

# Analisi statica e verifica del software

Simone Colli, Manuel Di Agostino

`simone.colli@studenti.unipr.it`, `manuel.diagostino@studenti.unipr.it`

Appunti del corso tenuto dal **Prof. Vincenzo Arceri**

Università degli Studi di Parma

Anno Accademico 2025/2026

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Cos'è l'affidabilità del software?	3
1.2	Perché l'affidabilità del software è importante?	3
1.3	Validazione e Verifica	4
1.4	Verifica del Software	4
<b>2</b>	<b>Background matematico</b>	<b>7</b>
2.1	Set notation	7
2.2	Partial order	7
2.3	Powerset	8
2.4	Diagramma di Hasse	9
2.5	Upper e lower bounds	10
2.6	Supremum e Infimum	11
2.7	Proprietà di lub e glb	11
2.8	Lattice	11
2.9	Set lattice	12
2.10	Complete lattice	13
2.11	Relations	14
2.12	Functions (or maps)	15
2.12.1	Monotonia, Embedding e Isomorfismo	16
2.12.2	Funzioni che preservano join e meet	17
2.13	Chains	18
2.13.1	Ascending Chain Condition (ACC)	19
2.14	Mappe continue	19
2.15	Fixpoint	20
2.15.1	Iterazioni di una funzione	20
2.16	Teoremi di Punto Fisso	20
<b>3</b>	<b>Modellazione dei programmi</b>	<b>23</b>
3.1	Poset nell'analisi statica	23
3.2	Punti fissi	24
3.3	IMP	25
3.3.1	Semantica concreta	25
3.4	Semantica delle tracce	25

# 1 Introduzione

## 1.1 Cos'è l'affidabilità del software?

### Definizione 1.1: Affidabilità del Software (IEEE 610.121990)

"The ability of a system or component to perform its required functions under stated conditions for a specified period of time"

Ovvero:

"La capacità di un sistema o componente di eseguire le funzioni richieste in condizioni specificate per un periodo di tempo specificato"

### Definizione 1.2: Gestione dell'Affidabilità del Software (IEEE 982.11988)

"The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources, schedule and performance"

Ovvero:

"Il processo di ottimizzazione dell'affidabilità del software attraverso un programma che enfatizza la prevenzione degli errori software, il rilevamento e la rimozione dei guasti, e l'uso di misurazioni per massimizzare l'affidabilità alla luce dei vincoli del progetto come risorse, tempi e prestazioni"

Sfruttando le definizioni Definizione 1.1 e Definizione 1.2, è possibile riassumere che l'affidabilità del software consiste in 3 attività:

- Prevenzione degli errori.
- Rilevamento e rimozione dei guasti.
- Valutazione per massimizzare l'affidabilità, in particolare valutazioni a supporto delle prime due attività.

## 1.2 Perché l'affidabilità del software è importante?

L'importanza dell'affidabilità del software deriva dalle enormi conseguenze economiche e operative che i bug possono causare.

Alcune stime indicano che:

- I bug nei software costano circa 60 miliardi di dollari all'anno negli Stati Uniti.
- L'economia mondiale perde circa 250 miliardi di dollari all'anno a causa di qualsiasi tipo di attacco palese (overt attack).
- I difetti nel software rendono la programmazione molto dolorosa.

Alcuni esempi di fallimenti dovuti a bug software includono:

- Il disastro del volo 501 di Ariane 5 nel 1996, che è esploso dopo 37 secondi dal decollo a causa di un errore di overflow durante la conversione da un float 64 bit ad un intero 16 bit. Questo evento ha causato una perdita di circa 370.000.000 dollari.
- Il bug del Pentium FDIV nel 1994, che ha causato errori di calcolo nelle divisioni in virgola mobile. Questo bug ha portato a una perdita di circa 475.000.000 dollari.
- L'aggiornamento difettoso di CrowdStrike nel 2024, che ha causato più di 5000 voli cancellati. Questo evento ha impattato su banche, governi e infrastrutture critiche. Le perdite economiche sono state stimate a circa 5.400.000.000 dollari.

La lista di esempi potrebbe continuare, ma l'importante è comprendere che l'affidabilità del software è cruciale per evitare perdite economiche significative e garantire il corretto funzionamento dei sistemi.

### 1.3 Validazione e Verifica

La validazione e la verifica sono due processi distinti ma complementari nell'ambito dello sviluppo del software, entrambi mirati a garantire che il software soddisfi i requisiti e funzioni correttamente.

La **validazione** si concentra sulla domanda: "Stiamo costruendo il prodotto giusto?". Essa verifica che il software soddisfi le esigenze e le aspettative degli utenti finali. La **verifica**, d'altra parte, si concentra sulla domanda: "Stiamo costruendo il prodotto nel modo giusto?". Essa assicura che il software sia sviluppato correttamente secondo le specifiche tecniche e i requisiti di progetto.

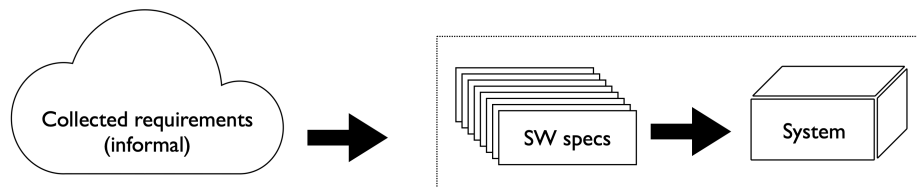


Figura 1: L'immagine illustra la differenza tra validazione e verifica.

#### Esempio 1.1: Risposta dell'ascensore

Un esempio pratico di validazione e verifica può essere trovato nel comportamento di un sistema di risposta dell'ascensore.

Una specifica validabile ma non verificabile potrebbe essere: "Se un utente preme il pulsante di richiesta a un piano  $i$ , un ascensore disponibile deve arrivare al piano  $i$  entro un tempo ragionevole".

Una specifica verificabile è: "Se un utente preme il pulsante di richiesta a un piano  $i$ , un ascensore disponibile deve arrivare al piano  $i$  entro 30 minuti".

### 1.4 Verifica del Software

Lo scopo della verifica del software è verificare alcune affermazioni di correttezza riguardante un programma:

- La **correttezza funzionale**, ovvero che il programma soddisfi i requisiti funzionali specificati.
- L'**assenza di errori non funzionali** (non cosa fa il programma, ma come lo fa), come ad esempio: soddisfacimento di vincoli spaziali e/o temporali, assenza di errori a runtime, soddisfacimento di vincoli di sicurezza, ecc.

## Bad news: theres no silver bullet

Non esiste un metodo di verifica che sia contemporaneamente:

- **Automatico**: non richiede interazione umana.
- **Potente**: è in grado di dimostrare proprietà non banali (*non-trivial properties*).
- **Corretto (Sound)**: non dimostra mai una proprietà se questa non è vera.
- **Completo (Complete)**: dimostra sempre la proprietà se essa è vera e non fallisce mai nel dichiarare la correttezza di un programma corretto.

Il **teorema di Rice** afferma che tutte le proprietà non banali del comportamento di un programma scritto in un linguaggio di programmazione Turing-completo sono indecidibili. L'**ind decidibilità** di una proprietà implica che non esiste alcun metodo di verifica automatico che sia allo stesso tempo **corretto** e **completo**.

## 2 Background matematico

Per poter discutere in modo rigoroso di tecniche di verifica del software, è necessario introdurre alcuni concetti matematici di base relativi alla teoria della computazione e alla logica formale.

### 2.1 Set notation

#### Definizione 2.1: Insieme

Un insieme (set) è una collezione di oggetti ben definiti e distinti. La collezione stessa è considerata un oggetto a sé stante.

Dato un insieme  $S$  è possibile dire che:

- $s \in S$  significa che l'elemento  $s$  appartiene all'insieme  $S$ .
- $S_1 \subseteq S_2 \triangleq \forall s \in S_1 \Rightarrow s \in S_2$  significa che l'insieme  $S_1$  è un sottoinsieme di  $S_2$  se ogni elemento di  $S_1$  appartiene anche a  $S_2$ .
- $S_1 \cup S_2 = \{s \mid s \in S_1 \vee s \in S_2\}$  è l'unione di due insiemi, ovvero l'insieme di tutti gli elementi che appartengono ad almeno uno dei due insiemi.
- $S_1 \cap S_2 = \{s \mid s \in S_1 \wedge s \in S_2\}$  è l'intersezione di due insiemi, ovvero l'insieme di tutti gli elementi che appartengono ad entrambi gli insiemi.

### 2.2 Partial order

#### Definizione 2.2: Ordine parziale

Un ordine parziale (partial order) è una relazione binaria  $\sqsubseteq$  su un insieme  $X$  che soddisfa le seguenti proprietà:

- Riflessività:  $\forall x \in X \Rightarrow x \sqsubseteq x$ , indica che ogni elemento è in relazione con se stesso.
- Anti-simmetria:  $\forall x, y \in X, (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$ , indica che se un elemento  $x$  è in relazione con un altro elemento  $y$  e viceversa, allora  $x$  e  $y$  sono lo stesso elemento.
- Transitività:  $\forall x, y, z \in X, (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$ , indica che se un elemento  $x$  è in relazione con un elemento  $y$ , e  $y$  è in relazione con un elemento  $z$ , allora  $x$  è in relazione con  $z$ .

#### Esempio 2.1: Esempio informale di ordine parziale su $\mathbb{Z}$

Informalmente è possibile considerare l'insieme dei numeri interi  $\mathbb{Z}$  con la relazione "minore o uguale di" ( $\leq$ ) come un esempio di ordine parziale. In questo caso è possibile verificare che:

- Riflessività: Per ogni numero intero  $i$ , vale che  $i \leq i$ .
- Anti-simmetria: Se  $i_1 \leq i_2$  e  $i_2 \leq i_1$ , allora  $i_1 = i_2$ .

- **Transitività:** Se  $i_1 \leq i_2$  e  $i_2 \leq i_3$ , allora  $i_1 \leq i_3$ .

## 2.3 Powerset

### Definizione 2.3: Insieme delle parti

Dato un insieme  $S$ , definito in Definizione 2.1, l'insieme delle parti (powerset) di  $S$  è l'insieme di tutti i sottoinsiemi di  $S$ , ed è indicato con  $\mathcal{P}(S)$ .

Dato un insieme  $S$  con  $|S| = n$  elementi è possibile determinare che l'insieme delle parti di  $S$  ha  $|\mathcal{P}(S)| = 2^n$  elementi.

### Esempio 2.2: Alcuni esempi di powerset

Dato l'insieme vuoto  $\emptyset$ , il suo powerset è:

$$\mathcal{P}(\emptyset) = \{\emptyset\}$$

ed ha  $2^0 = 1$  elemento.

Dato un insieme  $X = \{a, b\}$ , il suo powerset è:

$$\mathcal{P}(X) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$$

ed ha  $2^2 = 4$  elementi.

Dato l'insieme  $\mathbb{Z}$  dei numeri interi, il suo powerset è:

$$\mathcal{P}(\mathbb{Z}) = \{\emptyset, \{\dots, -2, -1, 0, 1, 2, \dots\}, \{0\}, \{1, 2, 3, \dots\}, \dots\}$$

ed ha un numero infinito di elementi.

Dato un powerset è possibile definire un ordine parziale  $\subseteq$  su di esso e verificare che esso soddisfa le proprietà di un ordine parziale:

- **Riflessività:**  $\forall X_1 \in \mathcal{P}(X). X_1 \subseteq X_1$
- **Anti-simmetria:**  $\forall X_1, X_2 \in \mathcal{P}(X). X_1 \subseteq X_2 \wedge X_2 \subseteq X_1 \Rightarrow X_1 = X_2$
- **Transitività:**  $\forall X_1, X_2, X_3 \in \mathcal{P}(X). X_1 \subseteq X_2 \wedge X_2 \subseteq X_3 \Rightarrow X_1 \subseteq X_3$

### Esercizio 2.1: Inclusione inversa su powerset

L'inclusione inversa su un powerset  $\mathcal{P}(X)$  è un ordine parziale? Per verificare se  $\supseteq$  è un ordine parziale su  $\mathcal{P}(X)$ , è necessario verificare se soddisfa le tre proprietà di un ordine parziale.

- **Riflessività:**  $\forall X_1 \in \mathcal{P}(X) \Rightarrow X_1 \supseteq X_1$ , indica che ogni insieme è sovrainsieme di se stesso.
- **Anti-simmetria:**  $\forall X_1, X_2 \in \mathcal{P}(X), (X_1 \supseteq X_2 \wedge X_2 \supseteq X_1) \Rightarrow X_1 = X_2$ , indica che se ogni elemento di  $X_1$  è anche in  $X_2$  e che ogni elemento di  $X_2$  è anche in  $X_1$ , allora  $X_1$  e



$X_2$  sono lo stesso insieme.

- **Transitività:**  $\forall X_1, X_2, X_3 \in \mathcal{P}(X), (X_1 \supseteq X_2 \wedge X_2 \supseteq X_3) \Rightarrow X_1 \supseteq X_3$ , indica che se  $X_1$  è un sovrainsieme di  $X_2$ , e  $X_2$  è un sovrainsieme di  $X_3$ , allora  $X_1$  è un sovrainsieme di  $X_3$ .

Poiché tutte e tre le proprietà sono soddisfatte, è possibile concludere che l'inclusione inversa  $\supseteq$  è un ordine parziale su il powerset  $\mathcal{P}(X)$ . Quindi  $\langle X, \supseteq \rangle$  è un poset.

## 2.4 Diagramma di Hasse

Il diagramma di Hasse è una rappresentazione grafica di un poset.

Sia  $X = \{x, y, z\}$  un insieme (Definizione 2.1), ed un poset  $\langle X, \sqsubseteq \rangle$  su di esso è possibile rappresentarlo tramite un diagramma di Hasse dove una linea che connette  $x$  e  $y$  indica che:

- $x \sqsubseteq y$
- $\nexists z \in X$  tale che  $x \sqsubseteq z \sqsubseteq y$

È possibile trovare la presenza di più livelli nel diagramma di Hasse, dove un elemento  $x$  è posizionato più in alto di un elemento  $y$  se  $x$  è “immediatamente maggiore” di  $y$ .

### Nota 2.1: Poset inverso

Un poset inverso è un poset ottenuto invertendo la relazione di ordine di un poset originale. Graficamente, questo si traduce nel riflettere il diagramma di Hasse rispetto ad un asse orizzontale, ovvero ruotando di 180 gradi il diagramma.

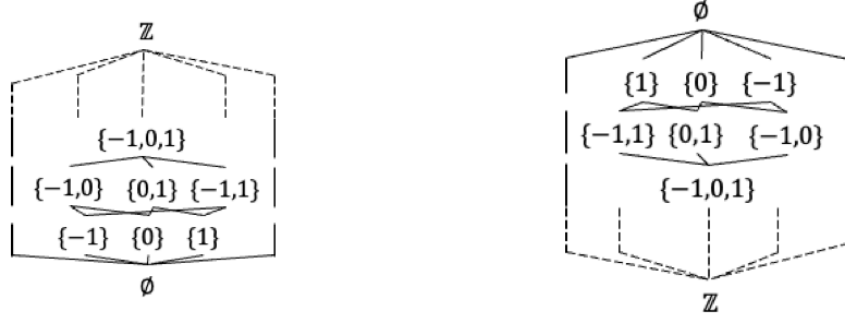


Figura 2: L'immagine illustra un esempio di diagramma di Hasse per un poset e il suo poset inverso.

## 2.5 Upper e lower bounds

### Definizione 2.4: Upper bound and least upper bound

Dato un poset  $\langle X, \sqsubseteq \rangle$  e un sottoinsieme  $Y \subseteq X$  è possibile dire che:

1.  $y \in X$  è un upperbound di  $Y$  se  $\forall y' \in Y, y' \sqsubseteq y$ , ovvero se ogni elemento dell'insieme  $Y$  è minore o uguale a  $y$ .
2.  $y$  è l'upperbound più piccolo tra tutti gli upperbound di  $Y$ , ovvero  $\forall y' \in UB(Y). y \sqsubseteq y'$ , dove  $UB(Y)$  è l'insieme di tutti gli upperbound di  $Y$ .

### Definizione 2.5: Lower bound e greatest lower bound

Dato un poset  $\langle X, \sqsubseteq \rangle$  e un sottoinsieme  $Y \subseteq X$  è possibile dire che:

1.  $y \in X$  è un lowerbound di  $Y$  se  $\forall y' \in Y, y \sqsubseteq y'$ , ovvero se ogni elemento dell'insieme  $Y$  è maggiore o uguale a  $y$ .
2.  $y$  è il lowerbound più grande tra tutti i lowerbound di  $Y$ , ovvero  $\forall y' \in LB(Y). y' \sqsubseteq y$ , dove  $LB(Y)$  è l'insieme di tutti i lowerbound di  $Y$ .

### Esercizio 2.2: Lub e glb su powerset

Dato il Poset  $\langle \mathcal{P}(X), \subseteq \rangle$ , e siano  $S_1, S_2 \in \mathcal{P}(X)$ :

1.  $S_1 \cup S_2$  è l'estremo superiore (lub) di  $\{S_1, S_2\}$ ?
2.  $S_1 \cap S_2$  è l'estremo inferiore (glb) di  $\{S_1, S_2\}$ ?

#### Soluzione esercizio 1

$S_1 \cup S_2$  è l'estremo superiore (lub) di  $\{S_1, S_2\}$ ?

Sia  $L = S_1 \cup S_2$ , che per essere lub deve soddisfare le condizioni elencate in Definizione 2.4 punto 1 e 2.

Punto 1:

$S_1 \subseteq L$  e  $S_2 \subseteq L$ .

Punto 2:

Sia  $U$  un qualsiasi insieme tale che  $S_1 \subseteq U$  e  $S_2 \subseteq U$ . Allora  $U$  deve per contenere gli elementi di  $S_1$  e  $S_2$  deve essere proprio  $(S_1 \cup S_2) \subseteq U$ .

#### Soluzione esercizio 2

$S_1 \cap S_2$  è l'estremo inferiore (glb) di  $\{S_1, S_2\}$ ?

## 2.6 Supremum e Infimum

### Definizione 2.6: Supremum

Dato un poset  $\langle X, \sqsubseteq \rangle$  e un sottoinsieme  $S \subseteq X$  il supremum (o least upper bound) di  $S$  è l'elemento più piccolo tra tutti gli upper bound di  $S$ .

### Definizione 2.7: Infimum

Dato un poset  $\langle X, \sqsubseteq \rangle$  e un sottoinsieme  $S \subseteq X$  l'infimum (o greatest lower bound) di  $S$  è l'elemento più grande tra tutti i lower bound di  $S$ .

## 2.7 Proprietà di lub e glb

### Proposizione 2.1: Proprietà di lub e glb

Dato un poset  $\langle X, \sqsubseteq \rangle$  è possibile definire:

- $\sqcup$  come il lub su  $X$ .
- $\sqcap$  come il glb su  $X$ .
- se  $\sqcup$  e  $\sqcap$  esistono sono unici.
- $\sqcup X$  esiste se e solo se  $X$  ha un elemento  $\top$  ( $\sqcup X = \top$ ).
- $\sqcap X$  esiste se e solo se  $X$  ha un elemento  $\perp$  ( $\sqcap X = \perp$ ).

## 2.8 Lattice

### Definizione 2.8: Lattice

Dato un poset  $\langle X, \sqsubseteq \rangle$ , esso è un reticolo (lattice) se soddisfa entrambe le seguenti proprietà:

- $\forall x, y \in X, \exists x \sqcup y$ , ovvero per ogni coppia di elementi qualsiasi di  $X$  deve esistere il loro lub. Un poset che soddisfa questa proprietà è detto “join semi lattice”.
- $\forall x, y \in X, \exists x \sqcap y$ , ovvero per ogni coppia di elementi qualsiasi di  $X$  deve esistere il loro glb. Un poset che soddisfa questa proprietà è detto “meet semi lattice”.

In un reticolo è presente la relazione d'ordine parziale  $\sqsubseteq$  su due elementi  $x, y \in X$ , ovvero  $x \sqsubseteq y$  se e solo se:

- $x \sqcup y = y$
- $x \sqcap y = x$

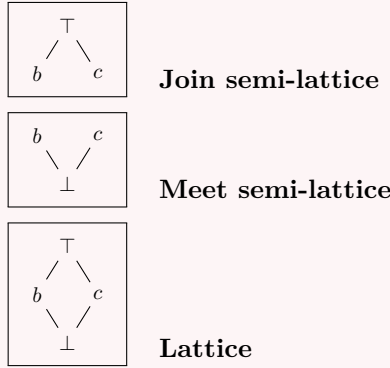


Figura 3: Rappresentazione grafica di Join semi-lattice, Meet semi-lattice e Lattice.

## 2.9 Set lattice

### Definizione 2.9: Set lattice

Le operazioni insiemistiche definiscono una struttura di reticolo. Dato il poset  $\langle \mathcal{P}(X), \subseteq \rangle$ , esso forma un reticolo rispetto alle operazioni di unione ( $\cup$ ) e intersezione ( $\cap$ ), denotato come:

$$\langle \mathcal{P}(X), \subseteq, \cup, \cap \rangle$$

Se l'insieme  $X$  è finito, allora il reticolo ha altezza finita.

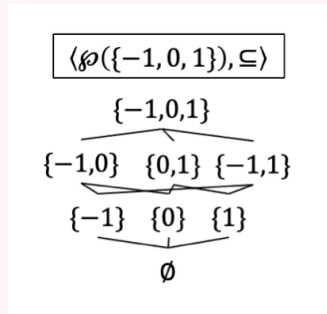


Figura 4: L'immagine illustra un esempio di reticolo su un poset finito.

### Nota 2.2: Top e Bottom nei reticoli

Un reticolo non possiede necessariamente un elemento Top ( $\top$ ) o un elemento Bottom ( $\perp$ ). Ad esempio, il reticolo  $\langle \mathbb{Z}, \leq, \max, \min \rangle$  non ha né un elemento massimo né un elemento minimo, poiché l'insieme dei numeri interi è illimitato in entrambe le direzioni.

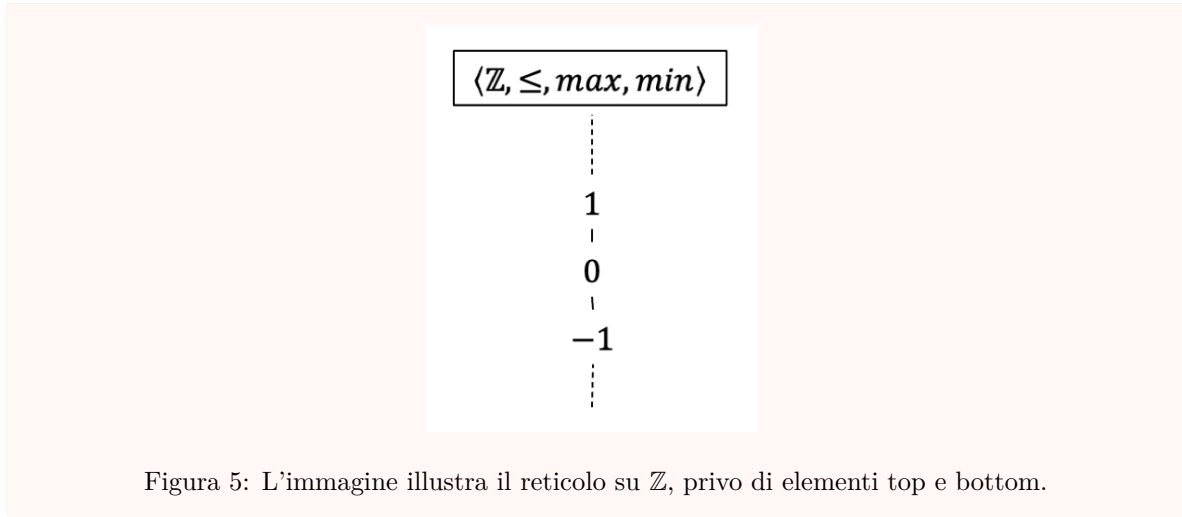


Figura 5: L'immagine illustra il reticolo su  $\mathbb{Z}$ , privo di elementi top e bottom.

## 2.10 Complete lattice

### Definizione 2.10: Complete lattice

Un reticolo  $\langle X, \sqsubseteq, \sqcup, \sqcap \rangle$  è detto completo se:

- Ogni sottoinsieme  $Y \subseteq X$  ha un lub in  $X$ .
- È presente un elemento bottom  $\perp$  in  $X$ .

### Esempio 2.3: Esempi di lattice completo e non completo

Considerando l'insieme dei numeri interi  $\mathbb{Z}$  con la relazione "minore o uguale di" ( $\leq$ ), si può osservare che

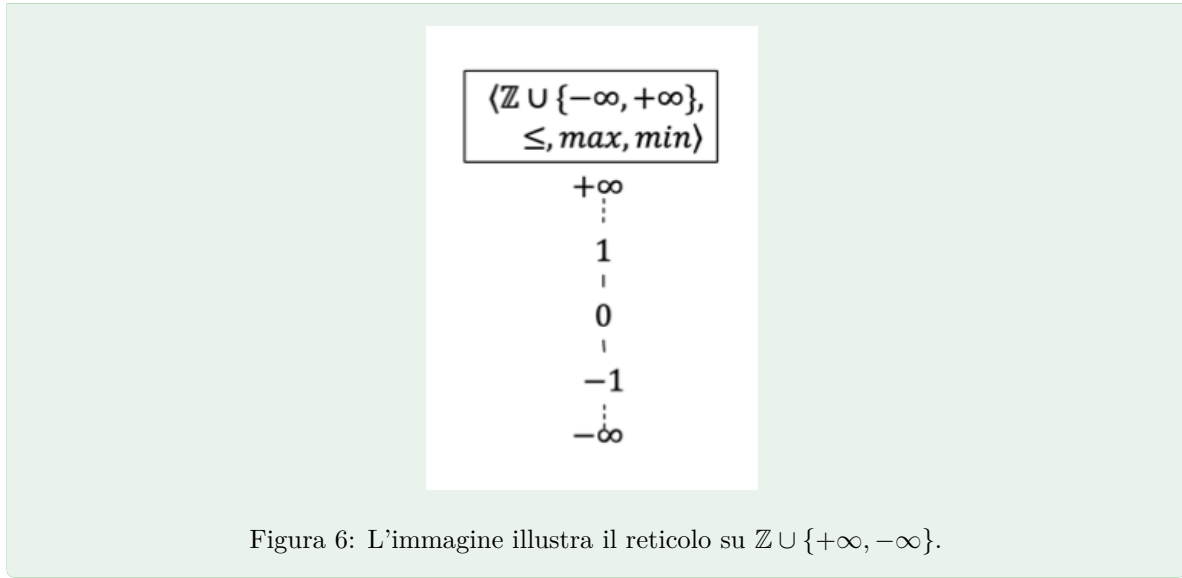
$$\langle \mathbb{Z}, \leq, \max, \min \rangle$$

è un reticolo ma non è completo. Infatti, non esiste un elemento top ( $\top$ ) in  $\mathbb{Z}$ , poiché non esiste un numero intero massimo.

Se all'insieme dei numeri interi si aggiungono gli elementi  $+\infty$  e  $-\infty$ , si ottiene l'insieme esteso  $\mathbb{Z} \cup \{+\infty, -\infty\}$ . In questo caso, il reticolo

$$\langle \mathbb{Z} \cup \{+\infty, -\infty\}, \leq, \max, \min \rangle$$

diventa un reticolo completo, poiché ora esistono sia un elemento top ( $+\infty$ ) che un elemento bottom ( $-\infty$ ).



#### Proposizione 2.2: Proprietà di un reticolo completo

Un reticolo completo possiede le seguenti proprietà:

- reticolo completo  $\langle X, \sqsubseteq, \sqcup, \sqcap \rangle$  ha un elemento top  $\top$  e un elemento bottom  $\perp$  e si denota come

$$\langle X, \sqsubseteq, \top, \perp, \sqcup, \sqcap \rangle$$

- I reticoli finiti sono completi.
- $\sqcup$  induce  $\sqcap : \sqcap S = \sqcup \{y \mid \forall x \in S. y \sqsubseteq x\}$
- $\sqcap$  induce  $\sqcup : \sqcup S = \sqcap \{y \mid \forall x \in S. x \sqsubseteq y\}$

## 2.11 Relations

#### Definizione 2.11: Prodotto cartesiano

Dati due insiemi (Definizione 2.1)  $X$  e  $Y$ , il loro prodotto cartesiano  $X \times Y$  è l'insieme di tutte le possibili coppie ordinate  $(x, y)$  dove il primo elemento appartiene a  $X$  e il secondo a  $Y$ .

$$X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$$

#### Definizione 2.12: Relazione

Una relazione  $R$  tra due insiemi  $X$  e  $Y$  è definita come un sottoinsieme del loro prodotto

cartesiano (Definizione 2.11):

$$R \subseteq X \times Y$$

### Nota 2.3: Notazione per le relazioni

Dato una relazione  $R \subseteq X \times Y$ , è possibile indicare che un elemento  $x \in X$  è in relazione con un elemento  $y \in Y$  tramite due notazioni:

- $(x, y) \in R$ , ovvero tramite relazione insiemistica.
- $x R y$ , ovvero tramite notazione infissa.

### Nota 2.4: Ordine parziale come relazione

Un ordine parziale  $\sqsubseteq$  (Definizione 2.2) è un caso particolare di relazione definita sullo stesso insieme  $X$ . Formalmente,  $\sqsubseteq$  è un sottoinsieme del prodotto cartesiano  $X \times X$  ( $\sqsubseteq \subseteq X \times X$ ). A differenza di una relazione generica, l'ordine parziale deve soddisfare le proprietà di riflessività, anti-simmetria e transitività (Definizione 2.2).

## 2.12 Functions (or maps)

Le funzioni (functions), sono un tipo particolare di relazione, e sono fondamentali per riuscire a modellare gli stati di un programma.

### Definizione 2.13: Funzione

Una funzione  $f$  tra due insiemi  $X$  e  $Y$  è un particolare tipo di relazione  $R \subseteq X \times Y$  che soddisfa la proprietà di unicità dell'immagine.

Formalmente, per ogni elemento  $x$  nel dominio, esiste al massimo un elemento  $y$  nel codominio tale che  $(x, y) \in f$ :

$$\forall (x, y) \in f, \nexists (x', y') \in f. (x = x' \wedge y \neq y')$$

Una funzione è definita al massimo una volta su un dato input.

### Esempio 2.4: Cerchio su un piano cartesiano

Un cerchio su un piano cartesiano non rappresenta una funzione, poiché un singolo valore di  $x$  può essere associato a due valori distinti di  $y$  (uno sopra l'asse  $x$  e uno sotto). Questo viola la proprietà di unicità dell'immagine definita per le funzioni.

### Nota 2.5: Non tutte le relazioni sono funzioni

Un ordine parziale (Definizione 2.2) non è generalmente una funzione, poiché un elemento può essere minore o uguale a molti altri elementi (uno-a-molti).

**Definizione 2.14: Notazione delle funzioni**

Dato  $f : X \rightarrow Y$ , dove  $X$  è il dominio e  $Y$  è il codominio:

- Notazione estensionale: Una funzione definita su punti specifici si scrive come:

$$f = [x_0 \mapsto y_0, x_1 \mapsto y_1, \dots, x_n \mapsto y_n]$$

che è equivalente all'insieme di coppie  $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ .

- Lambda notazione: Si utilizza  $\lambda x. espressione$  per definire una funzione anonima. Ad esempio  $\lambda x.f(x) \equiv f$ .

**2.12.1 Monotonia, Embedding e Isomorfismo****Definizione 2.15: Funzioni monotone, embedding ordinato e isomorfe**

Dati due posets  $\langle X, \sqsubseteq_X \rangle$  e  $\langle Y, \sqsubseteq_Y \rangle$ , e una funzione  $f : X \rightarrow Y$ , è possibile dire che:

- La funzione  $f$  è monotona se  $x_1 \sqsubseteq_X x_2 \Rightarrow f(x_1) \sqsubseteq_Y f(x_2)$ .
- La funzione  $f$  è un embedding ordinato se  $x_1 \sqsubseteq_X x_2 \iff f(x_1) \sqsubseteq_Y f(x_2)$ .
- La funzione  $f$  è un isomorfa se è un embedding ordinato ed è suriettiva, ovvero:  $\forall y \in Y, \exists x \in X. f(x) = y$ .

**Esempio 2.5: Esempio di funzione monotona, embedding ordinato e suriettiva**

Data  $f : \langle \mathbb{Z}, \leq \rangle \rightarrow \langle \mathbb{Z}, \leq \rangle$ , definita come  $f(x) = x + 1$ , è possibile osservare che:

- La funzione è monotona.
- La funzione è un embedding ordinato.
- La funzione è suriettiva.



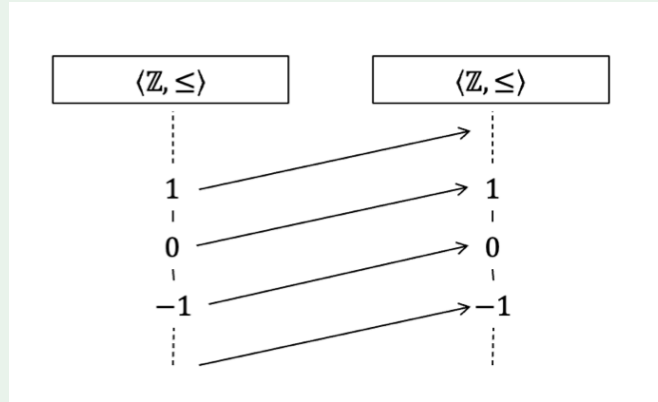


Figura 7: L'immagine illustra la funzione  $f(x) = x + 1$  come monotona, embedding ordinato e suriettiva.

#### Esempio 2.6: Esempio di funzione monotona ma che non è un embedding

Data  $f : \langle \mathcal{P}(\mathbb{Z}), \subseteq \rangle \rightarrow \langle \mathbb{Z}, \leq \rangle$ , definita come  $f(X) = \max(X)$ , è possibile osservare che:

- La funzione è monotona.
- La funzione non è un embedding.

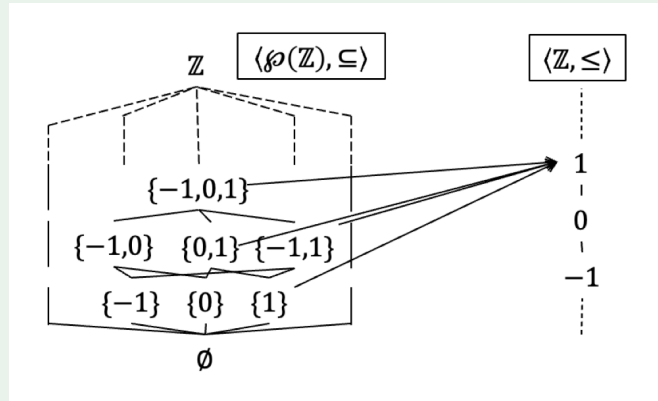


Figura 8: L'immagine illustra la funzione  $f(X) = \max(X)$  come monotona ma non un embedding.

#### 2.12.2 Funzioni che preservano join e meet

Dati due reticoli  $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X \rangle$  e  $\langle Y, \sqsubseteq_Y, \sqcup_Y, \sqcap_Y \rangle$ , e una funzione  $f : X \rightarrow Y$ , è possibile dire che:

- La funzione  $f$  preserva i join se  $f(x_1 \sqcup_X x_2) = f(x_1) \sqcup_Y f(x_2)$ .

- La funzione  $f$  preserva i meet se  $f(x_1 \sqcap_X x_2) = f(x_1) \sqcap_Y f(x_2)$ .

**Nota 2.6: Mantenimento della Monotonia**

Una funzione che preserva i join o i meet è monotona, ma una funzione che è monotona non necessariamente preserva i join o i meet.

## 2.13 Chains

**Definizione 2.16: Catena**

Dato un poset  $\langle X, \sqsubseteq \rangle$ , una catena (chain)  $S$  se tutti gli elementi contenuti sono completamente ordinati, ovvero:

$$\forall x, y \in S. x \sqsubseteq y \vee y \sqsubseteq x$$

**Esempio 2.7: Esempio di catena**

Dato un poset  $\langle \mathcal{P}(\mathbb{Z}), \subseteq \rangle$  con reticolo rappresentato nell'immagine sottostante (9), esempi di catene sono:

- $\{\emptyset\}$
- $\{\emptyset, \{0, 1\}\}$
- $\{\{0\}, \{-1, 0, 1\}\}$

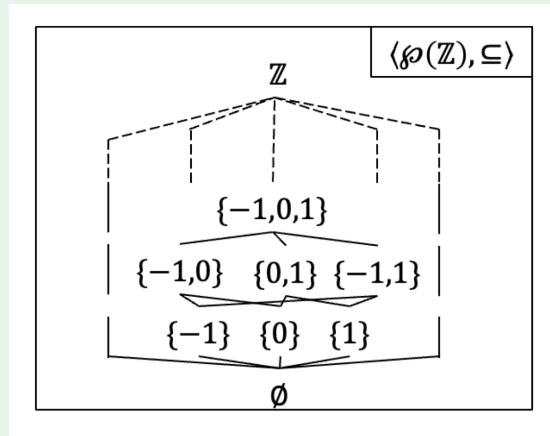


Figura 9: L'immagine illustra un reticolo su  $\mathcal{P}(\mathbb{Z})$ .

### 2.13.1 Ascending Chain Condition (ACC)

#### Definizione 2.17: Catena Ascendente

Una catena ascendente è una sequenza di elementi  $(l_n)_{n \in \mathbb{N}}$  indicizzata dai numeri naturali tali che:

$$i \leq j \implies l_i \sqsubseteq l_j$$

Ovvero  $l_0 \sqsubseteq l_1 \sqsubseteq l_2 \sqsubseteq \dots$

#### Definizione 2.18: Ascending Chain Condition (ACC)

Un poset soddisfa la Ascending Chain Condition (ACC) se ogni catena ascendente infinita si stabilizza (non è strettamente crescente per sempre).

Matematicamente, una catena  $(l_n)$  si stabilizza se:

$$\exists k \geq 0. \forall j \geq k. l_k = l_j$$

Questo significa che, dopo un certo indice  $k$ , tutti gli elementi successivi sono uguali a  $l_k$  (il punto fisso della sequenza).

#### Esempio 2.8: Esempio di catena ascendente infinita non è strettamente crescente

Un esempio di catena ascendente infinita non strettamente crescente è la seguente:

$$\exists k \geq 0. \forall j \geq k. l_k = l_j$$

Perché dopo un certo numero di passi  $k$ , tutti gli elementi successivi della catena sono uguali a  $l_k$ . Ovvero la catena si stabilizza.

## 2.14 Mappe continue

#### Definizione 2.19: Mappa continua

Dati due reticoli (o poset)  $\langle X, \sqsubseteq_X, \sqcup_X, \sqcap_X \rangle$  e  $\langle Y, \sqsubseteq_Y, \sqcup_Y, \sqcap_Y \rangle$  e una funzione  $f : X \rightarrow Y$  (Definizione 2.13), essa è detta continua se per ogni catena  $C \subseteq X$  (Definizione 2.16) tale che  $\sqcup_X C$  esiste, valgono le seguenti condizioni:

- $\sqcup_Y f(C)$  esiste;
- $f(\sqcup_X C) = \sqcup_Y f(C)$ .

#### Nota 2.7: Utilità delle mappe continue

Le mappe continue sono fondamentali nell'analisi statica perché permettono di applicare il Teorema di Kleene per calcolare i punti fissi (fixpoints) tramite iterazione.

## 2.15 Fixpoint

L'analisi statica riduce spesso il problema della verifica di un programma alla ricerca della soluzione di un sistema di equazioni ricorsive. La soluzione di tali equazioni corrisponde al concetto matematico di punto fisso.

### Definizione 2.20: Punto Fisso

Dato un insieme  $X$  e una funzione  $f : X \rightarrow X$ :

- Un elemento  $x \in X$  è un punto fisso di  $f$  se  $f(x) = x$ .
- L'insieme di tutti i punti fissi di  $f$  è denotato come  $Fix(f) = \{x \mid f(x) = x\}$ .

In un poset  $\langle X, \sqsubseteq \rangle$ , sono interessanti due tipi particolari di punti fissi:

- Least Fixpoint (lfp), ovvero il punto fisso più piccolo.

$$\forall y \in Fix(f). x \sqsubseteq y$$

- Greatest Fixpoint (gfp), ovvero il punto fisso più grande.

$$\forall y \in Fix(f). x \sqsupseteq y$$

### 2.15.1 Iterazioni di una funzione

Per calcolare un punto fisso, spesso si ricorre all'applicazione ripetuta della funzione.

### Definizione 2.21: Iterazioni

Le iterate di una funzione  $f : X \rightarrow X$  a partire da un elemento  $x$  sono definite per induzione come:

- $f^0(x) = x$
- $f^{n+1}(x) = f(f^n(x))$

### Nota 2.8: Stabilizzazione

Se l'insieme  $X$  è finito, la sequenza delle iterazioni  $f^0, f^1, \dots$  deve necessariamente stabilizzarsi (raggiungere un punto fisso o entrare in un ciclo) in al massimo  $|X|$  passi. Formalmente:

$$\forall k \geq |X|. \exists n < |X|. f^k(x) = f^n(x)$$

Se  $X$  è infinito, la sequenza potrebbe non terminare mai.

## 2.16 Teoremi di Punto Fisso

Esistono due teoremi principali che garantiscono l'esistenza (e talvolta la costruttibilità) dei punti fissi. La scelta del teorema dipende dalle proprietà della funzione  $f$  (Definizione 2.15)

**Teorema 2.1: Teorema di Knaster-Tarski**

Sia  $\langle X, \sqsubseteq, \sqcup, \sqcap \rangle$  un reticolo e sia  $f : X \rightarrow X$  una funzione monotona. Allora:

1.  $\langle Fix(f), \sqsubseteq, \sqcup, \sqcap \rangle$  è un reticolo completo.
2. Il least fixpoint (lfp) è dato dal meet di tutti i “post-fixpoints”, ovvero  $lfp(f) = \sqcap \{l \mid l \sqsupseteq f(l)\}$
3. Il greatest fixpoint (gfp) è dato dal join di tutti i “pre-fixpoints”, ovvero  $gfp(f) = \sqcup \{l \mid l \sqsubseteq f(l)\}$

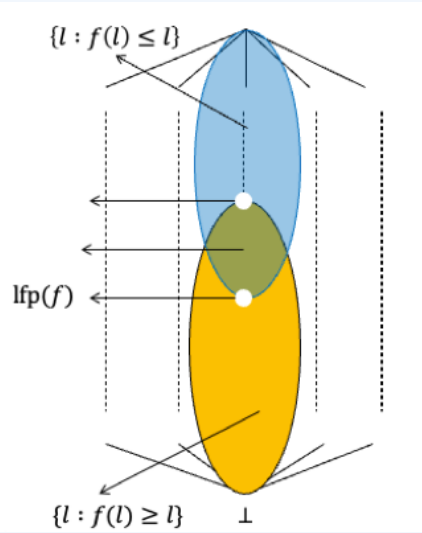


Figura 10: L'immagine illustra un esempio grafico del Teorema di Knaster-Tarski.

**Teorema 2.2: Teorema di Kleene**

Sia  $\langle X, \sqsubseteq, \perp, \top, \sqcup, \sqcap \rangle$  un reticolo completo e sia  $f : X \rightarrow X$  una funzione continua (Definizione 2.15). Allora  $f$  ha un least fixpoint che può essere calcolato costruttivamente come il limite della catena ascendente che parte dal bottom ( $\perp$ ):

$$lfp(f) = \bigsqcup_{n \geq 0} f^n(\perp)$$

**Nota 2.9: Confronto tra i Teoremi**

I due teoremi di punto fisso presentano differenze chiave:

- Knaster-Tarski: Richiede solo la monotonia. Questo teorema è più generale ma non fornisce un algoritmo costruttivo (dice solo “esiste ed è uguale all’intersezione di ...”).

- Kleene: Richiede la continuità (condizione più forte). Questo teorema fornisce un algoritmo iterativo  $(f(\perp), f(f(\perp)), \dots)$  per calcolare il risultato.

### 3 Modellazione dei programmi

L'obiettivo dell'analisi statica è certificare che un programma sia sicuro rispetto a determinati errori, come ad esempio la divisione per zero.

#### 3.1 Poset nell'analisi statica

L'idea fondamentale è calcolare i possibili valori che una variabile può assumere in ogni punto del programma, considerando tutte le possibili esecuzioni. Per fare questo è possibile utilizzare i reticoli (Definizione Definizione 2.8).

##### Esempio 3.1: Esempio di reticolo su un programma

Considerando il programma seguente:

```
1: i = read ();
2 if ( i != 0 )
3   j = 5 / i ;
4 else
5   j = 0;
6 return;
```

e definendo il reticolo dei valori interi come  $\langle \mathcal{P}(\mathbb{Z}), \subseteq, \cup, \cap \rangle$  è possibile calcolare i possibili valori delle variabili  $i$  e  $j$  in ogni punto del programma.

Program point	Valori
1	$i = \{\}, j = \{\}$
2	$i = \mathbb{Z}, j = \{\}$
3	$i = \mathbb{Z} \setminus \{0\}, j = \{5/i\}$
5	$i = \{0\}, j = \{0\}$
6	$i = \mathbb{Z}, j = \{-5, -2, -1, 0, 1, 2, 5\}$

Per ottenere i valori di  $j$  al punto 6, è stato necessario calcolare l'unione dei valori di  $j$  provenienti dai punti 3 e 5, utilizzando l'operazione di least upper bound (operazione di join, unione insiemistica in questo caso).

Per ottenere i valori di  $i$  al punto 2, è stato necessario considerare che la funzione `read()` può restituire qualsiasi valore intero. Dopo di che tramite un'operazione di "filtro"  $filter(i \neq 0)$  è possibile restringere i valori di  $i$  per entrare nel ramo vero dell'if. Per il calcolo del punto 3, invece è stato necessario applicare l'operazione di intersezione tra i valori di  $i$  al punto 2 e il filtro  $filter(x \neq 0)$  per ottenere i valori di  $i$  validi per quel ramo.

Ipotizzando che la funzione `read()` possa restituire valori interi in  $\{-1, 0, 1\}$ , è possibile riscrivere il codice del programma di esempio come:

```
1: i = {-1, 0, 1} ;
2 if ( i != 0 )
```

```

3  j = 5 / i ;
4  else
5  j = 0;
6  return;

```

Di conseguenza i valori delle variabili in ogni punto del programma diventano:

Program point	Valori
1	$i = \{\}, j = \{\}$
2	$i = [-1, 1], j = \{\}$
3	$i = [-1, -1] \cup [1, 1], j = [-5, 5]$
5	$i = [0, 0], j = [0, 0]$
6	$i = [-1, 1], j = [-5, 5]$

### 3.2 Punti fissi

Quando il programma contiene dei cicli (come **while**), il numero di esecuzioni possibili diventa arbitrario o infinito. Per analizzare queste strutture, utilizziamo il concetto di punto fisso (Definizione 2.20). L'analisi procede iterativamente accumulando informazioni (tramite il Join  $\cup$ ) finché lo stato non si stabilizza.

#### Esempio 3.2: Analisi di un ciclo su dominio infinito (Non terminazione)

Considerando il programma seguente che presenta un ciclo infinito che incrementa un contatore:

```

1: i = 0;
2: while (?)
3:   i++;
4: ...

```

Utilizzando il reticolo  $\langle \mathcal{P}(\mathbb{Z}), \subseteq, \cup, \cap \rangle$ , è possibile calcolare i valori di  $i$  al punto 2 (l'ingresso del ciclo). Lo stato al punto 2 è definito come l'unione del valore iniziale (da 1) e del valore di ritorno dal ciclo (da 3):

$$Val(2) = \{0\} \cup (Val(3) + 1)$$

Passo passo:



Iterazione	Valori al Punto 2	Valori al Punto 3
0	$\emptyset$	$\emptyset$
1	$\{0\}$	$\emptyset$
2	$\{0\}$	$\{0\}$
3	$\{0\} \cup \{0 + 1\} = \{0, 1\}$	$\{0\}$
4	$\{0, 1\}$	$\{0, 1\}$
5	$\{0, 1\} \cup \{1 + 1\} = \{0, 1, 2\}$	$\{0, 1\}$
...	...	...

Si nota che l'insieme continua a crescere  $(\{0, 1, 2, 3, \dots\})$  e, poiché il reticolo  $\mathcal{P}(\mathbb{Z})$  ha altezza infinita, l'algoritmo non raggiunge mai un punto fisso in tempo finito.

### 3.3 IMP

#### 3.3.1 Semantica concreta

#### 3.4 Semantica delle tracce