi tutti i costrutti del linguaggio.

3.5 Specializzazioni

All'interno di un linguaggio si possono inoltre aggiungere delle trasformazioni per motivi di performance, dette **specializzazioni**. Una specializzazione prende un $P(x, y)$ con x noto, e lo trasforma in $P_x(y)$ dove le computazioni relative ad x sono state svolte prima della computazione. Uno specializzatore è quindi un programma che realizza la seguente funzione:

$$Spec_L : (Prog^L \times D) \to Prog^L$$

tale che, dati un $P^L \in Prog^L$ e $d \in D$,

$$Spec_L(P^L, d) = P_d$$

e

$$I_L(P_L(d, Y)) = I_L(P_d, Y)$$

dove $Y \in D$ e I_L è l'interprete per L.

4 Semantica operazionale

4.1 Descrizione di un linguaggio

La descrizione di un linguaggio avviene su tre dimensioni:

1. **Sintassi**: È l'insieme delle regole che specificano come formare frasi ben fatte, come i caratteri/-sequenze di caratteri devono essere raggruppate per formare *token*.
   Un *token* è un blocco di testo categorizzato costituito da caratteri indivisibili detti *lessemi*, frequentemente definiti come espressioni regolari che vengono comprese e analizzate da un analizzatore lessicale (*lexer*).
   Nel gergo dei linguaggi di programmazione si definisce sintassi tutto ciò che è definito dalla grammatica, il rimanente costituisce la semantica. La definizione di grammatica comprende:

   (a) Alfabeto (simboli ammessi).
   (b) Descrizione lessicale (sequenza di simboli corretti che formano parole).
   (c) Descrizione di sequenze di parole che costituiscono frasi legali del linguaggio.

   La formalizzazione della sintassi di un linguaggio di programmazione avviene su tre possibili strutture:

   (a) Grammatiche libere dal contesto
   (b) Derivazioni e linguaggi
   (c) Alberi di derivazione

2. **Pragmatica**: È lo studio delle relazioni tra la lingua ed il contesto, che sono fondamentali per la comprensione della lingua stessa. Si analizza in che modo frasi corrette e sensate vengono usate. Frasi con lo stesso significato possono essere usate in modo diverso da utenti diversi a seconda del contesto. La pragmatica non viene definita a priori una volta sola, ma evolve con l'uso che viene fatto del linguaggio, tende più alle metodologie di progettazione del software fornisce un adeguato stile di programmazione.

3. **Semantica**: Riguarda la relazione tra segni e significato, si tratta di attribuire significato alle frasi e quindi **interpretare**, cioè definire cosa significa una frase corretta sintatticamente. Il significato è un'entità autonoma che esiste indipendentemente dai segni che usiamo per descriverlo. La semantica consta di 3 metodi formali per la sua descrizione:

   - Semantica operazionale
   - Semantica denotazionale
   - Semantica assiomatica

È difficile trovare equilibrio tra esattezza e flessibilità in modo da rimuovere ambiguità, ma ance lasciare spazio all'implementazione. Spesso si usa il linguaggio naturale.

La semantica operazionale descrive l'esecuzione di un programma attraverso **transizioni** definite direttamente sul linguaggio del programma.
È un tipo di formalismo simile all'interpretazione vera e propria, in cui abbiamo una macchina astratta e le istruzioni applicano transizioni di stato in questa.
C'è una sequenza di passi computazionali definita per ogni programma (anche non deterministica), che viene generata con l'applicazione di un insieme di *regole di inferenza*.
Si preoccupa di come vengono calcolati i risultati finali attraverso una variazione di configurazione del programma. Il modello matematico a cui fa riferimento è quello del sistema di transizione.

Un sistema di transizione è una struttura $(\Gamma, \rightarrow)$ dove Γ è un insieme di elementi γ chiamati **configurazioni** e la relazione binaria

$$\rightarrow \subseteq \Gamma \times \Gamma$$

è chiamata relazione di transizione.

4.2 Linguaggio IMP

Gli insiemi sintattici di base sono:

nome	descrizione	metavar
Tipi	`Typ = {bool, int}`	τ
Valori di verità	`bool = {tt, ff}`	t
Numeri	`int = {..., -1, 0, 1, ...}`	m, n
Identificatori	`Id = {rate, a25, b, x, ...}`	id, x
Operatori unari	`Uop = {not}`	uop
Operatori binari	`Bop = {+, -, x, or}`	bop

Le categorie sintattiche che ne derivano sono:

nome	descrizione	metavar
Costanti	`Con`	k
Espressioni	`Exp`	e
Dichiarazioni	`Dic`	d
Comandi	`Com`	c
Parametri attuali e formali	`AExp`	ae
Parametri attuali valutati	`ACon`	ak
Chiusure	`Form`	form
Tipi dei parametri attuali	`ATyp`	abs
Valori esprimibili	`EVal`$= \mathrm{bool} \cup \mathrm{int}$	$ev ::= \mathrm{k}$
Tipi esprimibili	`ETyp = Typ`	$\mathrm{et} ::= \tau$
Valori denotabili	`DVal`$= \mathrm{bool} \cup \mathrm{int} \cup \mathrm{Loc} \cup \mathrm{Abs}$	$\mathrm{dv} ::= \mathrm{k} \mid \mathrm{l} \mid \mathrm{abs}$
Tipi denotabili	`DTyp`$= \mathrm{Typ} \cup (\mathrm{Typ} \times \mathrm{Loc}) \cup (\mathrm{ATyp} \times \mathrm{Proc})$	$\mathrm{dt} ::= \tau \mid \tau\mathrm{loc} \mid \mathrm{aetproc}$
Valori memorizzabili	`SVal`$= \mathrm{bool} \cup \mathrm{int}$	$sv ::= \mathrm{k}$
Tipi memorizzabili	`STyp`$= \mathrm{bool} \cup \mathrm{int}$	$\mathrm{st} ::= \tau$

4.3 Dichiarazioni

Le dichiarazioni sono la categoria sintattica i cui elementi sono elaborati per produrre legami, ovvero un'associazione tra un **identificatore** e un **valore denotabile** oppure tra un **identificatore** e un **tipo denotabile**.

4.3.1 Locazioni

Le locazioni sono un meccanismo di *indirizzamento indiretto* tra identificatori e valori che possono memorizzare $(\mathrm{bool} \cup \mathrm{int})$.

Per ogni valore memorizzabile di tipo τ sia Loc_τ un insieme infinito di celle di memoria che possono memorizzare i valori di tipo τ. La collezione delle locazioni è quindi data da

$$\mathrm{Loc} = \bigcup_{\tau \in \mathrm{STyp}} \mathrm{Loc}_\tau$$

con $\mathrm{STyp} = \mathtt{bool} \cup \mathtt{int}$, ovvero l'insieme di elementi memorizzabili.

4.3.2 Memoria

Una memoria è un elemento dello spazio di funzioni

$$\mathit{Stores} = \bigcup_{L \subseteq \mathrm{Loc}} \mathrm{Store}_L$$

dove

$$\mathrm{Store}_L : L \to \mathit{SVal} \cup \{?, \bot\}$$

I simboli ? e $\bot$ indicano rispettivamente lo stato inutilizzato e indefinito. Il simbolo SVal denota i valori memorizzabili.

Alle memorie viene inoltre imposta la condizione:

$$\forall l \in L. \mathit{Stores}(l) \in \tau \iff l \in L \cap \mathrm{Loc}_\tau$$

per garantire che le locazioni siano utilizzate correttamente dal punto di vista dei tipi.
La memoria viene inoltre dinamicamente aggiornata durante l'esecuzione del programma. Dati $L, L' \subseteq \mathrm{Loc}, \sigma \in \mathrm{Store}_L$ e $\sigma' \in \mathrm{Store}'_L$, definiamo $\sigma[\sigma'] \in \mathrm{Store}_{L \cup L'}$ come:

$$\textbf{Aggiornamento:} \quad \sigma[\sigma'] = \begin{cases} \sigma'(l) & \text{if } l \in L' \\ \sigma(l) & \text{if } l \in (L - L') \end{cases}$$

Se $L \cap L' = \emptyset$ scriviamo σ, σ' al posto di $\sigma[\sigma']$.

4.3.3 Ambiente Dinamico

L'ambiente **dinamico** associa identificatori a valori denotabili, con $\bot$ associato all'identificatore che non è associato ad alcun valore. Gli ambienti sono modificati dalle dichiarazioni che a nuovi valori associano un valore o una locazione.

Un ambiente dinamico è un elemento dello spazio di funzioni

$$\mathit{Env} = \bigcup_{I \subseteq \mathrm{Id}} \mathit{Env}_I$$

dove

$$\mathit{Env}_I : I \to \mathit{DVal} \cup \mathrm{Loc} \cup \{\bot\}$$

con metavariabile ρ e DVal valori denotabili.

Le operazioni di aggiornamento degli ambienti $\rho[\rho'] \in \mathit{Env}_{I \cup I'}$ e $\rho, \rho' \in \mathit{Env}_{I \cup I'}$ sono definite come per le memorie.

4.3.4 Generatore di locazioni

Per dichiarare nuove variabili è necessario definire una funzione che ad ogni nuovo identificatore associ un locazione di memoria non utilizzata. Supponiamo quindi di poter ordinare le locazioni dello stesso tipo, associando ad ognuna di esse un valore naturale quando vengono utilizzate.

Un generatore di locazioni è una funzione

$$\mathit{New} : \mathrm{Loc} \times \mathrm{DTyp} \to \mathrm{Loc} \times \mathbb{N}$$

(con DTyp tipi denotabili) definita come la funzione che esegue l'associazione:

$$\mathit{New}(L, \tau) = \langle l, m \rangle, L \subseteq \mathrm{Loc}_\tau, l \in \mathrm{Loc}_\tau$$

con

$$m = \mathit{max}\{n | \exists \langle l, n \rangle . l \in L\} + 1$$

4.3.5 Ambiente Statico

Un ambiente statico (o di tipi) è un elemento dello spazio di funzioni TEnv definito da:

$$\mathrm{TEnv} = \bigcup_{I \subseteq Id} \mathrm{TEnv}_I$$

dove

$$\mathrm{TEnv}_I : I \to \mathrm{DTyp}$$

ha metavariabile Δ e DTyp è l'insieme dei tipi denotabili. Le operazioni d aggiornamento degli ambienti statici sono definiti come per le memorie.

4.3.6 Compatibilità di ambienti

Sia $\rho : I$ un ambiente dinamico e $\Delta : I$ un ambiente statico con $I \subseteq Id$. Gli ambienti ρ e Δ sono compatibili ($\rho : \Delta$) se e soltanto se

$$\forall(id) \in I.\big(\Delta(id) = \tau \wedge \rho(id) \in \tau\big) \vee \exists\tau.\big(\Delta(id) = \tau loc \wedge \rho(id) \in Loc_\tau\big)$$

4.3.7 Elaborazione ed equivalenza

La funzione di elaborazione $\mathrm{Elab} : \mathrm{Dic} \times \mathrm{Store} \to \mathrm{Env}$ descrive il comportamento dinamico delle dichiarazioni a partire da una memoria σ restituendo l'ambiente che esse generano. La funzione è così definita:

$$\mathrm{Elab}(d, \sigma) = \rho \iff \langle d, \sigma\rangle \to_d^* \langle \rho, \sigma'\rangle$$

L'equivalenza di due dichiarazioni $\equiv \subseteq \mathrm{Dic} \times \mathrm{Dic}$, è definita da

$$d_0 \equiv d_1 \iff \forall\sigma.(\mathrm{Elab}(d_0, \sigma) = \mathrm{Elab}(d_1, \sigma))$$

4.3.8 Binding

Processo tramite il quale vene effettuato il collegamento tra un entità software e il suo corrispondente valore.

I programmi sono formati da *entità*(variabili, istruzioni, routine...), le entità sono caratterizzate da *attributi*(var: nome, tipo, area di memoria, Routine: nome, parametri, modalità di passaggio parametri...)

Il binding *associa un valore alle entità* dei programmi. Per ogni entità le informazioni su esse sono contenute in un descrittore. Si ha quindi il collegamento tra indirizzi simbolici e indirizzi fisici. Binding di dati e istruzioni a indirizzi di memoria.
Ogni volta che si invoca una funzione:

1. Si crea una nuova attivazione(istanza) del servitore
2. Viene allocata memoria per i parametri e per le variabili locali
3. Si effettua il passaggio di parametri
4. Si trasferisce il controllo al servitore
5. Si esegue il codice della funzione

Al momento dell'invocazione viene creata dinamicamente una struttura dati detta **record di attivazione** che contiene i binding dei parametri e degli identificatori definiti localmente alla funzione. Questa struttura contiene tutto ciò che serve per la chiamata alla quale è associato, cioè parametri formali, variabili locali, indirizzi di ritorno e indirizzi del codice della funzione. Il record è deallocato/distrutto al termine dell'esecuzione della funzione.

4.4 Espressioni

Sono una categoria sintattica che compare in tutti i linguaggi di programmazione. Vengono valutate per ottenere un valore, un tipo esprimibile. I costituenti elementari sono i letterali (constanti e identificatori), composti mediante operatori.

4.4.1 Valutazione ed equivalenza

La funzione di valutazione

$$\mathrm{Eval} : \mathrm{Exp} \times \mathrm{Store} \to \mathrm{Con}$$

che descrive il comportamento dinamico delle espressioni a partire da una memoria σ restituendo il valore in cui esse sono valutate, è definita da

$$\mathrm{Eval}(e, \sigma) = k \iff \langle e, \sigma \rangle \to_e^* \langle k, \sigma \rangle$$

La funzione di equivalenza di espressioni $\equiv \subseteq \mathrm{Exp} \times \mathrm{Exp}$ è definita da

$$e_0 \equiv e_1 \iff \forall \sigma.(\mathrm{Eval}(e_0, \sigma) = \mathrm{Eval}(e_1, \sigma))$$

4.5 Comandi

4.5.1 Variabili e assegnamenti

Categoria sintattica i cui elementi sono eseguiti per aggiornare la memoria della macchina astratta che supporta il linguaggio. Il comando base è l'assegnamento che modifica il contenuto di una locazione. Una variabile è un contenitore di valori che ha un nome particolare. Il loro contenuto può essere modificato tramite un assegnamento. Un assegnamento è costruito utilizzando un termine come *left-value*, che denota la locazione da modificare, e un *right-value*, che contiene il nuovo valore da associare alla locazione.

4.5.2 Iterazione e ricorsione

Iterazione e ricorsione sono due meccanismi che permettono di ottenere formalismi di calcolo Turing-completi. L'iterazione può essere di due tipologie:

- *indeterminata* quando i cicli sono controllati logicamente (`while, repeat`)
- *determinata* quando i cicli sono controllati numericamente (`for, do while`)

La ricorsione è strutturata facendo chiamare ad una funzione se stessa per ripeterne l'esecuzione su un valore (o una serie di valori) fino a ricondursi ad un caso di base.

4.5.3 Esecuzione ed equivalenza

La funzione

$$\mathrm{Exec} : \mathrm{Com} \times \mathrm{Store} \to \mathrm{Store}$$

che descrive il comportamento dinamico dei comandi a partire da una memoria σ resituendo la memoria della configurazione iniziale, è definita da

$$\mathrm{Exec}(c, \sigma) = \sigma' \iff \langle c, \sigma \rangle \to_c^* \sigma'$$

L'equivalenza di comandi $\equiv \subseteq \mathrm{Com} \times \mathrm{Com}$ è definita da

$$c_0 \equiv c_1 \iff \forall \sigma.(\mathrm{Exec}(c_0, \sigma) = \mathrm{Exec}(c_1, \sigma))$$

4.6 Procedure

Lo scopo delle procedure è quello di abbreviare la scrittura di programmi che contengono sequenze di comandi ripetute, la cui unica differenza è data dal valore dei dati su cui lavorano. Questa differenza viene concretizzata con il meccanismo del passaggio di parametri. Negli ambienti viene legata una sequenza di comandi (il corpo della procedura) con il suo identificatore (nome della procedura).

4.6.1 Parametri

I parametri consentono il riutilizzo di una procedura su valori diversi dello stesso dato. I parametri sono:

- *formali* se si trovano nella dichiarazione/definizione della procedura
  `int f (int n) {return n+1;}`
- *attuali* quando vengono utilizzati nella chiamata effettiva della funzione
  `x = f(y+3);`

I parametri possono venire passati in due metodi:

- *per valore* quando il valore attuale è assegnato a quello formale; in questo caso il contenuto del valore attuale viene copiato in quello formale, che si comporta come una variabile locale alla procedura di cui fa parte (le modifiche al valore del parametro formale non influenzano il valore del parametro attuale)
- *per riferimento* quando un riferimento del valore attuale è passato al parametro formale (*aliasing*), e le modifiche a quello formale si ripercuotono su quello attuale.

4.6.2 Regole di scope

Con regola di scope (detto anche visibilità) si intende lo spazio di programma in cui una certa variabile/identificatore è visibile e può quindi essere richiamata e modificata.

Il problema dello scope di una variabile si verifica sostanzialmente in 3 casi:

1. in presenza di procedure (blocchi eseguiti in posizioni diverse dalla loro definizione)
2. in presenza di ambiente locale
3. in presenza di dichiarazioni con nome uguale che possano mascherare la dichiarazione precedente

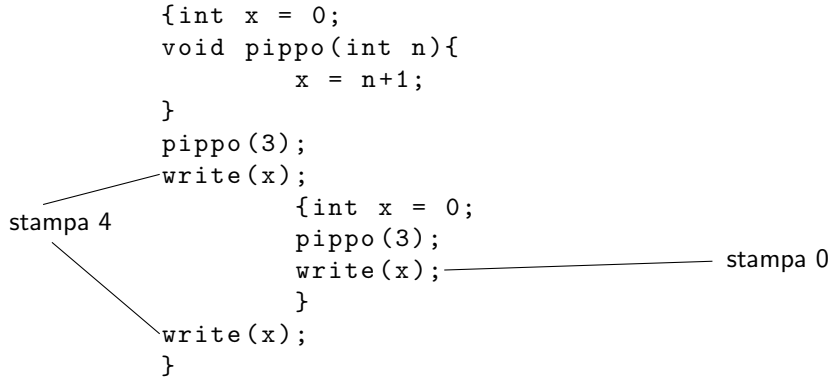
In generale un riferimento non locale in un generico blocco B può essere risolto in 2 modalità:

- nel blocco che include sintatticamente B (*scope statico*)
- nel blocco che è eseguito immediatamente prima di B (*scope dinamico*)

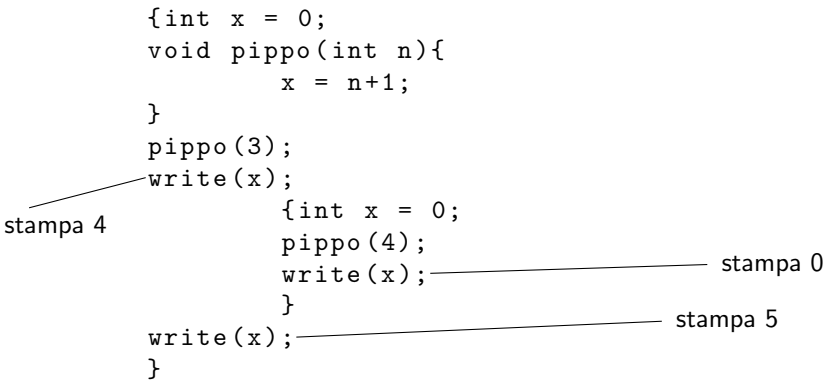
Scope Statico: ambiente che in ogni punto viene stabilito dal compilatore. Le dichiarazioni locali definiscono l'ambiente locale. L'utilizzo di un nome in un blocco comporta:

- l'utilizzo dell'ambiente locale se esiste, altrimenti
- si risale la gerarchia dei blocchi per cercare un ambiente non locale contenente un binding valido.

Scope statico 1: Un nome non locale è risolto nel blocco che testualmente lo racchiude

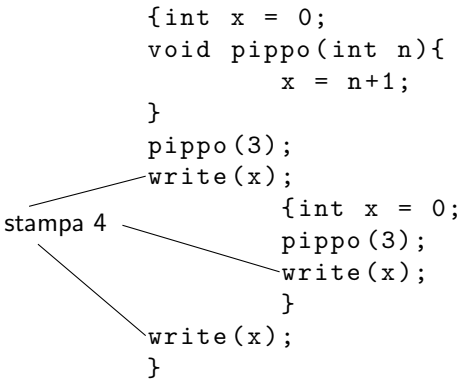


Scope statico 2: Un nome non locale è risolto nel blocco che testualmente lo racchiude



Scope Dinamico: l'associazione per un nome non locale X in un punto P del programma è la più recente tra quelle create per X ad essere attiva quando il flusso di controllo raggiunge P. Viene implementato con la ricerca nella pila dei **record di attivazione** dei blocchi/sottoprogrammi che hanno fatto raggiungere P. Per l'ambiente locale (e quello globale) non c'è differenza con lo scoping statico.

Scope dinamico 3: Un nome non locale è risolto nel blocco attivato più di recente e non ancora disattivato



5 Semantica denotazionale

5.1 Caratteristiche della semantica denotazionale

La semantica denotazionale consta di questi caratteri:

- *composizionalità*: la semantica di un costrutto è definita a partire dalla semantica dei suoi componenti
- assegna una denotazione al programma (funzione sui domini semantici)
- richiede un calcolo di minimo punto fisso, in quanto le funzioni di valutazione della semantica sono definite in modo ricorsivo

Nella semantica denotazionale si usa un altro dominio semantico, le *continuazioni*, funzioni

$$\mathcal{C} : \mathrm{store} \to \mathrm{store}$$

per trattare costrutti come i salti.

5.2 Sintassi astratta

La semantica formale viene solitamente definita su una rappresentazione dei programmi in *sintassi astratta*. Mentre nella sintassi concreta i costrutti del linguaggi sono rappresentati tramite stringhe, in sintassi astratta un costrutto è un'espressione (o albero) in termini di applicazione di un operatore astratto ad n operandi, che a loro volta sono espressioni. La sintassi astratta è definita specificano dei *domini sintattici*, ognuno dei quali possiede un nome e relative metavariabili.

La semantica denotazionale associa ad ogni costrutto sintattico la sua denotazione, ovvero una funzione che ha come dominio e codominio opportuni *domini semantici*. I domini semantici sono definiti da equazioni di dominio del tipo

$$\mathrm{nomeDominio} = \mathrm{espressioneDiDominio}$$

le quali sono costruite a partire da *costruttori di dominio* del tipo:

$$\mathrm{bool} = \{\mathrm{true}, \mathrm{false}\} \quad \text{enumerazione di valori}$$

oppure

$$\mathrm{val} = [\mathrm{int} + \mathrm{bool}] \quad \text{somma di domini}$$

Per ogni dominio esiste un metodo per garantire un ordinamento parziale degli stessi, garantendo che ogni dominio sia effettivamente un reticolo completo.

5.3 Domini semantici: lo stato

In un qualunque linguaggio ad alto livello lo stato deve comprendere un dominio chiamato *ambiente* (env), per modellare l'associazione tra gli identificatori e i valori che questi possono denotare. Il dominio env è definito quindi come

$$\mathrm{env} : \mathrm{ide} \to \mathrm{dval}$$

inoltre, essendo possibile che identificatori diversi denotino la stessa locazione di memoria, è necessaria un componente *memoria* (store), per modellare l'associazione tra locazioni e valori che possono essere memorizzati, quindi

$$\mathrm{store} : \mathrm{loc} \to \mathrm{mval}$$

5.4 Domini semantici: i valori

- dval: *valori denotabili*, dominio dei valori che possono essere denotati da un identificatore nell'ambiente.
- mval: *valori memorizzabili*, dominio dei valori che possono essere contenuti in una locazione di memoria.
- eval: *valori esprimibili*, dominio dei valori che possono essere ottenuti come semantica di un'espressione.

5.5 Dominio semantico fun

Le espressioni contengono l'astrazione $\mathtt{lambda}(\mathrm{I}, \mathrm{E})$ che rappresenta una funzione il cui corpo è l'espressione E_1 con parametro formale I
La semantica di tale costrutto è una funzione del dominio fun

$$fun = store \to dval \to eval$$

il valore di tipo $dval$ sarà il valore dell'argomento nella applicazione.

5.6 Funzioni di valutazione semantica

Per ogni dominio sintattico esiste una funzione di valutazione semantica,

$$\begin{aligned}
&\mathcal{E} : expr \to env \to store \to eval \quad \text{espressioni}\\
&\mathcal{C} : com \to env \to store \to store \quad \text{comandi}\\
&\mathcal{D} : dec \to env \to store \to (env \times store) \quad \text{dichiarazioni}\\
&\mathcal{P} : prog \to env \to store \to store \quad \text{programmi}
\end{aligned}$$

Le funzioni di valutazione semantica assegnano uno specifico significato ai vari costrutti del linguaggio, con una definizione data sui casi della sintassi astratta.

5.7 Denotazionale vs operazionale

A differenza della semantica denotazionale, la semantica operazionale non è sempre composizionale (Sez. 5.1), inoltre non assegna solo una denotazione (funzione tra domini semantici) al programma ma specifica anche come si raggiunge per un dato programma lo stato finale a partire da quello iniziale (tramite sistemi di transizione).

Le funzioni di valutazione della semantica sono in entrambi i casi definite in modo ricorsivo, ma per la semantica operazionale non è necessario alcun calcolo di punto fisso, in quanto questo viene determinato dall'interprete come chiusura transitiva delle transazioni che descrivono i passaggi di stato del programma.
Un'altra sostanziale differenza tra semantica operazionale e denotazionale è costituita dalla diversa definizione delle funzioni di valutazione della semantica, ad esempio:

$$\begin{aligned}
&\mathcal{C} : Com \to env \to store \to store && \text{(denotazionale)}\\
&\mathcal{C} : Com \times env \times store \to store && \text{(operazionale)}
\end{aligned}$$

e dal cambiamento dei domini di valori funzionali, come

$$\begin{aligned}
&fun = store \to dval \to eval && \text{(denotazionale)}\\
&fun = EXPR \times env && \text{(operazionale)}
\end{aligned}$$

5.8 Semantica denotazionale e paradigmi

5.8.1 Paradigma funzionale

- dominio sintattico: $expr$
- dominio semantico per lo stato: env
- $dval = eval$: gli $eval$ contengono sempre fun
- funzione di valutazione semantica: $\mathcal{E} : expr \to env \to eval$

5.8.2 Paradigma imperativo

- 3 domini sintattici: expr, com, dec
- 2 domini semantici per lo stato: env, store
- $\mathrm{dval} \neq \mathit{eval} \neq \mathrm{mval}$: i valori su cui si interpretano le astrazioni funzionali (fun) sono di solito dval. Le locazioni sono sempre denotabili.
- 3 funzioni di valutazione semantica:

$$\begin{aligned}\mathcal{E} &: \mathrm{expr} \to \mathit{env} \to \mathit{store} \to \mathit{eval} \\ \mathcal{C} &: \mathrm{com} \to \mathit{env} \to \mathit{store} \to \mathit{store} \\ \mathcal{D} &: \mathrm{dec} \to \mathit{env} \to \mathit{store} \to (\mathit{env} \times \mathit{store})\end{aligned}$$

5.8.3 Paradigma ad oggetti

- 4 domini sintattici: expr, com, dec, dichiarazioni di **classe**
- 3 domini semantici per lo stato: env, store, heap: per modellare i puntatori e gli oggetti
- $\mathrm{dval} \neq \mathit{eval} \neq \mathrm{mval}$: i valori su cui si interpretano le astrazioni funzionali (fun) sono di solito dval. Le locazioni sono sempre denotabili. Gli oggetti di solito appartengono a tutti e tre i domini.
- funzioni di valutazione semantica, prendono (e restituiscono) anche la heap:

$$\mathcal{C} : \mathrm{com} \to \mathit{env} \to \mathit{store} \to \mathrm{heap} \to (\mathit{store} \times \mathrm{heap})$$

6 Credits

Davide Bianchi (mail: `davideb1912@gmail.com`)
Matteo Danzi (mail: `matteodanziguitarman@hotmail.it`)