UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

TRASCRIZIONE DEGLI APPUNTI DI “Fondamenti di linguaggi e specifica”

Modulo della *prof.ssa Maria Paola Bonacina*

A.A. 2020 – 2021

Creato da *Davide Zampieri*

- **Introduzione alla teoria dei domini** – *G. Winskel: The formal semantics of programming languages* [capitolo 8]

Semantica dei linguaggi di programmazione.
Abbiamo tre approcci classici:

- Semantica *operazionale*.
- Semantica *denotazionale*.
- Semantica *assiomatica*.

Semantica denotazionale.
Sia *P* un linguaggio di programmazione. Per ogni programma *A* scritto in *P* si associa una funzione *f* tale che per ogni ingresso *n* per *A*, se *A* termina e produce l'uscita *q* allora *f* è definita su *n* e *f(n)=q*, e viceversa. Ciò indica una *corrispondenza biunivoca tra programmi e funzioni*. L'obiettivo sarà quello di costruire funzioni che corrispondano ai costrutti sintattici del linguaggio (denotazione).

Teoria dei domini.
Una *relazione di ordinamento* di un insieme è una relazione binaria tra elementi appartenenti all'insieme. Essa è *riflessiva*, *antisimmetrica* e *transitiva*. Il simbolo che useremo sarà: $\sqsubseteq$. Una relazione è *totale* se tutti gli elementi sono confrontabili, ovvero se tutti gli elementi possono essere messi su una linea (ad es. la relazione tra i numeri naturali). Viceversa, una relazione è *parziale* quando ci sono elementi non confrontabili (ad es. la relazione tra i due valori booleani).

Diagrammi di Hasse.
Per descrivere le relazioni di ordinamento si usa spesso il *diagramma di Hasse*, ovvero un grafo in cui si mettono in relazione gli elementi secondo il loro ordinamento posizionando in alto gli elementi più grandi.

Minorante e maggiorante.
Dato un insieme *U* con ordinamento $\sqsubseteq$, supponiamo di prendere un sottoinsieme di *U* e chiamarlo *A*. Si dice *minorante* di *A* un elemento *x* in *U* tale che *x* è $\sqsubseteq$ a tutti gli elementi di *A*. Si dice invece *maggiorante* di *A* un elemento *x* in *U* tale che tutti gli elementi di *A* sono $\sqsubseteq$ a *x*.

Minimo e massimo.
Se *x* è un minorante di *A* e *x* appartiene ad *A*, allora *x* è l'elemento *minimo* di *A*. Se invece *x* è un maggiorante di *A* e *x* appartiene ad *A*, allora *x* è l'elemento *massimo* di *A*.

Estremo inferiore e superiore.
Il più grande dei minoranti, se esiste, si dice *estremo inferiore* (greatest lower bound, glb). Il più piccolo dei maggioranti, se esiste, si dice invece *estremo superiore* (least upper bound, lub). Posso avere infiniti minoranti e maggioranti, ma l'estremo inferiore e l'estremo superiore sono i più precisi nell'approssimazione da sopra e da sotto.

Reticolo.
(U, $\sqsubseteq$) è un reticolo se per ogni coppia (x, y) di elementi di U sono definiti l'estremo inferiore e l'estremo superiore come:

- $inf(\{x, y\}) = x \sqcap y$
- $sup(\{x, y\}) = x \sqcup y$

Lambda calcolo.
Le *motivazioni* alla base della teoria dei domini sono:

- Trovare uno spazio dove definire e risolvere *equazioni ricorsive*.
- Trovare un modello per il *lambda calcolo*.

Il lambda calcolo è un formalismo usato per scrivere le funzioni. Ciò che ci interessa del lambda calcolo è la *lambda astrazione*, cioè il fatto di poter scrivere le funzioni come *funzioni anonime* (ad es. invece di scrivere *succ(x)* potremo semplicemente scrivere $\lambda x.x+1$). Un'altra proprietà interessante del lambda calcolo è il fatto di poter passare altre funzioni ai *parametri* di una funzione (ad es. la funzione fattoriale può essere definita dal punto fisso del funzionale $\vartheta = \lambda f.\ \lambda n.\ n=0 \rightarrow 1,\ n*f(n-1)$ come $fact = \vartheta(fact) = \lambda n.\ n=0 \rightarrow 1,\ n*fact(n-1)$).

Reticolo completo.
Un reticolo *S* si dice *completo* se per ogni sottoinsieme *A* di *S* esistono *inf(A)* e *sup(A)*. Siccome *S* è sottoinsieme di *S*, l'estremo inferiore e l'estremo superiore saranno definiti come:

- $inf(S) = min(S) = \bot$
- $sup(S) = max(S) = \top$

Esempio
$<\mathbb{N}, \sqsubseteq>$ è un reticolo perché per ogni coppia di numeri naturali (*n*, *m*) esiste $inf(\{n, m\}) = min(\{n, m\})$ e $sup(\{n, m\}) = max(\{n, m\})$; ma non è un reticolo completo perché esiste un sottoinsieme di $\mathbb{N}$ che non ha estremo superiore (ad es. $A = \{n \mid n \geq 100\}$).

Domini di funzioni.
Un reticolo completo (o insieme parzialmente ordinato completo) è detto anche *dominio*. Se D_1 e D_2 sono due domini, allora $[D_1 \rightarrow D_2] = \{f \mid f: D_1 \rightarrow D_2\}$ è l'insieme di tutte le funzioni che vanno da D_1 a D_2.

Ordinamento punto a punto.
L'*ordinamento punto a punto* è diverso dall'ordinamento di sottoinsieme (inclusione), infatti per due funzioni *f* e *g* appartenenti al dominio $[D \rightarrow D]$ potremo dire che $f \sqsubseteq g$ solo se per ogni *x* in *D* vale che $f(x) \sqsubseteq g(x)$. Inoltre, l'inclusione implica l'ordinamento punto a punto (ma non il contrario). Un caso particolare in cui l'ordinamento punto a punto implica l'inclusione è quello del *dominio piatto*, in quanto gli elementi di tale dominio sono tutti equi-comparabili (cioè senza un ordinamento stretto).

Catene di funzioni.
Data una *catena* di funzioni $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots \sqsubseteq f_i \sqsubseteq f_{i+1} \sqsubseteq \ldots$, per qualsiasi *x* del dominio *D* possiamo costruire la catena delle immagini $f_0(x) \sqsubseteq f_1(x) \sqsubseteq f_2(x) \sqsubseteq \ldots \sqsubseteq f_i(x) \sqsubseteq f_{i+1}(x) \sqsubseteq \ldots$. Inoltre, in *D* ogni catena ammette un *estremo superiore* definito come $sup(\{f_i\}_{i\geq 0})(x) = sup(\{f_i(x)\}_{i\geq 0})$. Esistono anche casi particolari di catene quali:

- *Catene finite* $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \sqsubseteq \ldots \sqsubseteq f_k$, in cui l'estremo superiore è pari a $sup(\{f_i\}_{i=0\ldots k}) = f_k$.
- *Catene stazionarie* $\forall i \geq 0\ \exists n\ \forall i \geq n.\ f_i(x) = f_n(x)$, che dopo un certo elemento diventano costanti.

Infine, come conseguenza dell'ordinamento si ha che $\forall i \geq 0.\ Dom(f_i) \subseteq Dom(f_{i+1})$.

Funzioni monotone e continue.
Una funzione si dice *monotona* se $\forall x, y.\ x \sqsubseteq y \rightarrow f(x) \sqsubseteq f(y)$. Intuitivamente, quello che si fa è escludere che una funzione avente in ingresso un valore indefinito (ad es. derivato da una computazione non terminante) restituisca come risultato un valore definito. Una funzione si dice *continua* se la sua applicazione all'estremo superiore di una catena ascendente corrisponde all'estremo superiore della catena delle immagini della funzione stessa, ovvero per ogni catena ascendente $x_0 \sqsubseteq x_1 \sqsubseteq \ldots \sqsubseteq x_i \sqsubseteq \ldots$ vale che $f(sup(\{x_i\}_{i\geq 0})) = sup(\{f(x_i)\}_{i\geq 0})$. Tutte le funzioni che useremo per definire le denotazioni dei costrutti dei linguaggi di programmazione saranno *funzioni monotone continue*. È quindi opportuno arricchire la definizione di continuità con la monotonia, ovvero vogliamo che le funzioni continue siano anche monotone.

Funzioni di ordine superiore.
Per scrivere denotazioni di programmi come funzioni ci servono *funzioni di ordine superiore* (funzionali), ovvero funzioni che prendono come argomento altre funzioni $[D \to D]$. Ma siccome $|[D \to D]| = |D|$, allora possiamo vedere le funzioni stesse come dei *dati*.

Prodotto di domini.
Il *prodotto* di due domini D_1 e D_2 è anch'esso un dominio. Ciò significa che se possiamo usare un dominio per modellare un certo tipo di dato, possiamo usarlo anche per modellare tuple di dati.

Somma di domini.
La *somma* di due domini D_1 e D_2 è anch'essa un dominio. Per somma si intende l'*unione disgiunta* dei due domini, ovvero l'unione senza intersezione. Per forzare l'unione ad essere disgiunta bisogna usare due *funzioni di iniezione*, le quali "iniettano" i due domini nel dominio somma. Infine, se i minimi dei due domini sono incomparabili bisogna fare l'*operazione di sollevamento*, ovvero aggiungere un nuovo minimo che sia comune a entrambi i domini di cui voglio costruire la somma.

➢ La semantica denotazionale di IMP – *G. Winskel: The formal semantics of programming languages* [capitolo 5]

Punto fisso.
Data una funzione $f: D \to D$, un elemento $x \in D$ si dice:

- *Punto fisso*, se $f(x) = x$.
- *Punto prefisso*, se $f(x) \sqsubseteq x$.
- *Punto postfisso*, se $x \sqsubseteq f(x)$.

Si osserva poi che ogni punto fisso è sia un punto prefisso che un punto postfisso. I punti fissi sono importanti perché per catturare la computazione di un ciclo andremo a costruire una funzione il cui punto fisso descriverà la *denotazione del ciclo*.

Minimo e massimo punto fisso.
Il *minimo punto fisso* di una funzione *mpf(f)* è definito come quell'elemento x tale che $f(x) = x$ e $\forall y.\ f(y) = y$ vale che $x \sqsubseteq y$, ovvero x è minore di qualsiasi altro punto fisso. Invece, il *massimo punto fisso* di una funzione *MPF(f)* è definito come quell'elemento x tale che $f(x) = x$ e $\forall y.\ f(y) = y$ vale che $y \sqsubseteq x$, ovvero x è maggiore di qualsiasi altro punto fisso.

Teorema del punto fisso per funzioni monotone su reticoli completi (Knaster-Tarski).
Se D è un reticolo completo, ogni funzione monotona $f: D \to D$ ammette:

- $mpf(f) = inf\{x \mid f(x) \sqsubseteq x\} = inf\{x \mid f(x) = x\}$.
- $MPF(f) = sup\{x \mid x \sqsubseteq f(x)\} = sup\{x \mid f(x) = x\}$.

Teorema del punto fisso per funzioni continue su domini (Kleene).
Siccome avremo spesso a che fare con i domini, ovvero strutture meno ricche dei reticoli, ci servirà questo secondo teorema del punto fisso in cui indeboliamo la struttura ma rafforziamo la funzione (andando a chiedere che sia anche continua oltre che monotona). Se D è un dominio, ogni funzione continua $f: D \to D$ ammette: $mpf(f) = sup\{f^n(\bot)\}_{n \geq 0}$.

Esempio di equazione ricorsiva: il fattoriale.

Possiamo definire il *fattoriale* come il punto fisso di una funzione di ordine superiore ϑ così definita:

$$\vartheta = \lambda f \in [D \to D].\ \lambda n \in D.\ n=0 \to 1,\ n*f(n-1)$$

Si noti che il risultato di ϑ appartiene al dominio *D*. Un *funzionale* si può quindi vedere come qualcosa che prende una funzione e restituisce un elemento, ma anche come qualcosa che prende una funzione e restituisce un'altra funzione. Se si passa la funzione $fattoriale = \lambda n.\ n=0 \to 1,\ n*fattoriale(n-1)$ come primo argomento di ϑ si ottiene l'equazione ricorsiva $\vartheta(fattoriale) = fattoriale$. Risolveremo tale equazione per punto fisso di ϑ, ovvero applicheremo il teorema del punto fisso per funzioni continue su domini.

Dimostrazione
Le funzioni costanti, le funzioni dell'aritmetica e la composizione di funzioni sono tutte funzioni *continue*. Si può inoltre dimostrare che anche la funzione condizionale è una funzione *continua*. Il teorema del punto fisso di Kleene dice che esiste un minimo punto fisso per ϑ e che esso è uguale all'estremo superiore della catena ottenuta applicando ϑ all'indefinito per *i* volte: $mpf(\vartheta) = sup_{i\geq 0}\{\vartheta^i(\perp)\}$. Si può quindi dimostrare che il minimo punto fisso corrisponde proprio alla funzione fattoriale: $$\vartheta^1(\perp)(x) = \begin{cases} 1 & \text{se } x = 0 \\ \perp & \text{altrimenti} \end{cases}$$ $$\vartheta^2(\perp)(x) = \begin{cases} 1 & \text{se } x = 0 \\ 1 * \vartheta^1(\perp)(0) = 1 & \text{se } x = 1 \\ \perp & \text{altrimenti} \end{cases}$$ $$\vartheta^i(\perp) = \lambda n.\ n = 0 \to 1,\ n * \vartheta^{i-1}(\perp)(n-1)$$ In sostanza, si calcola di volta in volta un'approssimazione della funzione fattoriale che restituisce il fattoriale di *0,1,…,i-1* per *x=0,1,…,i-1* e indefinito altrimenti. L'estremo superiore della catena formata da tali approssimazioni è proprio la *funzione fattoriale*.

Semantica denotazionale di un linguaggio modello di tipo imperativo.

Il linguaggio modello adottato si chiama *IMP*. Tale linguaggio permette di lavorare su *valori numerici*, *valori booleani* e *celle di memoria* (locazioni). Esiste inoltre la nozione di *stato* della memoria, ovvero una mappa che assegna ad ogni cella di memoria il rispettivo valore. Andremo quindi a denotare le *espressioni aritmetiche*, le *espressioni booleane* e i *comandi*.

Linguaggio IMP		
COSTRUTTO	SINTASSI	DOMINIO
Valori numerici	*n, m, …*	*N*
Valori booleani	*vero, falso,* $\perp$	*B*
Celle di memoria	*X, Y, …*	*Loc*
Stato	σ*: Loc* $\to$ *N* $\cup$ *B*	Σ
Espressioni aritmetiche	$+, -, \times$	*AExp*
Espressioni booleane	$\neg, \wedge, \vee, =, \leq$	*BExp*
Comandi		*Com*
• Comando vuoto	*skip*	
• Assegnamento	*X := e*	
• Composizione	c_0 *;* c_1	
• Selezione	*if b then* c_0 *else* c_1	
• Iterazione	*while b do c*	

Denotazioni.
Le *denotazioni* sono funzioni che associano ad ogni espressione aritmetica, booleana o comando una *funzione* della forma presentata nella tabella sottostante.

Denotazioni di IMP	
FUNZIONE	REGOLE
$\mathcal{A}: AExp \to (\Sigma \to N)$	
• Valori numerici	$\mathcal{A}[\![n]\!] = \lambda\sigma.n$
• Celle di memoria	$\mathcal{A}[\![X]\!] = \lambda\sigma.\sigma(X)$
• Operatori aritmetici (*op = {+, -, ×}*)	$\mathcal{A}[\![a_0\ op\ a_1]\!] = \lambda\sigma.\mathcal{A}[\![a_0]\!]\sigma\ op\ \mathcal{A}[\![a_1]\!]\sigma$
$\mathcal{B}: BExp \to (\Sigma \to B)$	
• Valori booleani	$\mathcal{B}[\![vero]\!] = \lambda\sigma.vero$
	$\mathcal{B}[\![falso]\!] = \lambda\sigma.falso$
• Operatori booleani (*op = {∧, ∨}*)	$\mathcal{B}[\![\neg b]\!] = \lambda\sigma.\neg(\mathcal{B}[\![b]\!]\sigma)$
	$\mathcal{B}[\![b_0\ op\ b_1]\!] = \lambda\sigma.\mathcal{B}[\![b_0]\!]\sigma\ op\ \mathcal{B}[\![b_1]\!]\sigma$
	$\mathcal{B}[\![a_0 = a_1]\!] = \lambda\sigma.\mathcal{A}[\![a_0]\!]\sigma = \mathcal{A}[\![a_1]\!]\sigma$
	$\mathcal{B}[\![a_0 \leq a_1]\!] = \lambda\sigma.\mathcal{A}[\![a_0]\!]\sigma \leq \mathcal{A}[\![a_1]\!]\sigma$
$\mathcal{C}: Com \to (\Sigma \to \Sigma)$	
• Comando vuoto	$\mathcal{C}[\![skip]\!] = \lambda\sigma.\sigma$
• Assegnamento	$\mathcal{C}[\![X := a]\!] = \lambda\sigma.\sigma[X \leftarrow \mathcal{A}[\![a]\!]\sigma]$
• Composizione	$\mathcal{C}[\![c_0; c_1]\!] = \lambda\sigma.(\mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!])\sigma$
• Selezione	$\mathcal{C}[\![if\ b\ then\ c_0\ else\ c_1]\!] = \lambda\sigma.\mathcal{B}[\![b]\!]\sigma \to \mathcal{C}[\![c_0]\!]\sigma, \mathcal{C}[\![c_1]\!]\sigma$
• Iterazione (*w = while b do c*)	$\mathcal{C}[\![w]\!] = \mathcal{C}[\![if\ b\ then\ c; w\ else\ skip]\!] = \lambda\sigma.\mathcal{B}[\![b]\!]\sigma \to \mathcal{C}[\![c; w]\!]\sigma, \mathcal{C}[\![skip]\!]\sigma$

Approfondimento sulla semantica denotazionale del while.
Per definire la semantica denotazionale del while è stato applicato il concetto di *equazione ricorsiva*. Se risolvessimo quindi l'equazione $\Gamma(\mathcal{C}[\![w]\!]) = \mathcal{C}[\![w]\!]$ per punto fisso troveremmo di volta in volta un'*approssimazione* del while che fa al più *n+1* verifiche della condizione booleana (ovvero *n* esecuzioni del corpo del ciclo) e restituisce indefinito altrimenti.

$$\Gamma^{n+1}(\bot) = \Gamma\left(\Gamma^n(\bot)\right) = \lambda\sigma.\mathcal{B}[\![b]\!]\sigma \to (\Gamma^n(\bot) \circ \mathcal{C}[\![c]\!])\sigma, \sigma$$

L'*estremo superiore* della catena formata da tali approssimazioni sarà proprio la denotazione del while: $\mathcal{C}[\![w]\!] = sup_{n\geq 0}\{\Gamma^n(\bot)\}$.

Forma curried e uncurried di funzioni.
I domini di definizione delle funzioni si possono scrivere in due diverse *forme*:

- Versione curried *g: D_1 → [D_2 → D_3]*, ricordando che "→" associa a destra.
- Versione uncurried *g': $D_1 \times D_2$ → D_3*.

Per la *trasformazione* da una forma all'altra devono essere usate le seguenti funzioni:

- *curry = λf. λx. λy. f(x,y)* tale che *curry(g') = g*.
- *uncurry = λf. λ(x,y). f(x)(y)* tale che *uncurry(g) = g'*.

Esempi di applicazione	
somma: N → [N → N] tale che *somma = λx. λy. x+y*	*twice = λf. λx. f(f(x)): [D → D] → D → D*
somma': N × N → N tale che *somma'= λ(x,y). x+y*	*twice' = λ(f,x). f(f(x)): [D → D] × D → D*
curry(somma') = λx. λy. somma'(x,y) = λx. λy. x+y	*curry(twice') = λx. λy. twice'(x,y) = λx. λy. x(x(y))*
uncurry(somma) = λ(x,y). somma x y = λ(x,y). x+y	*uncurry(twice) = λ(x,y). twice x y = λ(x,y). x(x(y))*

➢ **Semantica assiomatica e regole di Hoare** – *G. Winskel: The formal semantics of programming languages* [capitoli 6 e 7]

Asserzioni e annotazioni.
Per definire la semantica assiomatica di IMP abbiamo bisogno di asserzioni e annotazioni. Le *asserzioni* contengono le annotazioni e consistono in formule logiche che sono vere tutte le volte che l'esecuzione del programma passa per il punto in cui si trovano. Le *annotazioni*, invece, si dividono in:

- *Pre-condizioni*, ovvero formule logiche che sono vere all'inizio del programma.
- *Post-condizioni*, ovvero formule logiche che sono vere alla fine del programma.

Un *programma* può essere anche solo una procedura, un ciclo, un blocco o un frammento di codice.

Invarianti di ciclo.
La semantica assiomatica verrà utilizzata per costruire gli *invarianti di ciclo*, ovvero formule logiche che sono vere:

- Prima di entrare nel ciclo.
- Dopo ogni esecuzione del ciclo.
- Quando si esce dal ciclo.

Struttura delle frasi.
Nella semantica assiomatica tratteremo *frasi* del tipo *{A} c {B}* dove *A* e *B* sono formule logiche e *c* è un comando. Inoltre, *A* è detta pre-condizione di *c* e *B* è detta post-condizione di *c*.

Verità delle formule logiche.
Per stabilire se le formule logiche sono vere occorre avere le nozioni di interpretazione e di modello. L'*interpretazione* dà un significato ai simboli presenti nelle formule logiche (connettivi, uguaglianze, operatori dell'aritmetica, costanti numeriche, variabili del programma), mentre il *modello* è l'interpretazione che soddisfa una formula (ad es. gli stati σ sono un'interpretazione delle variabili del programma e quindi possono essere visti come modelli perché soddisfano le formule logiche nei vari punti di programma).

Regole della logica di Hoare.
Nella tabella sottostante vengono presentate le *regole di inferenza* della logica di Hoare per la semantica assiomatica.

Regole di Hoare	
COSTRUTTO	REGOLA
Comando vuoto	$\{A\}\ skip\ \{A\}$
Assegnamento	$\{B[a]\}\ X := a\ \{B[X]\}$
Composizione	$\dfrac{\{A\}\ c_0\ \{D\} \quad \{D\}\ c_1\ \{B\}}{\{A\}\ c_0; c_1\ \{B\}}$
Selezione	$\dfrac{\{A \wedge b\}\ c_0\ \{D\} \quad \{A \wedge \neg b\}\ c_1\ \{D\}}{\{A\}\ if\ b\ then\ c_0\ else\ c_1\ \{D\}}$
Iterazione	$\dfrac{\{I \wedge b\}\ c\ \{I\}}{\{I\}\ while\ b\ do\ c\ \{I \wedge \neg b\}}$
Implicazione	$\dfrac{A \to A' \quad \{A'\}\ c\ \{B'\} \quad B \to B'}{\{A\}\ c\ \{B\}}$

Per *verificare programmi* che contengono cicli while useremo spesso la seguente regola che combina iterazione e implicazione:

$$\frac{A \to I \quad \dfrac{\{I \wedge b\}\ c\ \{I\}}{\{I\}\ while\ b\ do\ c\ \{I \wedge \neg b\}} \quad I \wedge \neg b \to D}{\{A\}\ while\ b\ do\ c\ \{D\}}$$

Correttezza e completezza.
Le regole della logica di Hoare sono *corrette*. Tuttavia, non possiamo dimostrare la *completezza* poiché per il teorema di incompletezza di Gödel non esiste un sistema completo per l'aritmetica del primo ordine. Possiamo però dire che le regole della logica di Hoare sono *relativamente complete* (Stephen Cook).

Pre-condizione più debole.
La pre-condizione più debole (weakest precondition, *wp*) di un comando *c* rispetto a una formula *G* è una funzione *wp: Assn × Com* → *Assn*, dove *Assn* è l'insieme delle asserzioni, tale che $wp[\![c, G]\!] = F$ se e solo se $\forall\sigma.\sigma \vDash F \rightarrow \mathcal{C}[\![c]\!]\sigma \vDash G \wedge \forall F'.(\forall\sigma.\sigma \vDash F' \rightarrow \mathcal{C}[\![c]\!]\sigma \vDash G) \rightarrow (F' \rightarrow F)$.

Regole per il calcolo di wp	
COSTRUTTO	REGOLA
Comando vuoto	$wp[\![skip, G]\!] = G$
Assegnamento	$wp[\![X := a, G[X]]\!] = G[a]$
Composizione	$wp[\![c_0; c_1, G]\!] = wp[\![c_0, wp[\![c_1, G]\!]]\!]$
Selezione	$wp[\![if\ b\ then\ c_0\ else\ c_1, G]\!] = \\ = (b \wedge wp[\![c_0, G]\!]) \vee (\neg b \wedge wp[\![c_1, G]\!])$
Assunzione	$wp[\![assume\ B, G]\!] = B \rightarrow G$

Si può notare che le regole sono state date per tutti i costrutti eccetto l'*iterazione*, in quanto c'è la necessità di spezzare il programma in *cammini elementari* (concetto che vedremo più avanti).

➢ **Correttezza dei programmi** – *A. R. Bradley, Z. Manna: The calculus of computation* [capitoli 5 e 6]

Quadro generale.
La semantica assiomatica fornisce la base per un approccio deduttivo alla *verifica dei programmi*. Nella realtà, per automatizzare la semantica assiomatica, vengono usati un compilatore verificatore e un dimostratore di teoremi. Il *compilatore verificatore* permette di passare, attraverso il calcolo delle *wp*, da un programma annotato alle condizioni di verifica. Il *dimostratore di teoremi*, invece, dimostra per assurdo la validità delle condizioni di verifica oppure segnala che la negazione di una condizione di verifica è soddisfacibile, ovvero che la condizione di verifica non è valida.

Nuovo linguaggio modello.
Arricchiamo ora il linguaggio modello *IMP* con array, record (o strutture), cicli for e funzioni (con passaggio dei parametri per valore) per formare il nuovo linguaggio *pi*. Di conseguenza, dovremo anche arricchire la definizione di *annotazione*. Diremo quindi che le annotazioni in *pi* possono essere:

- *Pre-condizioni* (anche di funzioni).
- *Post-condizioni* (anche di funzioni).
- *Invarianti di ciclo* (anche per il ciclo for in quanto si può tradurre in un equivalente ciclo while).
- *Asserzioni a tempo di esecuzione*.
- *Asserzioni relative alle chiamate di funzione*.

Pre-condizioni e post-condizioni di funzioni.
La *pre-condizione di una funzione* è una formula le cui variabili libere includono i parametri formali della funzione. La *post-condizione di una funzione*, invece, è una formula le cui variabili libere includono i parametri formali della funzione e una variabile speciale *rv* che contiene il valore restituito.

Asserzioni a tempo di esecuzione.
Sono asserzioni usate per evitare *errori* che si possono verificare a tempo di esecuzione, come ad esempio:

- Divisione per zero, *X div Y @y ≠ 0*.
- Modulo zero, *X mod Y @y ≠ 0*.
- Violazione dei limiti di un array, *a[i] @l ≤ i ≤ u*.
- De-referenziazione di un puntatore nullo.

Cammini elementari di un programma.
Per poter definire $wp[\![c, G]\!]$ per ogni comando *c* ammesso dal linguaggio modello *pi* (quindi anche per cicli e funzioni) e per ogni formula *G* in modo che il calcolo di $wp[\![c, G]\!]$ stesso sia eseguibile in modo automatico dal compilatore verificatore bisogna scomporre il codice nei cosiddetti *cammini elementari* (basic paths). Un cammino elementare è una sequenza di comandi che comincia con una formula che è o una *pre-condizione* o un *invariante* e finisce con una formula che è o una *post-condizione* o un *invariante* o un'*asserzione*.

Condizione di verifica.
Con tutto quello che abbiamo visto finora possiamo dedurre che per il calcolo della *pre-condizione più debole di un cammino elementare* basta considerare le regole per il calcolo della pre-condizione più debole di assegnamenti, assunzioni e concatenazione di comandi. Dopo aver calcolato la pre-condizione più debole di un cammino elementare, non resta che applicare la *condizione di verifica* $F \rightarrow wp[\![c_0; c_1; \ldots; c_k, G]\!]$, la cui validità assicura che ogni qual volta il programma raggiunge l'inizio del cammino elementare formato dai comandi $c_0; c_1; \ldots; c_k$ in uno stato σ tale che $\sigma \vDash F$, l'esecuzione di tale cammino elementare porterà in uno stato σ' tale che $\sigma' \vDash G$.

Prototipo di una funzione.
Il *prototipo* di una funzione può essere schematizzato come:

- Una *pre-condizione* sui parametri formali della funzione — *@pre: F[p_1, …, p_n]*
- La *definizione* della funzione con i tipi di tutti i parametri formali — *$tipo_0$ f($tipo_1$ p_1, …, $tipo_n$ p_n)*
- Una *post-condizione* sui parametri formali e sul valore di ritorno — *@post: G[p_1, …, p_n, rv]*

Asserzioni relative alle chiamate di funzione.
Per poter applicare la tecnica dei cammini elementari a funzioni che chiamano altre funzioni ci servono:

- *Asserzioni in chiamata*, ovvero istanze della pre-condizione della funzione chiamata dove le variabili libere (cioè i parametri formali) sono istanziate con i parametri attuali della particolare chiamata.
- *Sommari di chiamata*, ovvero istanze della post-condizione della funzione chiamata dove le variabili libere (cioè i parametri formali) sono istanziate con i parametri attuali della particolare chiamata e la variabile speciale *rv* per il valore restituito è istanziata con una variabile nuova.

In questo modo i nostri *cammini elementari* potranno o finire con un'asserzione in chiamata di funzione (cioè con lo stato in cui vogliamo che si trovi il programma appena prima che venga chiamata la funzione) o attraversare la chiamata di funzione assumendo il sommario di chiamata (cioè lo stato in cui vogliamo che si trovi il programma appena dopo aver chiamato la funzione).

Esempi

PROTOTIPO DELLA FUNZIONE	TIPO DI CHIAMATA	CAMMINI ELEMENTARI
@pre: x ≥ 0 *int g(int x);* *@post: rv ≥ x*	*w := g(n+1);*	Cammino 1: … *@R: n+1 ≥ 0*
		Cammino 2: … *assume v ≥ n+1;* *w := v;* …
	return g(n+1);	Cammino 1: … *@R: n+1 ≥ 0*
		Cammino 2: … *assume v ≥ n+1;* *rv := v;* …