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 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.

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

4.2.1 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

4.2.2 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:

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

4.2.3 Sequenza

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

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

4.2.4 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:*

$$\frac{}{\{p[E/x]\}\ x := E\ \{p\}}$$

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 $x := y + 1$ 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:*

$$\{y + 1 > 0\}\ x := y + 1\ \{x > 0\}$$

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 $x := x+1$ con E pari a $x+2$ e con p pari a $\{x > 0 \wedge x \leq y\}$ come postcondizione data e cerco la precondizione. Quindi la tripla completa sarebbe:*

$$\{x + 2 > 0 \wedge x + 2 \leq y\}\ x := x + 1\ \{x > 0 \wedge x \leq y\}$$

e anche questa tripla è garantita dalla regola di sostituzione

Per proseguire bisogna definire il concetto di **invariante**.

Definizione 16. *La quinta regola è quella relativa all'iterazione*