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.

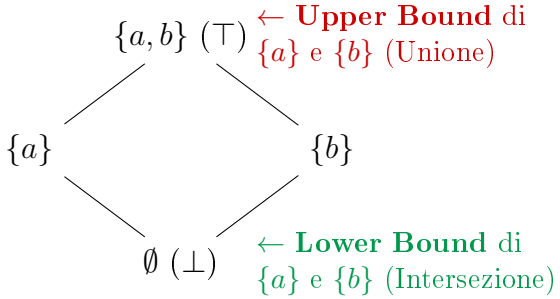
5.2 Applicazione nel Powerset

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.

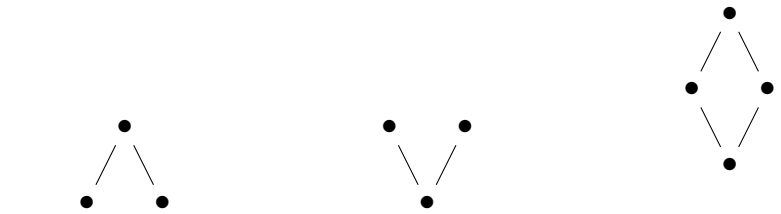
6.1 Lattice (Reticolo)

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)

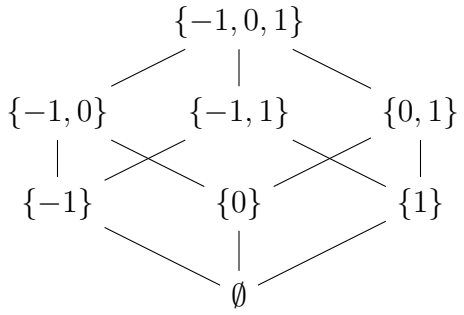
È l'"istanza concreta" più comune di un reticolo.

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".

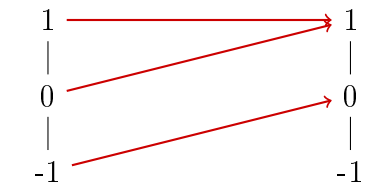
Definizione 7.4 (**Embedding Ordinato**)**.** È una condizione più forte della monotonia. La funzione preserva l'ordine in entrambe le direzioni:

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

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.

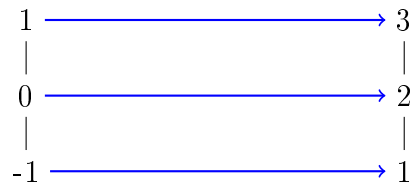
Funzione Monotona



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

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

Embedding Ordinato



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

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:

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

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 ($\bot$):

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

  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 :

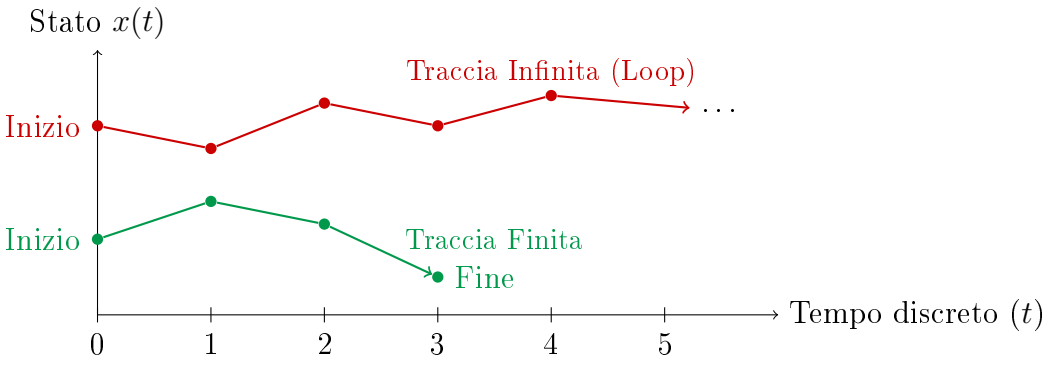
$$lfp(F_P) = \bigcup_{n \in \mathbb{N}} F_P^n(\bot)$$

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

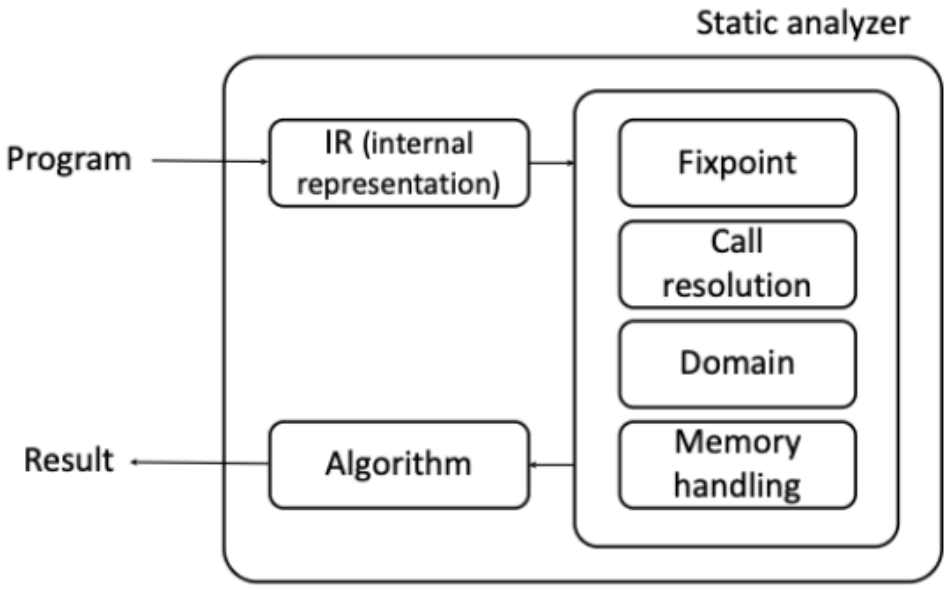
Un analizzatore statico è un software complesso composto da diversi moduli che lavorano insieme per trasformare il codice sorgente in risultati di analisi (come la segnalazione di bug o garanzie di correttezza).

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 $P^\sharp$).

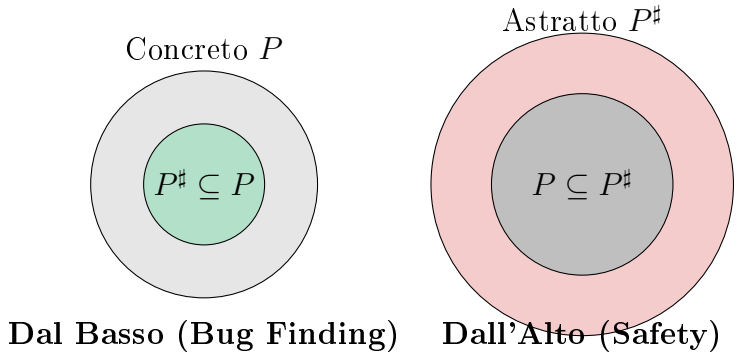
17.2 2. Approssimazione dall'Alto (Over-Approximation)

$$P \subseteq P^\sharp$$

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 $D^\sharp = \{\bot, (-), (0), (+), \top\}$.

- $\alpha(\{1, 5, 100\}) = (+)$
- $\alpha(\{-5, 2\}) = \top$ (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. $x_0 \to x_1 \to x_2 \dots$). 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).

20 Il Ponte tra Concreto e Astratto

Per analizzare un programma, dobbiamo muoverci tra due mondi:

1. **Dominio Concreto (C):** La realtà precisa ma incalcolabile. Esempio: l'insieme dei numeri interi $\wp(\mathbb{Z})$.
2. **Dominio Astratto (A):** La semplificazione calcolabile. Esempio: il dominio dei Segni o degli Intervalli.

Per passare da un mondo all'altro senza fare errori logici, usiamo due funzioni speciali chiamate α e γ.

20.1 Le Funzioni α e γ

Definizione 20.1 (**Astrazione** α)**.** $\alpha : C \to A$. Dato un insieme di valori concreti, restituisce il valore astratto "migliore" che li descrive.

$$\alpha(\{1, 2, 5\}) = [1, 5] \quad \text{(negli Intervalli)}$$

Definizione 20.2 (**Concretizzazione** γ)**.** $\gamma : A \to C$. Dato un valore astratto, restituisce l'insieme di **tutti** i valori concreti che esso rappresenta.

$$\gamma([1, 5]) = \{1, 2, 3, 4, 5\}$$

In parole semplici: *Nota che γ restituisce $\{1, 2, 3, 4, 5\}$ anche se l'insieme originale era solo $\{1, 2, 5\}$. I numeri 3 e 4 sono "rumore" o approssimazione introdotta dall'astrazione. Non possiamo evitarlo se vogliamo semplificare.*

21 Connessione di Galois (GC)

Una Connessione di Galois è la formalizzazione matematica che garantisce che α e γ siano "sincronizzate" correttamente per garantire la **Soundness** (sicurezza).

Definizione 21.1 (**Connessione di Galois**)**.** Due posets $\langle C, \subseteq \rangle$ e $\langle A, \sqsubseteq \rangle$ formano una Connessione di Galois (α, γ) se valgono due proprietà:

1. $\gamma \circ \alpha$ **è Estensiva:** $\forall c \in C, \quad c \subseteq \gamma(\alpha(c))$
2. $\alpha \circ \gamma$ **è Riduttiva:** $\forall a \in A, \quad \alpha(\gamma(a)) \sqsubseteq a$

Inoltre, entrambe le funzioni devono essere **monotone** (preservare l'ordine).

21.1 Spiegazione Intuitiva delle Proprietà

1. Estensività ($\gamma \circ \alpha \supseteq Id$): "Non perdere la verità" Se prendo i dati reali (c), li astraggo (α) e poi torno indietro al concreto (γ), l'insieme che ottengo deve **contenere** i dati di partenza.

- $c = \{1, 5\}$

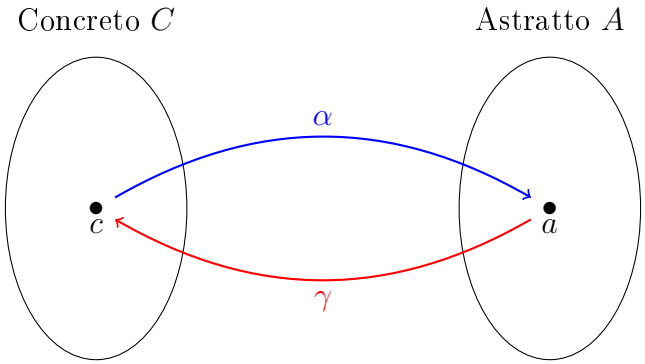
- $\alpha(c) = [1, 5]$
- $\gamma([1, 5]) = \{1, 2, 3, 4, 5\}$

$\{1, 5\} \subseteq \{1, 2, 3, 4, 5\}$. La proprietà è rispettata. Se mancasse il 5, l'analisi sarebbe sbagliata (unsound).

2. Riduttività ($\alpha \circ \gamma \sqsubseteq Id$): "Non perdere precisione inutilmente" Se parto da un'astrazione (a), la concretizzo (γ) e poi la ri-astraggo (α), non devo ottenere un'astrazione peggiore (più grande) di quella di partenza.

- $a = [1, 5]$
- $\gamma(a) = \{1, 2, 3, 4, 5\}$
- $\alpha(\{1, \ldots, 5\}) = [1, 5]$

$[1, 5] \sqsubseteq [1, 5]$. Sono uguali, quindi la proprietà è rispettata.



$$\alpha(c) \sqsubseteq a \iff c \subseteq \gamma(a)$$

22 Galois Insertion (GI)

Spesso nel dominio astratto ci sono elementi "inutili" o ridondanti. Quando li rimuoviamo tutti, otteniamo una forma speciale di connessione.

Definizione 22.1 (**Galois Insertion**)**.** Una Connessione di Galois è una **Insertion** se $\alpha \circ \gamma$ è l'**Identità**:

$$\forall a \in A, \quad \alpha(\gamma(a)) = a$$

In parole semplici: *Significa che ogni elemento del dominio astratto rappresenta un insieme di valori concreti unico. Non ci sono due etichette diverse per dire la stessa cosa. Esempio: Se avessi nel dominio astratto sia* $[1, 2]$ *che un simbolo speciale* K *che vuol dire "intervallo tra 1 e 2",* γ *mapperebbe entrambi a* $\{1, 2\}$*. Ri-astraendo* $\{1, 2\}$ *però* α *dovrebbe sceglierne uno solo. Se* $\alpha(\gamma(K)) = [1, 2] \neq K$*, non è una Insertion.*

23 Best Abstraction (Migliore Astrazione)

Una proprietà fondamentale garantita dalle Connessioni di Galois è l'esistenza della **Best Abstraction**.

Definizione 23.1. Per ogni insieme concreto $c \in C$, l'elemento $\alpha(c)$ è la **migliore** (più precisa) approssimazione possibile di c nel dominio A.

$$\forall a \in A, \quad c \subseteq \gamma(a) \implies \alpha(c) \sqsubseteq a$$

23.1 Quando la Best Abstraction non esiste?

Ci sono casi in cui non possiamo costruire una Connessione di Galois perché manca l'elemento "migliore".

Esempio: Il Cerchio e i Poliedri. Immagina di voler astrarre un **cerchio** usando dei **poliedri** (figure con lati dritti, come quadrati, esagoni, ecc.).

- Puoi racchiudere il cerchio in un quadrato.
- Puoi racchiuderlo in un ottagono (più preciso).
- Puoi usare un poligono a 100 lati (ancora più preciso).

Non esiste un "poliedro più piccolo possibile" che contiene il cerchio, perché puoi sempre tagliarne un pezzetto in più aggiungendo un lato. Quindi, tra il dominio delle figure geometriche curve e quello dei poliedri **non esiste una Connessione di Galois**.

Possibile domanda d'esame:
Spiegare la differenza tra Galois Connection e Galois Insertion. Fornire un esempio intuitivo di quando la proprietà di riduttività ($\alpha \circ \gamma \sqsubseteq Id$) diventa un'uguaglianza.

24 Semantica Astratta sui Control Flow Graphs (CFG)

Fino ad ora abbiamo visto la semantica concreta (le tracce di esecuzione). Per l'analisi statica vera e propria, è più comodo ragionare direttamente sul **Control Flow Graph (CFG)**.

24.1 Stati di Ingresso e Uscita

Dato un CFG, per ogni nodo (o punto del programma) x_i, vogliamo calcolare due insiemi di stati:

- **Entry State (Stato di Ingresso):** È lo stato della memoria *prima* che l'istruzione del nodo venga eseguita. Si calcola unendo (Join/LUB) gli stati di uscita di tutti i nodi predecessori.
$$entry(x_i) = \bigsqcup \{exit(x_j) \mid x_j \to x_i\}$$
- **Exit State (Stato di Uscita):** È il risultato dell'applicazione della semantica (astratta o concreta) dell'istruzione allo stato di ingresso.
$$exit(x_i) = [\![\text{stmt}(x_i)]\!](entry(x_i))$$

In parole semplici: *Immagina il CFG come una rete di tubi. L'acqua (i dati) scorre da un nodo all'altro. In ogni nodo, l'acqua che entra è la miscela di tutta l'acqua che arriva dai tubi precedenti. Il nodo poi "tratta" l'acqua (esegue l'istruzione) e la manda fuori.*

25 Sistemi di Equazioni e Punto Fisso

L'analisi statica si riduce a risolvere un sistema di equazioni ricorsive.

Definizione 25.1 (Sistema di Equazioni Semantiche). Dato un CFG con m nodi, il sistema è definito come:

$$F = \{x_i = f_i(x_1, \ldots, x_m) \mid i = 1, \ldots, m\}$$

dove ogni x_i rappresenta lo stato associato al nodo i e f_i è la funzione che calcola tale stato basandosi sugli altri.

L'obiettivo è trovare la **Soluzione Minima** (Least Fixpoint) di questo sistema.

- Nel dominio concreto $\wp(\mathbb{Z})$, calcolare questo punto fisso è spesso **indecidibile** o richiede tempo infinito.
- Ecco perché passiamo al dominio astratto.

26 Approssimazione e Soundness

Poiché non possiamo calcolare il punto fisso concreto, introduciamo un dominio astratto A che approssima il concreto C.

26.1 Soundness della Funzione Astratta

Affinché l'analisi sia affidabile, la funzione astratta $f^\sharp$ deve essere una **Sound Abstraction** della funzione concreta f.

Definizione 26.1 (Soundness). Siano C e A legati da una connessione di Galois (α, γ). Una funzione $f^\sharp : A \to A$ è sound rispetto a $f : C \to C$ se:

$$\forall c \in C, \quad \alpha(f(c)) \sqsubseteq_A f^\sharp(\alpha(c))$$

Oppure equivalentemente:

$$\forall a \in A, \quad f(\gamma(a)) \sqsubseteq_C \gamma(f^\sharp(a))$$

In parole semplici: *Significa che l'operazione astratta non deve mai essere "più ottimista" della realtà. Se nella realtà $2 + 2 = 4$, la mia somma astratta deve dire "il risultato è positivo" o "è pari", non può dire "è negativo". Deve includere la verità.*

27 Teoremi di Trasferimento del Punto Fisso

Una volta definito il sistema astratto, due teoremi ci garantiscono che il risultato che calcoleremo sarà corretto rispetto al programma reale.

27.1 Teorema di Approssimazione di Kleene

Se la funzione astratta è **continua**, possiamo calcolare il punto fisso iterando la funzione partendo dal basso ($\bot$). Il limite della sequenza è un'approssimazione corretta.

$$lfp(f) \sqsubseteq \gamma(\text{limite iterazioni astratte})$$

27.2 Teorema di Approssimazione di Tarski

Questo è più generale (richiede solo la **monotonia**). Afferma che qualsiasi **post-fixpoint** astratto è un'approssimazione sicura.

Teorema 27.1 (Approssimazione di Tarski). Se a è un post-fixpoint di $f^\sharp$ (cioè $f^\sharp(a) \sqsubseteq a$), allora:

$$lfp(f) \sqsubseteq \gamma(a)$$

In parole semplici: *Questo teorema è potentissimo. Ci dice che non serve trovare esattamente il punto fisso minimo astratto (che potrebbe essere difficile). Basta trovare un punto "stabile" dove $f^\sharp(a)$ è più piccolo o uguale ad a. Qualsiasi punto stabile è una garanzia di sicurezza per il programma reale.*

Possibile domanda d'esame:
Enunciare il Teorema di Approssimazione di Tarski e spiegare perché è fondamentale per la terminazione dell'analisi statica (suggerimento: widening).

28 Il Problema del Filter (Assume)

Nell'analisi statica classica, quando il programma incontra una diramazione condizionale (es. `if (x > 0)`), l'analizzatore deve dividere il flusso in due rami:

1. Il ramo **True**, dove assumiamo che $x > 0$.
2. Il ramo **False**, dove assumiamo che $x \leq 0$ (la negazione).

Tradizionalmente, questo viene fatto usando l'operatore `filter` (o `assume`).

Definizione 28.1 (**Filter Operator**)**.**

$$\text{filter} : A \times \text{Pred} \to A$$

Dato uno stato astratto A e un predicato c, restituisce un nuovo stato astratto raffinato in cui c è vero.

28.1 Inefficienze dell'approccio classico

Per gestire un `if`, l'analizzatore deve fare queste operazioni: 1. **Copia:** Duplicare lo stato astratto A in memoria (perché serve sia per il ramo True che per il False). 2. **Filter Positivo:** Calcolare $\texttt{filter}(A, c)$. 3. **Filter Negativo:** Calcolare $\texttt{filter}(A, \neg c)$.

In parole semplici: *Immagina di dover dividere un mazzo di carte in "Rosse" e "Nere". L'approccio classico è: 1. Fai una fotocopia dell'intero mazzo. 2. Sul mazzo originale, scorri e butta via le Nere (tieni le Rosse). 3. Sulla fotocopia, scorri e butta via le Rosse (tieni le Nere). È uno spreco! Hai duplicato il lavoro e lo spazio.*

29 L'Operatore Split

Per risolvere queste inefficienze, è stato introdotto l'operatore **Split**.

Definizione 29.1 (**Split Operator**)**.** L'operatore split è una funzione che processa il vincolo e la sua negazione in un unico passaggio atomico:

$$\text{split} : A \times \text{Pred} \to A \times A$$

Prende in input uno stato astratto e un predicato, e restituisce una **coppia** di stati: (A_{true}, A_{false}).

29.1 Vantaggi dello Split

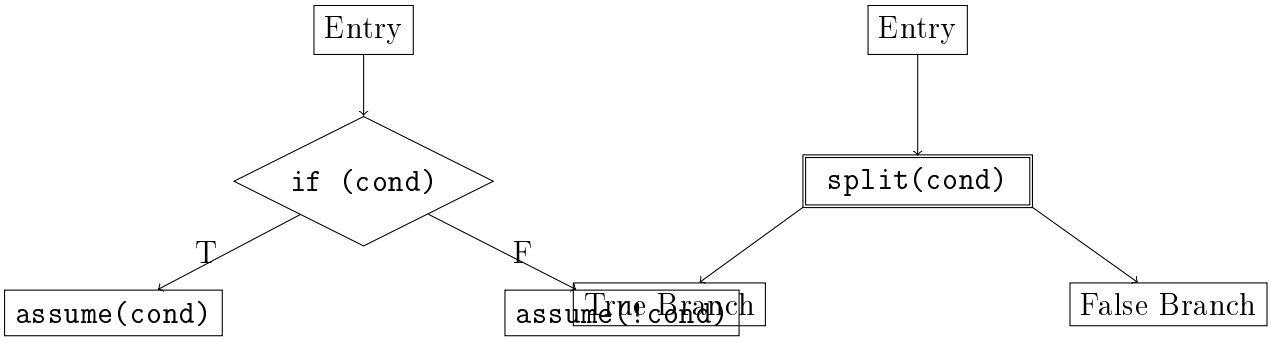
L'uso di `split` porta due grandi vantaggi:

- **Eliminazione del lavoro duplicato:** Il dominio astratto (es. Poliedri) può calcolare le due uscite simultaneamente, riutilizzando i calcoli intermedi.
- **Gestione della memoria:** Non serve copiare esplicitamente lo stato astratto prima dell'operazione. La gestione della memoria è delegata internamente alla libreria del dominio (che può essere più intelligente e fare copie "lazy" o parziali).

In parole semplici: *Tornando all'esempio delle carte: lo Split prende il mazzo e, in un solo passaggio, sposta le Rosse a sinistra e le Nere a destra. Nessuna fotocopia inutile, nessun doppio passaggio.*

30 Implementazione nel CFG

L'adozione dello Split richiede una modifica strutturale al **Control Flow Graph (CFG)** usato dall'analizzatore.



Standard CFG (Filter) **Optimized CFG (Split)**

- **Approccio Classico:** Il nodo condizionale ha due archi uscenti. Su ogni arco c'è un'istruzione `assume` (filter) separata. L'analizzatore deve visitare prima un arco, poi tornare indietro, copiare lo stato e visitare l'altro.
- **Approccio Split:** Il nodo condizionale è sostituito da un blocco `split(cond)`. Questo blocco calcola internamente le due uscite e le propaga direttamente ai successori.

31 Valutazione Sperimentale

L'approccio è stato implementato in framework reali (come *LiSA* o *Crab*) e testato. I risultati mostrano:

- **Miglioramento del tempo di analisi:** Riduzione significativa dei tempi di calcolo.
- **Riduzione della memoria:** Minore picco di utilizzo della RAM grazie alla gestione ottimizzata delle copie degli stati.

Possibile domanda d'esame:
Spiegare la differenza concettuale e operativa tra l'operatore `filter` e l'operatore `split`. Perché `split` è considerato un'ottimizzazione?

32 Introduzione all'Analisi Dataflow

L'analisi Dataflow è una tecnica per raccogliere informazioni sullo stato del programma in ogni punto della sua esecuzione, senza eseguirlo realmente. Si basa sul **Control Flow Graph (CFG)**.

L'idea chiave è definire come l'informazione:

1. Viene **Generata (GEN)** da un'istruzione.
2. Viene **Uccisa (KILL)** da un'istruzione.
3. Fluisce attraverso il grafo (unendo i rami).

32.1 Monotone Frameworks

La maggior parte delle analisi dataflow rientra nel quadro dei "Monotone Frameworks". Un framework è definito da:

- Un reticolo completo L (il dominio dei dati, es. insiemi di variabili).
- Funzioni di trasferimento $f_n : L \to L$ (come cambia lo stato nel nodo n).

33 Classificazione delle Analisi

Le analisi si distinguono in base a due assi fondamentali:

- **Direzione del Flusso:**
  - **Forward:** L'informazione segue le frecce del CFG (dall'inizio alla fine).
  - **Backward:** L'informazione risale le frecce (dalla fine all'inizio).
- **Tipo di Approssimazione (Join):**
  - **May ($\cup$):** L'informazione è valida se è vera in *almeno un* percorso ("potrebbe succedere").
  - **Must ($\cap$):** L'informazione è valida solo se è vera in *tutti* i percorsi ("deve succedere").

	May (Unione $\cup$)	**Must (Intersezione $\cap$)**
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Very Busy Expressions

Possibile domanda d'esame:
Riempire la tabella di classificazione delle analisi dataflow indicando per ogni cella il nome dell'analisi corrispondente.

34 Le 4 Analisi Classiche

34.1 1. Reaching Definitions (RD)

Tipo: Forward, May ($\cup$). **Obiettivo:** Per ogni punto del programma, determinare quali assegnamenti ($x := \dots$) fatti in precedenza potrebbero ancora essere validi (non sovrascritti).

- **Gen(n):** Se n è $x := a$, genera la definizione n.
- **Kill(n):** Se n è $x := a$, uccide tutte le altre definizioni di x nel programma.
- **Equazione:** $Out(n) = Gen(n) \cup (In(n) \setminus Kill(n))$.
- **Merge:** $In(n) = \bigcup_{p \in pred(n)} Out(p)$.

In parole semplici: *Serve per il "debugging" (chi ha dato questo valore a x?) e per trovare variabili non inizializzate.*

34.2 2. Available Expressions (AE)

Tipo: Forward, Must ($\cap$). **Obiettivo:** Capire quali espressioni (es. $a+b$) sono state già calcolate e non modificate.

- **Gen(n):** Se n calcola $a+b$, genera l'espressione $a+b$.
- **Kill(n):** Se n modifica a o b, uccide tutte le espressioni che contengono a o b.
- **Merge:** $In(n) = \bigcap_{p \in pred(n)} Out(p)$. (Deve essere disponibile da *tutti* i percorsi).

In parole semplici: *Serve per l'ottimizzazione: se $a+b$ è disponibile, non serve ricalcolarlo, basta usare il valore vecchio.*

34.3 3. Live Variables (LV)

Tipo: Backward, May ($\cup$). **Obiettivo:** Capire se il valore attuale di una variabile x verrà letto in futuro.

- **Direzione:** Si parte da `Exit` e si torna indietro.
- **Gen(n):** Se n legge x (es. $y := x + 1$), rende x "viva".
- **Kill(n):** Se n sovrascrive x (es. $x := 5$), rende la vecchia x "morta" (perché il valore precedente viene perso prima di essere letto).
- **Equazione:** $In(n) = Gen(n) \cup (Out(n) \setminus Kill(n))$.
- **Merge:** $Out(n) = \bigcup_{s \in succ(n)} In(s)$.

In parole semplici: *Fondamentale per la **Dead Code Elimination**. Se assegno $x := 10$ ma poi x non è "viva" (nessuno la legge), posso cancellare l'istruzione.*

34.4 4. Very Busy Expressions (VBE)

Tipo: Backward, Must ($\cap$). **Obiettivo:** Capire quali espressioni verranno sicuramente calcolate in futuro su tutti i percorsi, prima che le loro variabili cambino.

In parole semplici: *Serve per la* ***Code Hoisting*** *(spostamento del codice). Se $a + b$ viene calcolato sia nel ramo* `then` *che nel ramo* `else`*, posso spostare il calcolo prima dell'*`if` *per risparmiare spazio (codice più piccolo).*

Possibile domanda d'esame:
Descrivere l'analisi Live Variables: specificare se è Forward/Backward, May/-Must e fornire l'equazione di trasferimento.

35 Il Costo dei Domini Relazionali

Abbiamo visto che i domini relazionali (come Poliedri o Ottagoni) sono molto precisi ma computazionalmente pesanti (es. complessità $O(n^3)$ o esponenziale rispetto al numero di variabili n).

In parole semplici: *Se il tuo programma ha 1000 variabili, usare i Poliedri è impossibile: la matrice delle relazioni sarebbe gigantesca e i calcoli lentissimi. Tuttavia, in molti punti del programma, la maggior parte delle variabili non ha vincoli interessanti (vale $\top$ o non è in relazione con le altre). È uno spreco di risorse tracciare relazioni per variabili "inutili".*

36 L'approccio LUV (Likely Unconstrained Variables)

Per risolvere questo problema, è stata introdotta una tecnica chiamata **LUV**. L'idea è identificare quali variabili sono "probabilmente non vincolate" (cioè il loro valore è $\top$) e rimuoverle temporaneamente dal dominio relazionale pesante.

36.1 Struttura e Componenti

L'approccio si basa su due componenti:

1. **Dominio Sottostante (Underlying Domain):** Il dominio relazionale costoso (es. Poliedri) che vogliamo ottimizzare.
2. **Oracolo (Oracle):** Un'analisi molto economica e veloce (es. un'analisi sintattica o basata su intervalli) che "predice" quali variabili sono $\top$.

Definizione 36.1 (LUV Wrapper). Il dominio LUV agisce come un "guscio" attorno al dominio relazionale. Prima di eseguire un'operazione costosa (come un Join o un Meet):

1. Interroga l'**Oracolo**: "Quali variabili sono $\top$?"
2. Semplifica lo stato del dominio relazionale rimuovendo quelle variabili (operazione di `forget` o `havoc`).
3. Esegue l'operazione sul dominio semplificato (che ora ha dimensione minore).

37 Funzionamento dell'Oracolo

L'Oracolo deve essere **veloce**. Non deve fare calcoli complessi, altrimenti perdiamo il vantaggio dell'ottimizzazione. Solitamente si basa su:

- **Analisi Sintattica:** Se una variabile è appena stata dichiarata ma non inizializzata, o è un input dell'utente (`read()`), è sicuramente non vincolata.
- **Analisi Non-Relazionale:** Si può eseguire una veloce analisi a Intervalli prima. Se l'intervallo di x è $[-\infty, +\infty]$, allora l'oracolo suggerisce che x è $\top$ anche per i poliedri.

37.1 Cosa succede se l'Oracolo sbaglia?

Qui entra in gioco il concetto di **Soundness**.

- **Caso 1: Oracolo Preciso.** L'oracolo dice che x è $\top$ e x lo è davvero. Risparmiamo tempo e il risultato è esatto.
- **Caso 2: Oracolo Impreciso (Pessimista).** L'oracolo dice che x è $\top$, ma in realtà x aveva un vincolo (es. $x > 5$).
  - L'analizzatore si fida dell'oracolo e "dimentica" il vincolo su x (lo imposta a $\top$).
  - **Conseguenza:** Perdiamo precisione (risultato più astratto).
  - **Sicurezza:** L'analisi rimane **Sound** (corretta), perché $\top$ include il valore vero (Sovra-approssimazione).

In parole semplici: *L'oracolo non deve essere perfetto, deve essere utile. Se ogni tanto "butta via" qualche informazione vera per sbaglio, l'analisi sarà meno precisa ma comunque sicura e molto più veloce. È un trade-off tra **Precisione** e **Performance**.*

38 Esempio di Ottimizzazione

Consideriamo un programma con variabili x, y, z.

$$\text{Stato attuale (Poliedri): } \{x < y, \quad z = 5\}$$

Arriva l'istruzione `w = input()`. Il sistema deve calcolare il nuovo stato.

- **Senza LUV:** Il dominio Poliedri aggiunge una dimensione per w. La matrice passa da 3×3 a 4×4. Calcola le relazioni (nessuna) per w. Costoso.
- **Con LUV:**
  1. L'Oracolo analizza `input()` e dice: "w sarà sicuramente non vincolata ($\top$)".
  2. L'Oracolo guarda z. Magari un'analisi precedente dice che z non viene usata nel prossimo blocco. Dice: "z è irrilevante".
  3. Il dominio LUV rimuove z e ignora w. Passa al dominio sottostante solo le variabili x, y.
  4. Il calcolo avviene su una matrice 2×2 (molto più veloce).

Possibile domanda d'esame:
Spiegare il ruolo dell'Oracolo nella tecnica LUV (Likely Unconstrained Variables). Cosa succede alla correttezza (soundness) e alla precisione dell'analisi se l'oracolo classifica erroneamente una variabile vincolata come "non vincolata"?

39 Architettura dei Domini in LiSA

In LiSA, l'astrazione è modulare. Non esiste un unico oggetto che rappresenta "lo stato del programma", ma una gerarchia di componenti che collaborano.

39.1 Lo Stato Astratto (AbstractState)

L'interfaccia principale è `AbstractState`. Essa combina due informazioni fondamentali:

1. **Heap Domain:** Dove sono i dati in memoria? (Gestione di puntatori, alias, heap).
2. **Value Domain:** Quanto valgono i dati? (Intervalli, Segni, ecc.).

In parole semplici: *Immagina un oggetto Java* `p` *con un campo* `x`.

- *L'**Heap Domain** dice: "Il campo* `x` *dell'oggetto* `p` *corrisponde alla locazione di memoria astratta* `loc_1`*".*
- *Il **Value Domain** dice: "La locazione* `loc_1` *ha valore nell'intervallo* $[0, 10]$*".*

40 Semantic Domain e Value Domain

Per implementare un nuovo dominio numerico (es. Segni) in LiSA, bisogna estendere l'interfaccia `ValueDomain` (o una sua sottoclasse).

40.1 Metodi Fondamentali da Implementare

Ogni dominio deve fornire l'implementazione per le operazioni dell'Interpretazione Astratta:

40.1.1 1. Operazioni di Reticolo

- `lub(other)`: Calcola il Least Upper Bound (Join).
- `le(other)`: Verifica l'ordine parziale ($\sqsubseteq$).
- `top()` e `bottom()`: Restituiscono gli elementi $\top$ e $\bot$.

40.1.2 2. Semantica degli Statement

- `assign(id, expression, ...)`: Gestisce l'assegnamento $x = e$. Il dominio deve valutare l'espressione e e aggiornare il valore associato all'identificatore id.
- `smallStepSemantics(expression, ...)`: Valuta un'espressione aritmetica o logica. Es: Se l'espressione è `a + b`, il dominio deve chiamare la sua somma astratta.
- `assume(expression, ...)`: Gestisce i condizionali (es. `if (x > 0)`). Raffina lo stato astratto tenendo solo i valori che soddisfano la condizione.

Listing 1: Esempio di eval in un dominio astratto

```
@Override
public Domain eval(ValueExpression expression, ...) {
    if (expression instanceof BinaryExpression) {
        BinaryExpression bin = (BinaryExpression) expression;
        if (bin.getOperator() == NumericOperator.PLUS)
            // Chiama la somma astratta interna
            return left.plus(right);
    }
    return top(); // Se non so gestirlo, restituisco Top
}
```

41 Gestione della Memoria: Heap Domain

Se analizziamo linguaggi come Java o C, non abbiamo solo variabili intere, ma oggetti e puntatori. Il `ValueDomain` non sa cosa sia un puntatore.

41.1 Il Problema degli Alias

Codice: `a = new A(); b = a; b.x = 5;` L'analisi deve capire che modificando `b.x` stiamo modificando anche `a.x`.

41.2 Heap Domain in LiSA

L'interfaccia `HeapDomain` si occupa di tradurre espressioni complesse (accessi ai campi, array) in **Identificatori Astratti** univoci.

- **MonolithicHeap:** L'implementazione più semplice (e imprecisa). Tratta tutto l'heap come un unico "calderone". Ogni modifica a un campo potrebbe modificare qualsiasi altro campo. Molto veloce, ma poco preciso (soundness garantita, ma troppi falsi positivi).
- **PointBasedHeap:** Più precisa. Crea un grafo di allocazioni astratte. Distingue oggetti creati in punti diversi del codice (Allocation Sites).

Possibile domanda d'esame:
In LiSA, qual è la responsabilità del componente `HeapDomain` rispetto al `ValueDomain`? Perché sono separati?

42 Widening e Terminazione in LiSA

In LiSA, il widening non è applicato ciecamente a ogni iterazione. Si usa una strategia basata sui **WidenPoints**.

Definizione 42.1 (**Widening Point**)**.** I punti del CFG dove applicare il widening per rompere i cicli. Solitamente sono le **teste dei cicli** (loop headers).

Listing 2: Logica del Fixpoint in LiSA

```
// Pseudo-codice semplificato
State current = entryState;
State old = ...;

if (isLoopHead(node)) {
    // Se siamo in un ciclo, usiamo il widening per convergere
    current = old.widening(current);
} else {
    // Altrimenti usiamo il normale Join (Lub)
    current = old.lub(current);
}
```

In parole semplici: *LiSA calcola automaticamente dove sono i cicli nel grafo e applica il widening solo lì. Questo preserva la precisione nei rami senza cicli (dove il Join basta) e garantisce la terminazione dove serve.*

43 Introduzione ai Domini Numerici

In questo capitolo esploriamo come l'analisi statica gestisce i numeri. Poiché tracciare ogni singolo numero intero possibile è troppo costoso (o impossibile), usiamo delle astrazioni chiamate **Domini Numerici**.

Si dividono in due grandi famiglie:

1. **Non-Relazionali:** Astraggono il valore di ogni variabile separatamente (es. $x \in [0, 10]$). Sono veloci (lineari) ma meno precisi.
2. **Relazionali:** Astraggono le relazioni tra variabili (es. $x < y$). Sono più precisi ma molto più lenti (es. Poliedri, Ottagoni).

44 Il Dominio dei Segni (Sign Domain)

È il dominio più semplice. Ci interessa solo sapere se un numero è positivo, negativo o zero.

44.1 Definizione del Reticolo

Il dominio $D^\sharp$ è composto da:

- $\bot$: Insieme vuoto (impossibile).
- $\{< 0\}, \{= 0\}, \{> 0\}$: Segni base.
- $\geq 0, \leq 0, \neq 0$: Unioni dei precedenti.
- $\top$: Qualsiasi numero ($\mathbb{Z}$).

44.2 Aritmetica Astratta

Dobbiamo ridefinire le operazioni per questi simboli. Esempio della somma ($+^\sharp$):

$+$	$<\mathbf{0}$	$=\mathbf{0}$	$>\mathbf{0}$
$<\mathbf{0}$	< 0	< 0	$\top$
$=\mathbf{0}$	< 0	$= 0$	> 0
$>\mathbf{0}$	$\top$	> 0	> 0

In parole semplici: *La somma di un positivo e un negativo ($> 0 + < 0$) restituisce $\top$ perché il risultato dipende dai valori precisi (es. $5 + (-2) > 0$ ma $2 + (-5) < 0$). Qui perdiamo precisione.*

45 Il Dominio della Constant Propagation (CP)

Serve a capire se una variabile ha un valore costante fisso durante l'esecuzione.

45.1 Reticolo "Flat" (Piatto)

- **Top ($\top$):** La variabile non è costante (può assumere più valori).
- **Interi ($k \in \mathbb{Z}$):** La variabile vale esattamente k.
- **Bottom ($\bot$):** Codice morto.

Operazione di Join ($\sqcup$):

$$d_1 \sqcup d_2 = \begin{cases} d_1 & \text{se } d_1 = d_2 \\ \top & \text{se } d_1 \neq d_2 \end{cases}$$

Esempio: $5 \sqcup 5 = 5$, ma $5 \sqcup 3 = \top$.

46 Il Dominio degli Intervalli (Intervals)

È il dominio "standard" per l'analisi numerica. Rappresenta i valori come un range $[min, max]$.

46.1 Definizione

Un elemento astratto è una coppia $[l, u]$ dove $l \in \mathbb{Z} \cup \{-\infty\}$, $u \in \mathbb{Z} \cup \{+\infty\}$ e $l \leq u$.

46.2 Operazioni Principali

- **Join ($\sqcup$) - Unione:** È il più piccolo intervallo che contiene entrambi (Convex Hull).

$$[a, b] \sqcup [c, d] = [\min(a, c), \max(b, d)]$$

- **Meet ($\sqcap$) - Intersezione:** Serve per i filtri (es. `if (x > 5)`).

$$[a, b] \sqcap [c, d] = [\max(a, c), \min(b, d)]$$

- **Sottrazione Astratta:**

$$[a, b] -^\sharp [c, d] = [a - d, \quad b - c]$$

  Nota: Gli estremi si incrociano. Per ottenere il risultato minimo, devo prendere il minimo minuendo (a) e sottrarre il massimo sottraendo (d).

47 Il Problema della Terminazione (Widening)

Il dominio degli intervalli ha **catene ascendenti infinite**:

$$[0, 0] \sqsubseteq [0, 1] \sqsubseteq [0, 2] \cdots \sqsubseteq [0, \infty]$$

Un ciclo `while(x < 100) x++` potrebbe non terminare l'analisi in tempo utile.

47.1 Widening (∇)

L'operatore di Widening serve a "saltare" a infinito per garantire la terminazione. Dato un vecchio intervallo $[l_1, u_1]$ e uno nuovo $[l_2, u_2]$:

- Se $l_2 < l_1$ (scende) $\implies$ Nuovo limite $-\infty$.
- Se $u_2 > u_1$ (sale) $\implies$ Nuovo limite $+\infty$.
- Altrimenti mantieni i vecchi limiti.

48 Tabella Riassuntiva

Dominio	Tipo	Precisione	Costo	Catene Infinite?
Segni	Non-Rel	Bassa	Bassissimo	No
Costanti	Non-Rel	Media	Basso	No
Intervalli	Non-Rel	Alta	Medio	**Sì (Widening)**
Poliedri	Relazionale	Altissima	Esponenziale	Sì

Possibile domanda d'esame:
Perché nel dominio degli Intervalli l'operazione di sottrazione è definita come $[a, b] - [c, d] = [a - d, b - c]$ e non come $[a - c, b - d]$?
Risposta: Per garantire la **Soundness**. Il limite inferiore del risultato deve essere il valore più piccolo possibile. Per minimizzare $x - y$, devo prendere il x più piccolo possibile (a) e sottrarre il y più grande possibile (d).

49 Il Problema della Non-Terminazione

Molti domini astratti utili (come **Intervalli** o **Poliedri**) hanno un'altezza infinita. Questo significa che esistono catene di valori che crescono all'infinito senza mai stabilizzarsi.

Consideriamo questo ciclo semplice:

```
x = 0;
while (x < 100) {
    x++;
}
```

L'analisi classica (usando il Join $\sqcup$) calcola le iterazioni così:

1. Iterazione 0: $[0, 0]$
2. Iterazione 1: $[0, 0] \sqcup [1, 1] = [0, 1]$
3. Iterazione 2: $[0, 1] \sqcup [2, 2] = [0, 2]$
4. ...
5. Iterazione 99: $[0, 99]$

Se il ciclo fosse `while(true)`, l'analisi non finirebbe mai di calcolare $[0, \infty]$. Dobbiamo forzare la convergenza.

50 L'Operatore di Widening (∇)

Per risolvere il problema, sostituiamo l'operatore di Join ($\sqcup$) con il **Widening** (∇) nei punti di ciclo.

Definizione 50.1 (**Proprietà del Widening**). Un operatore $\nabla : A \times A \to A$ è un widening se:

1. **Soundness:** È un limite superiore (Upper Bound).
$$x \sqsubseteq (x \nabla y) \quad \text{e} \quad y \sqsubseteq (x \nabla y)$$
2. **Terminazione:** Per ogni sequenza crescente $x_0 \sqsubseteq x_1 \ldots$, la sequenza definita da $y_0 = x_0,\ y_{n+1} = y_n \nabla x_{n+1}$ si stabilizza in un tempo finito.

In parole semplici: *Il Join ($\sqcup$) è prudente: cerca il più piccolo insieme che contiene tutto. Il Widening (∇) è impaziente: se vede che un valore sta crescendo, "salta alle conclusioni" ed estende l'insieme fino all'infinito (o a un valore molto grande) per finire in fretta.*

50.1 Esempio: Widening sugli Intervalli

Dato un vecchio intervallo $I_1 = [l_1, u_1]$ e il risultato della nuova iterazione $I_2 = [l_2, u_2]$. Il widening controlla se i limiti sono stabili:

$$[l_1, u_1] \nabla [l_2, u_2] = [L, U]$$

Dove:

- $L = l_1$ se $l_2 \geq l_1$ (stabile), altrimenti $-\infty$.
- $U = u_1$ se $u_2 \leq u_1$ (stabile), altrimenti $+\infty$.

Riapplichiamolo al ciclo precedente:

1. Iterazione 0: $W_0 = [0, 0]$.
2. Iterazione 1: Calcolo $[0, 0] \sqcup [1, 1] = [0, 1]$.
3. Applico Widening: $W_0 \nabla [0, 1]$.
   - Limite inferiore $0 \to 0$ (Stabile).
   - Limite superiore $0 \to 1$ (Instabile/Cresciuto!).
4. Risultato: $[0, +\infty]$.
5. **Stop.** L'analisi ha trovato un punto fisso (stabile) in soli 2 passaggi.

51 Recuperare Precisione: Narrowing (Δ)

Il Widening è fantastico per terminare, ma è brutale. Nell'esempio sopra, il risultato è $x \in [0, +\infty]$. Ma il codice diceva `while(x < 100)`. Sappiamo che x non supererà mai 100. Abbiamo perso questa informazione "saltando" a infinito.

Per recuperarla, usiamo l'operatore di **Narrowing (Δ)**.

Definizione 51.1 (**Narrowing**). Il Narrowing serve a migliorare (restringere) un'approssimazione post-fixpoint ottenuta col widening. Si basa sul fatto che applicando un altro giro di semantica del ciclo partendo da $[0, +\infty]$, il vincolo `x < 100` taglierà via l'infinito.

51.1 Come funziona il ciclo completo

1. **Fase Ascendente (Widening):** Saliamo velocemente fino a trovare un'approssimazione sicura ma grossolana.

   $$\text{Risultato: } [0, +\infty]$$

2. **Fase Discendente (Narrowing):** Prendiamo $[0, +\infty]$ e lo facciamo passare *una volta sola* attraverso il corpo del ciclo (che contiene la guardia `x < 100`).

   $$[0, +\infty] \cap (-\infty, 99] = [0, 99]$$

   Il risultato raffinato $[0, 99]$ è ancora un punto fisso valido, ma molto più preciso.

52 Widening con Soglie (Thresholds)

Saltare subito a $\pm\infty$ è spesso troppo aggressivo. Una tecnica comune è il **Widening con Soglie**.

Invece di saltare a ∞, saltiamo a un insieme predefinito di valori "interessanti" $T = \{0, 1, 2, 5, 10, 100, \dots\}$.

Esempio: Se l'intervallo cresce da $[0, 1]$ a $[0, 2]$, invece di dire $[0, +\infty]$, proviamo a dire "forse arriva fino a 10?". Risultato: $[0, 10]$. Se poi supera anche 10, proviamo 100, e così via.

Possibile domanda d'esame:
Domanda: Spiegare perché l'uso del solo operatore di Join ($\sqcup$) non garantisce la terminazione dell'analisi statica sul dominio degli Intervalli. Qual è il ruolo del Widening e come interagisce con il Narrowing?
Risposta Chiave: Il dominio degli intervalli contiene catene ascendenti infinite (es. $[0, 1], [0, 2], \dots$). Il Join non può "indovinare" il limite e continua a calcolare passo passo. Il Widening garantisce la terminazione estrapolando il limite (spesso a ∞). Il Narrowing recupera la precisione persa rifinendo il risultato ottenuto dal Widening.

53 Perché serve il Narrowing?

Nel capitolo precedente abbiamo visto che il **Widening** (∇) ci permette di analizzare i cicli in tempo finito, ma spesso produce risultati troppo grossolani (sovrastima eccessiva).

Esempio classico:

```
x = 0;
while (x < 10) { x++; }
```

Il Widening, vedendo x crescere ($[0,1], [0,2] \dots$), "salta" a $[0, +\infty]$.

- **Soundness:** È corretto? Sì, tutti i valori reali $(0 \dots 10)$ sono in $[0, +\infty]$.
- **Precisione:** È preciso? No. Sappiamo che x non supererà mai 10.

Il **Narrowing** (Δ) serve a correggere questa sovrastima partendo dal risultato sicuro del Widening e migliorandolo.

54 Definizione e Funzionamento

Il Narrowing è un operatore che raffina un'approssimazione sicura (un Post-Fixpoint).

Definizione 54.1 (Operatore di Narrowing Δ). Un operatore $\Delta : A \times A \to A$ è un narrowing se, dato un punto fisso approssimato y (dove $F(y) \sqsubseteq y$):

$$y \Delta F(y) \sqsubseteq y$$

E garantisce che per ogni catena discendente $y_{n+1} = y_n \Delta F(y_n)$, questa non scenda mai "sotto" al vero punto fisso (preserva la soundness).

In parole semplici: *Il Narrowing prende il risultato "esagerato" del Widening (y) e prova a eseguire un altro giro del ciclo ($F(y)$). Se facendo un altro giro ottengo un risultato più piccolo di quello che avevo, significa che posso "stringere" il mio insieme senza perdere sicurezza.*

55 L'Algoritmo: Ascendente e Discendente

L'analisi statica di un ciclo avviene in due fasi distinte:

55.1 Fase 1: Sequenza Ascendente (Widening)

Si parte da $\bot$ e si usa ∇ per trovare rapidamente un limite superiore sicuro A.

$$X_0 = \bot, \quad X_{i+1} = X_i \nabla F(X_i)$$

Risultato: Un punto fisso A (es. $[0, +\infty]$) tale che $F(A) \sqsubseteq A$.

55.2 Fase 2: Sequenza Discendente (Narrowing)

Si parte da A e si usa Δ per scendere verso il punto fisso minimo reale.

$$Y_0 = A, \quad Y_{i+1} = Y_i \Delta F(Y_i)$$

56 Esempio Pratico Completo

Analizziamo il ciclo `while (x < 10) x++` con il dominio Intervalli.

56.1 1. Fase Ascendente (Widening)

- Iter 1: $[0, 0]$
- Iter 2: $[0, 0] \sqcup [1, 1] = [0, 1]$. Il limite sale.
- Widening: $[0, 0] \nabla [0, 1] = [0, +\infty]$.
- Verifica stabilità: Se entro con $[0, +\infty]$, il corpo del ciclo (limitato da $x < 10$) produce numeri fino a 9. Unito all'ingresso 0, il nuovo stato è $[0, 9]$.
- Poiché $[0, 9] \sqsubseteq [0, +\infty]$, abbiamo raggiunto un **Post-Fixpoint**. Ci fermiamo.

Stato attuale: $x \in [0, +\infty]$. (Sicuro, ma brutto).

56.2 2. Fase Discendente (Narrowing)

Ora applichiamo il Narrowing per stringere il risultato. Il Narrowing sugli intervalli è spesso definito semplicemente come l'intersezione: $[a, b] \Delta [c, d] = [a, b] \sqcap [c, d]$.

- **Start:** $Y_0 = [0, +\infty]$.
- **Calcolo $\mathbf{F}(Y_0)$:** Eseguiamo un giro del ciclo partendo da Y_0.
  - Filtro guardia ($x < 10$): L'intervallo diventa $[0, 9]$.
  - Corpo ($x++$): Diventa $[1, 10]$.
  - Unione con l'ingresso del ciclo ($x = 0$): $[0, 0] \sqcup [1, 10] = [0, 10]$.
- **Applicazione Narrowing:**

$$Y_1 = Y_0 \Delta F(Y_0) = [0, +\infty] \sqcap [0, 10] = [0, 10]$$

- **Risultato:** Abbiamo recuperato il limite superiore corretto!

57 Non-Terminazione del Narrowing

Attenzione: mentre il Widening **deve** garantire la terminazione, il Narrowing teoricamente potrebbe scendere all'infinito senza mai fermarsi (se il dominio ha catene discendenti infinite, come i Razionali $\mathbb{Q}$ o i Reali $\mathbb{R}$).

Tuttavia, nel dominio degli Intervalli Interi ($\mathbb{Z}$), le catene discendenti sono finite o si stabilizzano presto. Nella pratica degli analizzatori statici (come LiSA), si applica il Narrowing solo per un numero fissato di passi (es. 2 o 3 volte) e poi ci si accontenta.

Possibile domanda d'esame:
Domanda: Che relazione c'è tra Widening e Narrowing? È possibile usare il Narrowing senza aver prima usato il Widening?
Risposta: Widening e Narrowing sono complementari.

- Il Widening calcola un'approssimazione **dal basso verso l'alto** (over-approximation) per garantire la terminazione.
- Il Narrowing raffina quell'approssimazione **dall'alto verso il basso** per recuperare precisione.

Non ha senso usare il Narrowing senza Widening (o senza un post-fixpoint di partenza), perché il Narrowing richiede di partire già da un insieme "sicuro" che contiene la soluzione. Se partissi da $\bot$, non avrei nulla da restringere.

58 Definizione e Struttura

I **Domini Non Relazionali** (o a *Attributi Indipendenti*) sono il modo più semplice per analizzare lo stato di un programma con multiple variabili.

L'idea fondamentale è l'**astrazione indipendente**: ogni variabile viene analizzata isolatamente, dimenticando completamente le relazioni tra di esse.

58.1 Lifting Funzionale

Se abbiamo un dominio astratto di base D (es. Intervalli), lo stato astratto del programma $S^\sharp$ è definito come una funzione che mappa ogni variabile del programma (Var) in un elemento di D.

$$S^\sharp : Var \to D$$

Oppure, visto come prodotto cartesiano:

$$S^\sharp = D \times D \times \cdots \times D \quad (|Var| \text{ volte})$$

In parole semplici: *Immagina un foglio Excel.*

- *La colonna A descrive la variabile x (es. è positiva).*
- *La colonna B descrive la variabile y (es. è pari).*

Non c'è nessuna cella che dice "La colonna A è maggiore della colonna B". Le colonne non si parlano.

59 Operazioni Puntuali (Pointwise Operations)

Poiché le variabili sono indipendenti, tutte le operazioni di reticolo (Join, Meet, Order) vengono eseguite "punto per punto" (variabile per variabile).

Dati due stati S_1 e S_2:

- **Ordinamento ($\sqsubseteq$):** $S_1 \sqsubseteq S_2 \iff \forall v \in Var,\ S_1(v) \sqsubseteq_D S_2(v)$.
- **Join ($\sqcup$):** $(S_1 \sqcup S_2)(v) = S_1(v) \sqcup_D S_2(v)$.
- **Widening (∇):** $(S_1 \nabla S_2)(v) = S_1(v) \nabla_D S_2(v)$.

Esempio (Intervalli):

$$\begin{aligned} S_1 &= \{x \in [0,2],\ y \in [10,20]\} \\ S_2 &= \{x \in [2,4],\ y \in [10,10]\} \\ S_1 \sqcup S_2 &= \{x \in [0,4],\ y \in [10,20]\} \end{aligned}$$

Calcoliamo il join di x con x, e di y con y.

60 Il Limite della Precisione (Relazioni)

Il difetto principale è l'incapacità di rappresentare vincoli che coinvolgono più variabili contemporaneamente.

Esempio del "Quadrato":

$$x \in [0, 10], \quad y \in [0, 10]$$

Geometricamente, questo stato rappresenta un quadrato nel piano cartesiano. Se aggiungiamo il vincolo (filtro):

$$\text{assume}(x == y)$$

Un dominio non relazionale **non può imparare nulla** da questo vincolo (o impara pochissimo).

- Non può dire "adesso so che x è uguale a y".
- Continuerà a dire $x \in [0, 10]$ e $y \in [0, 10]$.
- Geometricamente, include ancora punti come $(x = 2, y = 8)$, che violano la condizione $x = y$.

In parole semplici: *Un dominio non relazionale approssima qualsiasi figura geometrica complessa usando solo un **Iper-Rettangolo** (o "Box") allineato agli assi. Non può rappresentare linee oblique o cerchi.*

61 Gestione di Bottom (Smash Product)

Un dettaglio tecnico cruciale riguarda la gestione dello stato $\bot$ (impossibile).

Se analizzando un programma scopriamo che **una sola** variabile x assume il valore $\bot$ (insieme vuoto), cosa succede all'intero stato?

Definizione 61.1 (**Smash Product**)**.** In un dominio ben formato, se anche solo una variabile è $\bot$, l'intero stato collassa a $\bot$ (Bottom).

$$(\dots, v_i \mapsto \bot, \dots) \equiv \bot_{state}$$

In parole semplici: *Se la variabile x è "impossibile" (insieme vuoto), significa che l'esecuzione non può aver raggiunto quel punto. Se non siamo lì, allora non esistono nemmeno y o z. L'intero stato è irraggiungibile.*

62 Confronto: Relazionale vs Non-Relazionale

Caratteristica	Non-Relazionale	Relazionale
Esempi	Segni, Intervalli, Costanti	Poliedri, Ottagoni, Congruenze
Rappresentazione	$Var \to D$ (Funzione)	Matrici, Grafi, Sistemi di eq.
Forma Geometrica	Scatole (Box) assi-parallele	Forme con lati obliqui
Complessità	Lineare $O(\|Var\|)$	Cubica o Esponenziale
Precisione	Bassa su $x = y$ o $x < y$	Alta

Possibile domanda d'esame:
Domanda: Dato lo stato astratto di Intervalli $S = \{x \in [0, 10], y \in [0, 10]\}$, quale sarebbe il risultato dell'operazione assume$(x < y)$ in un dominio non relazionale puro?
Risposta: Nella maggior parte dei casi, lo stato rimarrebbe invariato o subirebbe minime modifiche. Un dominio non relazionale non può memorizzare la relazione $x < y$. Potrebbe al massimo raffinare gli estremi se x fosse $[100, 200]$ e y $[0, 10]$, scoprendo un'incompatibilità ($\perp$). Ma con $[0, 10]$ per entrambi, il dominio non può escludere i casi in cui $x \geq y$ senza perdere la rappresentazione a intervalli indipendenti.