Modelli della Concorrenza

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Le immagini presenti in questi appunti sono tratte dalle slides del corso e tutti i diritti delle stesse sono da destinarsi ai docenti del corso stesso.

Capitolo 2

Introduzione alla Concorrenza

La **concorrenza** è presente in diversi aspetti della quotidianità. Un primo esempio di **sistema concorrente** non legato all'informatica è la *cellula vivente*: può essere vista come un dispositivo che trasforma e manipola dati per ottenere un risultato. I vari processi all'interno di una cellula avvengono in modo concorrente, il che la rende un *sistema asincrono*. Un secondo esempio può essere quello dell'*orchestra musicale*: i vari componenti suonano spesso simultaneamente rappresentando un *sistema sincrono* (ovvero un sistema che funziona avendo una sorta di "cronometro" condiviso dai vari attori). Un esempio informatico invece è un *processore multicore* (anche se in realtà anche se fosse *monocore* sarebbe comunque un sistema concorrente per ovvie ragioni). Anche una *rete di calcolatori* è un modello concorrente, nonché i *modelli sociali umani*.

Caratteristiche comuni

I modelli concorrenti hanno alcuni aspetti comuni, tra cui:

- competizione per l'accesso alle risorse condivise
- cooperazione per un fine comune (che può portare a competizione)
- coordinamento di attività diverse
- sincronia e asincronia

Studio

Durante lo studio e la progettazione di sistemi concorrenti si hanno diversi problemi peculiari che rendono il tutto molto complesso. Un sistema concorrente mal progettato può avere effetti catastrofici.

Per poter sviluppare modelli concorrenti si necessita innanzitutto di:

- **linguaggi**, per specificare e rappresentare sistemi concorrenti.
  - **linguaggi di programmazione** (con l'uso di *thread*, *mutex*, scambio di messaggi, etc... con i vari problemi di *race condition*, uso di variabili condivise etc...).
  - linguaggi rappresentativi, come ad esempio una *partitura musicale* (nella quale si visualizza bene la natura *sincrona*).
  - **task graph (*grafo delle attività*)**, nel quale i nodi sono le attività (o eventi) mentre gli archi rappresentano una *relazione d'ordine parziale*, come per esempio una *relazione di precedenza* sui nodi.
  - **algebre di processi**, simile ad un sistema di equazioni, con simboli che rappresentano eventi del sistema concorrente e operatori atti a comporre fra loro i vari sottoprocessi del sistema concorrente. Ogni "equazione" descrive un processo che costituisce un elemento di un sistema concorrente.
  - **modelli**, per modellare sistemi concorrenti in astratto. Un esempio è dato dalle **reti di Petri**, che modellano un sistema concorrente partendo dalle nozioni di *stato locale* di uno dei componenti del sistema e di *evento locale* che ha un effetto su alcune componenti (e non tutte). Si ha quindi rappresentato un *sistema dinamico* che si evolve nel tempo e la cui evoluzione è rappresentata tramite *relazioni di flusso*).
- **logica**, per analizzare e specificare sistemi concorrenti.
- **model-checking**, per validare formule relative a proprietà di sistemi concorrenti.

Capitolo 3

Logica

Si ringrazia Marco Natali [1] *per questo ripasso.*

La logica è lo studio del ragionamento e dell'argomentazione e, in particolare, dei procedimenti inferenziali, rivolti a chiarire quali procedimenti di pensiero siano validi e quali no. Vi sono molteplici tipologie di logiche, come ad esempio la logica classica e le logiche costruttive, tutte accomunate dall'essere composte da 3 elementi:

- **Linguaggio**: insieme di simboli utilizzati nella Logica per definire le cose.
- **Sintassi**: insieme di regole che determina quali elementi appartengono o meno al linguaggio.
- **Semantica**: permette di dare un significato alle formule del linguaggio e determinare se rappresentano o meno la verità.

3.1 Logica proposizionale

Ci occupiamo della *logica classica* che si compone in *logica proposizionale* e *logica predicativa*. La logica proposizionale è quindi un tipo di logica classica che presenta come caratteristica principale quella di essere un linguaggio limitato, ovvero caratterizzato dal poter esprimere soltanto proposizioni senza possibilità di estensione ad una classe di persone.

[1] Link al repository: https://github.com/bigboss98/Appunti-1/tree/master/Primo Anno/Fondamenti

3.1.1 Sintassi

Il linguaggio di una logica proposizionale è composto dai seguenti elementi:

- Variabili Proposizionali atomiche (o elementari): $P, Q, R, p_i, \ldots$.
- Connettivi Proposizionali: $\wedge, \vee, \neg, \implies, \iff$
- Simboli Ausiliari: "(" e ")" (detti delimitatori)
- Costanti: T (*True, Vero,* $\top$) e F (*False, Falso,* $\bot$)

La sintassi di un linguaggio è composta da una serie di formule ben formate (FBF) definite induttivamente nel seguente modo:

1. Le costanti e le variabili proposizionali: $\top, \bot, p_i \in FBF$.
2. Se A e $B \in FBF$ allora $(A \wedge B)$,$(A \vee B)$,$(\neg A)$,$(A \implies B)$, $(A \iff B)$ sono delle formule ben formate.
3. nient'altro è una formula

In una formula ben formata le parentesi sono bilanciate.

Esempio 1. *Vediamo degli esempi:*

- $(P \wedge Q) \in FBF$ *è una formula ben formata*
- $(PQ \wedge R) \notin FBF$ *in quanto non si rispetta la sintassi del linguaggio definita.*

Definizione 1. *Sia* $A \in FBF$*, l'insieme delle sottoformule di* A *è definito come segue:*

1. *Se* A *è una costante o variabile proposizionale allora* A *stessa è la sua sottoformula.*
2. *Se* A *è una formula del tipo* $(\neg A')$ *allora le sottoformule di* A *sono* A *stessa e le sottoformule di* A'*;* $\neg$ *è detto connettivo principale e* A' *sottoformula immediata di* A*.*
3. *Se* A *è una formula del tipo* $B \circ C$*, allora le sottoformule di* A *sono* A *stessa e le sottoformule di* B *e* C*;* $\circ$ *è il connettivo principale e* B *e* C *sono le due sottoformule immediate di* A*.*

È possibile ridurre ed eliminare delle parentesi attraverso l'introduzione della precedenza tra gli operatori, definita come segue:

$$\neg, \wedge, \vee, \implies, \iff$$

In assenza di parentesi una formula va parentizzata privilegiando le sottoformule i cui connettivi principali hanno la precedenza più alta.
In caso di parità di precedenza vi è la convenzione di associare da destra a sinistra. Segue un esempio:

$$\neg A \wedge (\neg B \implies C) \vee D \text{ diventa } ((\neg A) \wedge ((\neg B) \implies C) \vee D)$$

Albero Sintattico

Definizione 2. *Un albero sintattico T è un albero binario coi nodi etichettati da simboli di L, che rappresenta la scomposizione di una formula ben formata X definita come segue:*

1. Se X è una formula atomica, l'albero binario che la rappresenta è composto soltanto dal nodo etichettato con X
2. Se $X = A \circ B$, X è rappresentata da un albero binario che ha la radice etichettata con $\circ$, i cui figli sinistri e destri sono la rappresentazione di A e B
3. Se $X = \neg A$, X è rappresentato dall'albero binario con radice etichettata con $\neg$, il cui figlio è la rappresentazione di A

Poiché una formula è definita mediante un albero sintattico, le proprietà di una formula possono essere dimostrate mediante induzione strutturale sulla formula, ossia dimostrare che la proprietà di una formula soddisfi i seguenti 3 casi:

- è verificata la proprietà per tutte le formule atomo A
- supposta verifica la proprietà per A, si verifica che la proprietà è verificata per $\neg A$
- supposta la proprietà verificata per A_1 e A_2, si verifica che la proprietà è verifica per $A_1 \circ A_2$, per ogni connettivo $\circ$.

3.1.2 Semantica

La semantica di una logica consente di dare un significato e un'interpretazione alle formule del Linguaggio.

Definizione 3. *Sia data una formula proposizionale P e sia $P_1, \ldots, P_n$, l'insieme degli atomi che compaiono nella formula A. Si definisce come* interpretazione *una funzione $v : \{P_1, \ldots, P_n\} \mapsto \{T, F\}$ che attribuisce un valore di verità a ciascun atomo della formula A.*
*$v : P \to \{0, 1\}$ è un'**assegnazione booleana***

I connettivi della Logica Proposizionale hanno i seguenti valori di verità:

A	B	$A \wedge B$	$A \vee B$	$\neg A$	$A \Longrightarrow B$	$A \Longleftrightarrow B$
F	F	F	F	T	T	T
F	T	F	T	T	T	F
T	F	F	T	F	F	F
T	T	T	T	F	T	T

Essendo ogni formula A definita mediante un unico albero sintattico, l'interpretazione v è ben definita e ciò comporta che data una formula A e un'interpretazione v, eseguendo la definizione induttiva dei valori di verità, si ottiene un unico $v(A)$.

Definizione 4. *Una formula nella logica proposizionale può essere di diversi tipi:*

- ***Valida o Tautologica:** la formula è soddisfatta da qualsiasi valutazione della Formula*
- ***Soddisfacibile NON Tautologica:** la formula è soddisfatta da qualche valutazione della formula ma non da tutte.*
- ***Falsificabile:** la formula non è soddisfatta da qualche valutazione della formula.*
- ***Contraddizione:** la formula non viene mai soddisfatta*

Teorema 1. *Si ha che:*

- *A è una formula valida se e solo se $\neg A$ è insoddisfacibile.*
- *A è soddisfacibile se e solo se $\neg A$ è falsificabile*

Modelli e decidibilità

Si definisce *modello*, indicato con $M \models A$, tutte le valutazioni booleane che rendono vera la formula A. Si definisce *contromodello*, indicato con $\not\models$, tutte le valutazioni booleane che rendono falsa la formula A.

La logica proposizionale è decidibile (posso sempre verificare il significato di una formula). Esiste infatti una procedura effettiva che stabilisce la validità o no di una formula, o se questa ad esempio è una tautologia. In particolare il verificare se una proposizione è tautologica o meno è l'operazione di decidibilità principale che si svolge nel calcolo proposizionale.

Definizione 5. *Se $M \models A$ per tutti gli M, allora A è una tautologia e si indica $\models A$.*

Definizione 6. *Se $M \models A$ per qualche M, allora A è soddisfacibile.*

Definizione 7. *Se $M \models A$ non è soddisfatta da nessun M, allora A è insoddisfacibile.*

3.1.3 Equivalenze Logiche

Definizione 8. *Date due formule A e B, si dice che A è* logicamente equivalente *a B, indicato con $A \equiv B$, se e solo se per ogni interpretazione v risulta $v(A) = v(B)$.*

Nella logica proposizionale sono definite le seguenti equivalenze logiche, indicate con $\equiv$:

1. **Idempotenza:**

$$A \vee A \equiv A$$
$$A \wedge A \equiv A$$

2. **Associatività:**

$$A \vee (B \vee C) \equiv (A \vee B) \vee C$$
$$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$$

3. **Commutatività:**

$$A \vee B \equiv B \vee A$$
$$A \wedge B \equiv B \wedge A$$

4. **Distributività:**

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$
$$A \wedge (B \vee C) \equiv (A \wedge B \vee (A \wedge C)$$

5. **Assorbimento:**

$$A \vee (A \wedge B) \equiv AA \wedge (A \vee B) \equiv A$$

6. **Doppia negazione:**

$$\neg\neg A \equiv A$$

7. **Leggi di De Morgan:**

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

8. **Terzo escluso:**

$$A \vee \neg A \equiv T$$

9. **Contrapposizione:**

$$A \Longrightarrow B \equiv \neg B \Longrightarrow \neg A$$

10. **Contraddizione**

$$A \wedge \neg A \equiv F$$

Completezza di insiemi di Connettivi

Un insieme di connettivi logici è completo se mediante i suoi connettivi si può esprimere un qualunque altro connettivo. Nella logica proposizionale valgono anche le seguenti equivalenze, utili per ridurre il linguaggio:

$$(A \Longrightarrow B) \equiv (\neg A \vee B)$$
$$(A \vee B) \equiv \neg(\neg A \wedge \neg B)$$
$$(A \wedge B) \equiv \neg(\neg A \vee \neg B)$$
$$(A \Longleftrightarrow B) \equiv (A \Longrightarrow B) \wedge (B \Longrightarrow A)$$

L'insieme dei connettivi $\{\neg, \vee, \wedge\}$, $\{\neg, \wedge\}$ e $\{\neg, \vee\}$ sono completi.

Capitolo 4

Correttezza di programmi sequenziali

Introduciamo l'argomento con un esempio.

Esempio 2. *Definiamo una funzione in C che riceve un vettore, un intero (la lunghezza del vettore) e e restituisce un ulteriore numero intero. Chiedendoci*

Listing 1 Esempio di funzione in C

```
int f(int n, const int v[]) {
  int x = v[0];
  int h = 1;
  while (h < n) {
    if (x < v[h])
    x = v[h];
    h = h + 1;
  }
  return x;
}
```

cosa fa la funzione scopriamo che si occupa di cercare il massimo in un vettore.
La strategia della funzione è quella di spostarsi lungo il vettore e conservare in x il valore massimo fino ad ora trovato. Arrivati alla fine del vettore so che in x avrò il valore massimo.
Più formalmente suppongo che n sia $n > 0$, per dire che ho almeno un elemento nel vettore. Suppongo inoltre che $v[i] \in \mathbb{Z}$, $\forall i \in \{0, \ldots, n-1\}$. Abbiamo fissato le ***condizioni iniziali****.*
All'inizio x è il massimo del sotto-vettore con solo il primo elemento ($v[0..0]$)

e dopo l'assegnamento di h in $v[o..h-1]$, che, con $h=1$ mi conferma che x è il massimo in $v[0..0]$.
Al termine di una certa iterazione x contiene il massimo tra i valori compresi tra $v[0]$ e $v[h-1]$ (detto altrimenti il massimo in $v[0..h-1]$). Inoltre al fine di una certa iterazione mi aspetto che $h \leq n$.
Possiamo quindi dire che quando esco dal ciclo x è il massimo in $v[0..h-1]$ ma in questo momento $h=n$ e quindi x è il massimo del vettore.
Consideriamo ora la parte iterativa. All'inizio di ogni iterazione suppongo che x è il massimo in $v[0..h-1]$. Dopo l'istruzione di scelta x è il massimo in $v[0..h]$, comunque sia andata la scelta. Alla fine dell'iterazione, dopo l'incremento di h, avrò ancora che x è il massimo in $v[0..h-1]$. Ragiono quindi per induzione. Se all'inizio dell'iterazione e alla fine ho la stessa asserzione, ed è vera prima di iniziare l'iterazione, posso dire che ho una ***proprietà invariante*** *e vale anche al termine dell'ultima iterazione e quindi vale anche alla fine dell'esecuzione del programma.*

Nell'esempio notiamo in primis l'assenza di formalità. Si introducono quindi concetti:

1. **precondizione** che nell'esempio è fatta da $n > 0$ e $v[i] \in \mathbb{Z},\ \forall i \in \{0, \ldots, n-1\}$
2. **postcondizione** che nell'esempio si ritrova con l'asserzione x è il massimo in $v[0..h-1]$ e $h = n$, che scritto in modo formale diventa:

$$\left.\begin{array}{l} v[i] \leq x,\ \forall i \in \{0, \ldots, n-1\} \\ \exists\, i \in \{0, \ldots, n-1\} \text{ t.c. } v[i] = x \end{array}\right\} x = max(v[0..n-1])$$

Queste formule possono essere rese come formule proposizionali, tramite una congiunzione logica:

$$\begin{cases} v[0] \leq x \wedge v[1] \leq x \wedge \ldots v[n-1] \leq x \\ v[0] = x \vee v[1] = x \vee \ldots v[n-1] = x \end{cases}$$

Abbiamo studiato lo stato della memoria del programma in un certo istante tramite formule.

Definizione 9. *Definiamo* ***stato della memoria*** *come:*

$$s : V \to \mathbb{Z}$$

ovvero una funzione che mappa le variabili del programma (poste nell'insieme V) in $\mathbb{Z}$.

Fissato uno stato della memoria e una formula posso validare una formula in quello stato osservando le variabili e le relazioni aritmetiche della formula. Data una formula ϕ e uno stato s posso sapere se ϕ è valida in s.
Una formula che gode della **proprietà invariante** se vera all'inizio dell'iterazione è vera anche alla fine della stessa. Per capire se è invariante basta vedere lo stato di una formula ad inizio e fine di una iterazione.
L'esecuzione di una istruzione cambia lo stato della memoria. Potrebbe però accadere che una serie di istruzioni non facciano terminare il programma, perciò quanto detto sopra è in realtà un'approssimazione della realtà.

Definizione 10. *Definiamo la* ***specifica di correttezza di un programma*** *con la tripla:*

$$\alpha \ P \ \beta$$

dove:

- *α e β sono formule (definite con tutte le simbologie aritmetiche tra variabili, sia di conto che di relazione).*
- *P è un "programma" (anche un frammento o una singola istruzione) che modifica lo stato della memoria.*

α è la ***precondizione****, che supponiamo verificata nello stato iniziale e β è la* ***postcondizione****, che supponiamo valida dopo l'esecuzione del programma.*

Durante il corso useremo un linguaggio imperativo non reale semplificato. Dovremo anche definire una logica, definendo un apparato deduttivo, un insieme di regole per costruire dimostrazioni derivando nuove formule da quelle preesistenti. Useremo la **logica di Hoare**. Le formule della logica di Hoare sono triple di tipo ($\alpha P \beta$) quindi si tratta di una logica di tipo diverso anche se si appoggia su quella proposizionale.

4.1 Linguaggio semplificato

Definiamo quindi il linguaggio di programmazione imperativo semplificato che andremo ad utilizzare. Si userà una grammatica formale.
L'elemento fondamentale di questo linguaggio è il **comando**, che indica o una singola istruzione o un gruppo di istruzioni strutturate. Il simbolo usato nella grammatica per indicare un comando è "C". Un comando viene costruito tramite le **produzioni**, introdotte da "::=". Il comando più semplice è l'**assegnamento**, che usa l'operatore ":=" per assegnare un valore ad una variabile. Con il simbolo "E" indichiamo un simbolo non terminale della grammatica che sta per *espressione*. Una volta costruito semplici espressioni

possiamo combinarle, eseguendole in sequenza, inserendo un ";" tra due comandi.
In merito all'istruzione di scelta abbiamo l'istruzione "if", seguito da un'espressione booleana, seguito da "then", seguita da un comando, seguita da "else", seguita da un comando, e il tutto viene concluso da "endif". Qui abbiamo una prima semplificazioni dicendo che l'*else* è obbligatorio. Per l'iterazione abbiamo il "while", seguito da un'espressione booleana, seguito da "do" con poi il comando, il tutto concluso da endwhile. Infine abbiamo una istruzione speciale, chiamata "skip", che non fa nulla e avanza il *program counter* (con essa posso saltare il ramo alternativo dell'*if-else*).
Un'espressione booleana "B" può essere la costante "true", la costante "false" o del tipo "not B", "B and B", "B or B", "E < E" (e le altre), "E = E". Non si hanno tipi di dato ma supporremo di avere a che fare solo con *interi*. Non si ha la funzione di nozione o di classe. Questo linguaggio è comunque *Turing Complete*.

Listing 2 Esempio di programma D

```
x := a; y := b;
while x != y do
if x < y then
y := y - x;
else
x := x - y;
endif
endwhile
```

Cerchiamo di capire se il programma D, sopra definito, soddisfa la tripla:

$$\{a > 0 \land b > 0\}\ D\ \{x = MCD(a, b)\}$$

Quindi mi chiedo se eseguendo il programma con uno stato della memoria dove a e b sono due interi positivi (precondizione) allora, alla fine dell'esecuzione di D, x sarà il massimo comune divisore tra a e b (postcondizione). Dobbiamo dimostrare la tripla e qualora non fosse vera bisogna confutarla trovando un caso in cui non è verificata (trovando uno stato che soddisfi la precondizione ma che, una volta eseguito il programma, la postcondizione non sia verificata).

4.2 Logica di Hoare

Definizione 11. *Una **dimostrazione**, in una logica data, è una sequenza di formule di quella logica che sono o assiomi (formule che riteniamo vere*

a priori) o formule derivate dalle precedenti tramite una regola di inferenza. Una regola di inferenza mi manda da un insieme di formule $\alpha_1, \alpha_2, \ldots, \alpha_n$ ad una nuova formula α e si indica con:

$$\frac{\alpha_1, \alpha_2, \ldots, \alpha_n}{\alpha}$$

Che si legge come: "se ho già derivato $\alpha_1, \alpha_2, \ldots, \alpha_n$ sono autorizzato a derivare α".
*Nel nostro caso ogni α_i è una tripla della **logica di Hoare**.*
Una dimostrazione si ottiene quindi applicando le regole di derivazione fino ad arrivare, se si riesce, ad una soluzione.

4.2.1 Regole di derivazione

Vediamo quindi le regole di derivazione, che sono associate alle regole del linguaggio sopra definito.

Skip

Definizione 12. *Partiamo con la regola per l'istruzione **skip**, che non facendo nulla non cambia lo stato della memoria e quindi la regola di derivazione non ha nessuna premessa:*

$$\frac{}{\{p\}\ skip\ \{p\}}$$

con p che è una formula proposizionale. Dopo lo skip p vale se valeva prima dello skip

Implicazione

Definizione 13. *La seconda è una regola che non ha un rapporto diretto con il linguaggio e chiameremo **regola di conseguenza (o dell'implicazione)**. Ha due premesse: una tripla appartenente alla logica di Hoare e una implicazione della logica proposizionale.*

$$\frac{p \implies p' \quad \{p'\}\ C\ \{q\}}{\{p\}\ C\ \{q\}}$$

Ovvero se eseguo C partendo da uno stato in cui vale p' allora dopo varrà q. Ma sappiamo anche che p implica p'. Quindi se nel mio stato della memoria vale p e quindi anche p' per l'implicazione. Posso quindi dire che se ho p ed eseguo C ottengo q.
Ho anche una forma speculare:

$$\frac{\{p\}\ C\ \{q'\} \quad q' \implies q}{\{p\}\ C\ \{q\}}$$

Sequenza

Definizione 14. *La terza regola è legata alla struttura di* ***sequenza*** *dei programmi:*

$$\frac{\{p\}\ C_1\ \{q\}\quad \{q\}\ C_2\ \{r\}}{\{p\}\ C_1; C_2\ \{r\}}$$

Assegnamento

Definizione 15. *La quarta regola riguarda l'****assegnamento****. Questa è l'unica regola non banale (come lo è lo skip) che non ha premesse. È la regola base per derivare le triple necessarie alle altre regole. Quindi, avendo E come espressione, x una variabile e p come una postcondizione, ovvero una formula che contiene gli identificatori di diverse variabili:*

$$\frac{}{\{p[E/x]\}\ x \coloneqq E\ \{p\}}$$

Dove con $p[E/x]$, come precondizione, indichiamo una ***sostituzione*** *indicante che cerchiamo in p tutte le occorrenze x e le sostituiamo con E.*

Esempio 3. *Se ho $x \coloneqq y+1$ con E pari a $y+1$ e con p pari a $\{x > 0\}$ come postcondizione data e cerco la precondizione. Quindi la tripla completa sarebbe:*

$$\{y + 1 > 0\}\ x \coloneqq y + 1\ \{x > 0\}$$

E la tripla sappiamo che è vera *(se so che $y+1$ è positivo, dopo che a x assegno $y+1$ posso essere sicuro che anche x è positivo).*

Esempio 4. *Se ho $x \coloneqq x+1$ con E pari a $x+2$ e con p pari a $\{x > 0 \land x \leq y\}$ come postcondizione data e cerco la precondizione. Quindi la tripla completa sarebbe:*

$$\{x + 2 > 0 \land x + 2 \leq y\}\ x \coloneqq x + 1\ \{x > 0 \land x \leq y\}$$

e anche questa tripla è garantita dalla regola di sostituzione

Istruzione di scelta

Definizione 16. *La quinta regola è quella relativa all'****istruzione di scelta*** *che è della forma:*

$$\{p\}\ \textit{if } B \textit{ then } C \textit{ else } D \textit{ endif}\ \{q\}$$

Se la condizione B è vera eseguo C altrimenti D. In entrambi i casi alla fine deve valere la postcondizione q. Separiamo i due casi:

- *se suppongo vere p e B eseguo C arrivando in q:*

$$\{p \wedge B\}\ C\ \{q\}$$

- *se suppongo vera p ma falsa B avrò:*

$$\{p \wedge \neg B\}\ D\ \{q\}$$

Ricavo quindi la formula generale:

$$\frac{\{p \wedge B\}\ C\ \{q\} \qquad \{p \wedge \neg B\}\ D\ \{q\}}{\{p\}\ \textit{if } B \textit{ then } C \textit{ else } D \textit{ endif}\ \{q\}}$$

Come notazione usiamo che:

$$\vdash \{p\}\ C\ \{q\}$$

dove $\vdash$ segnala che la tripla è stata **dimostrata/derivabile** con le regole di derivazione (si parla quindi di *sintassi*, viene infatti ignorato il significato ma si cerca solo di applicare le regole, ottenendo al conclusione come risultato di una catena di regole).
Come notazione usiamo anche che:

$$\vDash \{p\}\ C\ \{q\}$$

dove $\vDash$ indica che la tripla è **vera** (si parla quindi di *semantica*, riferendosi al significato).
Dato che si ha **completezza** e **correttezza** dell'apparato deduttivo si ha hanno due situazioni.

- *ogni tripla **derivabile** è anche **vera** in qualsiasi interpretazione*
- *ogni tripla **vera** vorremmo fosse anche **derivabile** e il discorso verrà approfondito in seguito per la logica di Hoare*

$\vdash$ può avere a pedice una sigla per la regola rappresentata, ad esempio $\vdash_{ass}$ per l'assegnamento.

Esempio 5. *Vediamo qualche esempio di dimostrazione. Dimostro che la tripla seguente sia vera:*

$$\{y \geq 0\}\ x \coloneqq 2 \cdot y + 1\ \{x > 0\}$$

uso la regola di assegnamento *(che non ha premesse) partendo dalla postcondizione. Sostituisco e ottengo:*

$$\vdash_{ass} \{2 \cdot y + 1 > 0\} \ x := 2 \cdot y + 1 \ \{x > 0\}$$

Procedo usando la regola di implicazione *(sapendo che* $y \geq 0 \implies 2y+1 > 0$*):*

$$\vdash_{impl} \{y \geq 0\} \ x := 2 \cdot y + 1 \ \{x > 0\}$$

Dimostrando quindi che la tripla è ***vera****.*

Esempio 6. *Vediamo qualche esempio di dimostrazione. Dimostro che la tripla seguente sia vera:*

$$\{z > 0\} \ x := (y \cdot z) + 1 \ \{x > 0\}$$

e vediamo che la tripla non è valida in quanto y potrebbe essere negativo e non portare alla positività di x. Formalmente cerchiamo un controesempio cercando di non soddisfare la postcondizione. Scelgo in memoria $z = 1$ *e* $y = -2$*. Abbiamo fatto quindi un ragionamento semantico. Provo a anche sintatticamente e giungo a:*

$$\vdash \{(y \cdot z + 1) > 0\} \ x := (y \cdot z) + 1 \ \{x > 0\}$$

che non è vero, quindi non posso proseguire.

Esempio 7. *Vediamo qualche esempio di dimostrazione. Dimostro che la tripla seguente sia vera:*

$$\{s = x^i\} \ i := i + 1, \ s := s \cdot x \ \{s = x^i\}$$

anche se uguali precondizione e postcondizione fanno riferimento a due momenti della memoria diversi e quindi i valori saranno diversi.
Abbiamo a che fare con un ***invariante*** *in quanto la formula non varia tra precondizione e postcondizione anche se i valori saranno diversi (in mezzo al processo posso violare comunque l'invarianza).*
Ragioniamo in modo puramente sintattico. Dobbiamo applicare due volte l'assegnamento (e spesso serve dopo anche la regola di implicazione) e una volta la sequenza. Anche qui partiamo dalla postcondizione risalendo via via alle precondizioni:

$$\vdash_{ass} \{sx = x^i\} \ s := s \cdot x \ \{s = x^i\}$$

Ho creato quindi la condizione intermedia tra i due assegnamenti e proseguendo ho:

$$\vdash_{ass} \{sx = x^{i+1}\} \ i := i + 1 \ \{sx = x^i\}$$

per transitività si può scrivere:

$$\{sx = x^{i+1}\}\ i := i+1,\ s := s \cdot x\ \{s = x^i\}$$

ma non coincide con quanto voglio dimostrare. Ragiono quindi in modo algebrico. In $\{sx = x^{i+1}\}\ i := i+1\ \{sx = x^i\}$ *ho infatti:*

$$\{s\cancel{x} = x^{i+\cancel{1}}\} \rightarrow \{sx = x^i\},\ se\ x \neq 0$$

e quindi la formula iniziale è dimostrata.

Esempio 8. *Vediamo qualche esempio di dimostrazione. Dimostro che la tripla seguente sia vera:*

$$\{\top\}\ if\ x < 0\ then\ y := -2 \cdot x\ else\ y := 2 * x\ endif\ \{y \geq 0\}$$

Diciamo che C *rappresenta* $y := -2 \cdot x$ *e* D *rappresenta* $y := 2 * x$ *per praticità. Indichiamo la condizione booleana* $x < 0$ *con* B*. Tutta la condizione di scelta la chiamiamo* S *Quindi avremo:*

$$\{\top\}\ if\ B\ then\ C\ else\ D\ endif\ \{y \geq 0\}$$

che in modo ancora più compatto sarebbe:

$$\{\top\}\ S\ \{y \geq 0\}$$

Applico quindi la regola di derivazione al primo caso: *Ho quindi:*

$$\{\top \wedge x < 0\}\ C\ y \geq 0$$

che è uguale a:

$$\{x < 0\}\ C\ y \geq 0$$

Procedo ora con l'assegnamento:

$$\vdash_{ass} \{-2 \cdot x \geq 0\}\ y := -2 \cdot x\ \{y \geq 0\}$$

che però equivale algebricamente a:

$$\vdash_{ass} \{x \leq 0\}\ y := -2 \cdot x\ \{y \geq 0\}$$

ma siccome $x < 0 \implies x \geq 0$ *uso la regola dell'implicazione:*

$$\vdash_{impl} \{x < 0\}\ y := -2 \cdot x\ \{y \geq 0\}$$

Passo al secondo caso:

$$\{\top \wedge x \geq 0\}\ D\ y \geq 0$$

che è uguale a:

$$\{x \geq 0\}\ D\ y \geq 0$$

Procedo ora con l'assegnamento:

$$\vdash_{ass} \{2 \cdot x \geq 0\}\ y \coloneqq 2 \cdot x\ \{y \geq 0\}$$

che però equivale algebricamente a:

$$\vdash_{ass} \{x \geq 0\}\ y \coloneqq -2 \cdot x\ \{y \geq 0\}$$

e quindi è dimostrabile che:

$$\vdash \{\top\}\ S\ \{y \geq 0\}$$

Iterazione

Definizione 17. *Definiamo:*

- ***correttezza parziale**, dove la tripla viene letta supponendo a priori che l'esecuzione termini*
- ***correttezza totale**, dove la tripla viene letta dovendo anche dimostrare che l'esecuzione termini. Si procede quindi prima dimostrando la correttezza parziale aggiungendo poi la dimostrazione per l'esecuzione finita*

Definizione 18. *La sesta regola è quella relativa all'iterazione. Abbiamo quindi:*

$$\{p\}\ \textit{while}\ B\ \textit{do}\ C\ \textit{endwhile}\ \{q\}$$

ma non posso determinare a priori quante volte si eseguirà C, che modifica lo stato della memoria. Per comodità $\textit{while}\ B\ \textit{do}\ C\ \textit{endwhile}$ la chiameremo W, quindi in modo compatto abbiamo:

$$\{p\}\ W\ \{q\}$$

Partiamo con la correttezza parziale supponendo a priori che l'esecuzione termini. Si ha quindi che in q sicuramente B è falsa, altrimenti non si avrebbe terminazione. Quindi in realtà abbiamo:

$$\{p\}\ W\ \{q \wedge \neg B\}$$

Ipotizziamo che all'inizio B sia vera, quindi la precondizione sarà $\{i \wedge B\}$, con i rappresentante una nuova formula. In questi stati eseguiremo C. Suppongo di poter derivare, tramite C eseguito una sola volta, nuovamente i:

$$\{i \wedge B\}\ C\ \{i\}$$

Se vale questa tripla significa che i è un ***invariante*** *per C e quindi i viene chiamata* ***invariante di ciclo****. Nello stato raggiunto dopo C la condizione B può essere valida o meno. Qualora valga dopo la singola esecuzione di C allora avrei ancora $\{i \wedge B\}$ e dovrei eseguire nuovamente C e dopo varrà ancora i sicuramente e bisogna ristudiare B per capire come procedere, in quanto i resterà vera per qualsiasi numero di iterazioni di C. Si ha che i resterà vera, teoricamente, anche una volta "usciti" dal ciclo. Qualora non valga B si ha che:*

$$\{i \wedge \neg B\}\ W\ \{i \wedge \neg B\}$$

(dove si nota che i resta vera ma B impedisce di tornare nel ciclo).
Si ha quindi che:

$$\frac{\{i \wedge B\}\ C\ \{i\}}{\{i\}\ W\ \{i \wedge \neg B\}}$$

che è la ***regola dell'iterazione****. Scritta in modo completo:*

$$\frac{\{i \wedge B\}\ C\ \{i\}}{\{i\}\ \textit{while}\ B\ \textit{do}\ C\ \textit{endwhile}\ \{i \wedge \neg B\}}$$

Una nozione più forte di ***invariante*** *può essere espressa dicendo che*

$$\{i\}\ C\ \{i\}$$

che si differenzia da quella di ***invariante di ciclo*** *(dove mi interessa sapere che un invariante sia vero prima del ciclo anche se questa non è un proprietà intrinseca degli invarianti):*

$$\{i \wedge B\}\ C\ \{i\}$$

Ricordiamo che stiamo dando per scontata la ***terminazione*** *tramite la* ***correttezza parziale****.*
Facciamo qualche osservazione:

- *nella precondizione della conclusione non si ha B, in quanto il corpo dell'iterazione può anche non essere mai eseguito*

- *data un'istruzione iterativa posso avere più di un invariante. Si ha inoltre che ogni formula iterativa ha l'**invariante banale** $i = \top$. Possiamo avere anche una formula in cui compaiono variabili che non sono modificate della funzione iterativa e quindi l'intera formula è un invariante di ciclo. Studiamo quindi gli invarianti "più utili"*

- *nei casi pratici non consideriamo ovviamente iterazioni isolate ma iterazioni inserite in un programma. In questi casi quindi la scelta di un invariante adeguato dipende sia dall'iterazione che dall'intero contesto.*

  Esempio 9. *Ho un programma su cui voglio dimostrare:*

  $$\{p\}\ C;\ W;\ D\ \{q\}$$

  con W iterazione e C, D comandi.
  Spezziamo quindi il programma per fare la dimostrazione, tramite la regola della sequenza:

  $$\{p\}\ C\ \{r\}\ W\ \{z\}\ D\ \{q\}$$

  r sarà quindi la precondizione dell'iterazione W che porterà a z che sarà precondizione di D.
  Sapendo che la regola di derivazione per W è:

  $$\{i\}\ W\ \{i \wedge \neg B$$

  Confronto la tripla con la "catena" sopra espressa. Si nota che r dovrà implicare l'invariante per W.

Esempio 10. *Vediamo quindi un esempio completo.*
Si prenda il seguente programma:

Listing 3 Programma P

```
i := 0; s := 1;
while i < N do
  i := i + 1;
  s := s * x;
endwhile
```

Per comodità chiamo A i due assegnamenti iniziali, W l'iterazione e C il corpo dell'iterazione.
Ci proponiamo di derivare:

$$\{N \geq 0\}\ P\ \{s := x^N\}$$

x e N sono variabili, nella realtà costanti non venendo mai modificate da P, e il loro valore fa parte dello stato della memoria.
Concentriamoci in primis sull'iterazione cercando un invariante. Una strategia semplice è quella di simulare i primi passi di una esecuzione. Supponendo N comunque non nullo abbiamo, seguendo le due variabili i e s nelle varie iterazioni (procedendo verso destra nella tabella), procedendo in modo simbolico per s (usando quindi il simbolo x direttamente):

i	0	1	2	3	...
s	1	x	x^2	x^3	...

Quindi $s = x^i$ è il nostro invariante. Dimostro questa ipotesi è vera, dimostrando:

$$\{s = x^i \wedge i < N\}\ C\ \{s = x^i\}$$

sapendo che essa è la premessa alla regola di iterazione.
Procedo quindi con la dimostrazione, avendo l'assegnamento:

$$\vdash_{ass} \{sx = x^{i+1}\}\ C\ \{s = x^i\}$$

infatti i due assegnamenti sono indipendenti*, permettendo di applicare in una sola volta due regole di assegnamento.*
Sapendo che $s\cancel{x} = x^{i+\cancel{1}} \implies s = x^i$. Ho quindi mostrato che:

$$\vdash \{s = x^i\}\ C\ \{s = x^i\}$$

*dimostrando che ho effettivamente l'invariante $s = x^i$ (e nel dettaglio abbiamo trovato un **invariante forte**, che lo è a priori rispetto alla condizione booleana B).*
Vogliamo però ottenere come precondizione $\{s = x^i \wedge i < N\}$. Sapendo però che $s = x^i \wedge i < N$ è una regola "più forte" di $s = x^i$ (se è vera la prima sicuramente è vera anche la seconda) posso usare l'implicazione e dire che:

$$\{s = x^i \wedge i < N\} \implies \{s = x^i\}$$

quindi:

$$\vdash_{impl} \{s = x^i \wedge i < N\}\ C\ \{s = x^i\}$$

posso quindi applicare la regola dell'iterazione avendo la premessa corretta e ottenendo quindi:

$$\vdash_{iter} \{s = x^i\}\ W\ \{s = x^i \wedge i \geq N\}$$

ma la postcondizione non ci sta implicando $s = x^N$, per avere:

$$\{N \geq 0\}\ P\ \{s := x^N\}$$

bisogna quindi ***rafforzare*** *l'invariante, in modo che l'invariante implichi che alla fine dell'esecuzione $i = N$.*
Procediamo quindi con una nuova ipotesi, cercando di dimostrare che $i \leq N$ è un invariante:

$$\{i \leq N \wedge i < N\}\ C\ \{i \leq N\}$$

Questa precondizione contiene una formula che è più restrittiva dell'altra. Procedo con l'assegnamento (CAPIRE QUESTA PARTE):

$$\vdash_{ass} \{i + 1 \leq N\}\ C\ \{i \leq N\}$$

ma sapendo che $i < N \implies i + 1 \leq N$ ottengo:

$$\vdash_{impl} \{i < N\}\ C\ \{i \leq N\}$$

Dimostrando quindi che è un invariante.
Passo quindi all'iterazione:

$$\vdash_{iter} \{i \leq N\}\ W\ \{i \leq N \wedge i \geq N\}$$

e la postcondizione è uguale a dire che $i = N$.
Considerando quindi i due invarianti ottengo:

$$\vdash_{iter} \{s = x^i \wedge i \leq N\}\ W\ \{s = x^i \wedge i = N\}$$

Dobbiamo però dimostrare anche la parte degli assegnamenti iniziali:

$$\{N \geq 0\}\ A\ \{s = x^i \wedge i \leq N\}$$

Bisogna infatti vedere se anche le condizioni iniziali permettono all'invariante doppio di restare tale.
Applico quindi due volte l'assegnamento (partendo dal fondo); il primo con $s := 1$:

$$\vdash_{ass} \{1 = x^i \wedge i \leq N\}\ s := 1\ \{s = x^i \wedge i \leq N\}$$

Passo quindi a $i := 0$ seguendo il nuovo ordine delle condizioni:

$$\vdash_{ass} \{1 = x^0 \wedge 0 \leq N\}\ i := 0\ \{1 = x^i \wedge i \leq N\}$$

Applico quindi la sequenza con la prima parte (sapendo $1 = x^0$ è sempre vero):

$$\vdash_{seq} \{N \geq 0\}\ A\ \{s = x^i \wedge i \leq N\}$$

completando la dimostrazioni.
Da notare solo che x deve essere non nullo.

***Altri esempi sono presenti sulla pagina del corso**.*

4.2.2 Correttezza totale

Analizziamo ora il caso in cui non si abbia certezza di terminazione di un'iterazione:

$$\{p\} \text{ while } B \text{ do } C \text{ endwhile } \{q\}$$

Distinguiamo i due tipi di correttezza, dal punto di vista della derivabilità, tramite:

$$\vdash^{parz} \{p\} \text{ C } \{q\}$$

$$e$$

$$\vdash^{tot} \{p\} \text{ C } \{q\}$$

Dal punto di vista semantico non ha invece senso distinguere di due casi e quindi si ha solo:

$$\vDash \{p\} \text{ C } \{q\}$$

In quanto dal punto di vista semantico o si ha terminazione o non si ha, non si hanno casistiche differenti a seconda di correttezza totale o parziale.
Bisognerà quindi dimostrare la terminazione.
Studiamo il caso semplice dove:

$$W = \text{ while } B \text{ do } C \text{ endwhile}$$

Cerchiamo un'espressione aritmetica E, dove compaiono le variabili del programma, costanti numeriche e operazioni aritmetiche. Cerchiamo anche un **invariante di ciclo** i (che quindi è una formula) per W. E e i devono soddisfare due condizioni:

1. $i \implies E \geq 0$, quindi se vale i allora l'espressione aritmetica ha valore ≥ 0 (il valore di E lo ottengo eseguendo le operazioni sulle costanti e sui valori, in quello stato della memoria, delle variabili)

2. $\vdash^{tot} \{i \wedge B \wedge E = k\} \text{ C } \{i \wedge E < k\}$, dove nella precondizione abbiamo la congiunzione logica tra l'invariante i, la condizione di ciclo B e tra $E = k$, avendo prima assegnato a k il valore effettivo di E nello stato in cui iniziamo a computare C. k quindi non deve essere una variabile del programma e non deve apparire in C (rappresentante il corpo della singola iterazione). Se vale la condizione 1 e l'invariante sappiamo che $E \geq 0 \implies k \geq 0$. La postcondizione è formata dall'invariante i e dal fatto che il valore di E dopo l'esecuzione sia strettamente minore di k (che è il valore di E prima dell'esecuzione). In pratica E decresce ad ogni singola iterazione, ovvero ad ogni esecuzione di C.

La notazione $\vdash^{tot}$ serve a escludere che C abbia altri cicli annidati (portando a dover dimostrare che anche i cicli interni terminano). Per praticità quindi supponiamo di non avere cicli interni (stiamo partendo dal ciclo più interno)

In pratica, nella seconda condizione, uso E per concludere che una singola esecuzione dell'iterazione mi porta in uno stato in cui vale l'invariante, dove può valere o meno B (che potrebbe portare all'uscita dall'iterazione) ma in cui E ha un valore diverso, minore a quello di partenza ma mai minore di 0 per la prima condizione (in quanto l'invariante è sempre valido). Quindi, ad un certo punto, E raggiungerà il valore minimo e in quel momento o B è falsa o si ha una contraddizione (E dovrebbe diventare negativo), quindi si ha la terminazione.
Quindi se abbiamo dimostrato le due condizioni posso dire:

$$\vdash^{tot} \{i\}\ \mathrm{W}\ \{i \wedge \neg B\}$$

che è anche la conclusione della regola di derivazione dell'iterazione in caso di correttezza parziale.
Possiamo anche osservare due cose:

1. E non è una formula logica ma un'espressione aritmetica con valore numerico ($E \geq 0$ è una formula logica)

2. qualora si abbia $E = 0$ non si hanno problemi, 0 è solo un esempio, potremmo riscrivere la prima condizione come $E \geq n$, con n qualsiasi numero intero, basta che sia ben definito per poter permettere la ripetizione finita delle iterazioni

Chiameremo questa espressione aritmetica è **variante**.
L'invariante della correttezza totale non è sempre quello di quella parziale, magari è uguale, magari diverso o anche uguale solo in parte

Esempio 11. *Vediamo un esempio chiarificatore.*
Dato il programma (una sorta di conto alla rovescia):

Listing 4 Programma P

```
while x > 5 do
  x := x - 1;
endwhile
```

studio la correttezza totale della tripla:

$$\{x > 5\}\ P\ \{x = 5\}$$

Innanzitutto cerco un variante E, che decresce ad ogni singola iterazione è un invariante i come descritti sopra, quindi che, qualora ci sia l'invariante i allora $E \geq 0$.
Un primo invariante "banale" è $i = x \geq 5$.
In merito ad E notiamo che nel corpo dell'iterazione abbiamo un solo comando, dove il valore della variabile x viene decrementato, quindi un primo candidato per E potrebbe essere proprio x.
Per comodità scegliamo però come variante $x - 5$ per avere una corrispondenza diretta con l'invariante $x \geq 5$, essendo derivato dallo stesso invariante (basandoci in primis sulla prima condizione).
***Non sempre posso ottenere una corrispondenza tra variante e invariante**.*
Presi questo invariante e questo variante verifichiamo le due condizioni:

1. $x \geq 5 \implies x - 5 \geq 0$ *è ovviamente corretto perché si ottiene che* $x \geq 5 \implies x \geq 5$ *(infatti questo è il motivo per cui si è scelto* $x - 5$ *come variante)*

2. $\vdash^{tot} \{x \geq 5 \wedge x > 5 \wedge x - 5 = k\}\ x := x - 1\ \{x \geq 5 \wedge x - 5 < k\}$
   Applico quindi la regola dell'assegnamento (che si applica anche per la correttezza totale) e ottengo la precondizione per la postcondizione $\{x \geq 5 \wedge x - 5 < k\}$*:*

   $$\{x - 1 \geq 5 \wedge x - 1 - 5 < k\} = \{x \geq 6 \wedge x - 6 < k\}$$

   che però è diversa da $\{x \geq 5 \wedge x > 5 \wedge x - 5 = k\}$.
   Cerco quindi di applicare la regola dell'implicazione. Riguardando la precondizione originale noto che $x > 5$ *è "più forte" di* $x \geq 5$ *e quindi quest'ultimo può essere trascurato. Ricordando che stiamo lavorando su valori interi si ha che* $x > 5 \implies x \geq 6$*. Inoltre, sempre per il fatto che lavoriamo su numeri interi,*
   $x - 5 = k \implies x - 6 < k$ *e quindi:*

   $$\{x \geq 5 \wedge x > 5 \wedge x - 5 = k\} \implies \{x \geq 6 \wedge x - 6 < k\}$$

Abbiamo quindi dimostrato che il programma termina e vale la tripla iniziale. Qualora avessimo dovuto dimostrare la correttezza parziale si sarebbe dovuto procedere riutilizzando lo stesso invariante e procedendo studiando la negazione di B, ovvero $x \leq 5$ (che insieme danno la postcondizione).

Esempio 12. *vediamo un esempio non si ha alcuna variabile che decrementa nel corpo dell'iterazione:*

Listing 5 Programma P

```
while x < 5 do
  x := x + 1;
endwhile
```

studio la correttezza totale della tripla:

$$\{x < 5\}\ P\ \{x = 5\}$$

Dal punto di vista della correttezza parziale ragiono come al solito mentre per la correttezza totale la situazione è un po' diversa.
Ovviamente non posso usare x come variante ma posso sicuramente usare, per esempio, $-x$, che sicuramente decresce visto che x cresce, in entrambi i casi ad ogni iterazione. Riprendendo l'esempio sopra quindi posso usare $x \leq 5$ come invariante e, al posto di $-x$ che comunque andrebbe bene, secondo una ragionamento simile allo scorso esempio, $5 - x$ come variante (si nota che anche questo E è ricavabile da i, anche se questo non è sempre attuabile). Il resto della dimostrazione è analoga all'esempio precedente.

Correttezza e Completezza

In generale quando si sviluppa una logica, con un apparato deduttivo e un'interpretazione delle formule, siamo interessati a due proprietà generali della logica e dell'apparato deduttivo:

1. **correttezza**, ovvero il fatto che tutto quello che si può derivare con l'apparato deduttivo è effettivamente vero, ovvero $vdash \implies \vDash$. Quindi se una tripla è derivabile è anche vera

2. **completezza**, ovvero il fatto che l'apparato deduttivo sia in grado di derivare tutte le formule vere (ovvero tutte le triple) vere. Si ha quindi $\vDash \implies \vdash$

Per la logica di Hoare vale la correttezza ma vale una proprietà di completezza è *relativa* in quanto nel corso di una dimostrazione di completezza occorre a volte usare deduzioni della logica proposizionale ma tali formule parlano anche di operazioni aritmetiche. Si ha che l'aritmetica può essere formalizzata attraverso un linguaggio logico con degli assiomi e delle regole di inferenza. Si hanno però i **teoremi di incompletezza dell'aritmetica** di Gödel che dicono che l'aritmetica come teoria matematica è incompleta in quanto ci sono formule scritte nel linguaggio dell'aritmetica che sono vere ma non sono dimostrabili. Questa incompletezza si riverbera sulla logica di Hoare, che comunque, dal punto di vista deduttivo sulle triple, è completa.

4.2.3 Precondizione più debole

Dato un comando C e una formula q che interpretiamo come post condizione di C. Si cerca p tale per cui:

$$\vdash \{p\}\ \mathrm{C}\ \{q\}$$

Sia valida e quindi vera.
Ovviamente il caso interessante è quello totale.

Esempio 13. *Suppongo di avere come comando un singolo assegnamento* $y := 2 \cdot x - 1$ *e come postcondizione* $\{y > x\}$.
Cerco possibili precondizioni che, dopo l'assegnamento, portino a quella postcondizione. Ovviamente si avrebbero infiniti valori di x *che consentono di arrivare alla postcondizione (nonché altrettanti che non lo permettono).*

Si cerca quindi la "migliore" precondizione per arrivare ad una data postcondizione ma per farlo ci serve un criterio di confronto.
Introduciamo quindi alcuni simboli e nozioni utili:

- V è l'insieme della variabili di C
- $\Sigma = \{\sigma |\, \sigma : V \to \mathbb{Z}\}$ è l'insieme degli stati della memoria (ricordando che posso avere solo valori interi nel nostro linguaggio)
- Π è l'insieme di tutte le formule sull'insieme V
- $\sigma \vDash p$ significa che la formula p è vera nello stato σ e si ha che $\vDash \subseteq \Sigma \times \Pi$, con $\vDash$ che indica la veridicità di una formula in uno stato
- $t(\sigma) = \{p \in \Pi |\, \sigma \vDash p\}$ come la funzione che assegna ad uno stato l'insieme delle proposizioni, ovvero tutte le formule, che sono vere in σ
- $m(p) = \{\sigma \in \Sigma |\, \sigma \vDash p\}$ come la funzione che associa ad una formula l'insieme di tutti gli stati che soddisfano la formula

Tra le due funzioni finali si ha una sorta di "dualità" e agiscono in modo speculare tra loro. Se, partendo da Σ, applico $t(\sigma)$ e scopriamo che un certo stato p appartiene a Π e a tale p associamo l'insieme dato da $m(p)$ si avrà che $\sigma \in m(p)$.
Si possono fare altre osservazioni su queste due funzioni.
Prendiamo due formule p e q e cerco di capire se è possibile avere $m(p) = m(q)$.
La risposta è positiva e si parla, in caso, di *equivalenza tra formule*. Fissati

due stati σ_1 e σ_2 mi chiedo se possano appartenere allo stesso insieme di formule. In questo caso la risposta è negativa, in quanto sei i due stati sono distinti allora ci deve essere almeno una variabile x che ha valore diverso nei due stati ma allora è facile trovare una formula che separa i due stati e tale formula potrà essere vera solo in uno dei due stati.
Dato $S \subseteq \Sigma$ e $F \subseteq \Pi$ (ragiono quindi su sottoinsiemi) si ha che:

- $t(S) = \{p \in \Pi | \forall \sigma \in S,\ \sigma \vDash p\} = \bigcap_{\sigma \in S} t(\sigma)$ (ragiono quindi su tutti gli stati di S e quindi sulle formule che sono vere in tutti questi stati)
- $m(F) = \{\sigma \in \Sigma | \forall p \in F\ \ \sigma \vDash p\} = \bigcap_{p \in \Pi} m(p)$ (ragiono quindi su tutte le formule di Π e quindi su tutti gli stati in devono essere soddisfatte tutte queste formule)

Abbiamo quindi esteso il dominio delle due funzioni a sottoinsiemi di stati e sottoinsiemi di formule.
Si può anche dimostrare che $S \subseteq m(t(s))$ e che $F \subseteq t(m(F))$. Inoltre, dati $A \subseteq B$, si può dimostrare che $m(B) \subseteq m(A)$ (se ho un insieme più grande di formule ho meno stati che le soddisfano tutte).
Si ha un rapporto tra la logica proposizionale (coi suoi connettivi logici) e l'insieme di stati (p e q sono formule di Π):

- $m(\neg p) = \Sigma \setminus m(p)$, quindi il complemento insiemistico di p
- $m(p \vee q) = m(p) \cup m(q)$, quindi all'unione dei due insiemi
- $m(p \wedge q) = m(p) \cap m(q)$ (da verificare) quindi all'intersezione dei due insiemi

L'implicazione merita un discorso a parte in quanto ha due "nature":

- **operatore logico**, che comporta che $p \implies q = \neg p \vee q$ e in tal caso si ha, in linea a quanto detto per gli altri connettivi:

$$m(p \implies q) = m(\neg p) \cup m(q)$$

- **relazione tra formule**, dove se p implica q (in tutti in casi in cui è vera p è vera q) allora $m(p) \subseteq m(q)$, intendendo che q è "più debole" (ovvero come formula ci da una conoscenza inferiore essendo q corrispondente ad un insieme più grande di stati) di p. Si ha a che fare con una relazione algebrica tra le due formule. Più "debole" non significa comunque "peggiore"

Posso quindi capire come definire la precondizione migliore per ottenere una tripla valida, insieme al comando C e alla postcondizione.
Un modo è quello di definire la migliore precondizione come la precondizione più debole p che presa come precondizione forma una tripla valida:

$$\vdash \{p\}\ \mathrm{C}\ \{q\}$$

se p è la precondizione più debole allora corrisponde al più grande insieme di stati tale che la tripla sia valida.
Questa è una scelta ragionevole perché è la precondizione che impone meno vincoli sullo stato iniziale, infatti determina tutti gli stati iniziali che garantiscono il raggiungimento della postcondizione. È sempre possibile calcolare la precondizione più debole.
Indichiamo, fissati C comando e q formula di postcondizione, con $wp(C, q)$ la precondizione più debole (**weakest precondition (*wp*)**).

Teorema 2 (proprietà fondamentale della condizione più debole)**.** *Si ha che* $\vDash \{p\}\ C\ \{q\}$ *(quindi la tripla è vera) sse:*

$$p \implies wp(C, q)$$

Teorema 3. *L'**esistenza** della precondizione più debole è garantita.*
*È garantita inoltre l'**unicità** della precondizione più debole (a meno di eventuali equivalenze logiche).*

Vediamo quindi la *regola di calcolo.*
Si hanno diversi casi a seconda dei tipi di comando:

- **assegnamento:** in questo caso la precondizione più debole è quella determinata dalla regola di derivazione introdotta per la correttezza parziale. Quindi dato un assegnamento del tipo $x := E$ e una postcondizione q ho che la precondizione più debole si ottiene sostituendo in q ogni occorrenza di x con l'espressione E, ottenendo quindi:
$$\{q[E/x]\}$$
- **sequenza:** in questo caso, se ho la sequenza di due comando C_1 e C_2 allora la precondizione più debole si calcola nel seguente modo: calcolo la precondizione più debole per C_2, ottenendo $wp(C_2, q)$, che sarà quindi la formula usata come postcondizione per calcolare la precondizione più debole di C_1, ovvero: $wp(C_1, wp(C_2, q))$. Questo non è altro che la condizione più debole della sequenza:
$$wp((C_1; C_2), q) \equiv wp(C_1, wp(C_2, q))$$

- **scelta:** in questo caso, avendo:

  $$S = \text{ if } B \text{ then } C \text{ else } D \text{ endif}$$

  fisso la postcondizione q e procedo come per la regola di derivazione. Separo i due casi in cui sia vera B o meno. Se B è vera devo eseguire C quindi calcolo, eseguendo C la precondizione più debole $wp(C, q)$. Qualora B non sia vera calcolo, eseguendo D, la precondizione più debole $wp(D, q)$. Nel complesso quindi, facendo la disgiunzione dei due casi, ottengo:

  $$wp(S, q) \equiv (B \wedge wp(C, q)) \vee (\neg B \wedge wp(D, q))$$

- **iterazione:** in questo caso, avendo:

  $$W = \text{while } B \text{ do } C \text{ endwhile}$$

  È il caso più complicato.
  Si ha che se $\neg B$ è nello stato iniziale dll'iterazione allora il corpo non viene eseguito (arrivando direttamente alla postcondizione), invece se vale B si avrà che W equivale a $C; W$, in quanto sicuramente almeno una volta eseguo C. Ma a questo punto potrei rifare lo stesso discorso se B è ancora vera (questo fino a che non ho $\neg B$, uscendo dal ciclo e arrivando a q), ragionando su W di $C; W$. Si arriva di fatto ad una regola ricorsiva:

  $$wp(W, q) \equiv (\neg B \wedge q) \vee (B \wedge wp((C; W), q))$$

  Applico quindi la regola della sequenza per la condizione più debole, arrivando a:

  $$wp(W, q) \equiv (\neg B \wedge q) \vee (B \wedge wp(C, wp(W, q)))$$

  Si nota che questa definizione ricorsiva ha un problema grave: non si ha un **caso base** che risolva la catena di chiamate ricorsive.
  Purtroppo non si può usare il "trucco" dell'invariante come nella regola di definizione e questo comporta che non si ha un vero e proprio algoritmo unico ed effettivo per questa regola di ricerca della precondizione più debole

Si hanno dei casi estremi, ad esempio:

- precondizioni più deboli sempre false, ad esempio:

$$wp(x := 5, x < 0) \equiv \bot$$

ovvero se assegno a x il valore 5 non avrò mai x negativo, ovvero ho un insieme vuoto $\emptyset$ di stati che garantiscono quanto detto

- precondizioni più deboli sempre vere, ad esempio

$$wp(x := 5, x \geq 0) \equiv \top$$

ovvero se assegno a x il valore 5 si avrà che x è sempre positivo, quindi è vero per qualunque stato iniziale

Esempi vari

Esempio 14. *Vediamo un primo esempio semplice con solo la sequenza di due assegnamenti:*

Listing 6 Programma P

```
x := x+1;
y := y*x;
```

Dove i due assegnamenti sono rispettivamente A e B e si vuole la postcondizione $q = x < y$. Cerco quindi $wp(P, q)$.
Procediamo con la formula della sequenza (che ricordiamo procede "a ritroso").

- *Parto dal secondo assegnamento, B, e calcolo $wp(B, q)$.*
  Procedo con la regola e applico la sostituzione, ottenendo:

$$r = wp(B, q) = x < y \cdot x$$

quindi se $x < y \cdot x$ vale prima dell'esecuzione di B allora sicuramente si raggiunge uno stato finale dove vale la postcondizione q

- *lo stato in cui eseguiamo B, che abbiamo sopra chiamato r, è quello prodotto dall'esecuzione di A. Dobbiamo quindi calcolare $wp(A, r)$.*
  Procedo quindi ancora con l'assegnamento, ottenendo:

$$wp(A, r) \equiv x + 1 < y(x + 1)$$

che quindi, per composizione, è anche la precondizione più debole per P e q. Bisogna però fare dei controlli.
Uso quindi le regole delle disequazioni:

- *caso 1:* $x + 1 > 0 \implies y > 1$
- *caso 2:* $x + 1 = 0 \implies 0 < 0$ *e quindi non vale* q
- *caso 3:* $x + 1 < 0 \implies y < 0$

Costruisco quindi la precondizione più debole prendendo la disgiunzione logica dei due casi validi:

$$wp(P, q) \equiv (x \geq 0 \land y > 1) \lor (x < -1 \land y < 1)$$

(sempre ricordando che lavoriamo con interi)

Esempio 15. *Vediamo un altro esempio:*

Listing 7 Programma P

```
x := x+a;
y := y-1;
```

Dove i due assegnamenti sono rispettivamente A e B e si vuole la postcondizione $q = (x = (b - y) \cdot a)$. Cerco quindi $wp(P, q)$.
Nella postcondizione ho una variabile b esterna alla sequenza di interesse fissato a priori.
Come prima ottengo:

$$r = wp(B, q) \equiv (x = (b - y + 1) \cdot a) = (x = (b - y) \cdot a)$$

e quindi:

$$wp(A, r) \equiv (x + a = (b - y) \cdot a + a)$$

quindi $x = (b - y) \cdot a$ è un invariante ed è anche la postcondizione, Quindi q è un invariante, accezione più forte del termine, per P.

Esempio 16. *Vediamo un altro esempio:*

Listing 8 Programma P

```
if y = 0 then
  x := 0;
else
  x := x * y;
endif
```

Dove i due comandi sono rispettivamente C e D e la condizione booleana è B. Come postcondizione voglio $q = (x = y)$.
Per il caso della scelta bisogna calcolare i due casi e usare poi la disgiunzione tra essi.
Calcolo quindi:

1. *il caso in cui vale B: $wp(C, q) = (0 = y)$, usando l'assegnamento*
2. *il caso in cui vale $\neg B$: $wp(D, q) = (x \cdot y) = y$, usando assegnamento. La formula però comporta che ho due casi possibili: $y = 0 \vee x = 1$ (il secondo appunto se ho $y \neq 0$)*

Applico quindi la regola della scelta, ottenendo:

$$wp(P, q) \equiv (y = 0 \wedge y = 0) \vee (y \neq 0 \wedge (y = 0 \vee x = 1))$$

Semplificando si ha che:

$$wp(P, q) \equiv (y = 0) \vee (x = 1 \wedge y \neq 0)$$

Esempio 17. *Vediamo un altro esempio:*

Listing 9 Programma P

```
while x > 0 do
  x := x - 1
endwhile
```

Con la condizione B e il comando C, nel corpo dell'iterazione W. Come postcondizione si vuole $q = (x = 0)$.
Seguendo le modalità discusse in merito alle tecniche relative al calcolo della precondizione più debole per l'iterazione, vediamo che:

$$(\neg B \wedge q) \equiv (x \leq 0 \wedge x = 0) \equiv (x = 0)$$

Per comodità chiamo $wp(W, q)$ r.
Dobbiamo ora studiare $B \wedge wp(C, wp(W, q))$ che quindi per noi è, per l'alias appena definito, $B \wedge wp(C, r)$:

$$B \wedge wp(C, r) \equiv x > 0 \wedge r[x - 1/x]$$

usando quindi l'assegnamento e quindi si ha che:

$$B \wedge wp(C, r) \equiv (x > 0) \wedge (x - 1 = 0 \vee (x - 1 > 0 \wedge r[x - 2/x]))$$

Siamo quindi in piena ricorsione.
Allo stato attuale ho quindi (usando ldots per non dover riscrivere tutto):

$$(\neg B \wedge q)(B \wedge wp(C, r)) \equiv (x = 0 \vee (x > 0 \wedge (\ldots)))$$

in realtà, andando avanti, si otterrebbe uno sviluppo regolare che porterebbe alla formula infinita:

$$(x = 0 \vee x = 1 \vee x = 2 \vee \ldots) \equiv x \geq 0$$

e quindi al precondizione più debole diventa:

$$wp(W, q) = (x \geq 0)$$

Si nota quindi come i casi di iterazione si risolvono solo con tecniche "non rigorose" ad hoc.

Estensioni del linguaggio

Abbiamo, in conclusione, sviluppato una tecnica di studio basata su un linguaggio di programmazione molto semplificato. Innanzitutto potremmo estendere il linguaggio usato, aggiungendo:

- il **do-while**:

  $$\text{do } C \text{ while } B \text{ endwhile}$$

  che nella realtà corrisponde a:

  $$C\text{; while } B \text{ do C endwhile}$$

- il **repeat-until**:

  $$\text{repeat } C \text{ until } B \text{ endrepeat}$$

  che nella realtà corrisponde a:

  $$C\text{; while not } B \text{ do } C \text{ endwhile}$$

- il **ciclo for**:

  $$\text{for}(D;\ B;\ F)\ C \text{ endfor}$$

  sempre convertendolo in un while (*ipotizzo così*):

  $$\text{while } B \text{ do } C;\ F \text{ endwhile}$$

- **procedure, metodi e funzioni**
- **array**, anche se solo in lettura se vogliamo applicare la logica di Hoare come l'abbiamo vista

Un altro limite è dato dal fatto che abbiamo usato solo il tipo intero, ma possiamo potenzialmente aggiungere anche gli altri senza cambiare le basi della logica introdotte.

4.2.4 Logica di Hoare per sviluppare programmi

Possiamo vedere le logiche di Hoare come **contratti** tra chi scrive il programma e l'utente. In questo contesto l'utente commissiona il programma specificando la postcondizione e chiede al programmatore di garantire che tale postcondizione venga garantita al termine dell'esecuzione. Per garantire la postcondizione q deve essere vera la precondizione p.
Vediamo un esempio.

Esempio 18. *Ci proponiamo di calcolare la radice quadrata intera di un certo $k \geq 0$, approssimando per difetto.*
Alla fine del programma voglio il risultato nella variabile x, quindi voglio:

$$\{k \geq 0\}\ P\ \{0 \leq x^2 \leq k < (x+1)^2)\}$$

con $(x+1)^2$ per la correttezza dell'approssimazione.
Ci si propone di fare il calcolo per approssimazioni successive, partendo da un valore più piccolo di quello corretto.
Possiamo spaccare q in:

$$0 \leq x^2 \leq k \quad e \quad k < (x+1)^2$$

la prima possiamo considerarla come invariante (anche solo $x^2 \leq k$).

All'inizio possiamo pensare che x dipenda da k per una certa funzione E. Si ha quindi un prototipo del genere (con un certo B, condizione booleana, e un certo F, incremento nel corpo, dipendenti da x e k):

Listing 10 Programma P

```
x := E(x);
while B(x,k) do
  x := F(x,k);
endwhile
```

Alla fine deve valere $x^2 \leq k \wedge \neg B$ per la regola dell'iterazione.
Come B posso porre $(x+1)^2 \leq k$ (ovvero la negazione della seconda parte della postcondizione, in modo da ottenere, con la negazione, q).
Si ottiene quindi:

Listing 11 Programma P

```
x := 0;
while (x+1)^2 <= k do
  x := x+1;
endwhile
```

Bisognerebbe comunque dimostrare invariante e terminazione per validare il programma.

Supponiamo un programma del tipo:

$$\{p\} \text{ A;W;C } \{q\}$$

che si risolve schematicamente con:

$$\{p\} \text{ A } \{inv\} \text{ W } \{inv \wedge \neg B\} \text{ C } \{q\}$$

Questo schema usa i cosiddetti **invarianti costruttivi**.

4.2.5 Ultime considerazioni

Vediamo qualche ultima considerazione sulla logica di Hoare.

- alcune applicazioni pratiche della logica di Hoare:
  - *java modelling language*, un linguaggio di specificazione scritto in Java, che permette di fare *design by contract* stabilendo delle precondizioni e delle postcondizioni
  - *Eiffel, programming by contract*
  - *assert.h* in C
- alcune applicazioni teoriche della logica di Hoare:
  - *semantica del programmi*, ovvero lo studio di cosa significa un programma, tramite:
    - *semantiche operazionali*, dove al programma viene associata una macchina astratta e, dato un programma viene associata una computazione sulla macchina astratta (come ad esempio la **Macchina di Turing** o le **Macchine a Registri**)
    - *semantiche assiomatiche*, dove vediamo, mano a mano che viene eseguito il programma, le asserzioni che vengono verificate
    - *semantiche denotazionali*, in cui un programma è visto come un trasformatore da input e output. Come una $f : I \to O$. SI basa sul **lambda calcolo**

- *semantiche operazionali strutturate*

Si hanno quindi i due punti chiave che devono essere garantiti:

1. **terminazione** del programma
2. **composizionalità** tra più comandi per ottenere un programma

Le triple di Hoare vivono ancora nella **programmazione by contract**.

Capitolo 5

Calculus of Communicating Systems

Fino ad ora abbiamo considerato programmi sequenziali e abbiamo studiato la loro verifica, tramite le triple di Hoare. Si passa ora dal sequenziale al **concorrente**. Dati due comandi s_1 ed s_2 si ha che essi sono specificati in esecuzione concorrente con:

$$s_1|s_2$$

L'esecuzione in concorrenza può portare a diverse complicanze qualora non venga rispettato, per esempio, un certo ordine di esecuzione. Si ha quindi il **non determinismo**, potendo avere più risultati a seconda dell'ordine di esecuzione. Si perde la **composizionalità**.

Esempio 19. *Vediamo un esempio.*
Siano:

$$s_1 = \{x = V\}\ \ x = 2\ \ \{x = 2\}$$

$$s_2 = \{x = V\}\ \ x = 3\ \ \{x = 3\}$$

con V indicante un valore qualunque.
Posso avere:

$$\{x = V\}\ \ s_1|s_2\ \ \{x = 2 \vee x = 3\}$$

avendo non determinismo.
Definiamo anche:

$$s_1' = \{x = V_0\}\ \ x = 1; x = x + 1\ \ \{x = 2\}$$

e vediamo, a conferma della perdita di composizionalità e determinismo che:

$$\{x = V\}\ \ s_1'|s_2\ \ \{x = 2 \vee x = 3 \vee x = 4\}$$

Si studierà quindi la **semantica della programmazione concorrente/parallela (o dei sistemi distribuiti)**.
Il problema è stato studiato da Hoare e Milner (secondo punti di vista diversi) alla fine degli anni '70. Hoare ha introdotto un nuovo paradigma di programmazione, il paradigma **CSP (*communicating sequential processes*)**, il linguaggio macchina dei cosiddetti *transputer*. Non si ha più una memoria condivisa ma un insieme di processi ciascuno con una sua memoria privata. Si ha un'interazione tra processi tramite lo scambio di messaggi del tipo *hand-shacking*, avendo quindi la sincronizzazione, con lo scambio di informazioni. Viene fatto anche un processo particolare rappresentante la **memoria condivisa**. Avremo quindi:

$$x|s_1|s_2$$

con x che rappresenta la memoria condivisa dai due processi.
Ad oggi diversi linguaggi di programmazione adottano una "soluzione mista" rispetto a questo paradigma.
Milner propose in modo completamente indipendente da Hoare qualcosa di molto analogo. Milner propose infatti il **lambda calcolo (λ-*calcolo*)** per passare dal sequenziale al concorrente. Studia in modo approfondito la composizionalità, sfruttando la composizione tra funzioni, cercando di non perderla nel concorrente. Introduce quindi una sorta di λ-*calcolo concorrente*, introducendo il **Calculus of Communicating Systems (*CCS*)**, in cui pensa ad un calcolo algebrico per sistemi comunicanti. Adotta anche lui un paradigma che studia un sistema (non solo dal punto di vista della programmazione) formato da componenti, chiamati *processi*. Questi processi comunicano tramite lo scambio sincrono di messaggi, con il modello *hand-shacking*. Con a senza nulla indiciamo un processo generico (a potrebbe essere qualsiasi cosa) che invia e con $\overline{a}$ un processo generico che riceve. Un sistema quindi è un insieme di processi il cui comportamento è gestito da un calcolo algebrico, si punta alle **algebre di processi**, ovvero linguaggi di specifica di sistemi concorrenti che si ispirano al calcolo dei sistemi comunicanti. I messaggi di scambio corrispondono ad uno scambio di valori di variabili e questo è rappresentabile dall'algebra. I processi possono interagire anche con l'**ambiente esterno**. Dato un sistema S, si scrive:

$$S = p_1|p_2|p_3$$

se S è formato dai processi p_1, p_2 e p_3, processi che sono *interagenti* "a due a due". Ogni processo ha comunque una memoria privata.
Grazie alla comunicazione con l'*ambiente esterno* non si ha più un **sistema chiuso**. Pnueli (che creò anche le logiche temporali) e Harvel hanno definito

questi sistemi **sistemi reattivi**, per indicare che i sistemi reagiscono all'ambiente esterno.
Milner risolve il problema della *composizionalità* tramite l'uso di diverse *porte* che permettono ad un processo di comunicare con altri o con l'ambiente esterno. Quindi ogni processo può essere visto come un insieme di sottoprocessi *interagenti* che però interagiscono tramite sincronizzazione con i processi esterni tramite una porta (un sottoprocesso può comunicare con più processi esterni tramite più porte). Bisogna comunque mantenere il comportamento complessivo. Per il processo esterno è come se sostituissi il processo con cui comunica con il suo sottoprocesso. Si introduce infatti l'**equivalenza all'osservazione**, che permette di sostituire un processo s_i con s_i' se sono equivalenti rispetto all'*osservazione* ovvero sse un qualsiasi osservatore esterno non è in grado di distinguere i due processi. In questo caso *osservare* significa **interagire con il sistema** dove agisce il processo (questa è un'idea di "osservare" derivante dalla fisica moderna). Questo deve essere valido **per ogni possibile osservatore**. Se questo è garantito la sostituzione di un processo non va ad inficiare l'esecuzione complessiva, senza incorrere in *deadlock* o altre problematiche.

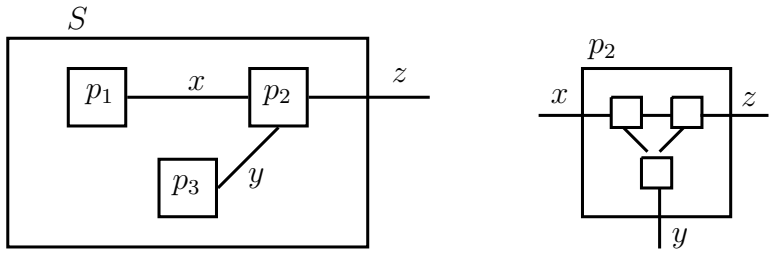

Figura 5.1: Rappresentazione stilizzata di un sistema S e del suo sottoprocesso p_2

Vedremo quindi:

- il calcolo "puro" di sistemi comunicanti CCS, definendone la semantica attraverso **Labeled Transition System,*LTS*** (*sistemi di transizione etichettati*), (avendo come nodi i processi e archi etichettati dalle azioni). Useremo le operazioni "+", "$\cdot$" e la ricorsione, che verranno definite tramite LTS
- l'equivalenza all'osservazione e la **bisimulazione**. Vedremo anche una tecnica di verifica della *bisimulazione*, tramite la **teoria dei giochi**