Definizioni Fondamentali

Analisi Statica e Verifica del Software

1 Set (Insieme)

Definizione 1.1 (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, valgono le seguenti notazioni fondamentali:

- **Appartenenza:** $s \in S$ indica che l'elemento s appartiene all'insieme S.
- **Sottoinsieme:** $S_1 \subseteq S_2 \iff \forall s \in S_1 \Rightarrow s \in S_2$.
- **Unione:** $S_1 \cup S_2 = \{s \mid s \in S_1 \vee s \in S_2\}$.
- **Intersezione:** $S_1 \cap S_2 = \{s \mid s \in S_1 \wedge s \in S_2\}$.

2 Partial Order (Ordine Parziale)

Definizione 2.1 (Ordine Parziale). Un ordine parziale è una relazione binaria $\sqsubseteq$ su un insieme X che soddisfa le seguenti tre proprietà per ogni $x, y, z \in X$:

1. **Riflessività:** $\forall x \in X \Rightarrow x \sqsubseteq x$.
2. **Anti-simmetria:** $(x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$.
3. **Transitività:** $(x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$.

3 Poset (Insieme Parzialmente Ordinato)

Definizione 3.1 (Poset). Un *Poset* (Partially Ordered Set) è la coppia formata da un insieme X e da una relazione di ordine parziale definita su di esso. Si denota formalmente come:

$$\langle X, \sqsubseteq \rangle$$

Esempio: L'insieme dei numeri interi con la relazione "minore o uguale" è un poset: $\langle \mathbb{Z}, \leq \rangle$.

4 Powerset (Insieme delle Parti)

Definizione 4.1 (Powerset). Dato un insieme S, l'insieme delle parti (*powerset*) di S, indicato con $\wp(S)$ (o $\mathcal{P}(S)$), è l'insieme di tutti i sottoinsiemi di S:

$$\wp(S) = \{U \mid U \subseteq S\}$$

Cardinalità: Dato un insieme S con $|S| = n$ elementi, il suo powerset ha $|\wp(S)| = 2^n$ elementi.
Struttura di Poset: Un powerset forma naturalmente un poset rispetto alla relazione di inclusione insiemistica: $\langle \wp(S), \subseteq \rangle$.

5 Bounds e Operazioni nel Powerset

In un poset generico $\langle X, \sqsubseteq \rangle$, definiamo i concetti di limiti (bounds) per un sottoinsieme $Y \subseteq X$.

5.1 Definizioni Astratte

Definizione 5.1 (**Upper Bound e LUB**).

- **Upper Bound (Maggiorante):** Un elemento $u \in X$ è un upper bound di Y se è maggiore o uguale a tutti gli elementi di Y $(\forall y \in Y, y \sqsubseteq u)$.
- **Least Upper Bound (LUB o Join $\sqcup$):** È il più piccolo tra tutti gli upper bound. Se esiste, è unico.

Definizione 5.2 (**Lower Bound e GLB**).

- **Lower Bound (Minorante):** Un elemento $l \in X$ è un lower bound di Y se è minore o uguale a tutti gli elementi di Y $(\forall y \in Y, l \sqsubseteq y)$.
- **Greatest Lower Bound (GLB o Meet $\sqcap$):** È il più grande tra tutti i lower bound. Se esiste, è unico.

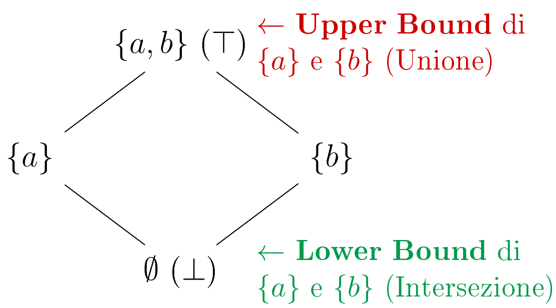
5.2 Applicazione nel Powerset

Nel caso specifico del Powerset $\langle \wp(S), \subseteq \rangle$, le operazioni astratte corrispondono alle operazioni insiemistiche classiche:

Concetto Astratto	**Simbolo**	**Nel Powerset ($\subseteq$)**
Ordine Parziale	$\sqsubseteq$	Inclusione ($\subseteq$)
Least Upper Bound (Join)	$\sqcup$	**Unione** ($\cup$)
Greatest Lower Bound (Meet)	$\sqcap$	**Intersezione** ($\cap$)
Elemento Top	$\top$	Insieme Universo (S)
Elemento Bottom	$\bot$	Insieme Vuoto ($\emptyset$)

5.3 Rappresentazione Grafica (Diagramma di Hasse)

Esempio del poset $\wp(\{a, b\})$ ordinato per inclusione.



6 Reticoli (Lattices)

Un reticolo è un tipo speciale di poset "molto ordinato", in cui non ci si perde mai né salendo né scendendo.

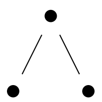
6.1 Lattice (Reticolo)

Definizione 6.1 (Lattice). Un poset $\langle X, \sqsubseteq \rangle$ è un **Lattice** se, per **ogni coppia** di elementi $x, y \in X$, esistono sempre:

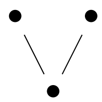
1. Il loro **Join** $x \sqcup y$ (Least Upper Bound).
2. Il loro **Meet** $x \sqcap y$ (Greatest Lower Bound).

Differenza visiva:

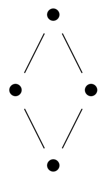
- **Join Semi-Lattice:** Converge sempre andando verso l'alto (come una V).
- **Meet Semi-Lattice:** Converge sempre andando verso il basso (come una $\wedge$).
- **Lattice:** È chiuso sia sopra che sotto (come un diamante).



Join Semi-Lattice



Meet Semi-Lattice



Lattice

6.2 Set Lattice (Reticolo di Insiemi)

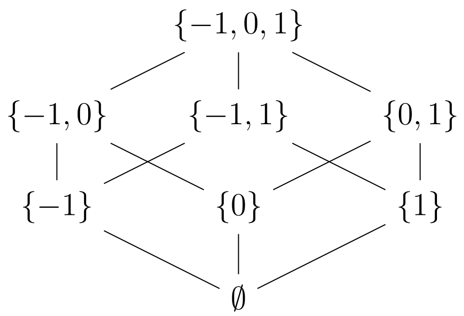
È l'"istanza concreta" più comune di un reticolo.

Definizione 6.2 (Set Lattice). Dato un insieme S, il **Set Lattice** è la struttura formata dal suo powerset $\wp(S)$ equipaggiato con le operazioni insiemistiche classiche:

$$\langle \wp(S), \subseteq, \cup, \cap \rangle$$

Caratteristiche:

- **Dominio:** Insieme delle parti.
- **Join ($\sqcup$):** Unione ($\cup$).
- **Meet ($\sqcap$):** Intersezione ($\cap$).



Esempio: Set Lattice su $\wp(\{-1, 0, 1\})$

6.3 Complete Lattice (Reticolo Completo)

Definizione 6.3 (**Complete Lattice**)**.** Un reticolo è **completo** se Join ($\sqcup$) e Meet ($\sqcap$) esistono per **qualsiasi sottoinsieme** del dominio, anche infinito.

Conseguenze importanti:

- Un reticolo completo ha sempre un elemento **Top** ($\top = \sqcup X$) e un elemento **Bottom** ($\bot = \sqcap X$).
- I reticoli finiti (come i Set Lattice su insiemi finiti) sono sempre completi.
- I numeri interi $\langle \mathbb{Z}, \leq \rangle$ sono un Lattice ma **non** completo (mancano $\top$ e $\bot$). Per renderlo completo bisogna aggiungere $+\infty$ e $-\infty$.

7 Relazioni e Funzioni (Maps)

Questi concetti matematici sono fondamentali per descrivere come cambiano gli stati di un programma durante l'esecuzione.

7.1 Relazioni

Definizione 7.1 (Relazione). Dati due insiemi X e Y, una **relazione** R è un sottoinsieme del prodotto cartesiano $X \times Y$:

$$R \subseteq X \times Y$$

Esempio: L'ordine parziale $\sqsubseteq$ è una relazione definita su un insieme con se stesso ($R \subseteq X \times X$).

7.2 Funzioni (Maps)

Definizione 7.2 (Funzione). Una **funzione** (o mappa) $f : X \to Y$ è una relazione speciale che soddisfa la proprietà di **unicità dell'immagine**: per ogni input $x \in X$, esiste al massimo un output $y \in Y$.

Le funzioni sono usate per modellare lo stato della memoria (es. mappa *variabile* $\to$ *valore*).

7.2.1 Proprietà delle Funzioni sui Poset

Quando le funzioni operano su insiemi ordinati (Poset), ci interessano tre proprietà chiave per l'analisi statica.

Definizione 7.3 (Monotonia). Una funzione $f : X \to Y$ è **monotona** se preserva l'ordine:

$$x_1 \sqsubseteq x_2 \implies f(x_1) \sqsubseteq f(x_2)$$

Intuitivamente: Se l'input "cresce" (diventa più grande o più preciso), l'output non può "decrescere".

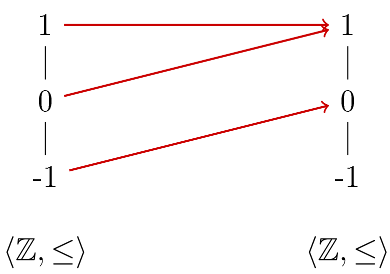
Definizione 7.4 (Embedding Ordinato). È una condizione più forte della monotonia. La funzione preserva l'ordine in entrambe le direzioni:

$$x_1 \sqsubseteq x_2 \iff f(x_1) \sqsubseteq f(x_2)$$

La struttura dell'ordine viene conservata perfettamente nel passaggio da X a Y.

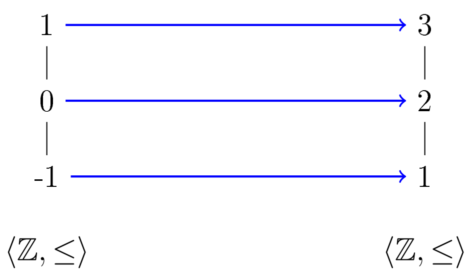
Definizione 7.5 (Isomorfismo). È un embedding ordinato che è anche **suriettivo** (copre tutto il codominio Y). Significa che i due poset X e Y sono strutturalmente identici.

Funzione Monotona



Se x sale, la freccia non scende mai ($f(x_1) \sqsubseteq f(x_2)$).

Embedding Ordinato



Struttura copiata identica ($f(x) = x + 2$). Iniettiva e bidirezionale.

8 Teoria del Punto Fisso e Terminazione

Questa sezione copre i concetti matematici necessari per garantire che l'analisi statica termini e produca risultati corretti, specialmente in presenza di cicli.

8.1 Chains (Catene)

Definizione 8.1 (**Catena**)**.** Dato un poset $\langle X, \sqsubseteq \rangle$, un sottoinsieme $C \subseteq X$ è una **catena** se è totalmente ordinato, ovvero tutti gli elementi sono confrontabili tra loro:

$$\forall x, y \in C \implies (x \sqsubseteq y \lor y \sqsubseteq x)$$

Catena Ascendente: Una sequenza indicizzata $(l_n)_{n \in \mathbb{N}}$ tale che $i \leq j \implies l_i \sqsubseteq l_j$.

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

Nell'analisi statica, le catene ascendenti rappresentano l'accumulo progressivo di informazioni durante le iterazioni.

8.2 ACC (Ascending Chain Condition)

Questa proprietà è fondamentale per garantire la **terminazione** degli algoritmi di analisi.

Definizione 8.2 (**ACC**)**.** Un poset soddisfa la **Ascending Chain Condition** se ogni catena ascendente infinita alla fine si **stabilizza**. Esiste un indice k oltre il quale il valore non cambia più:

$$\exists k \geq 0 \text{ tale che } \forall j \geq k, l_k = l_j$$

Intuizione: Non è possibile "crescere" all'infinito. Se il dominio ha la ACC (es. è finito), l'analizzatore non andrà mai in loop infinito.

8.3 Mappe Continue

Definizione 8.3 (Funzione Continua). Siano X e Y due CPO (Complete Partial Orders). Una funzione $f : X \to Y$ è **continua** se preserva i limiti delle catene (i LUB):

$$f\left(\bigsqcup C\right) = \bigsqcup_{c \in C} f(c)$$

Nota: La continuità è una condizione più forte della monotonia. È necessaria per applicare il Teorema di Kleene.

8.4 Fixpoint (Punto Fisso)

I punti fissi forniscono la semantica dei costrutti ciclici (loop).

Definizione 8.4 (Punto Fisso). Dato un insieme X e una funzione $f : X \to X$, un elemento $x \in X$ è un punto fisso se:

$$f(x) = x$$

Nell'analisi statica cerchiamo il **Least Fixpoint (lfp)**, ovvero il punto fisso più piccolo, che rappresenta l'insieme minimo dei comportamenti possibili del programma.

8.4.1 Teoremi fondamentali

- **Teorema di Knaster-Tarski:** Se $\langle X, \sqsubseteq \rangle$ è un reticolo completo e f è **monotona**, allora l'insieme dei punti fissi è un reticolo completo e il lfp esiste.
  Limite: Non dice come calcolarlo (non costruttivo).
- **Teorema di Kleene:** Se $\langle X, \sqsubseteq \rangle$ è un CPO e f è **continua**, allora il lfp è il limite dell'iterazione partendo dal basso ($\perp$):

  $$lfp(f) = \bigsqcup_{n \in \mathbb{N}} f^n(\perp)$$

  Vantaggio: Fornisce un algoritmo costruttivo (iterativo) usato dagli analizzatori.

9 Il Linguaggio IMP

IMP è un linguaggio imperativo minimale ("toy language") utilizzato per modellare formalmente la semantica senza la complessità dei linguaggi reali. È Turing-completo.

9.1 Sintassi

La sintassi è divisa in tre categorie sintattiche:

1. **Espressioni Aritmetiche (e):** Calcolano valori interi.

$$e ::= x \mid n \mid e_1 \; op_a \; e_2$$

   Dove: x è una variabile, $n \in \mathbb{Z}$, $op_a \in \{+, -, *, \div\}$.

2. **Espressioni Booleane (b):** Calcolano valori di verità (condizioni).

$$b ::= \text{true} \mid \text{false} \mid \neg b \mid b_1 \; op_b \; b_2 \mid e_1 \; op_c \; e_2$$

   Dove: $op_b \in \{\wedge, \vee\}$, $op_c \in \{==, <, >, \leq\}$.

3. **Statement (Comandi, s):** Modificano lo stato della memoria.

$$s ::= x := e \mid \text{skip} \mid s_1; s_2 \mid \text{if } b \text{ then } s_1 \text{ else } s_2 \mid \text{while } b \text{ do } s$$

9.2 Esempio di programma IMP

Un programma che calcola la somma dei numeri da 0 a x:

```
y := 0;
while x > 0 do
    y := y + x;
    x := x - 1
```

10 Semantica delle Tracce

10.1 Definizione di Traccia

Definizione 10.1 (**Traccia**)**.** Una traccia $\tau \in X^\infty$ è una sequenza di stati che rappresenta una singola evoluzione del programma:

- **Finite:** L'esecuzione termina in uno stato finale (es. il programma finisce).
- **Infinite:** L'esecuzione non termina (es. un ciclo `while(true)`).
- **Parziali:** Rappresentano un prefisso dell'esecuzione fino a un certo punto intermedio (es. se interrompiamo l'analisi).

10.2 Semantica Concreta come Reticolo

L'insieme di **tutte** le possibili esecuzioni (semantica concreta) viene modellato utilizzando un reticolo basato sul powerset delle tracce:

$$\langle \wp(X^\infty), \subseteq, \cup, \cap \rangle$$

L'obiettivo è calcolare questo insieme per catturare sia le esecuzioni che terminano correttamente, sia quelle che divergono (loop infiniti).

10.3 Calcolo tramite Least Fixpoint (lfp)

La semantica del programma P si calcola cercando il **Least Fixpoint** della sua funzione di transizione F_P. Utilizzando il Teorema di Kleene, procediamo in modo iterativo partendo dal basso :

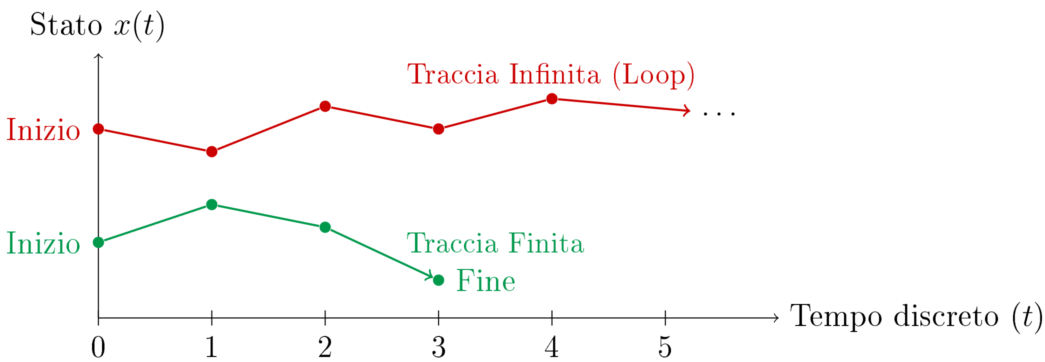
$$lfp(F_P) = \bigcup_{n \in \mathbb{N}} F_P^n(\bot)$$

La procedura iterativa: L'iterazione costruisce la semantica passo dopo passo:

1. **Passo 0 ($F^0(\bot) = \bot$):** Stato vuoto o non inizializzato.
2. **Passo 1 ($F^1(\bot)$):** Tracce di lunghezza 1 (stati iniziali).
3. **Passo k ($F^k(\bot)$):** Insieme delle esecuzioni parziali di lunghezza fino a k.

Il processo termina quando si raggiunge un punto fisso, ovvero quando l'aggiunta di un nuovo passo non scopre nuove tracce ($F^{k+1} = F^k$).

Rappresentazione Grafica delle Tracce:



11 Panoramica di un Analizzatore Statico

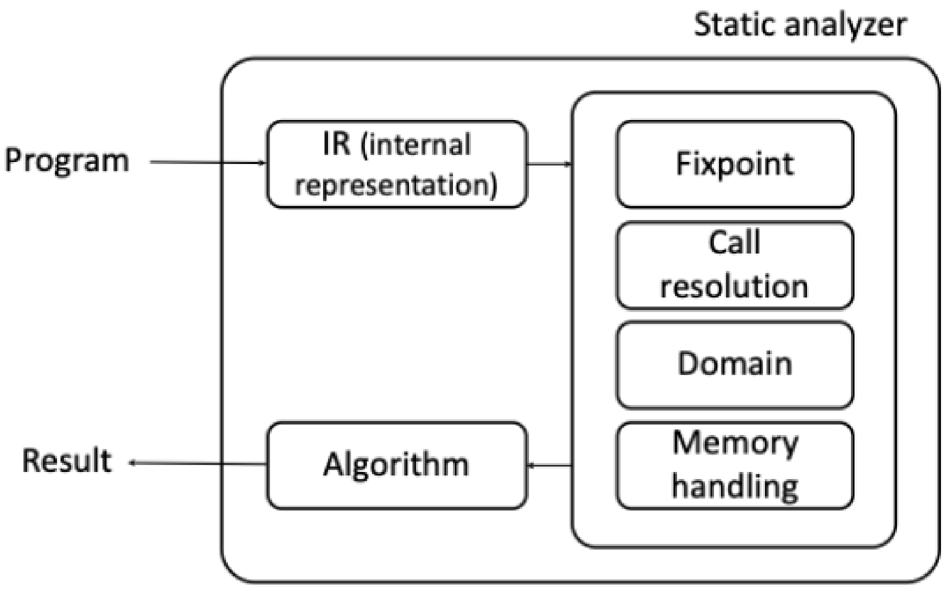
Un analizzatore statico è un software complesso composto da diversi moduli che lavorano insieme per trasformare il codice sorgente in risultati di analisi (come la segnalazione di bug o garanzie di correttezza).

11.1 Componenti Principali

Il flusso di lavoro di un analizzatore statico segue questi passaggi logici:

1. **Programma (Input):** Il codice sorgente che si vuole analizzare.
2. **IR (Rappresentazione Interna):** Il codice viene tradotto in una forma intermedia (es. Control Flow Graph) più facile da manipolare per l'analisi.
3. **Engine di Analisi:** Il "cuore" del sistema. Gestisce l'algoritmo di punto fisso, la risoluzione delle chiamate a funzione e la gestione della memoria.
4. **Dominio Astratto:** L'astrazione dei dati su cui lavora l'algoritmo (es. Intervalli, Segni, Costanti).
5. **Risultato:** L'output dell'analisi (warnings, report).

In parole semplici: *L'idea fondamentale è la* ***modularità****. Se vuoi analizzare un nuovo linguaggio (es. passare da Java a C++) o cambiare il tipo di analisi (es. da intervalli a segni), non devi riscrivere tutto. I componenti devono essere indipendenti: l'algoritmo di punto fisso non deve sapere che linguaggio stai analizzando, deve solo lavorare sul grafo astratto (IR).*



Possibile domanda d'esame:
Quali sono i componenti principali di un analizzatore statico e perché è importante che siano modulari?

12 LiSA (Library for Static Analysis)

LiSA è una libreria open-source scritta in Java per costruire analizzatori statici basati sull'Interpretazione Astratta.

12.1 Architettura e Flusso di Lavoro

In LiSA, il processo di analisi avviene attraverso i seguenti step:

1. **Front-end:** Un componente specifico per ogni linguaggio (Java, C, IMP, ecc.) traduce il codice sorgente nei CFG (Control Flow Graphs) di LiSA.
2. **LiSA Core:** Il motore di analisi generico che prende i CFG e li analizza. Include:
   - **CFG Fixpoint:** L'algoritmo che calcola il punto fisso sul grafo.
   - **Statement Semantics:** La logica per interpretare le istruzioni.
   - **Domain:** L'implementazione dei domini astratti (es. Intervalli).
   - **Memory Handling:** La gestione della memoria (heap, stack).
3. **Checks:** Controlli finali sui risultati per generare warnings.

Approfondimento: La Metafora della Catena di Montaggio

Immagina LiSA come una **catena di montaggio universale** per analizzare programmi. Il suo obiettivo è prendere codice scritto in lingue diverse (Java, C, Python) e capire se è corretto, usando sempre lo stesso "macchinario" centrale.

Ecco la spiegazione dettagliata dei tre step:

1. Front-end: Il Traduttore Universale Il problema principale dell'analisi statica è che ogni linguaggio di programmazione ha una sintassi diversa. Scrivere un analizzatore per Java e uno per C richiederebbe di riscrivere tutto da zero due volte.

- **Cosa fa:** Il Front-end agisce come un interprete. Prende il codice sorgente (il file `.java` o `.c`) e lo traduce in un linguaggio che LiSA capisce: il **CFG (Control Flow Graph)**.
- **Perché è utile:** Una volta che il codice è diventato un CFG di LiSA, il resto dell'analizzatore non deve più preoccuparsi se l'originale era Java o C. Vede solo nodi (istruzioni) e archi (flusso).

2. LiSA Core: Il Motore di Analisi Questo è il cervello del sistema. Una volta ricevuto il CFG "tradotto", il Core deve eseguirlo in modo astratto per trovare le proprietà del programma. È composto da quattro pezzi fondamentali che lavorano insieme:

- **A. CFG Fixpoint (L'Algoritmo):** Immagina questo componente come il **direttore dei lavori**.
  - Il suo compito è percorrere il grafo (CFG) e propagare le informazioni da un nodo all'altro.

- Usa un algoritmo iterativo (basato sulla *worklist*). Continua a far girare le informazioni nel grafo finché queste non si stabilizzano (raggiungono il *punto fisso*), cioè finché non cambiano più.

- **B. Statement Semantics (Il Dizionario dei Significati):** Il grafo contiene istruzioni generiche. Questo componente spiega al motore *cosa* significano quelle istruzioni.
  - *Esempio:* Se nel grafo c'è un nodo che dice `x = a + b`, la *Statement Semantics* dice: "Attenzione, questo simbolo `+` significa 'somma aritmetica', non concatenazione". Traduce la sintassi in un'operazione logica che il dominio può capire.

- **C. Domain (La Lente di Ingrandimento):** Questo è il componente che decide **cosa** stiamo osservando dei dati.
  - Il motore chiede: "Ho la variabile `x`, come la rappresento?".
  - Se il Dominio è **Intervalli**, risponde: "Rappresentala come `[min, max]`".
  - Se il Dominio è **Segni**, risponde: "Dimmi solo se è `+` o `-`".
  - È qui che avvengono i calcoli veri (es. `[1, 5] + [2, 3] = [3, 8]`).

- **D. Memory Handling (La Mappa):** Questo componente gestisce **dove** sono salvati i dati.
  - In un programma reale, abbiamo variabili locali (nello stack) e oggetti dinamici (nello heap).
  - LiSA deve sapere che la variabile `x` in una funzione è diversa dalla variabile `x` in un'altra, o che `p.val` si riferisce a una specifica cella di memoria. Il *Memory Handling* trasforma nomi complessi in indirizzi astratti univoci.

3. Checks: L'Ispettore Finale Una volta che il Core ha finito di girare, abbiamo un grafo "decorato": ogni punto del programma ha associato uno stato astratto (es. "qui `x` vale `[0, 10]`").

- **Cosa fa:** I *Checks* scorrono questi risultati e verificano se violano delle regole.
- **Esempio:** Se c'è un'istruzione `y = 10 / x` e l'analisi ha calcolato che in quel punto `x` vale `[0, 5]`, il Check vede che lo `0` è incluso nell'intervallo e lancia un allarme (Warning): "Possibile divisione per zero!".

Esempio 12.1 (Riassunto pratico). Immagina di analizzare `x = 10 / y`:

1. **Front-end:** Legge il file e crea un grafo con un nodo per la divisione.
2. **LiSA Core:**
   - **Fixpoint:** Arriva al nodo della divisione portandosi dietro le informazioni precedenti.
   - **Memory:** Capisce quale `x` e quale `y` stiamo usando.
   - **Semantics:** Capisce che `/` è una divisione matematica.
   - **Domain:** Calcola il risultato astratto (es. `Intervallo(10) / Intervallo(y)`).
3. **Checks:** Controlla se il divisore `y` poteva essere `0`. Se sì, ti avvisa.

12.2 Struttura del CFG in LiSA

Un Control Flow Graph in LiSA è costituito da:

- **Nodi:** Rappresentano gli statement (istruzioni) del programma.
- **Archi:** Collegano i nodi e rappresentano il flusso. Possono essere:
  - **Sequential Edge:** Flusso normale sequenziale.
  - **True Edge:** Preso quando una condizione è vera (ramo `then`).
  - **False Edge:** Preso quando una condizione è falsa (ramo `else`).

In parole semplici: *Usare i CFG permette a LiSA di "dimenticare" la sintassi specifica del linguaggio originale (es. come si scrive un ciclo* `while` *in C vs Python) e lavorare su una struttura a grafo unificata.*

> **Possibile domanda d'esame:**
> Descrivere i tipi di archi presenti in un CFG di LiSA e la loro funzione.

13 Sintassi vs Semantica

Un concetto chiave in LiSA è la distinzione tra lo statement sintattico e il suo significato semantico.

Definizione 13.1 (**Rewriting Semantico**). Diverse operazioni sintattiche possono avere lo stesso significato semantico. LiSA utilizza un linguaggio interno di **Espressioni Simboliche** per uniformare queste differenze.

Esempio: In Java, l'operatore `+` può significare:

- Somma aritmetica: `1 + 2`
- Concatenazione di stringhe: `"a" + "b"`

Il dominio astratto non deve preoccuparsi della sintassi. LiSA traduce lo statement in un'espressione simbolica "atomica":

- Se sono numeri → `ArithmSum`
- Se sono stringhe → `StringConcat`

In questo modo, il dominio astratto implementa solo la logica per `ArithmSum` o `StringConcat`, indipendentemente da come erano scritte nel codice originale.

Possibile domanda d'esame:
Perché LiSA distingue tra lo statement sintattico e la sua semantica tramite espressioni simboliche? Fornire un esempio.

14 Implementazione dell'Analisi Dataflow

LiSA fornisce un'architettura specifica per implementare analisi dataflow (come quelle viste nel Cap. 4).

14.1 Algoritmo Generico (Worklist)

Le analisi dataflow in LiSA sfruttano un algoritmo di punto fisso generico basato su **worklist**. L'implementazione è divisa in due classi principali per separare la logica:

- **DataflowDomain:** Gestisce la logica generale del reticolo e l'operazione di Join.
  - `PossibleForwardDataflowDomain`: Implementa analisi **May** (usa l'Unione $\cup$).
  - `DefiniteForwardDataflowDomain`: Implementa analisi **Must** (usa l'Intersezione $\cap$).
- **DataflowElement:** Rappresenta il singolo elemento tracciato (es. una variabile viva, un'espressione disponibile). Definisce le operazioni specifiche **Gen** e **Kill**.

In parole semplici: *Questa struttura permette di implementare una nuova analisi (es. Reaching Definitions) scrivendo solo la logica di Gen/Kill nel `DataflowElement`, mentre il `DataflowDomain` gestisce automaticamente il calcolo del punto fisso e la propagazione nel grafo.*

Nota: Attualmente LiSA supporta nativamente solo analisi **Forward**.

15 Perché Approssimare?

La verifica del software ha un problema fondamentale: il **Teorema di Rice**. Esso afferma che qualsiasi proprietà non banale (interessante) del comportamento di un programma è **indecidibile**.

In parole semplici: *Non esiste e non esisterà mai un programma che possa prendere un qualsiasi codice e dirti con certezza assoluta "Sì, è corretto" o "No, ha un bug" in tempo finito. Se vogliamo un'analisi automatica che termini sempre, dobbiamo rinunciare alla perfezione e accettare l'**approssimazione**.*

15.1 Soundness e Decidibilità

L'obiettivo dell'analisi statica è trovare un'approssimazione $[\![P]\!]^\sharp$ della semantica reale $[\![P]\!]$ che soddisfi due requisiti:

1. **Soundness (Correttezza):** $[\![P]\!] \subseteq [\![P]\!]^\sharp$. L'approssimazione deve includere *tutti* i comportamenti reali (anche a costo di includerne alcuni che non esistono).
2. **Decidibilità:** Verificare le proprietà sull'approssimazione deve essere possibile in tempo finito.

16 Astrazione: Oggetti e Proprietà

L'astrazione è il processo di sostituire oggetti concreti con descrizioni semplificate (proprietà).

16.1 Reticolo delle Proprietà

Nel mondo concreto, una proprietà è semplicemente un **insieme di oggetti** che la soddisfano. Esempio: La proprietà "essere pari" è l'insieme $\{\dots, -2, 0, 2, 4, \dots\}$.

Definizione 16.1 (Reticolo delle Proprietà). L'insieme di tutte le possibili proprietà $\wp(\Sigma)$ forma un reticolo completo dove:

- L'ordine parziale $\subseteq$ è l'implicazione logica.
- Il Join $\cup$ è l'unione (disgiunzione logica).
- Il Meet $\cap$ è l'intersezione (congiunzione logica).
- $\top = \Sigma$ (Vero/Tutto), $\bot = \emptyset$ (Falso/Impossibile).

17 Direzione dell'Astrazione

Quando approssimiamo un insieme concreto P con un insieme astratto $P^\sharp$, possiamo sbagliare "per eccesso" o "per difetto".

17.1 1. Approssimazione dal Basso (Under-Approximation)

$$P^\sharp \subseteq P$$

L'insieme astratto è più piccolo del reale.

- Se $x \in P^\sharp \implies$ Sicuramente $x \in P$ (**YES**).
- Se $x \notin P^\sharp \implies$ Non so (**Don't Know**).

Uso: Testing e Bug Finding. Se trovo un bug nell'astrazione, il bug esiste davvero. Non posso garantire la sicurezza (potrei mancare dei bug fuori da $P^\sharp$).

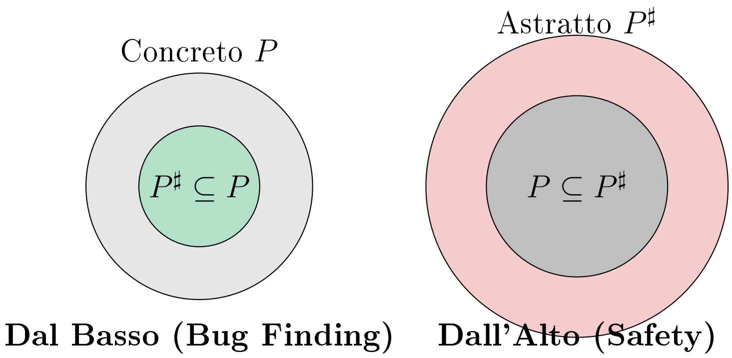
17.2 2. Approssimazione dall'Alto (Over-Approximation)

$$P \subseteq P^\sharp$$

L'insieme astratto è più grande del reale (include comportamenti "spuri").

- Se $x \notin P^\sharp \implies$ Sicuramente $x \notin P$ (**NO** - Sicuro).
- Se $x \in P^\sharp \implies$ Potrebbe essere vero o un Falso Positivo (**Don't Know**).

Uso: Safety Verification (Analisi Statica classica). Se l'astrazione dice "nessun errore", il programma è sicuro. Se segnala un errore, potrebbe essere un falso allarme.



Possibile domanda d'esame:
Qual è la differenza tra approssimazione dal basso e dall'alto? Quale delle due garantisce la "Soundness" per la verifica di sicurezza e perché?

18 Esempio: Il Dominio dei Segni

Un esempio classico di astrazione dall'alto per i numeri interi.

Definizione 18.1 (Dominio dei Segni). Il dominio astratto è $D^\sharp = \{\perp, (-), (0), (+), \top\}$.

- $\alpha(\{1, 5, 100\}) = (+)$
- $\alpha(\{-5, 2\}) = \top$ (Non so il segno, contiene sia positivi che negativi).

Perdita di precisione: Le operazioni devono essere ridefinite per mantenere la soundness.

- $(+) +^\sharp (+) = (+)$ (Preciso).
- $(+) +^\sharp (-) = \top$ (Impreciso: $5 + (-2) > 0$, ma $2 + (-5) < 0$. Non possiamo saperlo).

19 Gerarchia delle Semantiche

L'analisi statica può essere vista come una serie di astrazioni successive, partendo dalla realtà più dettagliata fino a quella più astratta.

19.1 1. Semantica delle Tracce (Trace Semantics)

È la realtà completa. Una traccia è una sequenza di stati nel tempo (es. $x_0 \to x_1 \to x_2 \ldots$). Distingue ogni singola esecuzione e l'ordine temporale degli eventi. È incalcolabile.

19.2 2. Semantica Collettrice (Collecting Semantics)

Astrazione: Dimentichiamo l'ordine temporale e le relazioni tra stati successivi. Per ogni punto del programma (riga di codice), raccogliamo l'insieme di **tutti** i possibili stati che possono verificarsi lì, indipendentemente da come ci siamo arrivati.

In parole semplici: *Immagina di scattare una foto a tutte le possibili variabili ogni volta che il programma passa per la riga 10. Mettiamo tutti questi valori in un sacco. Perdiamo la storia ("ci sono arrivato dopo il ciclo o prima?"), ma sappiamo quali valori sono possibili. Questo introduce "rumore" o tracce spurie.*

19.3 3. Semantica Astratta (Abstract Semantics)

Astrazione: Sostituiamo gli insiemi di stati (che possono essere infiniti o complessi) con oggetti geometrici o logici calcolabili (Intervalli, Segni, Poliedri).

> **Possibile domanda d'esame:**
> Cosa si intende per Semantica Collettrice e quale informazione si perde passando dalla Semantica delle Tracce a quest'ultima?

Risposta:

La Semantica Collettrice associa ad ogni punto del programma (es. riga di codice o nodo del CFG) l'insieme di tutti gli stati di memoria raggiungibili in quel punto in qualsiasi possibile esecuzione. Matematicamente, dimentica la dimensione temporale t e proietta gli stati sulla struttura del programma (i punti di controllo).

Quale informazione si perde?

Passando dalla Semantica delle Tracce a quella Collettrice si perde l'ordine temporale e le relazioni tra stati successivi.

- **Perdita della Storia:** Non sappiamo più "da dove veniamo". Se lo stato S_1 e lo stato S_2 sono entrambi possibili alla riga L, la semantica collettrice non ci dice se S_1 appare sempre prima di S_2 o viceversa.
- **Tracce Spurie (Rumore):** Poiché perdiamo il legame causa-effetto tra uno stato e il successivo, l'analisi potrebbe ammettere dei cammini "fantasma" (*spurious traces*) che saltano da uno stato valido all'altro in modi che nella realtà non avvengono mai (es. saltare dallo stato dell'iterazione 10 allo stato dell'iterazione 2).