Logica per la Programmazione

Lezione 14

- Introduzione alla Logica di Hoare
- Linguaggio di Programmazione Imperativo: Sintassi e Semantica
- Tripla di Hoare soddisfatta

Introduzione

- Dall'inizio del corso ad ora abbiamo introdotto, un po' alla volta, un linguaggio logico sempre più ricco:
  - connettivi logici (Calcolo Proposizionale)
  - termini e quantificatori (Logica del Primo Ordine)
  - uguaglianza e disuguaglianze
  - insiemi e intervalli
  - quantificatori funzionali
- Per ognuna di queste estensioni abbiamo presentato:
  - la sintassi con grammatiche in BNF
  - la semantica (tabelle di verità, interpretazioni e modelli)
  - esempi di formalizzazione di enunciati
  - alcune leggi ed esempi di dimostrazioni

Sui Linguaggi di Programmazione

- In questa parte finale del corso sfruttiamo la logica introdotta per fornire una semantica formale di un semplice linguaggio di programmazione imperativo
  Perché?
- La presentazione di un **linguaggio di programmazione** di solito consiste nel dare la **sintassi** e una **semantica**
  - **sintassi** spesso fornita con strumenti formali (es. grammatica in Backus-Naur Form (BNF))
  - **semantica** spesso data in modo informale
    - solitamente di stile operazionale
    - comprensibile per non esperti
    - ma lascia spazio a ambiguità
    - non sufficiente per applicazioni "critiche"

Necessità di una Semantica Formale

- A volte è necessario dimostrare proprietà relative a programmi
  - Software per controllo di centrali nucleari, di armamenti, di apparecchiature mediche, software/protocolli per e-banking, per gestione carte di credito, ...
- Sono state proposte varie *semantiche formali* tra cui:
  - **operazionale**, definendo struttura degli stati e transizioni di stato
  - **denotazionale**, con domini semantici e funzioni di interpretazione per i costrutti del linguaggio (come quella del C a PRL)
  - **assiomatica**, annotando un programma con asserzioni (formule) che descrivono le proprietà e poi dimostrandone la correttezza con **regole di inferenza**
- Noi vedremo la **semantica assiomatica** di Hoare [1969] - Dijkstra [1976]

Cosa Vedremo...

- Nel nostro linguaggio di programmazione (minimo) avremo
  - **espressioni**: standard a valori interi o booleani
  - **comandi**: comando vuoto, assegnamento, condizionale, comando iterativo
- Daremo la sintassi con grammatica BNF
- Daremo la semantica
  - per le espressioni una semantica in stile denotazionale
  - per i comandi presenteremo una semantica operazionale in modo *informale*
- Infine daremo una semantica assiomatica del linguaggio in modo formale

Un Esempio di Programma

- Assumiamo che **a** : $array[0, n)$ of int
- Cosa calcola il seguente programma?

```
x, c := 0, 0;
while x < n do
    if (a[x] > 0)
        then c := c + 1
        else skip fi;
    x := x + 1
endw
```

- Il numero di elementi di **a** che sono maggiori di 0
- Tra poco saremo in grado di **dimostrarlo formalmente**!

Triple di Hoare

Dopo sintassi e semantica di espressioni e comandi, introdurremo le

Triple di Hoare

che permettono di annotare un programma con asserzioni

- definiremo quando una tripla è **soddisfatta**
- vedremo un **Proof System** (un insieme di **regole di inferenza**) per dimostrare che una tripla è soddisfatta (usando **induzione strutturale** sul programma)

Il programma che conta gli elementi maggiori di zero di un array, "Annotato"

```
{a :  array [0, n) of int }
x, c := 0,0;
{Inv :  c = # { j :  j ∈ [0, x) | a[j] > 0 } ∧ x ∈ [0, n] }
{t:n - x}
while x < n do
    if (a[x] > 0) then c := c + 1 else skip fi;
    x := x + 1
endw
{Inv ∧ ¬(x<n)}
{c = # {j :  j ∈ [0, n) | a[j] > 0}}
```

Saremo in grado di dimostrare che alla fine dell'esecuzione è vera l'ultima formula

Linguaggio Imperativo Minimo: sintassi (1)

Espressioni (a valori interi o booleani):

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \mathit{Const} \mid \mathit{Ide} \mid (\mathit{Exp}) \mid \mathit{Exp}\ \mathit{Op}\ \mathit{Exp} \mid \mathit{not}\ \mathit{Exp} \\
\mathit{Op} & ::= & + \mid - \mid * \mid \mathit{div} \mid \mathit{mod} \mid \\
 & & = \mid \neq \mid < \mid \leq \mid > \mid \geq \mid \mathit{or} \mid \mathit{and} \\
\mathit{Const} & ::= & \mathit{Num} \mid \mathit{Bool} \\
\mathit{Bool} & ::= & \mathit{true} \mid \mathit{false} \\
\mathit{Num} & ::= & 0 \mid -1 \mid 1 \mid \ldots
\end{array}
$$

Linguaggio Imperativo Minimo: sintassi (2)

Comandi:

$$
\begin{array}{lcl}
\mathit{Com} & ::= & \textbf{skip} \mid \\
 & & \mathit{Ide\_List} := \mathit{Exp\_List} \mid \\
 & & \mathit{Com}\ ;\ \mathit{Com} \mid \\
 & & \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Com}\ \textbf{else}\ \mathit{Com}\ \textbf{fi} \mid \\
 & & \textbf{while}\ \mathit{Exp}\ \textbf{do}\ \mathit{Com}\ \textbf{endw} \\
 & & \\
\mathit{Ide\_List} & ::= & \mathit{Ide} \mid \mathit{Ide}, \mathit{Ide\_List} \\
 & & \\
\mathit{Exp\_List} & ::= & \mathit{Exp} \mid \mathit{Exp}, \mathit{Exp\_List}
\end{array}
$$

Stato di un Programma

- Uno stato di un programma è una funzione da identificatori di variabili a valori (booleani e interi)

$$\sigma : \mathit{Ide} \to \mathbb{Z} \cup \mathbb{B}$$

- Indichiamo con *State* l'insieme degli stati

$$\mathit{State} = [\mathit{Ide} \to \mathbb{Z} \cup \mathbb{B}]$$

- Poiché in uno stato l'insieme delle variabili è finito, si può usare una rappresentazione estensionale. Esempio:

$$\sigma = \{x \mapsto 18, y \mapsto \mathit{true}, z \mapsto -8\}$$

- Lo stato rappresenta quindi *in modo astratto* lo stato della memoria usata dal programma.
- Non si possono modellare concetti come *aliasing* (due variabili che denotano la stessa cella di memoria) e *puntatori*.

Semantica (valore) delle Espressioni

- Come visto, le espressioni possono contenere variabili
- Il **valore** di una espressione dipende dal valore associato alle variabili, quindi dipende dallo stato.
- Per calcolare il valore delle espressioni definiamo la **funzione di interpretazione semantica**:

$$\mathcal{E} : \mathtt{Exp} \times \mathit{State} \to \mathbb{Z} \cup \mathbb{B}$$

- $\mathcal{E}(e, \sigma)$ denota il valore dell'espressione e nello stato σ
- La funzione $\mathcal{E}$ è definita in modo induttivo (sulla struttura delle espressioni)

Semantica delle Espressioni

$$
\begin{array}{rcll}
\mathcal{E}(\mathit{true},\sigma) & = & \mathbf{tt} & \\
\mathcal{E}(\mathit{false},\sigma) & = & \mathbf{ff} & \\
\mathcal{E}(n,\sigma) & = & \mathbf{n} & \mathit{se} \quad n \in \mathtt{Num} \\
\mathcal{E}(x,\sigma) & = & \sigma(x) & \mathit{se} \quad x \in \mathtt{Ide} \\
\mathcal{E}(E\ \mathit{op}\ E',\sigma) & = & \mathcal{E}(E,\sigma)\ \mathbf{op}\ \mathcal{E}(E',\sigma) & \mathit{se} \quad \mathit{op} \in \{+,-,\mathit{div},\mathit{mod},=,\neq,<,>,\leq,\geq\} \\
 & & & \quad\ \mathcal{E}(E,\sigma) \in \mathbb{Z} \ \ \mathit{e} \ \ \mathcal{E}(E',\sigma) \in \mathbb{Z} \\
\mathcal{E}(E\ \mathit{op}\ E',\sigma) & = & \mathcal{E}(E,\sigma)\ \mathbf{op}\ \mathcal{E}(E',\sigma) & \mathit{se} \quad \mathit{op} \in \{\mathit{and},\mathit{or},=,\neq\} \\
 & & & \quad\ \mathcal{E}(E,\sigma) \in \mathbb{B} \ \ \mathit{e} \ \ \mathcal{E}(E',\sigma) \in \mathbb{B} \\
\mathcal{E}(\mathit{not}\ E,\sigma) & = & \neg\mathcal{E}(E,\sigma) & \mathit{se} \quad \mathcal{E}(E,\sigma) \in \mathbb{B} \\
\mathcal{E}((E),\sigma) & = & \mathcal{E}(E,\sigma) &
\end{array}
$$

Espressioni: esempi

Consideriamo lo stato

$$\sigma = \{x \mapsto 18, y \mapsto \mathit{true}, z \mapsto -8\}$$

- $\mathcal{E}(x + z, \sigma) = \mathcal{E}(x, \sigma) + \mathcal{E}(z, \sigma) = 18 - 8 = 10$
- $\mathcal{E}(y, \sigma) = \mathit{true}$
- $\mathcal{E}(5 + y, \sigma) = \mathcal{E}(5, \sigma) + \mathcal{E}(y, \sigma) = 5 + \mathit{true} = ?$
- Quindi $\mathcal{E}$ è una funzione **parziale**!!!

Significato Informale dei Comandi

- L'esecuzione di un comando semplice (tipo un assegnamento singolo o multiplo) tipicamente ha l'effetto di cambiare lo stato della memoria
- I comandi composti (sequenza, condizionale, iterazione) hanno "solo" il ruolo di controllare il flusso di esecuzione di comandi semplici. Naturalmente il loro effetto cambia al cambiare dello stato in cui vengono eseguiti.
- In generale, possiamo dire che l'esecuzione di un comando causa una transizione (un "passaggio") da uno stato (quello in cui inizia l'esecuzione del comando) ad un altro (quello in cui l'esecuzione termina).

Semantica Informale dei Comandi (1)

- L'esecuzione di **skip** a partire dallo stato σ porta nello stato σ
- L'esecuzione dell'assegnamento $x_1, ..., x_n := E_1, ..., E_n$ a partire dallo stato σ porta nello stato

$$\sigma[\mathcal{E}(E_1, \sigma)/x_1, ..., \mathcal{E}(E_n, \sigma)/x_n]$$

- L'esecuzione del comando **C;C'** a partire dallo stato σ porta nello stato σ' ottenuto eseguendo **C'** a partire dallo stato σ'' ottenuto dall'esecuzione di **C** nello stato σ

Semantica Informale dei Comandi (2)

- L'esecuzione del comando **if E then C else C' fi** a partire da uno stato σ porta
  - nello stato σ' che si ottiene dall'esecuzione di **C** a partire da σ, se $\mathcal{E}(E, \sigma) = \mathbf{tt}$
  - nello stato σ' che si ottiene dall'esecuzione di **C'** in σ, se invece $\mathcal{E}(E, \sigma) = \mathbf{ff}$
- L'esecuzione del comando **while E do C endw** a partire da σ porta in σ se $\mathcal{E}(E, \sigma) = \mathbf{ff}$, altrimenti porta nello stato σ' ottenuto dall'esecuzione di **while E do C endw** a partire dallo stato σ'' ottenuto con l'esecuzione di **C** nello stato σ

Terminazione dei comandi

Attenzione: non tutti i comandi terminano con successo, cioè portano in uno stato σ' partendo da uno stato σ:

- [Ciclo infinito:] Esempio: **while** *true* **do** *skip* **endw**
- [Errore di tipo:] Esempio: un assegnamento $x := z + y$ eseguito in uno stato in cui $\mathcal{E}(z + y, \sigma)$ non è definito, come:

$$\sigma = \{x \mapsto 18, y \mapsto \mathit{true}, z \mapsto -8\}$$

- [Operazione non definita:] Esempio: un assegnamento come $x := y \text{ div } z$ eseguito in uno stato in cui $\mathcal{E}(y \text{ div } z, \sigma)$ non è definito, come:

$$\sigma = \{x \mapsto 0, y \mapsto 7, z \mapsto 0\}$$

C'è qualche differenza tra gli ultimi due casi? Quale?

Triple di Hoare

- Una **Tripla di Hoare** ha la forma

$$\{Q\}\ C\ \{R\}$$

dove C è un **comando** del linguaggio, mentre Q e R sono **asserzioni**, cioè formule della Logica del Primo Ordine (estesa con le notazioni introdotte in precedenza) in cui possono comparire variabili dello stato

- Il dominio di interpretazione delle asserzioni comprende $\mathbb{Z}$, $\mathbb{B}$ e i sottoinsiemi di $\mathbb{Z}$
- **Significato intuitivo**: la tripla $\{Q\}\ C\ \{R\}$ è **soddisfatta** se dato un qualunque stato σ che soddisfi Q, l'esecuzione del comando C a partire da σ termina, e lo stato finale soddisfa R
- **Attenzione**: diciamo termina per termina con successo
- **Esempio**

$$\{x > 1\}\ x := x + 1\ \{x > 2\}$$

Notazione

- $\mathit{free}(P)$ è l'insieme delle variabili libere che compaiono in una **asserzione** P

$$\mathit{free}(x > y \wedge z \leq 1) = \{x, y, z\}$$

- Sia P un'asserzione e σ uno stato. Usiamo P^σ per indicare l'asserzione P istanziata sullo stato σ, ovvero in cui a tutte le variabili sono sostituiti i loro valori nello stato
- Poiché l'interpretazione del linguaggio delle asserzioni è fissata, dato uno stato si può valutare l'asserzione, ovvero il suo valore di verità
- Esempio: $\sigma = \{x \mapsto 18, y \mapsto \mathit{true}, z \mapsto -8\}$

$$P = (x > 0 \wedge z < 0)$$
$$P^\sigma = (18 > 0 \wedge -8 < 0) \equiv \mathbf{T}$$

- Quindi uno stato determina univocamente un'interpretazione (nel senso della Logica dei Primo Ordine)

Stati come Interpretazioni e Modelli

- Scriviamo $\sigma \models P$ sse $P^\sigma \equiv \mathbf{T}$, quindi se σ è un **modello** di P.
  In questo caso diciamo che: lo stato σ **soddisfa l'asserzione** P
- Con $\{P\}$ indichiamo **l'insieme degli stati** che soddisfano P, ovvero

$$\{P\} = \{\sigma \mid \sigma \models P\}$$

Stati come Modelli: alcune proprietà utili

Sia E un'espressione, P un'asserzione e σ, σ' due stati. Valgono le seguenti proprietà:

- Se per ogni $x \in \mathit{free}(E), \sigma(x) = \sigma'(x)$ allora $\mathcal{E}(E, \sigma) = \mathcal{E}(E, \sigma')$
- Se per ogni $x \in \mathit{free}(P), \sigma(x) = \sigma'(x)$ allora

$$\sigma \models P \quad \text{se e solo se} \quad \sigma' \models P$$

- Per ogni variabile x

$$\sigma[\mathcal{E}(E, \sigma)/x] \models P \quad \text{se e solo se} \quad \sigma \models P[E/x]$$

- Esempio: $\sigma[5/x] \models (y = x)$ se e solo se $\sigma \models (y = x)[5/x]$ $(\equiv \sigma \models (y = 5))$

Triple di Hoare

Data la **tripla di Hoare** $\{Q\}\ C\ \{R\}$

- Q è detta precondizione
- R è detta postcondizione
- La **tripla è soddisfatta** se:
  - *per ogni* stato σ che soddisfa la precondizione Q (ovvero $\sigma \models Q$)
  - l'esecuzione del comando C a partire dallo stato σ
  - **termina** producendo *uno stato* σ', e
  - σ' soddisfa la postcondizione R (ovvero $\sigma' \models R$)

Interpretazione delle Triple di Hoare: **Semantica**

- Data la precondizione Q ed il comando C, determinare la postcondizione R in modo che sia soddisfatta $\{Q\}\ C\ \{R\}$ è un modo per descrivere il comportamento di C.
- Per esempio sia

$$C = \textbf{if } (x < 10) \textbf{ then } x := 1 \textbf{ else } x := 2 \textbf{ fi}$$

- Se consideriamo la precondizione $Q = (x > 10)$, allora la postcondizione $R = (x = 2)$ descrive il comportamento del comando

Interpretazione delle Triple di Hoare: **Correttezza**

- Dati il comando C, la precondizione Q e la postcondizione R **dimostrare** che la tripla $\{Q\}\ C\ \{R\}$ è **soddisfatta** corrisponde ad una *dimostrazione di correttezza* del comando C rispetto alle proprietà descritte dalla precondizione Q e dalla postcondizione R
- Per esempio sia C il comando (assumendo a come array di interi su $[0, n)$)

```
while (x < n) do
    if (a[x] > 0) then c := c + 1
        else skip fi;
    x := x + 1
endw
```

- La tripla $\{Q\}\ C\ \{R\}$ risulta soddisfatta per

$$Q = (x = 0 \wedge c = 0)$$
$$R = (c = \#\{j : j \in [0, n) \mid a[j] > 0\})$$

Interpretazione delle Triple di Hoare: **Specifica**

- Date sia la precondizione Q che la postcondizione R **determinare** un comando C che soddisfa la tripla $\{Q\}\ C\ \{R\}$. Questo equivale a *scrivere il programma che realizza le specifiche* Q ed R.
- Per esempio consideriamo le specifiche $Q = (x > 0)$ (la precondizione) ed $R = (y = 2 \times x)$ (la postcondizione)
- Possono essere soddisfatte dal comando

$$y := x * 2$$

oppure dal comando

$$y := x + x$$

oppure dal comando

$$y := x; y := y * 2$$

oppure dal comando

$$y := 10; x := 5 \text{ (!!!)}$$

ecc...

Proof System per Verifica di Triple: alcune Regole di Inferenza

$$(\textit{pre-post} \;\; o \;\; \textit{conseguenza}) \quad \frac{P \Rightarrow P' \quad \{P'\}\ C\ \{R'\} \quad R' \Rightarrow R}{\{P\}\ C\ \{R\}}$$

$$(\textit{pre}) \quad \frac{P \Rightarrow P' \qquad \{P'\}\ C\ \{R\}}{\{P\}\ C\ \{R\}}$$

$$(\textit{post}) \quad \frac{\{P\}\ C\ \{R'\} \qquad R' \Rightarrow R}{\{P\}\ C\ \{R\}}$$

La correttezza di queste regole segue immediatamente dalla definizione di "tripla soddisfatta"