Riassunto MFI 21/22

Sintassi dei linguaggi

Abbiamo studiato in corsi precedenti la sintassi di un linguaggio di programmazione, ora siamo interessati a definire una semantica. Studieremo i linguaggi come dei sistemi formali: essi si compongono di un alfabeto di simboli, un insieme di espressioni ben formate e delle **regole di inferenza**.

Partendo da una sintassi concreta (una grammatica context free):
Costanti n := 0 | 1 | …
Variabili x := x_0 | x_1 | …
Aexp a := n | x | a + a | a * a | …

Possiamo definire un insieme R di regole di inferenza, nella forma
R = {...|(e_1,...,e_n,e_{n+1})|...}
I cui elementi sono delle relazioni n+1 arie, quando scriviamo
(e_1,...,e_n,e_{n+1}) per un qualche n intendiamo:
e_1,...,e_n sono le **premesse** mentre e_{n+1} è la **conclusione**.

Scritto anche:
$$\text{r}\ \frac{e_1 \ \ldots\ e_n}{e_{n+1}}$$
Dove r denota il nome della regola, il significato è: se valgono tutte le premesse, allora vale la conclusione.

E se n = 0? in tal caso abbiamo una regola senza premesse e con solo una conclusione, essendo senza premesse questa regola vale sempre. In tale caso la regola viene chiamata **assioma**.

Le regole di inferenza mi permettono di scrivere una **derivazione** su un'espressione della grammatica: a partire dall'espressione cerco di raggiungere gli assiomi. Se riesco a costruire una derivazione per un'espressione allora l'espressione si dice **ben formata**.

Esempio di regole di inferenza per la grammatica precedentemente definita:

$$\frac{}{n \in Aexp} \qquad \frac{}{x \in Aexp}$$

$$\frac{a_1 \in Aexp \quad a_2 \in Aexp}{a_1 + a_2 \in Aexp}\ \text{regola del +}$$

Introduciamo ora la sintassi astratta (BNF): vogliamo togliere i dettagli futili, prevenire ambiguità e sfruttare la curryficazione.

nome := stringa
aexp := N n | V x | PLUS aexp aexp | TIMES aexp aexp

Dove N specifica che l'elemento è un numerale mentre V che l'elemento è una variabile.

Quindi una stringa in una sintassi concreta come "2 * (y + 3)" diventa:
TIMES (N 2) (PLUS (V y) (N 3))
Notiamo che PLUS/TIMES : aexp -> aexp -> aexp, ovvero due funzioni che dati due elementi aexp restituiscono un'espressione aritmetica.
Invece N : N -> aexp e V : stringa -> aexp.

Notiamo che abbiamo scritto una **definizione induttiva**:
Da un numero finito di clausole abbiamo generato un insieme infinito di espressioni aritmetiche.
"Una ***definizione*** è ***induttiva*** se costruita a partire da un numero finito di clausole che introducono i *costruttori* degli elementi dell'insieme, ossia di quelle costanti e simboli funzionali che, applicati ripetutamente gli uni agli altri secondo le regole della definizione, generano tutti e soli gli elementi dell'insieme definito."

Un esempio di definizione induttiva è la seguente, dei numeri naturali:
data N : Set where
  zero : N
  suc : N → N

Un esempio di definizione induttiva di una funzione è la seguente:
size : aexp -> N
size (N n) = 1
size (V x) = 1
size (PLUS a b) = 1 + size a + size b

Semantica delle espressioni

Ora che abbiamo definito una sintassi astratta per le espressioni aritmetiche possiamo definire una semantica, ovvero dargli un significato. Si noti che, anche se 0 + 0 = 0, PLUS (N 0) (N 0) != (N 0) da un punto di vista sintattico (sono due espressioni diverse nella sintassi) ma semanticamente uguali.

Ovviamente un'espressione aritmetica se valutata produrrà un numero intero. Prima di fare ciò, ci serve un modo per valutare le variabili, che numero può restituirmi una variabile come x? Visto che siamo in un linguaggio imperativo, il valore di x dipende dallo **stato**!

State s : vname -> valore.

Possiamo ora definire la semantica di un'espressione aritmetica mediante la funzione **aval**:
aval : aexp -> state -> valore
aval (N n) s = n
aval (V x) s = s x
aval (PLUS a b) s = (aval a s) + (aval b s)
Dove in questo caso la + rappresenta la somma di naturali

Proprietà della semantica in aval:
Definiamo una funzione Fv che rappresenta l'insieme delle variabili libere in un'espressione aritmetica.
Fv (N n) = {}
Fv (V x) = {x}
Fv (PLUS a b) = Fv(a) U Fv(b)

Lemma:

Se per ogni x ∈ Fv(a) gli stati s e s' coincidono, ovvero s(x) = s'(x), allora aval a s = aval a s'

In generale in un'espressione aritmetica tutte le variabili sono libere, e se ad una variabile libera volessimo assegnare un preciso valore?
Facciamo una **sostituzione**.
Con a[a'/x] intendiamo la sostituzione con a' di x in a.

E se non volessimo sostituire una variabile, ma esplicitare il suo valore in un certo stato?
Con s[x |-> n] indichiamo che la variabile x ha valore n nello stato s, mentre qualunque variabile y != x avrà valore s y.

Lemma di sostituzione:

aval (a[a'/x]) s = aval a s[x |-> aval a' s]

Definiamo similmente
bexp := Bc bool | NOT bexp | AND bexp bexp | LESS aexp aexp

E gli diamo una semantica mediante bval : bexp -> state -> bool
bval (Bc x) s = x
bval (Not b) s = not (bval b s)
bval (And a b) s = bval a s ∧ bval b s
bval (Less a b) s = aval a s ≤? aval b s

Semantica dei linguaggi

Ora che abbiamo introdotto espressioni aritmetiche e espressioni booleane possiamo passare a definire un linguaggio mediante dei **comandi**. Semantica BNF dei comandi com:
com := SKIP //comando che non fa nulla
| vname := aexp //assegnazione
| com ; com //composizione sequenziale
| IF bexp THEN com ELSE com //selezione
| WHILE bexp DO com //iterazione

E qual'é la semantica di un comando? Cosa comporta la sua esecuzione? Una trasformazione di stati! Definiamo quindi la semantica di un comando come la trasformazione di uno stato in un altro.

Potremmo definire una funzione induttiva cval, ma abbiamo un problema: non abbiamo sicurezza se una stringa di comandi termini (si noti che il problema di dire se un insieme di comandi terminerà o meno è non decidibile). Come facciamo? Introduciamo un nuovo strumento, la **semantica naturale/big-step.**
Usiamo la relazione (c, s) => t (di tipo com x state x state) con il seguente significato: "l'esecuzione di c in s termina e restituisce lo stato t". s viene detto stato **iniziale**.

La semantica big-step viene definita come un sistema formale, guardare su appunti le regole di inferenza.

proposizione:

qualunque s,t, non si può derivare che (while true do skip, s) => t

Equivalenza di programmi:

I comandi c1 e c2 si dicono **equivalenti** sse
qualunque s,t : (c1, s) => t ⇔ (c2, s) => t

Introduciamo ora un teorema fondamentale per le semantiche, la dimostrazione è scritta negli appunti:

Teorema determinismo della semantica naturale:

Per ogni c ∈ com, s,t,t' ∈ state
(c, s) => t e (c, s) => t' implica t = t'.

Quindi l'esecuzione di un comando su uno stato se termina produce sempre lo stesso stato.

Sfruttando il teorema del determinismo, possiamo definire una funzione parziale $[\ \_\ ]_{nat}$: com x state x state
dove $[c]_{nat}$ s =

t , se esiste una derivazione (c, s) => t
bottom , altrimenti

In generale [c] s ci restituisce lo stato in cui termina l'esecuzione di c a partire da s se l'esecuzione termina.

Potremmo essere interessati a studiare l'esecuzione di un comando in uno stato analizzandone la **riduzione**, per fare ciò viene introdotta una nuova semantica detta **small-step**, basata sui sistemi di riduzione:

Un sistema di riduzione è una coppia (S, →) dove S è un insieme di stati e → un sottoinsieme di S x S. Presi due stati x,y ∈ S si scrive x → y per abbreviare (x, y) ∈ →, "lo stato x si riduce nello stato y".

Un esempio di sistema di riduzione sono gli automi a stati finiti, in tal caso S rappresenta una coppia (stringa da leggere, stato) e la riduzione → si basa sulla funzione di transizione.

Nel caso di IMP S è una coppia (comando, stato) e la funzione di transizione rappresenta l'esecuzione parziale del comando con l'eventuale modifica dello stato, (c, s) → (c', s'), detto passo di calcolo.

In questo contesto il comando SKIP assume il significato di terminazione dell'esecuzione. Vedere appunti riduzione di assegnazione, composizione, IF false.
Caso interessante:

(WHILE b DO c, s) → (IF b THEN (c; WHILE b DO c) ELSE SKIP, s)

Definizione:

$(c, s) \xrightarrow{k} (c', t) \Leftrightarrow$ (c, s) si riduce a (c', t) in k passi.
$(c, s) \xrightarrow{*} (c', t) \Leftrightarrow \exists k$ t.c. $(c, s) \xrightarrow{k} (c', t)$

lemma determinismo small-step:

$\forall$ c,s se (c, s) $\rightarrow$ (c', s') e (c, s) $\rightarrow$ (c'', s'')
allora c' = c'' e s' = s''

Anche in questa semantica possiamo introdurre una definizione di terminazione: (c, s) termina $\Leftrightarrow$ $\exists$ t t.c. (c, s) $\overset{*}{\rightarrow}$ (SKIP, t). (c, s) cicla se esiste una sequenza infinita (c, s) $\rightarrow$ (c0, s0) $\rightarrow$ (c1, s1) $\rightarrow$ …

Possiamo allora definire una funzione $[\ \_\ ]_{sos}$: com x state x state
$[\ c\ ]_{sos}\ s$ =

t , se (c, s) $\overset{*}{\rightarrow}$ (SKIP, t)
bottom , se (c, s) cicla

Teorema di equivalenza di semantiche:

$\forall c \in$ com, s,t $\in$ state, $[\ c\ ]_{nat}\ s = [\ c\ ]_{sos}\ s$
alternativamente (c, s) => t $\Leftrightarrow$ (c, s) $\overset{*}{\rightarrow}$ (SKIP, t)

La dimostrazione si basa sul dimostrare le due implicazioni, sfruttando due lemmi per caso composizione. Dimostrazioni qui.

Lemma 1:

se (c1, s) $\overset{k}{\rightarrow}$ (SKIP, s') allora (c1; c2 , s) $\overset{k+1}{\rightarrow}$ (c2, s')

Lemma 2:

(c, s) => t $\Rightarrow$ (c, s) $\overset{*}{\rightarrow}$ (SKIP, t)

Lemma 3:

se (c1;c2 , s) $\overset{k}{\rightarrow}$ (SKIP, s'') allora $\exists$ s', k_0, k_1 t.c.
(c_1, s) $\overset{k0}{\rightarrow}$ (SKIP, s') e (c_2, s') $\overset{k1}{\rightarrow}$ (SKIP, s'') e k = k_0 + k_1 + 1

Lemma 4:

(c, s) $\overset{k}{\rightarrow}$ (SKIP, t) $\Rightarrow$ (c, s) => t

Dai lemmi 2 e 4 si dimostra $\Leftrightarrow$.

Compilatore per IMP

La compilazione di un insieme di comandi com deve produrre una sequenza di istruzioni per una macchina astratta. Definiamo queste istruzioni con una definizione induttiva.
In generale mi aspetto che

se (c, s) => t allora
p ⊦ (0, s, []) $^{*}\rightarrow$ (size p, t, [])

Dove (program counter, stato, stack) e p lista di istruzioni

Definiamo allora un tipo config: int x state x stack
Dove stack = val list //lista di valori
Definiamo le istruzioni, ispirandosi a quelle già viste in architettura:

instr := LOADI n
| LOAD x
| ADD
| JMP p
| JMPLESS p
| JMPGE p
| STORE x

E infine una funzione esecutore iexec : instr → config → config
La quale rappresenta la semantica delle istruzioni macchina.

iexec (LOADI n) (i, s, stk) = (i+1, s, n # stk) //ovvero n in cima allo stack
iexec (LOAD x) (i, s, stk) = (i+1, s, (s x) # stk)
iexec (ADD) (i, s, m # (n # stk)) = (i+1, s, (m+n) # stk)
iexec (STORE x) (i, s, n # stk) = (i+1, s[x |-> n], stk)
iexec (JMP p) (i, s, stk) = (i + 1+ p, s, stk)
iexec (JMPLESS p) (i, s, m # (n # stk)) =
(IF n < m THEN i + 1 + p ELSE i + 1, s, stk)
iexec (JMPGE p) (i, s, m # (n # stk)) =
(IF n ≥ m THEN i + 1 + p ELSE i + 1, s, stk)

Notiamo che abbiamo un program counter, quindi abbiamo necessità di poter leggere le parti di programma di una data posizione.

Introduciamo la funzione parziale lookup: int → prog → instr
lookup 0 (x # xs) = x
lookup i+1 (_ # xs) = lookup i xs

Ora possiamo definire cosa significa eseguire un'istruzione di p:

$$\frac{0 \le i < \text{size } p \qquad \text{iexec (lookup } i\ p)\ (i, s, stk) \equiv (i', s', stk')}{p \vdash (i, s, stk) \to (i', s', stk')}$$

Dove tipo p : prog = instr list e size p = lunghezza lista p

Def: $p \vdash (i, s, stk) \overset{*}{\to} (i', s', stk')$ la chiusura riflessiva e transitiva della relazione.

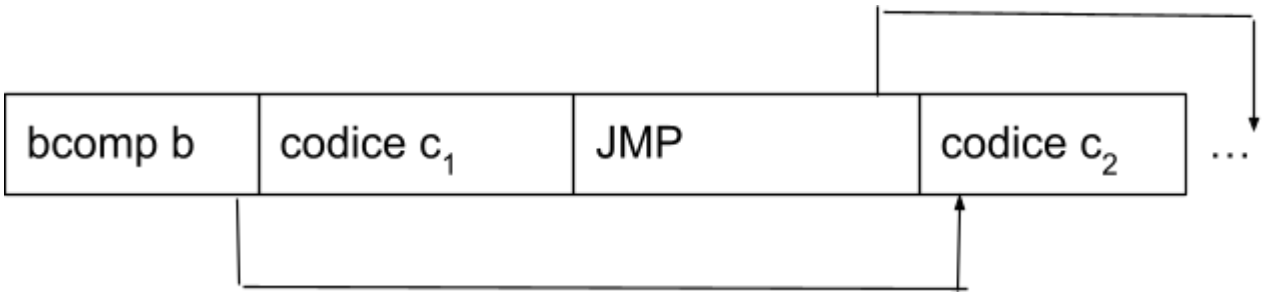
Ora che abbiamo definito le istruzioni e l'esecutore possiamo definire il compilatore per le espressioni aritmetiche acomp : aexp → instr
acomp (N n) = [LOADI n]
acomp (V x) = [LOAD x]
acomp (PLUS a_1 a_2) = acomp a_1 @ acomp a_2 @ [ADD]

Lemma 8.6:
acomp a $\vdash$ (0, s, stk) $\overset{*}{\to}$ (size (acomp a), s, (aval a s) # stk)
Quindi compilare e eseguire un'espressione a equivale a valutarla e caricare il suo valore in cima allo stack.

Lemma 8.8:
Sia pc' = size (bcomp b f n) + (IF bval b s = f THEN n ELSE 0)
dove f flag booleano, b var booleana e n offset.

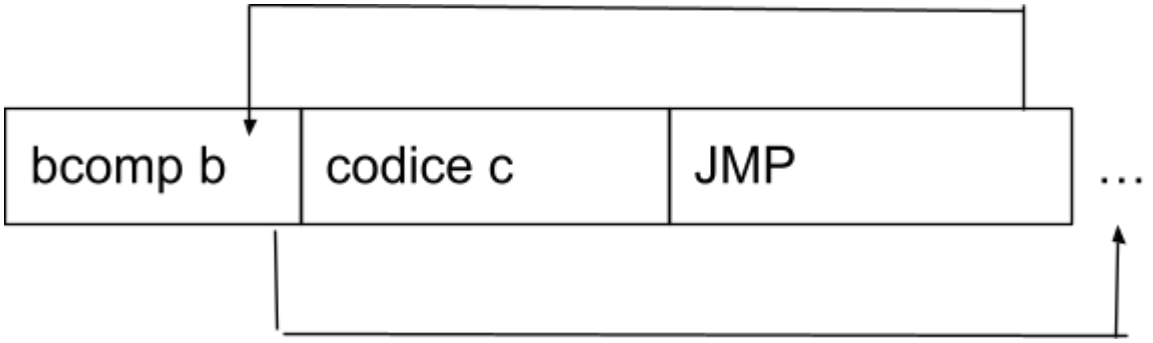
A pc' oltre alla size del codice che serve per eseguire la valutazione di b aggiungo il valore n se il valore di b equivale al flag f.
Ma a cosa serve? A gestire i salti degli if e dei while.

Immaginiamo le istruzioni per il comando IF b THEN c_1 ELSE c_2



Se if vero eseguo c_1 e salto, se if falso salto a c_2 e proseguo

Codice per il comando WHILE b do c



Quindi serve che la compilazione di un'espressione booleana tenga conto della nostra necessità di saltare blocchi di istruzioni in base al valore di b.
Sfruttiamo il lemma 8.8 per definire bcomp: bexp → bool → int → prog

bcomp (Bv v) f n = (IF v == f THEN [JMP n] ELSE [])
    //se valore booleano uguale al flag saltiamo, altrimenti no
bcomp (NOT b) f n = bcomp b -f n
bcomp (AND b_1 b_2) f n =
    [siano cb_2 = bcomp b_2 f n
        m = IF f THEN (size cb_2) ELSE (size cb_2 + n)
        cb_1 = bcomp b_1 false m]
    cb_1 @ cb_2
    /*idea:
    se b_1 vale false non serve valutare b_2, tutto AND varrà false
    quindi se b_1 == false salta di size cb_2 o size cb_2 + n in base al flag
    se b_1 invece vale true dobbiamo valutare b_2, e in cb_2 sarà deciso il salto: ricorda che TRUE && b_2 equivale a b_2*/

bcomp (LESS a_1 a_2) f n =

acomp a_1 @ acomp a_2 @ (IF f THEN [JMPLESS n] ELSE [JMPGE n])

//carica a_1 e a_2 in cima allo stack e in base a f calcola se sono minore o maggiore e uguali

Ora possiamo definire un **compilatore per i comandi**

ccomp: com → prog

ccomp SKIP = []

ccomp (x := a) = acomp a @ [STORE x]

ccomp (c_1 ; c_2) = ccomp c_1 @ cccomp c_2

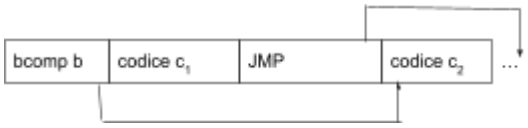
ccomp (IF b THEN c_1 ELSE c_2) = //sfruttando lemma 8.8

[siano cc_1 = ccomp c_1

cc_2 = ccomp c_2

cb = bcomp b false (size cc_1 + 1)]

cb @ cc_1 @ JMP (size cc_2) @ cc_2

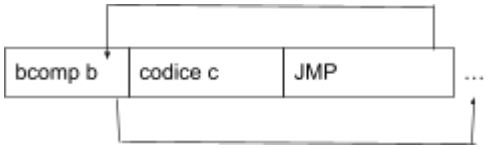


ccomp (WHILE b DO c) =

[siano cc = ccomp c

cb = bcomp b false (size cc + 1)]

cb @ cc @ JMP -(size cc + size cb + 1)



<u>Teorema</u>: correttezza del compilatore

∀ c : com, s,t : state, stk: stack

se (c, s) => t ⇒ ccomp c ⊦ (0, s, stk) *→ (size ccomp c, t, stk)

Si può dimostrare facendo induzione sulla derivazione di (c, s) => t

Si deve utilizzare il <u>lemma 8.5</u>:

se p ⊦ (0, s, stk) *→ (i', s', stk') e size p ≤ i'

e p' ⊦ (i' - size p, si, stki) *→ (i'', s'', stk'')

allora p @ p' ⊦ (0, s, stk) *→ (i'', s'', stk'')

Tipi per IMP

Scopo dell'introduzione di sistemi di tipo nei linguaggi di programmazione è prevenire alcuni errori durante l'esecuzione, come assegnamenti incoerenti o chiamata di funzioni o metodi con parametri scorretti. Ciò avviene mediante l'analisi statica del codice, o come si dice a tempo di compilazione. La proprietà di escludere una certa classe di errori è un caso di **safety**, ossia la garanzia che qualcosa di indesiderato non possa verificarsi in nessuna possibile esecuzione.

Che l'esecuzione di un codice sia esente da errori è in generale indecidibile; tuttavia i sistemi di tipo come quello esemplificato in questo paragrafo sono decidibili e sono condizione sufficiente (ma ovviamente non necessaria) perché un codice ben tipato non produca errori durante la sua esecuzione.

Estendiamo IMP in maniera tale che le espressioni aritmetiche possano produrre numeri interi e reali.

aexp := Ic i //numeri interi

| Rc r //numeri reali

| Vx

| PLUS a_1 a_2

Dove PLUS è una funzione polimorfa e PLUS (Ic i) (Rc r) produce un errore.

Se definiamo il tipo val come l'insieme dei valori di una funzione aritmetica:

val ::= Iv i | Rv r //valori interi e reali

Allora possiamo definire la seguente relazione:

tval a s v ⇔ La valutazione di a in s ha valore v

Da cui possiamo definire le nuove regole di inferenza:

---------- ---------- ----------
tval (Ic i) s (Iv i) tval (Rc r) s (Rv r) tval (V x) s (s x)

tval a_1 s (Iv i_1) tval a_2 s (Iv i_2)
---------- regola del +
tval (PLUS a_1 a_2) s (Iv (i_1 + i_2))

tval a_1 s (Rv i_1) tval a_2 s (Rv i_2)
---------- regola del +
tval (PLUS a_1 a_2) s (Rv (i_1 + i_2))

Dobbiamo anche modificare la valutazione delle espressioni booleana, un confronto come 1 == 1.0 non è permesso:

tbval b s v ⇔ la valutazione di b in s confronta valori omogenei e vale v

1) ----------
   tbval (Bc bv) s bv

   tbval (Bc b) s bv
2) ----------
   tbval (NOT b) s -bv

   tbval b_1 s bv_1 tbval b_2 s bv_2
3) ----------
   tbval (AND b_1 b_2) s (bv_1 ∧ bv_2)

   tval a_1 s (Iv i_1) tval a_2 s (Iv i_2)
4) ---------- //definita similmente per Rv
   tbval (LESS a_1 a_2) s (i_1 < i_2)

Anche la semantica small-step richiede delle modifiche, per evitare degli errori di tipo:

tval a s v (se espressione a corretta)

(x:= a, s) → (SKIP, s[x |-> v])

tbval b s true (espressione b corretta e restituisce true)

(IF b THEN c_1 ELSE c_2) → (c_1, s)

Sono simili i casi di if-false

Ora che abbiamo aggiunto i valori interi e reali alle espressioni aritmetiche possiamo munire IMP di un sistema di tipi. Il nostro obiettivo è di prevenire la seguente situazione: (c, s) ↛ ma c /= SKIP. Ovvero l'esecuzione del programma non è conclusa ma non può comunque proseguire per un qualche errore.

Avendo valori interi e reali introduciamo i tipi interi e reali:
Type := I_{ty} | R_{ty}

Per prima cosa introduciamo Γ, il contesto di un programma. E' una funzione vname → type che restituisce i tipi di una variabile. Quando in un programma c scriviamo int x vogliamo dire che in quel contesto il tipo della variabile x sarà I_{ty}.

Abbiamo 3 diverse relazioni in base al tipo delle espressioni:
Γ ⊢ a : z //espressione a ben tipata e di tipo z
Γ ⊢ b //espressione bool ben tipata (tipo booleano sottinteso)
Γ ⊢ c //comando c ben tipato in Γ

Introduciamo le regole di inferenza per questa relazioni per espressioni aritmetiche:

--- --- ---

Γ ⊢ I_c i : I_{ty} Γ ⊢ R_c r : R_{ty} Γ ⊢ V x : Γ x

//nel caso della variabili il tipo dipende dal contesto

Γ ⊢ a_1 : δ Γ ⊢ a_2 : δ

Γ ⊢ (PLUS a_1 a_2) : δ

Le regole delle espressioni booleane sono banali. L'unica interessante è quella riguardante il confronto LESS:

$$\frac{\Gamma \vdash a_1 : \delta \qquad \Gamma \vdash a_2 : \delta}{\Gamma \vdash (\text{LESS } a_1\ a_2)}$$

Dove ci si assicura che il tipo di a_1 e a_2 sia lo stesso.

Ora guardiamo le regole per i comandi:

1) $$\frac{}{\Gamma \vdash \text{SKIP}}$$

2) $$\frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x := a}$$ //a ben tipato e dello stesso tipo di x

3) $$\frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1 ; c_2}$$

4) $$\frac{\Gamma \vdash b \qquad \Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2}$$

5) $$\frac{\Gamma \vdash b \qquad \Gamma \vdash c}{\Gamma \vdash \text{WHILE } b \text{ DO } c}$$

Prima di concludere introduciamo una nuova relazione:

$$\Gamma \vdash s \Leftrightarrow \forall x : \text{vname}.\ \text{type}\ (s\ x) = \Gamma\ x$$

Ovvero per qualunque x s assegna ad x valori dello stesso tipo del tipo di x nel contesto Γ. In tal caso Γ e s di dicono **compatibili**.
Sarebbe un problema se nel contesto Γ s fosse una variabile intera ma contenesse valori reali (o viceversa).

Teorema di correttezza per Γ ⊦ c:

Supponiamo che Γ ⊦ c e Γ ⊦ s, allora

∀ c',t'. (c, s) $^{*}\rightarrow$ (c', t') -/> ⇒ c' = SKIP

Equivalente a:

∀ c',t'. (c, s) $^{*}\rightarrow$ (c', t') e c' ≠ SKIP

allora ∃ c'',t''. (c', t') → (c'', t'')

Per dimostrare questo teorema dobbiamo introdurre i lemmi 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, infine dimostriamo vera la seconda definizione del teorema di correttezza. Dimostrazioni dei lemmi si possono trovare qua.

L'idea di base è di dimostrare la seguenti proprietà:

1) preservazione del tipo: se partiamo da un comando corretto e uno stato compatibile, dopo k riduzione saremo in un comando c' corretto e uno stato s' compatibile.
2) progresso: se partiamo da un comando corretto e uno stato compatibile e c non è SKIP, devono esistere c' e s' tali che c e s si riducono a c' s'.

Lemma 9.2: preservazione del tipo per aexp

Γ ⊦ a : z && taval a s v && Γ ⊦ s

type v == z.

Lemma 9.3: progresso per aexp

Γ ⊦ a : z && Γ ⊦ s

allora ∃ v. taval a s v

Lemma 9.4: progresso per bexp

Γ ⊦ b && Γ ⊦ s

allora ∃ bv. tbval b s bv

Lemma 9.5: preservazione per com

Γ ⊦ c && (c, s) → (c', s')

allora Γ ⊦ c'

Lemma 9.6: preservazione per gli stati

$\Gamma \vdash c$ && $\Gamma \vdash s$ && $(c, s) \rightarrow (c', s')$
allora $\Gamma \vdash s'$

Lemma 9.7: progresso

$\Gamma \vdash c$ && $\Gamma \vdash s$ && $c \neq SKIP$
allora $\exists c', t'. (c, t) \rightarrow (c', t')$

teorema 9.8: correttezza del sistema di tipo

$\Gamma \vdash c$ e $\Gamma \vdash s$ e $(c, s) \overset{*}{\rightarrow} (c', t')$ e $c' \neq SKIP$
allora $\exists c'', t''. (c', t') \rightarrow (c'', t'')$

Tipi per la sicurezza

Introduciamo un sistema di tipi per IMP che garantisce la proprietà di sicurezza detta non-interferenza. Il metodo utilizzato è un caso particolare di information flow control mediante un sistema di analisi statica (qui un sistema di tipo), parte della cosiddetta language based security.

Supponiamo che ad ogni variabile sia attribuito un livello di sicurezza, che codifica in ordine crescente il grado di riservatezza dei dati ossia dei valori delle variabili. La proprietà di non interferenza garantisce che non possa esservi un flusso di informazioni da dati privati verso variabili pubblicamente accessibili.

Definiamo level = N, maggiore il livello maggiore la riservatezza
E una funzione per ottenere la riservatezza di una variabile:

sec: vname → level
sec low = 0 //dato pubblico
sec high = 1 //dato privato

Vi sono due tipi di violazioni:

1) esplicite: esempio low := high
2) implicite IF $high_1 < high_2$ THEN low := 0 ELSE low := 1

Il problema di 1 è ovvio: sto assegnando un valore con alta riservatezza ad una variabile con bassa riservatezza. Nel caso 2 non ottengo il valore di $high_1$ e $high_2$, ma comunque un'informazione sui due valori viene portata a delle variabili con meno riservatezza, e vogliamo evitare anche questo.

proprietà di non interferenza:

c soddisfa la proprietà di non interferenza se $\forall s_1,s_2$ stati i cui valori siano uguali per variabili di livello basso, se $(c, s_1) \Rightarrow t_1$ e $(c, s_2) \Rightarrow t_2$ allora t_1 e t_2 hanno gli stessi valori per le variabile di livello basso (pubbliche).

Le variabili pubbliche di s_1 e s_2 hanno gli stessi valori e vengono modificate dagli stessi comandi. Non essendo la loro modifica influenzata da variabili private (per proprietà) i loro valori devono essere gli stessi.

Calcoliamo il livello di sicurezza delle espressioni aritmetiche e booleane:

$sec_a : aexp \rightarrow level$
$sec_a\ (N\ n) = 0$
$sec_a\ (V\ x) = sec\ x$
$sec_a\ (PLUS\ a_1\ a_2) = max\ (sec_a\ a_1)\ (sec_a\ a_2)$

$sec_b : bexp \rightarrow level$
$sec_b\ (Bc\ x) = 0$
$sec_b\ (Not\ b) = sec_b\ b$
$sec_b\ (And\ b_1\ b_2) = max\ (sec_b\ b_1)\ (sec_b\ b_2)$
$sec_b\ (Less\ a_1\ a_2) = max\ (sec_a\ a_1)\ (sec_a\ a_2)$

Possiamo allora definire la proprietà di non inferenza mediante la seguente relazione: state x state x level

$$s = t\ (\leq l) \Leftrightarrow \forall x : vname.\ se\ (sec\ x) \leq l \Rightarrow s\ x = t\ x$$

Similmente:

$$s = t\ (< l) \Leftrightarrow \forall x : vname.\ se\ (sec\ x) < l \Rightarrow s\ x = t\ x$$

Se la relazione s = t (≤ l) viene preservata nell'esecuzione di un programma allora possiamo concludere che eventuali mutamenti del valore di variabili con livello di sicurezza > l non sarà osservabile considerando mutamenti dei valori di variabili di livello ≤ l.

Lemma 9.9:

$s = t\ (\leq l)\ \&\&\ sec_a\ a \leq l \Rightarrow aval\ a\ s = aval\ a\ t$

Lemma 9.10:

$s = t\ (\leq l)\ \&\&\ sec_b\ b \leq l \Rightarrow bval\ b\ s = bval\ b\ t$

Ora possiamo introdurre i **sistemi di tipo**: $L \vdash c$ (oppure $\vdash c : L$)
Questo ha due significati:

1) c non ha flussi di informazioni per variabili di livello < L
2) In c tutti i flussi di variabili di livello ≥ L sono sicuri, ovvero monotoni rispetto al livello (i flussi di informazioni sono o tra variabili dello stesso livello o da variabili pubbliche a private).

Definiamo questo sistema su IMP mediante regole di inferenza:

1) $\dfrac{}{L \vdash SKIP}$

2) $\dfrac{sec_a\ a \leq sec\ x \qquad L \leq sec\ x}{L \vdash x := a}$ //da pubblico a privato o stesso livello

3) $\dfrac{L \vdash c_1 \qquad L \vdash c_2}{L \vdash c_1 ; c_2}$

4) $\dfrac{max\ (sec_b\ b)\ (L) \vdash c_1 \qquad max\ (sec_b\ b)\ (L) \vdash c_2}{L \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2}$ //no violazioni implicite

Non solo c_1 e c_2 non devono avere flussi da privato a pubblico, ma non devo poter ottenere informazioni delle variabili in b a partire dai valori di c_1 e c_2

$$5)\ \frac{\max\ (sec_b\ b)\ (L) \vdash c}{L \vdash \text{WHILE b DO c}}$$

Vedere su appunti esempi di violazioni di sicurezza.

Vogliamo infine dimostrare che questo sistema è <u>corretto</u> per IMP, ovvero:

comandi eseguiti in stati correlati _ = _ (≤ L), quando terminando producono stati correlati _ = _ (≤ L).

Utilizziamo due lemmi per poi dimostrare un teorema che implica la correttezza. Dimostrazioni si possono trovare qua.

<u>Lemma 9.12: antimonomicità</u>

$L \vdash c \ \&\& \ L' \leq L \Rightarrow L' \vdash c$

<u>Lemma 9.13: confinamento</u>

$(c, s) \Rightarrow t \ \&\& \ L \vdash c \Rightarrow s = t\ (< L)$

<u>Teorema di non interferenza(9.14)</u>

$(c, s) \Rightarrow s' \ \&\& \ (c, t) \Rightarrow t' \ \&\& \ 0 \vdash c \ \&\& \ s = t\ (\leq L)$

allora $s' = t'\ (\leq L)$

Logica di Hoare

Siamo interessati alla **verifica funzionale**: data una specifica e un programma, dimostrare che il programma è corretto, ovvero soddisfa le specifiche per tutti gli input validi.

Come posso fare verifica funzionale?

⇒ Logiche dei programmi!

Noi ne studieremo due: logica di Hoare e logica di separazione (quest'ultima un'estensione della prima).

Un primo esempio di logica dei programmi sono le asserzioni di Floyd (citate per completezza). Logica di Hoare si ispira dalle asserzioni di Floyd.

Una relazione fondamentale per la logica di Hoare sono le triple di Hoare:

{P} c {Q}

Dove P pre-condizioni, Q post-condizioni e c comando. Qua le condizioni sono definite mediante un misto di aritemtica e logica del primo ordine.

Relazione:

$$\{P\}\ c\ \{Q\} \text{ valida } (\vDash \{P\}\ c\ \{Q\}) \Leftrightarrow$$
$$\forall I,\ \forall s,s' : \text{stati}.\ s \vDash^{I} P \wedge (c, s) \Rightarrow s' \Rightarrow s' \vDash^{I} Q$$

Ovvero $\vDash \{P\}\ c\ \{Q\}$ sse l'esecuzione di c in uno stato s che rende vera P quando termina produce uno stato s' che rende vera Q.

E a cosa serve I? Rappresenta l'**interpretazione** delle variabili logiche, a cui assegna un valore. Differenza fondamentale: s e s' sono stati, assegnano valori alle variabili di programma, I interpretazione, assegna valore alle variabili logiche. Lo rivedremo presto.

Ricordiamo alcune proprietà e definizioni di imp:

- [c] s = t se (c, s) => t, $\perp$ altrimenti
- determinismo, se (c, s) => t e (c, s) => t', allora t = t'.
- (c, s) => t sse (c, s) $\overset{*}{\rightarrow}$ (SKIP, t)

Ora possiamo dare un significato a s $\models_I$ P, ovvero lo stato s rende vera P nell'interpretazione I:
P è della forma : a = a'. dove a e a' sono termini in aexp'

Definiamo aexp' come aexp U variabili logiche.
es: X + (3 + y) con X variabile di programma e y var logica.

s $\models_I$ a = a' come viene valutato?
aval a[I(x)/x] s per tutte le variabili logiche, ovvero in a sostituisco tutte le variabili logiche con la loro interpretazione, per poi valutare a in s.

Ora posso valutare s $\models_I$ a = a':

$$s \models_I a = a' \Leftrightarrow \text{aval } a[I(x)/x]\ s = \text{aval } a'[I(x)/x]\ s$$

(casi indentici con operatori diversi come <, diverso ecc. basta sostituire = con il nuovo operatore)

Per operatore booleani?

$$s \models_I P \wedge Q \Leftrightarrow s \models_I P \wedge s \models_I Q$$

$$s \models_I \neg Q \Leftrightarrow s \not\models_I Q$$

Definizioni particolari vanno utilizzate per i quantificatori delle variabili booleane:

$$s \models_I \forall x.\ Q \Leftrightarrow \forall n \text{ Naturale. } I' = I\ [x \mid\!\!\rightarrow n].\ s \models_{I'} Q$$

$$s \models_I \exists x.\ Q \Leftrightarrow \exists n \text{ Naturale. } I' = I\ [x \mid\!\!\rightarrow n].\ s \models_{I'} Q$$

E nel caso del bottom?

$$\perp \models_I P \text{ vera per qualunque } I$$

Sfruttando quest'ultima definizione, possiamo anche scrivere in modo coinciso la relazione della tripla di Hoare come:

$$\vDash \{P\}\ c\ \{Q\} \Leftrightarrow \forall I,\ \forall s.\ s \vDash_I P \Rightarrow [\ c\ ]\ s \vDash_I Q$$

Vedere esempi di dimostrazione.

Siamo interessanti a dimostrare la <u>correttezza della logica di Hoare</u>:

$$\vdash \{P\}\ c\ \{Q\} \Rightarrow \vDash \{P\}\ c\ \{Q\}$$

ovvero: $\forall s.\ s \vDash_I P \Rightarrow [\ c\ ]\ s \vDash_I Q$

Introduciamo una serie di regole di inferenza sulla derivazione:

1) ---------- //SKIP
   $\vdash \{P\}$ SKIP $\{P\}$

2) ---------- //assegnamento
   $\vdash \{P[a/X]\}\ X := a\ \{P\}$

nb: a potrebbe contenere X!

es: {Y = 2} X := 2 {Y = X}
effettivamente (Y = X)[2/X] == (Y = 2)

<u>non</u> è vero il contrario, $\nvDash \{P\}\ X := a\ \{P[a/X]\}$
es {X = 0} X := 1 {X = 0[1/X]}. così facendo post-condizione : 1 = 0. Si possono trovare stati s che rendono vera P e falsa Q.

Caso interessante: {X = n} X := X + 1 {X = n + 1} è corretto, ma non mi basta l'assioma di assegnamento per ottenerlo

$P \rightarrow P'$ $\quad \{P'\}\ c\ \{Q\}$

3) ---------- //rafforzamento
   $\vdash \{P\}\ c\ \{Q\}$

Ora {X = n} X := X + 1 {X = n + 1} si può dimostrare sfruttando la regola di rafforzamento, perché $X = n \rightarrow X + 1 = n + 1$.

Ma come possiamo definire l'implicazione tra condizioni?

Definiamo $[P]^I = \{s \mid s \models^I P\}$. L'insieme degli stati che rendono vera P.

Ma allora:

$$P \rightarrow P' \Leftrightarrow \forall s.\ s \models^I P \Rightarrow s \models^I P'$$
$$P \rightarrow P' \Leftrightarrow [P]^I \subseteq [P']^I$$

L'insieme degli stati che rendono vera P deve essere contenuto nell'insieme degli stati che rendono vera P'.
Idea rafforzamento: Se tutti gli stati in [P'] se eseguiti in c producono stati che rendono vera Q e $P \rightarrow P'$ (quindi $[P]^I \subseteq [P']^I$), allora varrà la stessa cosa per gli stati in [P].

$$4)\ \frac{P \rightarrow P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \rightarrow Q}{\vdash \{P\}\ c\ \{Q\}}\ \text{//conseguenza}$$

Si dimostra banalmente
se $s \models^I P \Rightarrow s \models^I P'$ //per $P \rightarrow P'$
$\Rightarrow [c]\ s \models^I Q'$ //per $\{P'\}\ c\ \{Q'\}$
$\Rightarrow [c]\ s \models^I Q$ //per $Q' \rightarrow Q$

$$5)\ \frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\vdash \{P\}\ c_1 ; c_2\ \{Q\}}\ \text{//composizione}$$

$$6)\ \frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \text{IF b THEN } c_1 \text{ ELSE } c_2\ \{Q\}}\ \text{//selezione}$$

$$7)\ \frac{P \rightarrow I \qquad \{I \wedge b\}\ c\ \{I\} \qquad I \wedge \neg b \rightarrow Q}{\vdash \{P\}\ \text{WHILE b DO c } \{Q\}}\ \text{//iterazione}$$

Dove I rappresenta l'<u>invariante di ciclo</u>, condizione vera prima, durante e dopo l'esecuzione del ciclo.

Notiamo che possiamo applicare la regola di conseguenza alla regole di iterazione per ottenere una variante di quest'ultima:

$$\text{7.1)}\ \frac{\{P \wedge b\}\ c\ \{P\}}{\vdash \{P\}\ \text{WHILE b DO c}\ \{P \wedge \neg b\}}\ \text{//iterazione}$$

Banalmente sostituendo P a I e $P \wedge \neg b$ a Q sfruttiamo il fatto che $P \rightarrow P$ e $P \wedge \neg b \rightarrow P \wedge \neg b$ sono sempre vere, quindi si possono eliminare dalla regola.

Correttezza e Completezza della Logica di Hoare:

Per entrambe si può trovare qui la dimostrazione.

Teorema di correttezza di HL:

$$\vdash \{P\}\ c\ \{Q\} \Rightarrow \vDash \{P\}\ c\ \{Q\}$$

Ovvero: se si riesce a derivare che {P} c {Q} allora $\forall s.\ s \vDash^{l} P \Rightarrow [\ c\ ]\ s \vDash^{l} Q$.

La dimostrazione viene fatta per induzione su {P} c {Q}.

La correttezza di un sistema formale è un'aspetto fondamantale (non avrebbe senso lavorare su un sistema che non è corretto), ma perché ci interessa la completezza? Lo scopo della logica di Hoare è di poter creare dei programmi verificatori che dato in input un programma e delle specifiche ci dicano se il programma inserito sia corretto o no. Se non dimostriamo la completezza ci potremmo trovare nel seguente problema: fornito in input un programma corretto, il verificatore non riuscirebbe comunque a costruire una derivazione mediante HL, dando un risultato negativo alla verifica.

Teorema di completezza di HL:

$$\vDash \{P\}\ c\ \{Q\} \Rightarrow \vdash \{P\}\ c\ \{Q\}$$

Ma come si può dimostrare un qualcosa di simile?
Utilizziamo le predicate transform (inventate da Dijkstra)

Definiamo cos'é un predicate transform:

$[\ c\ ]\ (S) = \{c\ [s] \mid s \in S\}$ //capire perché c [s]

Quindi da un insieme S di stati creiamo un insieme [c] (S) dove ogni elemento s' corrisponde ad uno stato s a cui è stato applicato il comando c.

$[A]^i = \{ s \mid s \models^I A\}$ insieme di stati che rendono vera la condizione A.

Ora possiamo dare una nuova definizione di $\models^I \{P\}\ c\ \{Q\}$:

$$\models^I \{P\}\ c\ \{Q\} \Leftrightarrow [\ c\ ]\ ([P]^i) \subseteq [Q]^i$$

Ovvero se all'insieme di stati che rendono vera P applico il comando c, devo ottenere un sottoinsieme degli stati che rendono vera Q. Basta un solo stato che non renda vera Q per ottenere un errore di verifica.

Altra idea: <u>weakest liberal pre-condition</u>

$$s \models^I wlp\ (c, B) \Leftrightarrow [\ c\ ]\ s \models^I B$$

wlp(c, B) rappresenta la condizione più libera possibile t.c. se uno stato la soddisfa, allora se a suddetto stato viene eseguito un comando c, viene prodotto uno stato che soddisfa la condizione B.

$[\ wlp(c, B)\ ]^i$ = insieme di stati che soddisfano wlp(c, B) =

$$\{s \mid [\ c\ ]\ (s) \models^I B\} = [\ c\ ]^{-1}\ ([\ B\ ]^i)$$

Banalmente possiamo calcolare questa condizione prendendo l'insieme di stati che soddisfano B, per poi calcolare l'inverso di [c], così ottenendo l'insieme di stati a cui applicando c ottengo $[\ B\ ]^i$.

<u>Lemma di espressività delle asserzioni</u>:
Per qualunque comando c ed asserzione B, esiste A t.c.

$$\forall s, i.\ s \models^I A \Leftrightarrow s \models^I wlp\ (c, B)$$

Ovvero per ogni comando c e condizione B, posso trovare una condizione A equivalente a wlp(c, B).

Non si studia la dimostrazione, ma un suo passaggio fondamentale dimostra la seguente relazione:

$$\vdash \{\ wlp(c, B)\ \}\ c\ \{B\}$$

Ora sfruttando due lemmi si può arrivare alla dimostrazione della completezza di HL:

1) $\vDash \{ wlp(c, B) \}\ c\ \{B\}$
2) $\vDash \{A\}\ c\ \{B\} \Rightarrow A \rightarrow wlp(c, b)$

Le dimostrazioni di questi lemmi sono banali.

Ora posso dimostra il teorema di completezza, ovvero:

$$\vDash \{P\}\ c\ \{Q\} \Rightarrow \vdash \{P\}\ c\ \{Q\}$$

Ma visto che $\vDash \{P\}\ c\ \{Q\}$ implica $\vDash P \rightarrow wlp\ (c, Q)$, basta dimostrare che

$$\vDash \{P\}\ c\ \{Q\} \Rightarrow \vdash \{wlp(c, Q)\}\ c\ \{Q\}$$

Banalmente se questo vale per un insieme wlp(c, Q), varrà anche per qualunque sottoinsieme di wlp(c, Q).

La dimostrazione si costruisce per induzione su c, si trova qua.

Purtroppo sarebbe più corretto parlare di completezza parziale per i seguenti casi:

1) Sappiamo che se un comando non termina, esso restituisce lo stato $\perp$.

   Essendo che $\perp \vDash$ FALSE vale, abbiamo che
   {TRUE} WHILE true DO SKIP {FALSE} vale.

   In generale, {TRUE} c {FALSE} sse $\forall$ s. (c, s) =/> , e questo insieme di comandi non è ricorsivamente enumerabile. Di conseguenza abbiamo un insieme infinito di comandi che sono validi ma di cui non riusciamo a derivare $\vdash$ {TRUE} c {FALSE}, quindi la completezza è parziale.

2) {TRUE} SKIP {A} sse $\vDash$ A. Ma per il primo teorema di incompletezza di Godel sappiamo che non esiste una teoria formale per l'aritmetica t.c.

   $$\vDash A \Rightarrow \vdash A$$

Verification Condition

Ora possiamo iniziare a pensare a come creare un tool per la verifica formale di un programma in IMP.
Dobbiamo rendere la verifica automatizzata!

Intanto dobbiamo dare un tipo alle asserzioni:
assn : state → Bool
E data una assn P : P s ⇔ s ⊨ P
Riformulando la tripla di Hoare:
{P} c {Q} ⇔ ∀s, t. P s ∧ (c, s) => t ⇒Q t.

Altra definizione: s[a/X] = s[X |-> aval a s]

Infine ∀s. Q s → Q' s equivale a ⊨ Q → Q'

Ora possiamo definire una funzione wp, che rappresenta la weakest liberal pre-condition:

wp : com → assn → assn
wp c Q = λs. ∀t. (c, s) => t ⇒ Q t.

Effettivamente data wp c Q, se s soddisfa wp c Q allora l'applicazione di c in s deve restituire uno stato che soddisfa Q.

Lemma (weakest liberal pre-condition):
A meno di estensionalità, i seguenti predicati sono equivalenti:

1) wp SKIP Q = Q
2) wp (X := a) Q = λs. Q (s[a/x])
3) wp (c_1 ; c_2) Q = wp c_1 (wp c_2 Q)
4) wp (IF b THEN c_1 ELSE c_2) Q =
   λs. [IF (bval b s) THEN (wp c_1 Q) s ELSE (wp c_2 Q) s]
5) wp (WHILE b DO c) Q =
   λs. [IF (bval b s) THEN [wp (c; WHILE b DO c) Q] s ELSE Q s]

Principio di estensionalità:
Notiamo la seguente proprietà matematica: date due funzioni f,g : A → B se ∀x : A. f x == g x allora f = g. Se qualunque input otteniamo lo stesso risultato, le due funzioni si dicono equivalenti.

Possiamo allora definire i **comandi annotati**:
acom = com ∪ {I} WHILE b DO c
Dove I è un'asserzione assn e rappresenta l'invariante del ciclo.

Definiamo una funzione per eliminare le asserzioni.
strip : acom → com
strip ({I} WHILE b DO c) = WHILE b DO c
//altri comandi vengono copiati uguali

E definiamo una funzione pre, estendendo wp ai comandi annotati
pre : acom → assn → assn
pre SKIP Q = Q
pre (X := a) Q = λs. Q (s[a/x])
pre (c_1 ; c_2) Q = wp c_1 (wp c_2 Q)
pre (IF b THEN c_1 ELSE c_2) Q =
λs. [IF (bval b s) THEN (wp c_1 Q) s ELSE (wp c_2 Q) s]
pre ({I} WHILE b DO c) Q = I

E leghiamo il tutto ad Hoare la seguente proprietà di correttezza:

$$P \rightarrow \text{pre } c\ Q \Rightarrow \vdash \{P\}\ \text{strip } c\ \{Q\}$$

Quindi se P è un sottoinsieme delle weakest liberal precondition di c, Q allora vale la tripla di Hoare p, c, Q. Notiamo che c è un comando o un comando annotato.

Ma questa proprietà non è vera, perché non abbiamo il controllo degli invarianti del WHILE! Potrebbe venir scritta un'invariante non corretta o inutile.
Manca l'ipotesi che l'invariante in c sia tale.

E quand'é che I è un'invariante per WHILE b DO c data una certa condizione Q?

1) $\forall s.\ I\ s \wedge bval\ b\ s \Rightarrow (pre\ c\ I)\ s$
2) $\forall s.\ I\ s \wedge \neg bval\ b\ s \Rightarrow Q\ s$

Allora ci serve un modo di aggiungere dei predicati a sinistra dell'implicazione, in modo da poter essere certi che I sia un invariante. Essendo che il comando c può essere complesso, necessitiamo di una funzione che dato il comando e la post-condizione, generi una serie di condizioni messe in congiunzione per essere certi che $P \rightarrow pre\ c\ Q \Rightarrow \vdash \{P\}\ strip\ c\ \{Q\}$, per qualunque invariante.

Ricorda, **non** stiamo defininendo pre o post-condizione, stiamo rafforzando a sinistra l'implicazione. Necessitiamo di Q per verificare il secondo punto dell'invariante.
Definiamo una funzione <u>generatore di condizioni di verifica</u>:
Vc : acom → assn → bool
Vc SKIP Q = TRUE //non ci sono invarianti da verificare, vale true
Vc (X:=a) Q = TRUE //come sopra
Vc (c_1 ; c_2) Q = Vc c_1 (pre c_2 Q) ∧ Vc c_2 Q //postcond di c_1 è precond di c_1
Vc (IF b THEN c_1 ELSE c_2) Q = Vc c_1 Q ∧ Vc c_2 Q
//devo generare le condizioni sia di c_1 che di c_2
Vc ({I} WHILE b DO c) Q = $\forall s.[(I\ s \wedge (bval\ v\ s) \Rightarrow (pre\ c\ I)\ s) \wedge (I\ s \wedge \neg bval\ b\ s \Rightarrow Q\ s)] \wedge Vc\ c\ I$
//questo è il comando fondamentale di vc, andiamo ad aggiungere le clausole così che I sia un invariante per c e Q.

<u>Lemma (verification condition)</u>:

$$Vc\ c\ Q \Rightarrow \vdash \{pre\ c\ Q\}\ strip\ c\ \{Q\}$$

Ottimo! Data una weakest liberal pre-condition di c e Q possiamo sempre derivare $\vdash \{pre\ c\ Q\}\ strip\ c\ \{Q\}$, utilizzando l'invariante per poter verificare la correttezza dei cicli while.
La dimostrazione del Lemma si trova qua.

Separation Logic

Fino ad ora abbiamo utilizzato la logica di Hoare applicandola al semplice linguaggio IMP. Si può ovviamente contestare l'estrema semplicità del linguaggio IMP, il quale si compone soltanto di variabili e comandi.

Per iniziare possiamo pensare di introdurre il concetto di **puntatori**, grazie ai quali possiamo anche iniziare a introdurre qualche nozione di <u>struttura dati</u> in IMP.

<u>IMP esteso con puntatori</u>:
nuovo com := vecchio com $\cup$
| x := cons(a_1, a_2, …, a_n) //allocatore di un array di n interi
| x := [a] //lookup, valore in indirizzo a
| [a] := a' //mutation, assegno val a' in locazione a
| dispose(a) //deallocazione di indirizzo nel heap

<u>Cambia la semantica</u>! Ora non basta più il semplice concetto di state. Dobbiamo dividere la memoria delle variabili dalla memoria dei puntatori. Due nuove funzioni, store e heap. Il vecchio state viene rappresentato dall'unione di queste funzioni.

store : var_name → val
heap : loc → val
Dove val sono valori interi e Loc un numero naturale diverso da 0.

Sia h $\in$ heap, allora possiamo esplicitare i suoi componenti come
h = {$L_1 \rightarrow v_1$, $L_2 \rightarrow v_2$, … , $L_n \rightarrow v_n$} per un qualche n ≥ 0 e
dom(h) = {L_1, L_2, …, L_n}

<u>Notazione fondamentale</u>:
1) $h_1 \perp h_2$ = dom(h_1) ∩ dom(h_2) = ∅.
2) $h_1 \uplus h_2 = h_1 \cup h_2$ se $h_1 \perp h_2$, indefinito altrimenti.

Con cui testiamo che due heap siano definiti per posizioni diverse e in tale caso possiamo unirli in un unico heap.

Possiamo ora estendere la semantica small-step per IMP esteso, notiamo che ora lavoriamo con triple (comando, store, heap):

1) ---
(X := a, s, h) → (SKIP, s[X |-> aval a s], h)

aval a s = n n ∈ dom(h) h(n) = m
2) ---
(X := [a], s, h) → (SKIP, s[X |-> m], h)

aval a s = n n ∈ dom(h)
3) ---
([a] := a', s, h) → (SKIP, s, h[n |-> aval a' s])

Introduciamo delle regole per allocare e deallocare:

m,m+1, m+n-1 ∉ dom(h) aval a_i s = v_i i = 1…n
4) ---
(X := cons(a_1, a_2, …, a_n), s, h) →
(SKIP, s[X |-> m], h ⊎ {m → v_1, …, m+n-1 → v_n})

nella variabile X inserisco il puntatore al primo elemento del array

//un controllo che h sia della seguente forma
aval a s = n h = h' ⊎ {n → m}
5) ---
(dispose(a), s, h) → (SKIP, s, h')

<u>Introduciamo una nozione di errore</u>, i quali sono esplicitati all'utente:

aval a s = n n ∉ dom(h)
1) ---
(X := [a], s, h) → error

aval a s = n n ∉ dom(h)

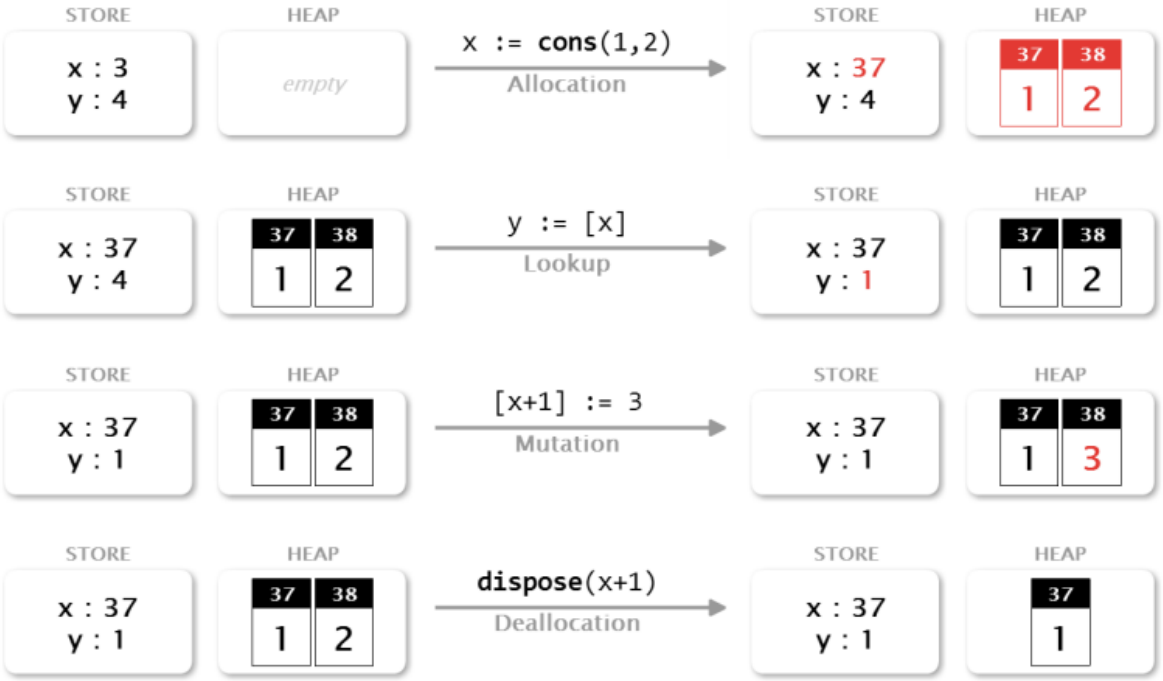
2) ----------
([a] := a', s, h) → error

aval a s = n n ∉ dom(h)

3) ----------
(dispose(a), s, h) → error

Sono sottinetese le propagazioni degli errori, esempio se c_1 porta ad un errore anche c_1 ; c_2 porta ad un errore.

Ora si può vedere un esempio di esecuzione con IMP esteso:



Ora che abbiamo esteso IMP per dare maggiore possibilità di programmazione, la logica di Hoare non è più sufficente per poter dimostrare la correttezza di un programma. Da qui nasce la Separation Logic.

Un caso come la regole di assegnazione:
{P[a/X]} X := a {P}

Continua a valere per le variabili {1 = 1} X := 1 {X = 1}
Ma ovviamente non vale più per i puntatori.

Prima di definire SL dobbiamo estendere le asserzioni:
$s, h \vDash emp$ se $h = \varnothing$
$s, h \vDash a \text{ |-> } a'$ se $h = \{aval\ a\ s \rightarrow aval\ a'\ s\}$
$s, h \vDash P * Q$ se $\exists\, h_1, h_2$ t.c. $[h = h_1 \uplus h_2 \wedge s, h_1 \vDash P \wedge s, h_2 \vDash Q]$
dove P e Q sono delle altre asserzioni.

La prima indica l'heap vuoto, la seconda che heap è composto da un unico assegnamento, da indirizzo a a indirizzo a', la terza che heap può essere diviso in due parti le quali soddisfano due diverse asserzioni P e Q.

Vediamo un esempio:
Nello store = {X → 20 e Y → 37} e heap = {20 → 0 e 37 → 1}
s, X |-> 0 * Y |-> 1 vale?

Diviamo lo heap in due parti: $\{20 \rightarrow 0\} \uplus \{37 \rightarrow 1\}$ e vediamo che
$s, \{20 \rightarrow 0\} \vDash \{X \text{ |-> } 0\}$ e $s, \{37 \rightarrow 1\} \vDash \{Y \text{ |-> } 1\}$ sono banalmente vere.
Quindi $s, h \vDash X \text{ |-> } 0 * Y \text{ |-> } 1$.

E se ipoteticamente s X = s Y? Allora entrambe le variabili farebbero riferimento ad un unico punto del heap. Ma non è possibile dare due valori ad una stessa cella! Infatti * richiede di proseguire il lavoro su due heap separati per questo motivo.

<u>Alcune abbreviazioni utili</u>:

1) $a \hookrightarrow a' \equiv a \text{ |-> } a' * true$
   ovvero $h = \{aval\ a\ s \rightarrow aval\ a'\ s\} \uplus h'$
   ed essendo che $s, h' \vDash true$ vale sempre, indico che in qualche parte del heap nella locazione a vi è il valore a'.
2) $a \text{ |-> } a_1, \ldots, a_n \equiv a \text{ |-> } a_1 * a+1 \text{|-> } a_2 * \ldots * a+n-1 \text{ |-> } a_n$
3) $a \hookrightarrow a_1, \ldots, a_n \equiv a \text{ |-> } a_1 * a+1 \text{|-> } a_2 * \ldots * a+n-1 \text{ |-> } a_n * true$
4) $a \text{ |-> } \_ \equiv \exists x.\ a \text{ |-> } x$

Vi sono alcune differenza tra P ∧ Q e P * Q, visto che la seconda impone che entrambe le asserzioni siano vere in due heap separati e distinti.
Esempio: in generale P ⇒ P ∧ P sempre vera, mentre P ⇒ P * P se P è vera saremmo in un caso Vero ⇒ Falso (notiamo che P * P non può mai essere vera perché non posso dividere heap in due modi separati che soddisfano la stessa condizione).
Possiamo ora dare una definizione di correttezza:
una tripla (c, s, h) si dice **safe** se (c, s, h) *--/> error, ovvero se non si riduce mai ad una situazione di errore.

Teorema di correttezza parziale di SL:

$\vDash_{sl} \{P\}\ c\ \{Q\} \Leftrightarrow$

$\forall I,s,h.\ s, h \vDash^{I} P \Rightarrow (c, s, h)$ safe $\wedge\ [\ c\ ]\ (s, h) \vDash^{I} Q$

dove $[\ c\ ]\ (s, h) = t, h'$ se $(c, s, h) \xrightarrow{*} (SKIP, t, h')$

$\perp$ altrimenti

Possiamo ora introdurre le regole per lo Heap:

1) ---------- //fetch
   $\{X = Y \wedge a \mid\!\!\to z\}\ X := [a]\ \{X = z \wedge a[Y/X] \mid\!\!\to z\}$
   //se a conteneva X dobbiamo togliere X in a per non perdere valore z

2) ---------- //mutation
   $\{a \mid\!\!\to \_\}\ [a] := a'\ \{a \mid\!\!\to a'\}$

3) ---------- //cons
   $\{X = Y\}\ X := cons(a_1, \ldots, a_n)\ \{X \mid\!\!\to a_1[Y/X] \ldots a_n[Y/X]\}$

4) ---------- //mutation
   $\{emp\}\ X := cons(a_1, \ldots, a_n)\ \{X \mid\!\!\to a_1 \ldots a_n\}$

5) ---------- //dispose
{a |-> _} dispose(a) {emp}

Introduciamo anche un teorema molto importate
Teorema di correttezza della Frame Rule:

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}}$$

La regola è vera quando fv R ∩ mod c = ∅.

Ovvero le variabili libere in R e le variabili modificate in c non hanno elementi in comune.

Ossia : $\vDash_{sl} \{P\}\ c\ \{Q\} \Rightarrow \vDash_{sl} \{P * R\}\ c\ \{Q * R\}$
Questo teorema ci permette di dividere il codice in pezzi per dimostrarli separatamente.

Vediamo un esempio di derivazione:
$\vDash_{sl}$ {X |-> _ * y |-> 3} [X] := 4 {X |-> 4 * y |-> 3} vale?

$$\frac{\dfrac{}{\{X \mapsto \_\}\ [X] := 4\ \{X \mapsto 4\}}\ \text{//mutation}}{\{X \mapsto \_ * Y \mapsto 3\}\ [X] := 4\ \{X \mapsto 4 * Y \mapsto 3\}}\ \text{//fr}$$

Il codice scritto è corretto.
Inoltre se s X = s Y allora per qualunque h $\nvDash$ X |-> _ * Y |-> 3 perché non posso dividere l'heap in due parti distinte (essendo che X e Y puntano alla stessa locazione di memoria), per def di *.

Una dimostrazione del teorema di correttezza si può trovare qua.

Vengono utilizzati i seguenti lemmi:
Lemma di monotonicità:

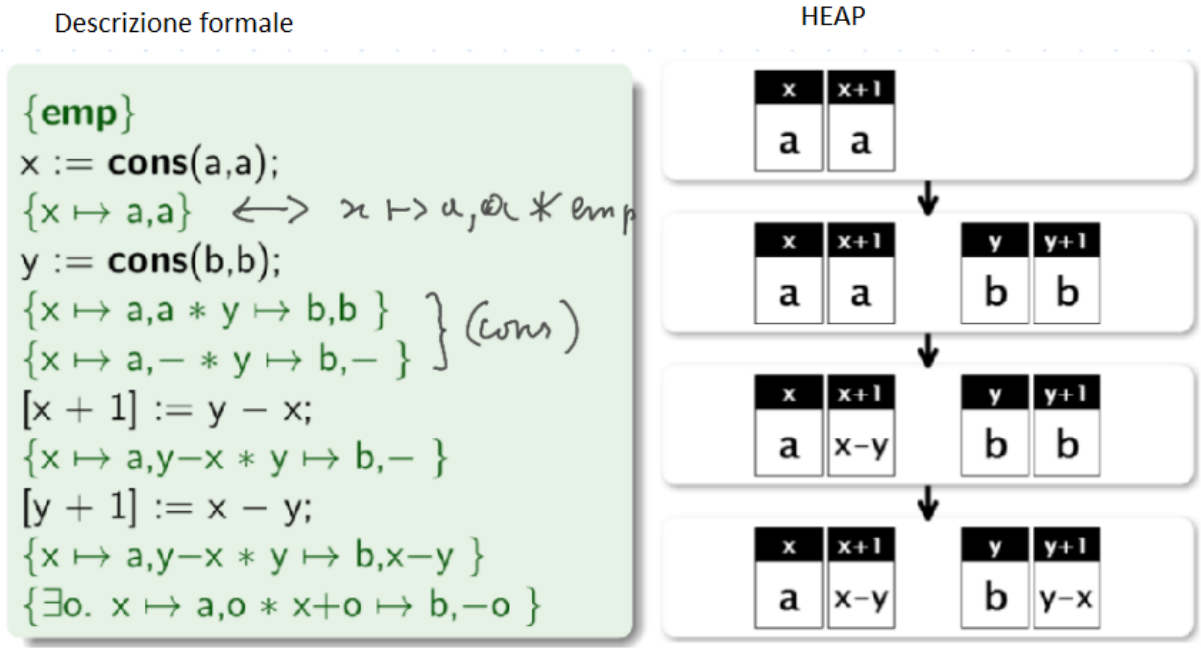
1) (c, s, h) safe $\Rightarrow$ $\forall$ h' tale che h $\perp$ h' . (c, s, h $\uplus$ h') safe.
2) $(c, s, h_0) \xrightarrow{*} (SKIP, s', h_0') \Rightarrow$
   $\forall h_1$ tale che $h_0 \perp h_1$. $(c, s, h_0 \uplus h_1) \xrightarrow{*} (SKIP, s', h_0 \uplus h_1)$
3) (c, s, h_0) riduce all'infinito $\Rightarrow$ $\forall h_1$ tale che $h_0 \perp h_1$. $(c, s, h_0 \uplus h_1)$ riduce all'infinito

Lemma frame property:

(c, s, h_0) safe $\wedge$ $(c, s, h_0 \uplus h_1) \xrightarrow{*} (SKIP, s', h')$

$\Rightarrow \exists\, h_0' \perp h_1 \,.\, (c, s, h_0) \xrightarrow{*} (SKIP, s', h_0') \wedge h' = h_0' \uplus h_1$

Ora immaginiamo di voler creare una derivazione di una tripla del tipo: {emp} *una serie di comandi* {*una serie di condizioni*}. Dobbiamo costruire una derivazione applicando le varie regole in base ai comandi. Si nota che invece che costruire una derivazione "ad albero", possiamo creare una serie di asserzioni inserite tra i comandi per dimostrare la post-condizione, molto più compatta.
Inoltre notiamo una corrispondenza tra l'heap e la sua descrizione formale.



Una trasformazione di stato (del heap) diventa una trasformazione di formule! Quindi formule elementari e semplici non sono altro che una descrizione minimamente astratta dello stato heap a livello logico.
derivazione ⇔ esecuzione!
Se riusciamo a formalizzare questa cosa possiamo costruire passi della derivazione mediante riscrittura delle formule in base ai comandi, andando a automatizzare la costruzione della derivazione. Questa si chiama **esecuzione simbolica**.

Introduciamo ora gli heap simbolici: una formula della SL

$H ::= \exists x . (P_1 \wedge P_2 \wedge \ldots \wedge P_n) \wedge (S_1 * S_2 * \ldots * S_n)$
dove $x = \cup fv(P_i) \cup \cup fv(S_i)$

Una chiusura esistenziale di una congiunzione di due elementi (che possono anche essere uno dei due o entrambi non presenti):

1) pura: P_i, sono una serie di condizioni del tipo var diversa/uguale a var o var diversa/uguale numero. Valore di default: True.
2) spaziale: S_i, descrizione del heap mediante l'operatore *. O rappresenta indirizzo |-> valore e (mediante lsne) una lista di valori da x a y.

Spaziali: descrivono lo spazio del heap. 2 descrive l'heap, 1 descrive lo store mediante equazioni e disequazioni.

Ora possiamo discutere l'esecuzione simbolica:
Per determinati comandi, detti atomici, cioè non composti da altri sottocomandi:
A ::= X := E | X := [E] | [E] := E' | X := cons(_) | dispose(E)

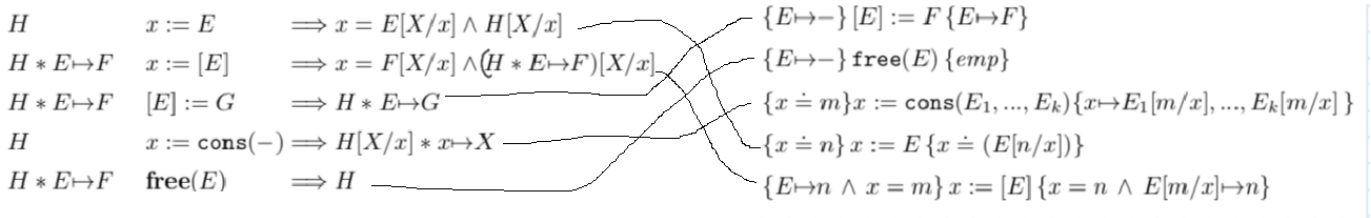
Cerchiamo di tradurre le trasformazioni di stati simbolici come fossero le triple in SL:

$$\vdash_{SL} \{H\}\ A\ \{H'\} \quad \Leftrightarrow \quad H, A \Rightarrow H'$$

Descrizioni di heap e stato diventano le pre e posto condizioni.

Definiamo anche la combinazione di due H:
se $H = (P_1 \ldots P_n) \wedge (S_1 \ldots S_n)$ e $H' = (P'_1 \ldots P'_n) \wedge (S'_1 \ldots S'_n)$
allora $H * H' = (P_1 \ldots P_n \wedge P'_1 \ldots P'_n) \wedge (S_1 \ldots S_n * S'_1 \ldots S'_n)$

Alcuni casi di relazione rappresentazione simbolica e tripla SL:



Nota: nelle triple di SL non si specifica l'intero heap, mediante frame rule posso dimostrare un comando in una specifica parte del heap. Senza frame rule tutte le triple farebbero riferimento a specifici heap, dovendoci preoccupare della restante parte.

Una tripla: {H} A_1 ; … ; A_n {H'} si dimostra mediante una sequenza di riduzioni, grazie alle quali eseguendo i comandi A_i H viene modificato man mano, e se arriviamo alla fine in modo che H_n = H' abbiamo trovato una derivazione.

Notiamo che essendo una composizione ci basiamo sulla banale regola in SL:

$$\frac{\{H\}\ A\ \{H''\} \quad \{H''\}\ A'\ \{H'\}}{\{H\}\ A\ ;\ A'\ \{H'\}}$$

E in esecuzione simbolica:

$$\frac{H, A_1 \Rightarrow H'}{H, A_1\ ;\ A_2 \Rightarrow H', A_2}$$

Differenza importante: Hoare tende a cercare H partendo da H', nell'esecuzione simbolica partiamo da H e eseguendo A arriviamo ad H'.

VeriFast

Aspetti fondamentali:

1) Modularizzazione, mediante contratti per tutte le funzioni.
2) Path condition.
3) Predicati.

Verifast sfrutta il concetto di stato simbolico:
Una tripla (S, H, π), dove:

1) S store simbolico, insieme finito di formule della forma $x_i = t_i$
   (nb: x_i != x_j quando i != j).
2) H heap simbolico, insieme di L |-> v e predicati P(v).
3) π path condition, insieme di formule in HOL che rappresentano restrizioni sui simboli logici in S. Servono a rappresentare condizioni che si generano mediante operazioni e fork.

Essendo un'esecuzione simbolica gli input su cui lavoriamo non sono definiti in maniera concreta.Una variabile x intera avrà un valore non specificato v. Se dovessimo trovare un pezzo di codice della forma *"IF (x > 5) A() ELSE B()"* si creeranno due esecuzioni in parallelo, una nel metodo A e un'altra nel metodo B. La prima avrà nella path condition una formula del tipo x > 5, la seconda una formula del tipo x ≤ 5.

Rimane invariata la definizione di Interpretazione:
I : varname → value
Dato un termine t, con $[\ t\ ]_I$ denotiamo il suo valore nell'attuale interpretazione, sicuramente un value.
L'interpretazione di un formula Φ, denotata con $[\ \Phi\ ]_I$, restituisce un booleano.

Diciamo che uno stato s e uno heap h rendono vero uno stato simbolico in una interpretazione I:
$s, h \vDash^I (\hat{S}, \hat{H}, \pi) \Leftrightarrow$

1) $\forall \Phi \in \pi.\ [\ \Phi\ ]_I = true$
2) $\forall (x = t) \in \hat{S}.\ I(x) = [\ t\ ]_I = s\ x$

3) $\hat{H}$ ben formato, ovvero :
    a) $\forall$ coppia (L |-> v, L' |-> v'). $[L]_I \; != \; [L']_I$
    b) $\forall$ L |-> v . h ($[L]_I$) = $[v]_I$

Banalmente: per 1 l'intepretazione deve rendere vere tutte le formule del path cond (ricorda: interpretazione è un'assegnazione di valori alle variabili). Per 2 l'intepretazione deve assegnare dei valori che siano coerenti con lo store simbolico, e s deve assegnare ad x lo stesso valore di I. Infine per 3 L'intepretazione deve assegnare dei valori corretti e coerenti allo heap simbolico.

L'esecuzione simbolica si basa sulle seguenti operazioni: produrre e consumare asserzioni nell'heap simbolico.

In un'esecuzione simbolica, per consumare un'asserzione a si intende cercare nel heap simbolico un chunk il quale soddisfa a, per poi eliminarlo del heap, verificando se il nuovo stato simbolico soddisfa le post-condizioni.

Similmente, la produzione consiste nell'estendere l'heap simbolico con un chunk che soddisfi l'asserzione a, per poi controllare che il risultato soddisfi le post-condizioni.

Definendole come funzioni:
produce(h, s, π, a, Q) ⇔
    $\exists$ h' t.c. h $\perp$ h' $\wedge$ π,s,h' $\vdash_{smtp}$ a $\wedge$ Q(s, h$\uplus$h', π') con π $\subseteq$ π'
dove SMTP rappresenta un verificatore di soddisfacibilità di formule.

consume(h, s, π, a, Q) ⇔
    $\exists$ h',h'' t.c. h=h'$\uplus$h'' $\wedge$ π,s,h' $\vdash_{smtp}$ a $\wedge$ Q(s, h'', π') con π $\subseteq$ π'

Ma allora anche la verifica di un programma può essere definito come una funzione verify
verify(h, s, π, c, Q) ⇔
  l'esec. simbolica di c in (h, s, π) termina in (h', s', π') t.c. Q(h', s', π').

Come funziona tutto questo in un contesto reale?

Immaginiamo di avere una funzione main che richiama un metodo A() durante un'esecuzione simbolica. Sappiamo che il metodo A richiede a_1 e assicura a_2.

Il metodo A viene verificato nella seguente maniera:

1) si assumono/producono tutte le pre-condizioni. In questo caso a_1
2) Nell'esecuzione dei comandi si modifica lo heap.
3) si derivano/consumano le post-condizioni. In questo caso a_2.

Nel contesto del main l'esecuzione non entra nel metodo A, ma semplicemente:

1) si derivano/consumano tutte le pre-condizioni a_1.
2) si assumono/producono tutte le post-condizioni a_2.

Alla fine dell'esecuzione del main ci aspettiamo che lo heap sia vuoto. Se l'esecuzione simbolica si conclude allora abbiamo una dimostrazione che il codice scritto è corretto.

Altri concetti fondamentali:

- predicati: per evitare di dover riscrivere nei contratti gli stessi chunk, possiamo scrivere un predicato, il quale abbrevia i suddetti chunk, preoccupando di aprirlo e chiuderlo durante l'esecuzione simbolica.
- tipi induttivi: si scrivono come in Haskell, composti da costanti basi e costanti costruttori. Esempio: liste di interi. Possiamo creare predicati utilizzando i tipi induttivi per una maggiore espressitività.
- Funzioni fix-point: funzioni primitive ricorsive su tipi induttivi. Deve essere composta da uno switch sul parametro. Da qui si possono introdurre nozioni di calcolo simbolico.
- lemma: funzioni fix-point con tipo di ritorno void e con un contratto. Le pre-condizioni diventano le ipotesi del lemma, le post-condizioni la sua tesi. Ovviamente dato un input, questo non può essere modificato. Trasformano degli heap chunk in altri heap chunk, in base alle tesi e ipotesi.
- invarianti: per dimostrare un ciclo corretto si utilizza un invariante. Prima si dimostra che l'invariante vale prima del ciclo. In seguito si cancella tutto l'heap e si suppone solo l'invariante, e si dimostra

che vale anche dopo una esecuzione. Infine si dimostra che l’invariante vale anche dopo il ciclo.