Analisi del Software

Corso tenuto dalla Professoressa Isabella Mastroeni

Università degli Studi di Verona

Alessio Gjergji

Indice

Capitolo 1

Analisi

L'analisi può essere costruita sia mediante l'elaborazione del codice sorgente che basandosi sul modello del programma. La scelta tra questi approcci dipende dalla situazione specifica.

Il Concetto di `CFG` (*Control Flow Graph, Grafo di Flusso di Controllo*) è affidabile per l'analisi statica, ma la sua precisione può variare a seconda di come viene costruito il modello del programma.

Ecco una panoramica delle due principali modalità di analisi:

- **Analisi Statica** (*senza esecuzione del codice*): Questo tipo di analisi è basato sulla struttura del codice sorgente e non richiede l'esecuzione del programma. Tuttavia, la sua principale limitazione è la precisione. L'analisi statica può fornire risultati decisi in modo certamente, ma può non essere completamente accurata nel catturare il comportamento reale del programma. È un'analisi basata sulla "visione teorica" del codice.
- **Analisi Dinamica** (*basata sull'esecuzione del codice*): In questo caso, l'analisi si basa sull'esecuzione effettiva del programma. Ciò fornisce una visione più accurata del comportamento, ma presenta sfide legate alla non terminazione e alla limitazione nell'eseguire tutti gli input possibili. L'analisi dinamica è solitamente basata su una serie limitata di input e non può garantire una copertura completa del comportamento del programma.

Entrambi gli approcci hanno limitazioni dovute al concetto di "non decidibilità" (**teorema di Rice**), che impedisce di ottenere risposte certe per alcune proprietà semantiche dei programmi.

L'obiettivo ideale dell'analisi sarebbe avere un sistema in grado di analizzare automaticamente qualsiasi programma in un linguaggio Turing completo come $\mathcal{L}$, fornendo risposte certe per tutte le proprietà semantiche in un tempo finito. Tuttavia, per ottenere un'analisi realistica, spesso è necessario fare delle compromissioni:

- **Automazione limitata**: Talvolta, per rendere l'analisi fattibile, è necessario rinunciare all'automazione completa e utilizzare un analizzatore solo su una classe limitata di programmi.

- **Accettazione dell'imprecisione controllata**: In alternativa, è possibile accettare un certo grado di imprecisione controllata nell'analisi, astrazione del risultato. Questo può essere fatto in modo completamente automatizzato ed è una caratteristica comune dell'analisi statica.

Tuttavia, è importante notare che togliendo l'automazione si perde la garanzia di rilevare tutti gli errori e le imprecisioni nell'analisi. Accettando l'imprecisione controllata, è comunque possibile ottenere risultati utili nell'ambito dell'analisi statica.

1.1 Teorema di Rice

Il Teorema di Rice è un importante risultato nella teoria della calcolabilità che dimostra le limitazioni fondamentali della verifica delle proprietà dei programmi. Il teorema afferma che, per qualsiasi proprietà non banale π di programmi, non esiste un algoritmo generale che possa determinare in modo decidibile se un programma p soddisfa la proprietà π. In altre parole, non è possibile costruire un analizzatore che, dato un programma p, determini in modo decidibile se p soddisfa π.

1.1.1 Definizione Formale

Per comprendere meglio il Teorema di Rice, introduciamo una definizione formale:

Sia $\mathcal{L}$ un linguaggio di programmazione e consideriamo l'insieme $\mathcal{P}$ di tutti i programmi validi scritti in $\mathcal{L}$. Ogni programma p in $\mathcal{P}$ è rappresentato da un numero naturale che funge da codifica univoca.

Definiamo una "proprietà di programma" π come un sottoinsieme di $\mathcal{P}$. In altre parole, π è un insieme di programmi che soddisfano una certa caratteristica o comportamento specifico.

Il Teorema di Rice afferma che, per ogni proprietà non banale π (ovvero, π non è vuota e $\pi \neq \mathcal{P}$), l'insieme $p \in \mathcal{P}, |, p$ soddisfa π è indecidibile.

Teorema di Rice

Sia $\mathcal{L}$ un linguaggio di programmazione Turing completo e sia π una proprietà non banale di programmi in $\mathcal{L}$. Allora, l'insieme $p \in \mathcal{L}$, p soddisfa π è indecidibile.

$$\forall \pi \text{ non banale } \exists \mathcal{L} \text{ Turing completo t.c. } \{p \in \mathcal{L} \mid p \text{ soddisfa } \pi\} \text{ è indecidibile}$$

Questo significa che non esiste un algoritmo generale che, dato un programma p, possa decidere in modo algoritmico se p appartiene all'insieme dei programmi che soddisfano la proprietà π.

1.1.2 Conseguenze

Il Teorema di Rice ha importanti conseguenze nella teoria della calcolabilità e nella verifica dei programmi. Dimostra che molte domande sulla correttezza o sul comportamento dei programmi non possono essere risolte in modo algoritmico generale. In altre parole, esistono

limitazioni intrinseche alla capacità di analizzare automaticamente i programmi per determinate proprietà. Questo teorema sottolinea l'importanza delle restrizioni sulle classi di problemi che possono essere risolti da algoritmi generici.

1.2 Approssimazione

L'approssimazione in analisi implica che una risposta non completamente accurata può ancora essere accettabile, a condizione che l'errore sia riconosciuto e possa essere descritto in modo controllato. Quindi, è importante distinguere tra l'inaccuratezza, che indica una leggera deviazione dalla precisione, e l'errore, che rappresenta una violazione significativa delle aspettative. π proprietà di analizzare, p programma, quindi Analisi$_\pi(p)$.

Nel contesto dell'analisi, consideriamo una proprietà π da analizzare su un programma p, che denotiamo come Analisi$\pi(p)$. Questo significa che per ogni programma p nel linguaggio $\mathcal{L}$, l'output dell'analisi, ovvero Analisi$\pi(p)$, sarà "true" solo se il programma p soddisfa la proprietà π. In altre parole, l'analisi ci dice se un programma rispetta la proprietà o meno.

$$\forall p \in \mathcal{L} \qquad Analisi_\pi(p) = true \Leftrightarrow p\, soddisfa\, \pi$$

Tuttavia, è importante notare che non sempre è possibile ottenere una bi-implicazione perfetta tra l'analisi e la proprietà. Ciò significa che, in alcuni casi, l'analisi potrebbe non essere in grado di fornire una risposta definitiva riguardo alla proprietà π, ma può comunque essere utilizzata per valutare in modo approssimativo e controllato l'aderenza del programma alla proprietà. Questo compromesso tra completezza e precisione è spesso necessario nell'ambito dell'analisi statica per rendere l'analisi praticamente realizzabile.

Soundness (*Correttezza*)

La correttezza (*o soundness*) dell'analisi implica che se l'output dell'analisi, cioè $Analisi_\pi(p)$, è "true," allora il programma p deve effettivamente soddisfare la proprietà π.

$$Analisi_\pi(p) = true \Rightarrow p\, soddisfa\, \pi$$

In altre parole, se l'analisi dice che un programma soddisfa la proprietà, questa affermazione è accurata. Tuttavia, è importante notare che se l'analisi restituisce "false," l'analizzatore potrebbe sovrastimare i programmi che non soddisfano la proprietà, includendo programmi che in realtà potrebbero soddisfarla. In termini pratici, ciò significa che l'analisi potrebbe essere conservativa nel rifiutare alcuni programmi.

Completezza

La completezza implica che se un programma p soddisfa effettivamente la proprietà π, allora l'output dell'analisi, cioè $Analisi_\pi(p)$, deve essere "true."

$$Analisi_\pi(p) = true \Leftarrow p\, soddisfa\, \pi$$

quindi

$$p\, soddisfa\, \pi \Rightarrow Analisi_\pi(p) = true$$

In altre parole, se un programma rispetta la proprietà, l'analisi dovrebbe essere in grado di riconoscerlo come tale e restituire una risposta positiva. Questo garantisce che l'analisi non dia risultati falsi negativi, ossia non manchi di riconoscere programmi che soddisfano la proprietà.

In sintesi, la correttezza ci assicura che l'analisi non dia risultati falsi positivi, mentre la completezza ci assicura che l'analisi non dia risultati falsi negativi. Tuttavia, spesso è difficile ottenere sia la completa correttezza che la completa completezza in un'analisi, e quindi si deve trovare un equilibrio tra queste due proprietà.

Supponiamo di avere un programma p nel linguaggio $\mathcal{L}$ e desideriamo determinare se una determinata proprietà π vale su di esso. In generale, non è sempre possibile condurre un'analisi diretta su p a causa della complessità del codice. Per affrontare questa sfida, trasformiamo il nostro codice in un modello astratto su cui possiamo applicare strumenti automatici come grafi di flusso di controllo (`CFG`), automi, e così via. Su questi modelli, spesso disponiamo di tecniche algoritmiche decidibili per stabilire se una versione adattata della proprietà π, denotata come π', vale sul programma p. Questa trasformazione del codice in un modello astratto consente di sfruttare strumenti automatizzati per ottenere informazioni sulla proprietà π'.

La perdita di precisione si verifica quando la risposta definitiva ottenuta dal modello astratto non si traduce in modo accurato nella risposta ottenuta direttamente dal programma originale. La precisione si perde nella transizione dall'affermazione:

$$\pi' \text{ vale sul modello di } p \Longrightarrow \pi \text{ vale su } p$$

In altre parole, non sempre possiamo garantire che se la proprietà π' vale sul modello astratto di p, allora la proprietà π vale direttamente sul programma p senza errori o imprecisioni.

1.2.1 Confronto tra analisi statica e analisi dinamica nella verifica delle specifiche dei programmi

Immaginiamo di avere un programma e una specifica, rappresentata dalla proprietà π, che vogliamo verificare. Nell'analisi statica, esaminiamo la proprietà π direttamente nel codice sorgente del programma senza eseguirlo. In altre parole, effettuiamo un'analisi basata solo sulla struttura e sulla sintassi del codice.

D'altra parte, l'analisi dinamica prende in input il programma e valuta la relazione tra gli input forniti e gli output generati durante l'esecuzione del programma. In questo caso, l'analisi

dinamica può non avere conoscenza completa del codice sorgente, ma si concentra sulla valutazione del comportamento effettivo del programma attraverso l'esecuzione (*questo approccio è spesso chiamato "testing"*).

Quindi, mentre l'analisi statica si basa sull'analisi del codice sorgente aperto per determinare se la proprietà π è verificata, l'analisi dinamica si concentra sull'esecuzione del programma e sull'osservazione del comportamento in relazione agli input dati.

1.2.2 Classificazione delle tecniche di analisi in base alle caratteristiche rinunciate per superare la non decidibilità

Per affrontare le limitazioni imposte dal Teorema di Rice, le tecniche di analisi possono rinunciare a alcune caratteristiche chiave. Queste caratteristiche includono:

- **Automatico**: La capacità di eseguire l'analisi senza intervento umano.
- $\forall p \in \mathcal{L}$: La capacità di analizzare qualsiasi programma nel linguaggio $\mathcal{L}$.
- **Corretto**: La capacità di fornire risultati accurati e privi di errori.
- **Completo**: La capacità di coprire tutti i possibili casi e fornire risposte definitive.

A seconda delle caratteristiche a cui si rinuncia, emergono diverse classi di analisi:

- Verifica (*Model checking*): Questa tecnica si basa su insiemi finiti di stati o comportamenti del sistema. Rinuncia alla possibilità di analizzare tutti i programmi, ma rimane automatica ed è corretta e completa all'interno del modello specifico.
- Analisi Conservative (*Statiche*): Le analisi conservative cercano di estrarre informazioni in modo statico dal programma. Forniscono una semantica approssimata ma conservativa del programma, il che significa che le proprietà approssimate implicano le proprietà concrete. Questa tecnica è automatica, lavora su programmi rappresentabili finitamente ed è corretta ma non completa (*operando solo su specifiche proprietà*).
- Bug finding (*Debugging*): Il debugging è una tecnica di supporto per gli sviluppatori che può fornire risposte con perdita sia di correttezza che di completezza. È principalmente utilizzato per identificare e risolvere errori durante lo sviluppo del software.
- Testing: Il testing è una tecnica dinamica che comporta l'esecuzione del programma su un insieme selezionato di input. Non ha limitazioni sul tipo di programmi che può analizzare, ma rinuncia alla correttezza, intesa come copertura completa degli input. Quando un test rileva una violazione della proprietà, si può concludere che la proprietà non è soddisfatta.

Queste diverse classi di analisi offrono trade-off tra automazione, capacità di analisi, precisione e completezza, e vengono utilizzate in base alle esigenze specifiche di verifica e analisi dei programmi.

Capitolo 2

Modelliamo programmi

2.1 Un semplice linguaggio imperativo `IMP`

Definiamo il linguaggio $\mathcal{L}$ dove:

$$\mathbb{V} = \mathbb{Z}$$
$$\mathbb{X} = \texttt{Var}$$

$$\begin{aligned}
\textit{Exp}\ \mathbb{E} &::= n \in \mathbb{V} \mid x \in \mathbb{X} \mid \mathbb{E} \oplus \mathbb{E} \\
\textit{Bool}\ \mathbb{B} &::= \texttt{true} \mid \texttt{false} \mid \mathbb{E} \oplus \mathbb{E} \\
\textit{Com}\ \mathbb{C} &::= \texttt{skip} \mid \mathbb{X} := \mathbb{E} \mid \mathbb{C}; \mathbb{C} \mid \texttt{if}\ \mathbb{B}\ \texttt{then}\ \mathbb{C}\ \texttt{else}\ \mathbb{C} \\
&\quad \mid \texttt{while}\ \mathbb{B} \mid \texttt{input(x)} \\
\textit{Programma}\ \mathbb{P} &::= \mathbb{C}
\end{aligned}$$

2.1.1 La semantica di `IMP`

La semantica è uno strumento formale che permette di dare significato ai programmi.

Semantica operazionale

La semantica operazionale è uno strumento formale che fornisce significato ai programmi attraverso la descrizione del comportamento passo dopo passo dell'interprete. Questo significa che il significato di un programma è descritto dalla sequenza dei singoli passi di computazione che esso compie.

Semantica denotazionale

La semantica denotazionale attribuisce significato ai programmi tramite una funzione che associa a ciascun programma un valore. In termini matematici, possiamo rappresentare questa idea come segue:

$$\texttt{intput} \xrightarrow[\text{semantica}]{} \texttt{output}$$

In altre parole, esiste una funzione $[\![.]\!]$ che mappa l'input del programma all'output. Questa approccio è composito, il che significa che possiamo definire il significato di programmi composti in termini dei loro componenti, come segue:

$$[\![.]\!] : \texttt{intput} \to \texttt{output}$$

$$[\![P_1; P_2]\!] = [\![P_2]\!] \oplus [\![P_1]\!]$$

Queste due forme di semantica, operazionale e denotazionale, sono strumenti utili per comprendere il significato dei programmi in modo dettagliato e matematico.

2.1.2 Lo stato della memoria

Nel contesto della programmazione, lo "stato" rappresenta una fotografia istantanea della configurazione della macchina (*astratta*) su cui viene eseguito un programma. Questo stato descrive l'associazione tra le variabili del programma e i valori che contengono. Formalmente, possiamo rappresentare lo stato come una funzione $\mathbb{M}$ che mappa le variabili ($\mathbb{X}$) ai loro valori ($\mathbb{V}$), come segue:

$$\mathbb{M} : \mathbb{X} \to \mathbb{V}$$

Durante l'esecuzione di un programma, viene generata una sequenza di stati che riflettono come il programma modifica lo stato della memoria nel corso del tempo. Questa sequenza di stati è essenziale per comprendere come il programma funziona e come influisce sullo stato della macchina. Nel contesto della modellazione formale, spesso ci riferiamo a questo processo come "esecuzione".

Per descrivere l'evoluzione di uno stato durante l'esecuzione di un programma, utilizziamo un modello chiamato "sistema di transizione", che è rappresentato da una coppia $\langle \Sigma, \to \rangle$. In questa coppia, Σ rappresenta l'insieme degli stati possibili e $\to$ rappresenta la relazione che specifica come un determinato stato può transizionare in un altro stato a seguito dell'esecuzione di un'azione del programma. Questo modello è fondamentale per analizzare il comportamento dinamico di un programma e comprendere come le modifiche di stato si verificano nel corso dell'esecuzione.

2.1.3 Semantica delle transizioni di stato

La semantica è definita come l'insieme di tutte le possibili sequenze di transizioni di stato (*eventualmente infinite*) a partire dagli stati iniziali, ovvero le esecuzioni delle istruzioni di un programma, indicate da sequenze di stati nel sistema di transizione.

Fornisce il significato dei programmi attraverso l'esecuzione delle loro istruzioni su un interprete (*cioè componendo gli effetti delle istruzioni*).

Il modello matematico si basa sull'utilizzo delle tracce in un sistema di transizione.

2.1.4 Semantica delle espressioni

La semantica delle espressioni è definita come segue:

$$[\![E]\!] : \mathbb{M} \to \mathbb{V}$$

a partire dalla memoria in $\mathbb{M}$ restituisce:

Valori

Il valore in $\mathbb{V}$ rappresentato da e. In altre parole:

$$m \in \mathbb{M} \quad , n \in \mathbb{V}$$

$$[\![n]\!](m) = n$$

$$[\![x]\!](m) = m(x)$$

Quindi, per un'espressione composta:

$$[\![e_1 \oplus e_2]\!](m) = f_{\oplus}([\![e_1]\!](m), [\![e_2]\!](m))$$

Dove il simbolo $\oplus$ è il simbolo sintattico e $f_{\oplus}$ è la funzione semantica.

Booleani

Per i valori booleani:

$$b \in \mathbb{B} \qquad [\![tt]\!](m) = tt \qquad [\![ff]\!](m) = ff$$

$$[\![b_1 \oplus b_2]\!](m) = f_{\oplus}([\![b_1]\!](m), [\![b_2]\!](m))$$

2.1.5 Semantica dei comandi

La semantica dei comandi è definita come segue:

$$c \in \mathbb{C}. \quad [\![c]\!] : \mathbb{M} \to \mathbb{M}$$

`skip`

L'istruzione `skip` rappresenta un comando nullo che non modifica lo stato della memoria.

$$[\![\texttt{skip}]\!](m) = m$$

Composizione

La composizione di due comandi c_1 e c_2 esegue prima c_1 e poi c_2. La semantica della composizione è data da:

$$[\![c_1; c_2]\!](m) = [\![c_2]\!]([\![c_1]\!](m)) = [\![c_2]\!] \oplus [\![c_1]\!](m)$$

Assegnamento

L'assegnamento dell'espressione e alla variabile x modifica lo stato della memoria mappando x al valore di e in m.

$$[\![x := e]\!](m) = m[x \mapsto [\![e]\!](m)]$$

`input`

L'istruzione `input(x)` rappresenta l'input di un valore n nella variabile x all'interno dello stato della memoria m.

$$[\![\texttt{input(x)}]\!](m) = m[x \mapsto n] \quad n \in \mathbb{V}$$

If-then-else

L'istruzione condizionale `if b then c_1 else c_2` esegue c_1 se la condizione b è vera, altrimenti esegue c_2.

$$[\![\texttt{if} \quad b \quad \texttt{then} \quad c_1 \quad \texttt{else} \quad c_2]\!](m) = \begin{cases} [\![c_1]\!](m) & \text{se } [\![b]\!](m) = \texttt{true} \\ [\![c_2]\!](m) & \text{se } [\![b]\!](m) = \texttt{false} \end{cases}$$

While

L'istruzione `while b do c` rappresenta un ciclo che continua a eseguire c fintanto che la condizione b è vera. La semantica di `while` è definita come segue:

$$[\![\texttt{while} \quad b \quad \texttt{do} \quad c]\!](m) = \begin{cases} [\![\texttt{while} \quad b \quad \texttt{do} \quad c]\!]([\![c]\!](m)) & \text{se } [\![b]\!](m) = \texttt{true} \\ m & \text{se } [\![b]\!](m) = \texttt{false} \end{cases}$$

Il while può comportare un ciclo infinito. Per gestire questa eventualità, si utilizza il concetto di traccia del programma e il punto di programma, raccogliendo gli stati raggiunti fino a quel punto. Questo permette di lavorare con proprietà degli input invece di manipolare singoli valori, affrontando problemi legati all'infinito.

2.2 Semantica Transazionale

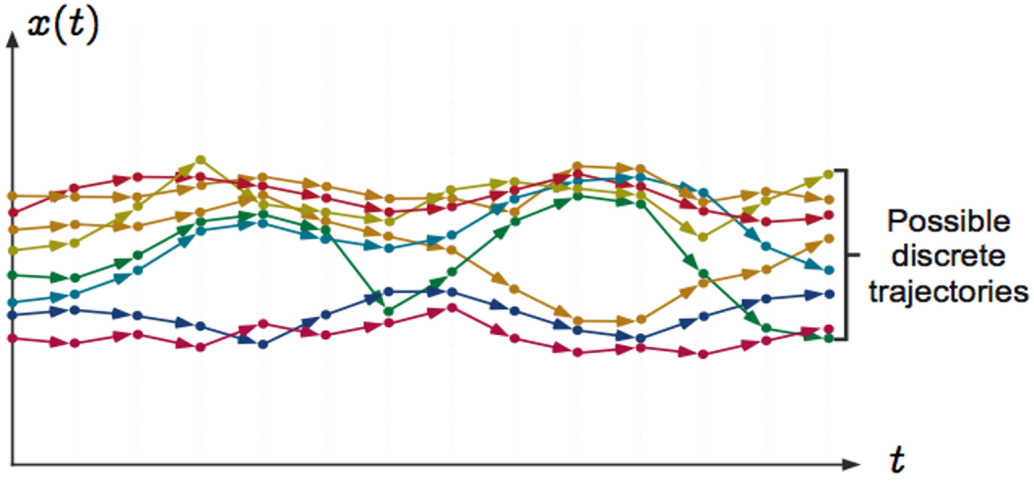
La semantica transazionale è un approccio alla comprensione del comportamento dei programmi attraverso la raccolta e l'analisi delle tracce di esecuzione. Questo approccio considera l'insieme di tutte le tracce di esecuzione possibili, partendo dagli stati iniziali del programma.

Questa raccolta di tracce è nota come "program trace semantics" (*semantica delle tracce di programma*).

Le tracce di programma forniscono una visione dettagliata dell'evoluzione del programma nel tempo, inclusi gli stati intermedi e le transizioni tra di essi. Questa analisi delle tracce è preziosa per comprendere come il programma risponde a diverse condizioni e input, e può rivelare informazioni importanti sul suo comportamento dinamico.

La semantica transazionale è particolarmente utile per affrontare problemi legati all'infinito, in quanto consente di lavorare con tracce e punti di programma invece di manipolare singoli valori. Questo approccio facilita la gestione delle esecuzioni potenzialmente infinite, fornendo una base solida per l'analisi formale dei programmi.

Nel complesso, la semantica transazionale fornisce uno strumento potente per la comprensione approfondita del comportamento dei programmi, evidenziando le variazioni nello stato della memoria nel corso dell'esecuzione e consentendo l'analisi delle proprietà attraverso l'osservazione delle tracce di programma.



2.3 Semantica come punto fisso

Semantica a punto fisso

Dato un dominio D di stati e una funzione F:

- D è un ordine parziale, cioè $\langle D, \leq \rangle$ è un *po-set*, dove D soddisfa le seguenti proprietà:
  - Riflessività: $\forall x \in D : x \leq x$.
  - Antisimmetria: $\forall x, y \in D : (x \leq y \land y \leq x) \Rightarrow x = y$.
  - Transitività: $\forall x, y, z \in D : (x \leq y \land y \leq z) \Rightarrow x \leq z$.
- $F : D \to D$ è una funzione totale e monotona, il che significa che per ogni x e y in D se $x \leq y$, allora $F(x) \leq F(y)$. Inoltre, la funzione F è iterabile, cioè può essere composta con se stessa più volte, ottenendo $F^n(x) = F(F(F(\ldots F(x)))$.

Un sistema di transizione è una coppia $\langle \Sigma, \tau \rangle$, dove Σ è un insieme non vuoto di stati e τ è una relazione di transizione che collega gli stati. In altre parole, un sistema di transizione rappresenta un insieme di stati e le relazioni tra di essi, ed è utilizzato per descrivere il comportamento di sistemi o programmi.

2.3.1 Punto fisso inferiore

- Il punto rosso $\odot$ rappresenta un stato bloccato.
- Il punto blu $\bullet$ rappresenta uno stato non bloccato.

Quindi, possiamo rappresentare l'evoluzione del sistema attraverso iterazioni. Iniziamo con un insieme vuoto di stati X^0: Nella prima iterazione, otteniamo l'insieme X^1 contenente uno stato bloccato:

$$X^0 = \varnothing$$

Nella prima iterazione, otteniamo l'insieme X^1 contenente uno stato bloccato:

$$X^1 = \{\odot\}$$

Nella seconda iterazione, aggiungiamo uno stato non bloccato con una transizione τ dall'insieme X^1 all'insieme X^2:

$$X^2 = \{\odot, \bullet \xrightarrow{\tau} \odot\} \qquad \text{dove } \{\odot\} \cup \bullet \xrightarrow{\tau} \{\odot\}$$

In questa iterazione, uno stato non bloccato può avanzare diventando uno stato bloccato. La notazione $\bullet \xrightarrow{\tau}$ indica una transizione che può verificarsi. Nella terza iterazione, continuiamo ad aggiungere stati e transizioni: Qui vediamo che gli stati non bloccati possono ancora avanzare tramite transizioni τ, ma alla fine possono diventare stati bloccati.

Tutti gli stati contenuti in X^3 rappresentano gli stati terminati del sistema, ossia quegli stati in cui il sistema non può avanzare ulteriormente.

Qui vediamo che gli stati non bloccati possono ancora avanzare tramite transizioni τ, ma alla fine possono diventare stati bloccati.

Tutti gli stati contenuti in X^3 rappresentano gli stati terminati del sistema, ossia quelli in cui il sistema non può avanzare ulteriormente.

La notazione finale, $lfp^{\subseteq}_{\varnothing} F^+$, rappresenta il calcolo del punto fisso inferiore di una funzione o di un operatore F in questo contesto. In questo calcolo, stiamo cercando il più piccolo insieme di stati che rimane invariato quando applichiamo l'operatore F iterativamente a partire da un insieme vuoto. Questo è fondamentale per identificare gli stati stabili o terminali in un sistema o un processo.

2.3.2 Punto fisso superiore

- Il punto rosso $\odot$ rappresenta uno stato bloccato.
- Il punto blu $\bullet$ rappresenta uno stato non bloccato.
- Il punto arancione $\bullet$ rappresenta uno stato non bloccato che può avanzare e diventare uno stato bloccato.

Ora, possiamo rappresentare l'evoluzione del sistema attraverso iterazioni. Iniziamo con un insieme iniziale X^0 che contiene uno stato non bloccato che può avanzare e diventare uno stato bloccato, con un numero di passi non definito:

$$X^0 = \{\bullet, \bullet \xrightarrow{?} \bullet, \bullet \xrightarrow{?} \bullet \xrightarrow{?} \bullet, \ldots, \bullet \xrightarrow{?} \bullet \ldots \bullet \xrightarrow{?} \bullet, \ldots\}$$

Nella prima iterazione, otteniamo l'insieme X^1, che include uno stato bloccato e uno stato non bloccato che può avanzare tramite una transizione τ:

$$X^1 = \{\odot, \bullet \xrightarrow{\tau} \bullet, \bullet \xrightarrow{\tau} \bullet \xrightarrow{?} \bullet, \ldots, \bullet \xrightarrow{\tau} \bullet \ldots \bullet \xrightarrow{?} \bullet, \ldots\}$$

Nella seconda iterazione, otteniamo l'insieme X^2, che include uno stato bloccato, uno stato non bloccato che può avanzare tramite una transizione τ, e uno stato non bloccato che può continuare a evolversi:

$$X^2 = \{\odot, \bullet \xrightarrow{\tau} \odot, \bullet \xrightarrow{\tau} \bullet \xrightarrow{\tau} \bullet, \ldots, \bullet \xrightarrow{\tau} \bullet \xrightarrow{\tau} \bullet \ldots \bullet \xrightarrow{?} \bullet, \ldots\}$$

Qui vediamo che gli stati non bloccati possono avanzare tramite transizioni τ, ma alla fine possono diventare stati bloccati.

L'insieme $\{\odot\} \cup \bullet \xrightarrow{\tau} \Sigma^+$ rappresenta il punto fisso superiore (*gfp*) in questo contesto. Il *gfp* rappresenta il più grande insieme di stati che rimane invariato quando applichiamo l'operatore Σ^+ iterativamente a partire da un insieme vuoto. In altre parole, è l'insieme più grande in cui gli stati non bloccati possono continuare a evolversi. Il *gfp* è fondamentale per identificare gli stati stabili o terminali in un sistema o un processo.

$$gfp^{\subseteq}_{\Sigma^\omega} F^\omega$$

2.3.3 Semantica dei comandi come punto fisso

La semantica dei comandi mappa un insieme di input in un insieme di stati.

$$[\![\mathbb{C}]\!]_{\wp} : \wp P(\mathbb{M}) \to \wp(\mathbb{M})$$

$$[\![\texttt{skip}]\!]_{\wp}(M) = M$$

$$[\![C_0; C_1]\!]_{\wp}(M) = [\![C_1]\!]_{\wp}([\![C_0]\!]_{\wp}(M))$$

$$[\![\texttt{x:= E}]\!]_{\wp}(M) = \{m[x \mapsto [\![E]\!]_M(m)] \mid m \in M\}$$

$$[\![\texttt{input(x)}]\!]_{\wp}(M) = \{m[x \mapsto n] \mid m \in M, n \in \mathbb{V}\}$$

$$[\![\texttt{if B then C else C'}]\!]_{\wp}(M) = [\![C_0]\!]_{\wp}(\mathcal{F}_B(M)) \cup [\![C_1]\!]_{\wp}(\mathcal{F}_{\neg B}(M))$$

$$[\![\texttt{while B do C}]\!]_{\wp}(M) = \mathcal{F}_{\neg B}\left(\bigcup_{i\geq 0}([\![C]\!]_{\wp} \circ \mathcal{F}_B)^i(M)\right)$$

Dove:

$$\mathcal{F}_B(M) = \{m \in M | [\![B]\!](m) = \texttt{true}\}$$

Semantica del ciclo

Dobbiamo partizionare l'esecuzione basandola sul numero di iterazioni che il ciclo esegue prima di uscire. L'insieme degli output è l'infinita unione della famiglia di insiemi M_i che denotano gli stati prodotti dal programma in esecuzione.

$$M_i = \mathcal{F}_{\neg B}\left(([\![C]\!]_{\wp} \circ \mathcal{F}_B)^i(M)\right)$$

Dove:

$$\bigcup_{i\geq 0} M_i = \bigcup_{i\geq 0} \mathcal{F}_{\neg B}\left(([\![C]\!]_{\wp} \circ \mathcal{F}_B)^i(M)\right) = \mathcal{F}_{\neg B}\left(\bigcup_{i\geq 0}([\![C]\!]_{\wp} \circ \mathcal{F}_B)^i(M)\right)$$

Notiamo che:

$$\mathcal{F}_{\neg B}(lfp_M F) \ \textit{dove}\ F : M' \mapsto M \cup [\![C]\!]_{\wp} \circ (\mathcal{F}_B(M'))$$

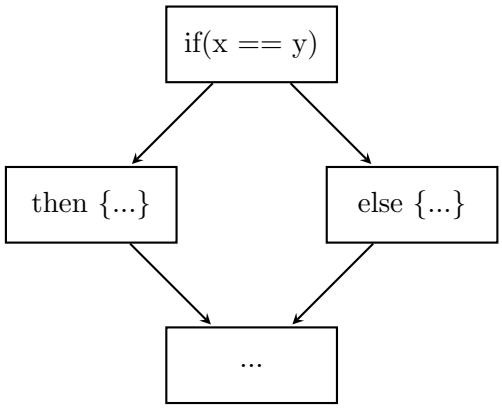
2.4 Il Control Flow Graph

Il *Control Flow Graph* (CFG) è un grafo diretto che rappresenta il flusso di controllo di un programma. Il grafo è generato dalla sintassi del programma. Lo scopo principale di tale grafo è quello di permettere di capire facilmente la struttura del codice rilevando codice morto, cicli infiniti, e altre caratteristiche del programma. È quindi utile per l'analisi statica del codice.

Il `CFG` è un grafo diretto $G = (N, E)$ dove:

- un nodo $n \in N$ rappresenta un blocco di codice, ovvero è una sequenza massimale di istruzioni con un singolo punto di ingresso, un singolo punto di uscita e senza diramazioni interne. Per semplicità, assumiamo un unico nodo d'ingresso n_0 e un unico nodo di uscita n_f.

- Un arco $e = (n_i, n_j) \in E$ rappresenta un possibile flusso di controllo tra due blocchi di codice.



(a) Esempio di CFG

```
if(x == y)
    then
        ...
    else
        ...
...
```

Figura 2.4.1: La figura generale

2.4.1 Blocchi di base

Blocco di base

Un blocco di base è la massima sequenza consecutiva di istruzioni senza diramazioni interne, con un singolo punto di ingresso, un singolo punto di uscita e senza salti all'interno del blocco.

Si tratta dell'unità di base per la costruzione del `CFG` e per l'analisi del flusso.

Le ottimizzazioni che è possibile attuare includono l'eliminazione della ridondanza e l'allocazione dei registri.

2.4.2 Identificare i blocchi di base

Questo è un processo di analisi del flusso di controllo per identificare i blocchi di base. Di seguito è riportata una spiegazione dettagliata basata sull'input fornito:

Identificazione dei leader:

- Il primo statement nella sequenza (*punto di ingresso*) è un leader.
- Ogni statement "s" che è la destinazione di un salto (*condizionale o incondizionale*) è un leader (*cioè esiste un "goto s"*).
- Ogni statement immediatamente successivo a un salto (*condizionale o incondizionale*) o a un return è un leader.

Creazione dei blocchi di base:

- Per ogni leader identificato, il suo blocco di base include il leader stesso e tutte le istruzioni fino al prossimo leader (*senza includerlo*) o fino alla fine del programma.

Questo processo consente di identificare i blocchi di base e di definire il flusso di controllo all'interno del programma.

2.4.3 Esempio

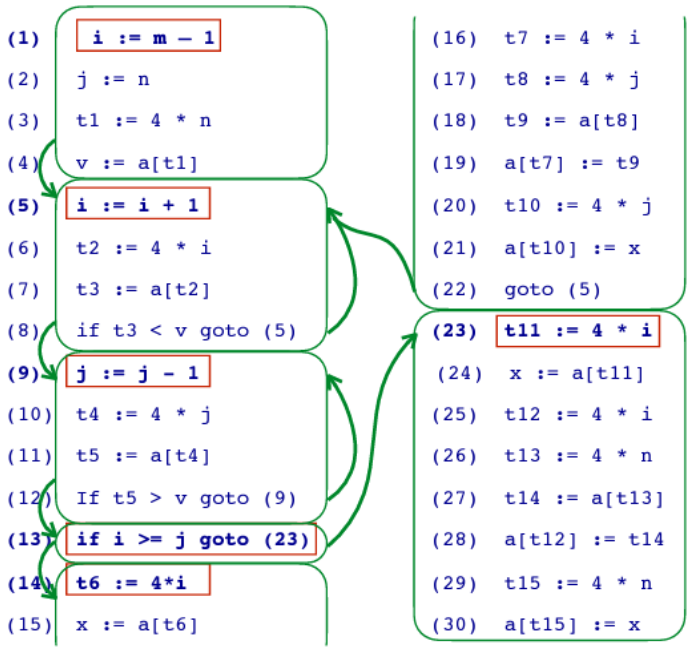
```
1. i := m - 1
2. j := n
3. t1 := 4 * n
4. v := a[t1]
5. i := m + 1
6. t2 := 4 * i
7. t3 := a[t2]
8. if t3 < v goto (5)
9. j:= j - 1
10. t4 := 4 * j
11. t5 := a[t4]
12. if t5 > v goto (9)
13. if i >= j goto (23)
14. t6 := 4 * i
15. x := a[t6]
16. t7 := 4 * i
17. t8 := 4 * j
18. t9 := a[t8]
19. a[t7] := t9
20. t10 := 4 * j
21. a[t10] := x
22. goto (5)
23. t11 := 4 * i
24. x := a[t11]
25. t12 := 4 * i
26. t13 := 4 * n
27. t14 := a[t13]
28. a[t12] := t14
29. t15 := 4 * n
30. a[t15] := x
```

```
1. i := m - 1
2. j := n
3. t1 := 4 * n
4. v := a[t1]
5. i := m + 1
6. t2 := 4 * i
7. t3 := a[t2]
8. if t3 < v goto (5)
9. j:= j - 1
10. t4 := 4 * j
11. t5 := a[t4]
12. if t5 > v goto (9)
13. if i >= j goto (23)
14. t6 := 4 * i
15. x := a[t6]
16. t7 := 4 * i
17. t8 := 4 * j
18. t9 := a[t8]
19. a[t7] := t9
20. t10 := 4 * j
21. a[t10] := x
22. goto (5)
23. t11 := 4 * i
24. x := a[t11]
25. t12 := 4 * i
26. t13 := 4 * n
27. t14 := a[t13]
28. a[t12] := t14
29. t15 := 4 * n
30. a[t15] := x
```

La partizione del codice intermedio in blocchi di base coinvolge diversi passaggi importanti per rappresentare il flusso di controllo in un programma. I passaggi chiave sono i seguenti:

- Aggiunta di archi corrispondenti ai flussi di controllo tra i blocchi.
- Trattamento dei costrutti come:
  - **Goto incondizionale**: Questo genera un collegamento diretto a un blocco specifico.
  - **Branch condizionale**: Può generare più archi uscenti da un blocco, a seconda delle possibili condizioni.
  - **Flusso sequenziale**: Se non ci sono ramificazioni alla fine di un blocco, il controllo passa semplicemente al blocco successivo.
- Aggiunta di nodi finti e archi, se necessario, per rappresentare i nodi di ingresso e di uscita nel caso in cui non siano unici.
- L'obiettivo è di semplificare al massimo gli algoritmi di analisi e trasformazione, assicurando che non ci siano archi che entrano nel nodo di ingresso n_0 o che escono dal nodo di uscita n_f.

Questi passaggi sono cruciali per modellare accuratamente il flusso di controllo all'interno di un programma e facilitare ulteriori analisi e ottimizzazioni.



Dato un `CFG` $= \langle N, E \rangle$, è definito come un insieme di nodi e archi, dove ogni arco rappresenta il flusso di controllo tra due nodi. Nel contesto di un `CFG` $=< N, E >$:

- Se esiste un arco $n_i \to n_j \in E$:
  - n_i è un predecessore di n_j
  - n_j è un successore di n_i
- Per qualsiasi nodo $n \in N$:
  - `Pred(n)`: l'insieme dei predecessori di n
  - `Succ(n)`: l'insieme dei successori di n
  - Un nodo di diramazione (`branch node`) è un nodo che ha più di un successore
  - Un nodo di unione, (`join node`) è un nodo che ha più di un predecessore

2.5 Il linguaggio `imp-CFG`

Spostando la caratteristica di controllo sulla struttura del grafo, il linguaggio non è più `IMP`, ma una versione leggermente modificata:

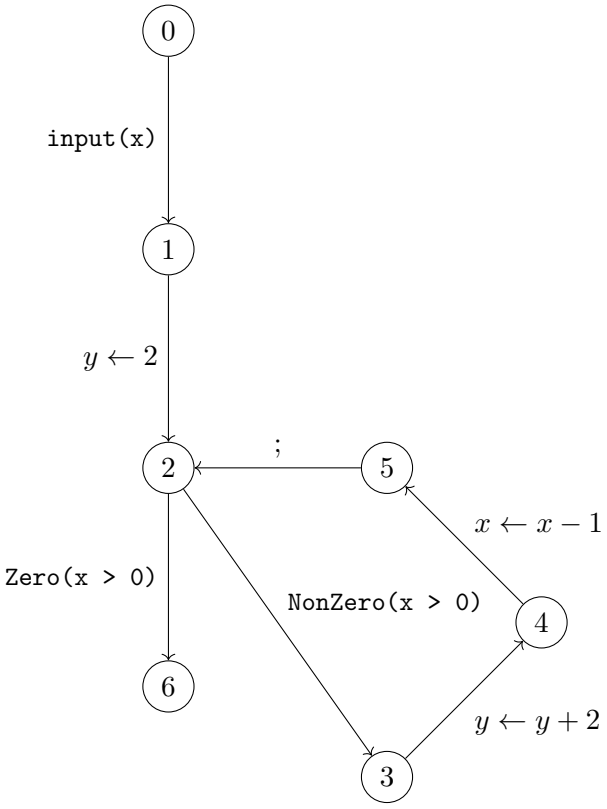
- I vertici corrispondono ai punti del programma
- Gli archi sono passi del calcolo etichettati con l'azione del programma corrispondente

- Le etichette delle istruzioni diventano etichette dei nodi

test:	`NonZero(e)` or `Zero(e)`
assignment:	$x \leftarrow e$
empty statement:	;
input:	`input(x)`

Esempio

```
input(x);
y := 2;
while(x > 0) {
  y := y + 2;
  x := x - 1;
}
```



Una dichiarazione condizionale o un ciclo all'interno di un grafo di flusso di controllo presenta due archi corrispondenti: l'arco etichettato con `NonZero` viene percorso se la condizione "e" è verificata (cioè se "e" viene valutata come un valore diverso da 0). L'arco etichettato con `Zero`, invece, viene percorso se la condizione non è soddisfatta.

Un arco è definito come $k = (u, \texttt{lab}, v)$, dove u rappresenta il vertice di partenza, v rappresenta il vertice di destinazione e `lab` rappresenta l'etichetta dell'arco. Questo arco rappresenta l'effetto della dichiarazione, ovvero la trasformazione dello stato prima dell'esecuzione dell'azione di etichettatura in uno stato successivo: **effetto dell'arco**.

2.5.1 Semantica del linguaggio `imp-CFG`

La semantica del linguaggio descrive la trasformazione dello stato prima e dopo l'esecuzione di un'azione del linguaggio, che viene riflessa nell'effetto complessivo della dichiarazione

all'interno del programma.

$$\begin{array}{lcll} [\![;]\!](m) & = & m & \\ [\![\texttt{NonZero(e)}]\!](m) & = & m & \textit{if } [\![e]\!](m) = \texttt{true} \\ [\![\texttt{Zero(e)}]\!](m) & = & m & \textit{if } [\![e]\!](m) = \texttt{false} \\ [\![x \leftarrow e]\!](m) & = & m[x \mapsto [\![e]\!](m)] & \\ [\![\texttt{input(x)}]\!](m) & = & m[x \mapsto m(x)] & \end{array}$$

2.5.2 Computazione del linguaggio `imp-CFG`

Una computazione è un percorso nel grafo di flusso di controllo, ovvero una sequenza di archi che iniziano dal nodo iniziale u e terminano in un nodo finale v. Il percorso è quindi una sequenza di archi:

$$\pi = k_1, k_2, \ldots, k_n = (u_i, \texttt{lab}_i, u_{i+1}), i = 1, \ldots, n-1, u = u_1, v = v_n$$

La trasformazione di stato corrispondente alle computazioni ottenuta dalla composizione degli effetti degli archi della computazione:

$$[\![\pi]\!] = [\![k_n]\!] \circ \cdots \circ [\![k_1]\!]$$

Capitolo 3

Significato di approssimare

3.1 L'idea di approssimazione

Immagina di avere due insiemi di oggetti: uno di questi, chiamiamolo $[\![P]\!]$, ha una caratteristica speciale che chiameremo Q. Ora, il punto cruciale è che non possiamo dire con certezza se un determinato oggetto appartiene a Q o meno. È come se avessimo un mucchio di oggetti e non riuscissimo a dire se uno specifico oggetto appartiene a un gruppo particolare o meno.

Quello che dobbiamo fare è trovare un modo per approssimare l'insieme $[\![P]\!]$ in modo da poter prendere decisioni più facili su Q. In altre parole, dobbiamo trovare un altro insieme, $[\![P]\!]^\sharp$, che contiene la maggior parte degli oggetti di $[\![P]\!]$, ma che sia più facile da analizzare. Questo insieme deve avere due caratteristiche importanti: tutti gli oggetti di $[\![P]\!]$ devono essere anche in $[\![P]\!]^\sharp$, e l'insieme $[\![P]\!]^\sharp$ deve essere tale che possiamo dire con certezza se un oggetto appartiene a Q o meno.

Quando abbiamo questo insieme $[\![P]\!]^\sharp$, possiamo utilizzarlo per fare deduzioni su Q. Se tutti gli oggetti in $[\![P]\!]^\sharp$ appartengono a Q, allora possiamo dire con sicurezza che tutti gli oggetti in $[\![P]\!]$ devono appartenere a Q. Ma se non tutti gli oggetti in $[\![P]\!]^\sharp$ appartengono a Q, non possiamo essere certi se gli oggetti in $[\![P]\!]$ appartengono o meno a Q.

In sostanza, il nostro obiettivo è rendere più facile prendere decisioni su questi oggetti, anche se non possiamo dire con certezza assoluta se un oggetto specifico appartiene a Q. Questo approccio ci consente di ragionare in modo più chiaro su questi insiemi e di trarre conclusioni ragionevoli su di essi.

La correttezza ci consente di sfruttare la decidibilità dell'approssimazione:

$$[\![P]\!] \subseteq Q \implies [\![P]\!]^\sharp \subseteq Q$$

Altrimenti, non possiamo saperlo con certezza!

3.1.1 Astrazione della semantica

Vediamo come costruire l'insieme $[\![P]\!]^\sharp$ a partire da $[\![P]\!]$. Specificheremo la semantica come una coppia: una funzione f (*con punto fisso*) e un dominio di calcolo D (*ordinato*).

- Astrazione del dominio di calcolo e delle relazioni tra oggetti concreti e astratti, ovvero l'osservazione astratta dei dati e come questi si relazionano tra loro.
- Astrazione del calcolo, con particolare attenzione all'astrazione del punto fisso, come la semantica manipola questi risultati astratti.

L'astrazione è il processo di sostituire qualcosa di concreto con una descrizione che considera alcune proprietà (*generalmente non tutte*), definita come modello astratto. Può descrivere alcune proprietà in modo preciso, ma non tutte.

Un'astrazione $\wp(\Sigma)$ di oggetti in Σ è $A \subseteq \wp(\Sigma)$ tale che:

- Gli elementi presenti nell'insieme A sono quelli descritti precisamente dall'astrazione, senza perdita di precisione.
- Gli elementi non presenti nell'insieme A devono essere rappresentati da altri elementi dell'insieme, con una perdita di precisione.

3.1.2 Oggetti

Nell'analisi/verifica dei programmi dobbiamo considerare oggetti che rappresentano parti dello stato di calcolo:

- Valori: Booleani, Interi,... $\mathcal{V}$
- Nomi di variabili $\mathbb{X}$
- Ambienti $\mathbb{X} \to \mathcal{V}$
- Stacks
- ...

Proprietà

Le proprietà sono insiemi di oggetti (che hanno quella proprietà). Esempi:

- Numeri naturali dispari: $\{1, 3, 4, \ldots, 2n + 1, \ldots\}$
- Numeri interi pari: $\{2z \mid z \in \mathbb{Z}\}$
- Valori delle variabili intere: $\{x \mid x \in \mathbb{X} \wedge \texttt{minint} < x < \texttt{maxint}\}$
- Proprietà di invarianza: di un programma con stati: Γ

$$I \in \wp(\Sigma)$$

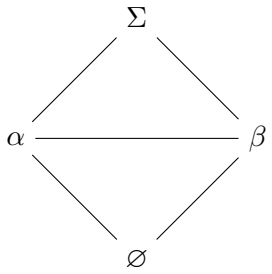
- ...

3.1.3 Proprietà

L'insieme delle proprietà di $\wp(\Sigma)$ degli oggetti in Σ è un reticolo distributivo completo,

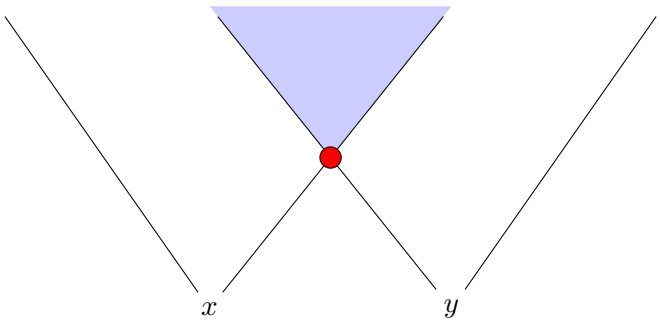
$$\langle \wp(\Sigma), \subseteq, \varnothing, \sigma, \Sigma, \cup, \cap, \neg \rangle$$

Nell'analisi di un sistema complesso, è essenziale considerare l'astrazione come un processo chiave per semplificare la comprensione. Quando si tratta di approssimare una proprietà concreta con un'astrazione, si aprono due possibili approcci. L'approccio di **approssimazione dal basso** implica che l'astrazione rappresenti un sottoinsieme della proprietà concreta, mentre l'approccio di **approssimazione dall'alto** (P) implica che l'astrazione rappresenti un sovrainsieme della proprietà concreta. Questi approcci possono essere visti come duali, sebbene l'analisi si concentri principalmente sull'approccio di approssimazione dall'alto, poiché trovare approssimazioni utili dal basso può essere più impegnativo e complesso.



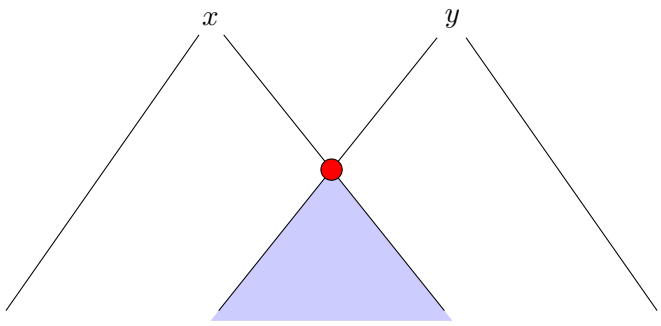
Least upper bound

Il least upper bound (`LUB`) di un insieme di elementi è il più piccolo elemento del reticolo che è maggiore o uguale a ciascun elemento dell'insieme ($X \vee Y$). Ovvero in $\wp(D)$ tale che $A \supseteq X$ e $A \supseteq Y$.



Greatest lower bound

Il greatest lower bound (`GLB`) di un insieme di elementi è il più grande elemento del reticolo che è minore o uguale a ciascun elemento dell'insieme ($X \wedge Y$). Ovvero in $\wp(D)$ tale che $A \subseteq X$ e $A \subseteq Y$.



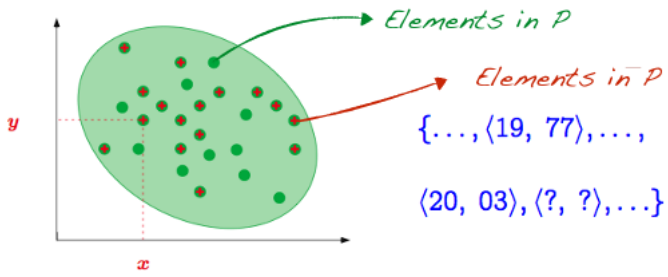
3.2 Approssimazione dei dati

Sia $P^\sharp$ una proprietà di D se e solo se $P^\sharp$ e $\wp(D)$. Vogliamo quindi capire la relazione tra gli elementi di D e $\wp(D)$ e poi, preso $D^\sharp \subseteq \wp(D)$ la relazione tra gli elementi di D e gli elementi di $D^\sharp$. Per approssimare D scegliamo un sottoinsieme $D^\sharp$ che fissa le proprietà che vogliamo osservare (*con precisione*). In generale $d \in D \implies d^\sharp \in D \subseteq \wp(D)$.

Potremmo quindi avere:

- $d \subseteq d^\sharp$ ovvero **over approximation**.
- $d \supseteq d^\sharp$ ovvero **under approximation**.

3.2.1 Approssimazione dal basso

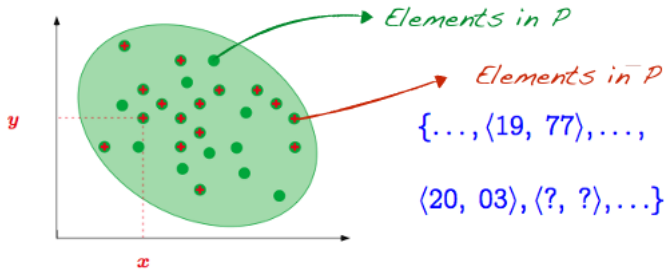


Per rispondere alla domanda $\langle x, y\rangle \in P$ utilizziamo un'astrazione $\bar{P}$, tale che $P \supseteq \bar{P}$.

- Se $\langle x, y\rangle \in \bar{P}$, quindi $d \subseteq d^\sharp$, allora $\langle x, y\rangle \in P$.
- Se $\langle x, y\rangle \notin \bar{P}$, quindi $d \supseteq d^\sharp$, allora non lo sappiamo.

In sintesi prendiamo un insieme più piccolo che comprende una sottoparte del nostro insieme di partenza e analizziamo tale insieme più piccolo. Se troviamo una risposta positiva allora abbiamo risposto alla domanda, altrimenti non lo sappiamo.

3.2.2 Approssimazione dall'alto



Per rispondere alla domanda $\langle x, y \rangle \in P$ utilizziamo un'astrazione $\bar{P}$, tale che $P \subseteq \bar{P}$.

- Se $\langle x, y \rangle \in \bar{P}$, quindi $d \supseteq d^\sharp$, allora non sappiamo rispondere.
- Se $\langle x, y \rangle \notin \bar{P}$, quindi $d \subseteq d^\sharp$, allora no.

In sintesi prendiamo un insieme più grande che comprende il nostro insieme di partenza e analizziamo tale insieme più grande. Più grande non è sinonimo di più complesso, ma spesso ricondurci a proprietà più generali potrebbe aiutarci nell'analisi, la rappresentazione estensionale potrebbe quindi risultare più semplice. Tale approccio ci permette di rispondere alla domanda solo che la proprietà non è soddisfatta per il nuovo insieme più grande, ovvero $P^\sharp$.

In sostanza:

Proprietà concrete

Le proprietà concrete sono un insieme di oggetti potenzialmente complessi, infiniti e non rappresentabili da un calcolatore.

Proprietà astratte

Le proprietà astratte sono un insieme più ampio di oggetti. A volte, l'ampiezza maggiore implica una maggiore estensibilità per la rappresentazione. Tuttavia, strutture più ampie ben scelte possono avere codifiche più semplici che possono essere sfruttate per la memorizzazione e il calcolo.

3.2.3 Minima astrazione

Assumendo che le proprietà astratte $P \in \wp(\Sigma)$ devono essere apporssimate dall'alto della proprietà astratta $\bar{P} \in A \subset \wp(\Sigma)$, tale che:

$$P \subseteq \bar{P}$$

Sappiamo che la più piccola proprietà $\bar{P}$ è la più precisa delle approssimazioni che possiamo avere. Ovviamente, la minima proprietà astratta potrebbe non non esistere per tutte le astrazioni A. Se questa minima approssimazione esiste è preferibile che sia il più precisa possibile,

se non esiste, può essere utilizzata una migliore alternativa che fornisce un'approssimazione più precisa.

3.2.4 Miglior astrazione

Una buona scelta per l'astrazione è quella che fornisce la miglior approssimazione per ogni proprietà concreta

$$P \subseteq \bar{P}$$

$$\forall \bar{P}' \in A.(P \subseteq \bar{P}') \implies (\bar{P} \subseteq \bar{P}')$$

Segue che la miglior approssimazione è la *greatest lower bound* di tutte le approssimazioni delle proprietà.

$$\bar{P} = \bigcap \{\bar{P}' \in A \mid P \subseteq \bar{P}'\} \in A$$

Tra tutti gli elementi più piccoli di quelli in X, è il più grande.

$$x = \texttt{glb}X \subseteq P \iff \forall l \in P.(\forall y \in X.l \leq y) \implies x \geq l$$

3.2.5 Esempio: `Sign`

Semantica concreta

Abbiamo a disposizione programmi che manipolano numeri interi: $f : \mathbb{Z} \to \mathbb{Z}$. Una delle proprietà osservate è quella di *sign*, ovvero che il risultato tra due numeri dipenderà dal segno dei due numeri. Dobbiamo indicare cosa inseriremo in $D^\sharp$ ovvero `Sign`.

- $+ = \{n \mid n > 0\} \in \wp(\mathbb{Z})$
- $- = \{n \mid n < 0\} \in \wp(\mathbb{Z})$
- $0 = \{0\} \in \wp(\mathbb{Z})$

Quindi:

$$\{+, 0, -\} \subseteq \wp(\mathbb{Z})$$

Semantica astratta

Il dominio astratto, noto come `Sign`, viene utilizzato per approssimare l'insieme di interi manipolati dai programmi. La funzione $f^\sharp$ manipola quindi i segni.

$$D^\sharp = \{+, -, 0, \mathbb{Z}, \varnothing\} = \texttt{Sign}$$

In `Sign`, gli interi possono essere rappresentati come:

- $x \subseteq \mathbb{Z} \to x^\sharp \in D^\sharp$ è il più piccolo insieme in $D^\sharp$ che contiene x.
- $\{-, 5, 4\} \to \mathbb{Z}$
- $\{3\} \to + \equiv \mathbb{Z}$
- $\{7\} \to + \equiv \mathbb{Z}^+$

- $\{-5\} \to - \equiv \mathbb{Z}^-$
- $\{-5, -6\} \to - \equiv \mathbb{Z}^-$

	$\mathbb{Z}$	$\mathbb{Z}^+$	$\mathbb{Z}^0$	$\mathbb{Z}^-$	$\varnothing$
$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$
$\mathbb{Z}^+$	$\mathbb{Z}$	$\mathbb{Z}^+$	$\mathbb{Z}^+$	$\mathbb{Z}$	$\mathbb{Z}^+$
$\mathbb{Z}^0$	$\mathbb{Z}$	$\mathbb{Z}^+$	$\mathbb{Z}^0$	$\mathbb{Z}^-$	$\mathbb{Z}^0$
$\mathbb{Z}^-$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}^-$	$\mathbb{Z}^-$	$\mathbb{Z}^-$
$\varnothing$	$\mathbb{Z}$	$\mathbb{Z}^+$	$\mathbb{Z}^0$	$\mathbb{Z}^-$	$\varnothing$

Figura 3.2.1: Operazioni di `Sign` relative alla somma.

	$\mathbb{Z}$	$\mathbb{Z}^+$	$\mathbb{Z}^0$	$\mathbb{Z}^-$	$\varnothing$
$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$
$\mathbb{Z}^+$	$\mathbb{Z}$	$\mathbb{Z}^+$	$\mathbb{Z}^0$	$\mathbb{Z}^-$	$\mathbb{Z}^+$
$\mathbb{Z}^0$	$\mathbb{Z}$	$\mathbb{Z}^0$	$\mathbb{Z}^0$	$\mathbb{Z}^0$	$\mathbb{Z}^0$
$\mathbb{Z}^-$	$\mathbb{Z}$	$\mathbb{Z}^-$	$\mathbb{Z}^0$	$\mathbb{Z}^+$	$\mathbb{Z}^-$
$\varnothing$	$\mathbb{Z}$	$\mathbb{Z}^+$	$\mathbb{Z}^0$	$\mathbb{Z}^-$	$\varnothing$

Figura 3.2.2: Operazioni di `Sign` relative alla moltiplicazione.

Per quanto riguarda la somma perdiamo informazioni solamente nel caso in cui si abbia un'operazione tra un numero positivo e uno negativo, poiché perdiamo le informazioni relative ai valori. Per quanto riguarda la moltiplicazione, invece, non perdiamo informazioni guardando la proprietà `Sign`, poiché è precisa sulla moltiplicazione.

3.3 Astrazione delle computazioni

Si tratta di approssimare la semantica sul dominio delle osservazioni. Una volta fissate queste osservazioni, osserviamo come la semantica opera su di esse.

Abbiamo già visto il significato di computazione, che ripetiamo.

Computazione

Una computazione è una traccia nel tempo dello stato del programma durante l'esecuzione. Quindi lo stato delle memorie nei vari punti del programma. A partire da uno stato iniziale noi abbiamo le possibili traiettorie di esecuzione del programma, talvolta infinite poiché potenzialmente divergenti.

In realtà le possibili traiettorie non sono continue, ma sono discrete, poiché fissiamo degli step di tempo che tipicamente corrispondono alle singole istruzioni del programma e ogni traccia è spezzata in questa sequenza di evoluzione.

3.3.1 Computazione di insiemi

Quello che avviene ricerca della **decidibilità** è quello di osservare le proprietà di interesse. Per osservare le proprietà di interesse, necessitiamo dell'osservazione di insiemi. L'insieme infatti rappresenta una proprietà che descrive un invariante di tutti gli elementi in esso contenuti.

Trasformando l'insieme di tracce in un'unica computazione che avviene tra insiemi. I punti rimangono comunque concreti, quindi dal punto di vista di ciò che possiamo calcolare, ovvero degli stati raggiungibili ad ogni passo di computazione, non cambia nulla, perché non perdiamo informazioni sugli stati raggiunti.

Questo calcolo però non vogliamo eseguirlo con il calcolo diretto, perché l'infinità delle traiettorie non viene assolutamente alterata, quindi stiamo potenzialmente gestendo insiemi potenzialmente infiniti.

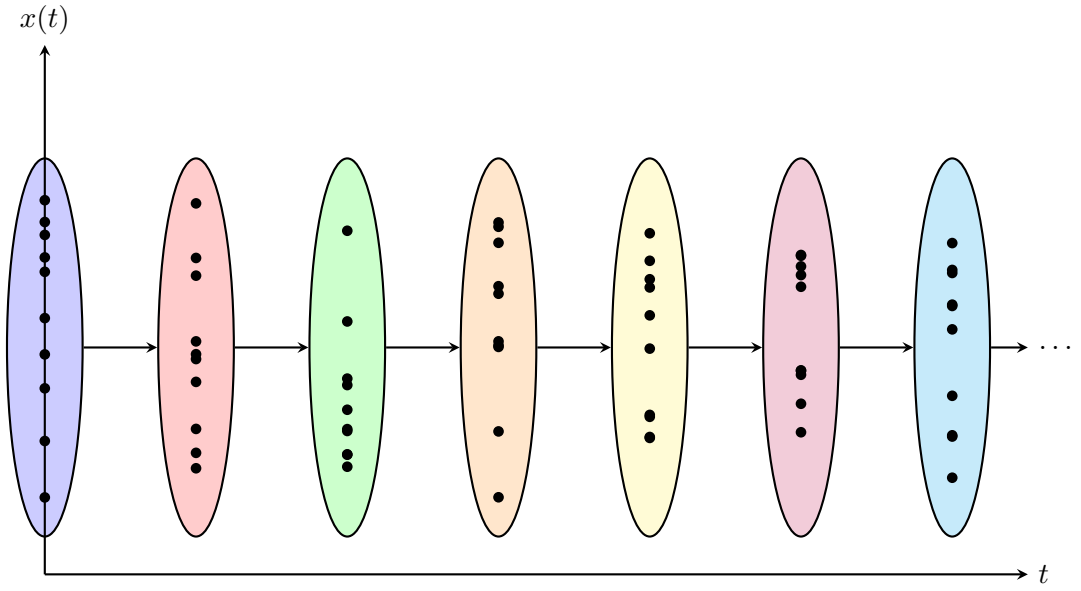
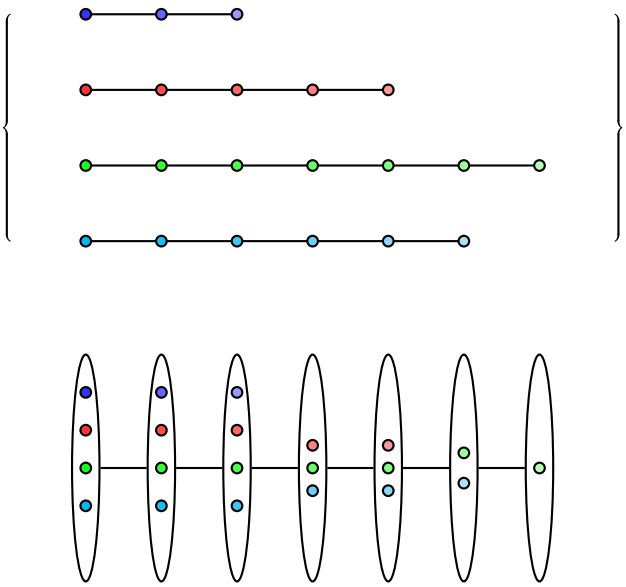


Figura 3.3.1: Traccia di una computazione di insiemi.

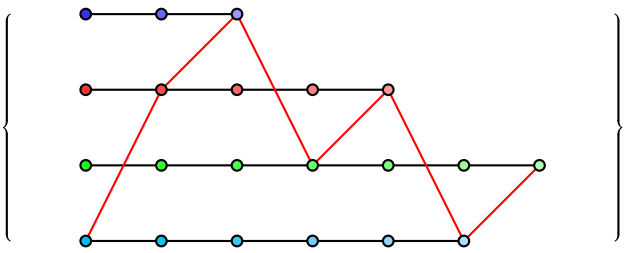
Il calcolo avviene quindi per punto fisso, partiamo quindi da un insieme di stati iniziali e andiamo via a via a collezionare tutti gli stati che raggiungiamo durante l'esecuzione, chiamato **reachability semantics** o **collecting semantics**.

3.3.2 Collecting semantics

Dal punto di vista della raggiungibilità degli stati, l'informazione è precisa, infatti l'insieme di stati raggiunti sono gli stessi che avremmo raggiunto con la semantica concreta. Di fatto, però, abbiamo una perdita di informazione dal punto di vista dell'insieme delle tracce che stiamo rappresentando.

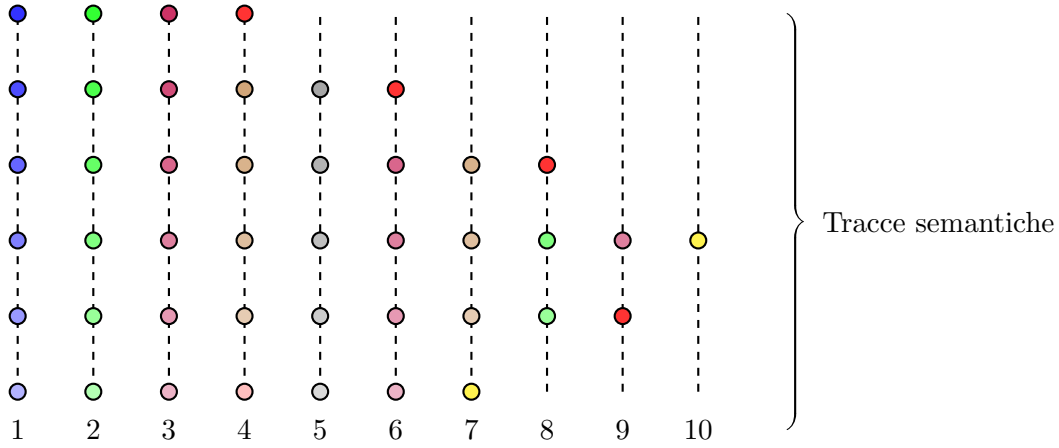


La semantica delle tracce mi colleziona l'insieme di tutte le tracce di computazione e la semantica delle collezioni invece considera per ogni passo di computazione l'insieme la proprietà raggiunta degli stati raggiunti. Abbiamo perso informazione rispetto alle tracce che rappresentiamo, in questo passaggio perdiamo la traccia che nello stato successivo raggiunge un determinato stato, poiché la traccia diventa unica.

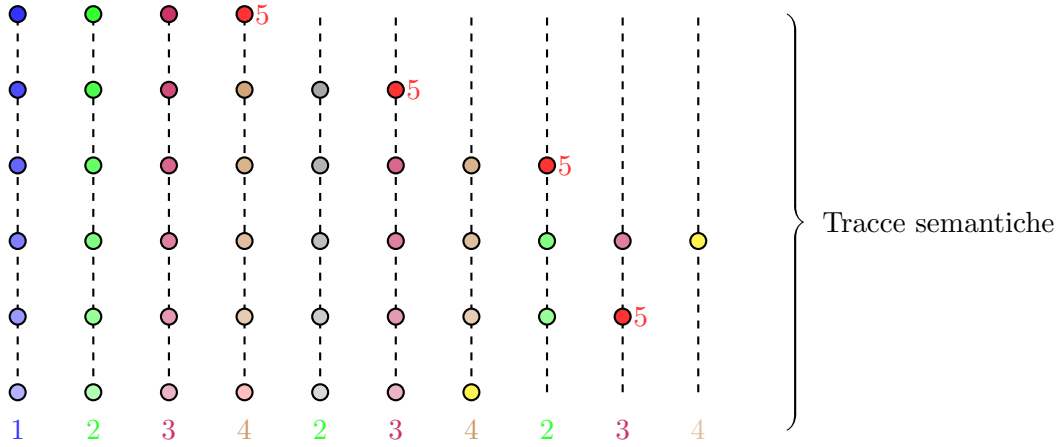


Abbiamo buttato via l'informazione che riguardava l'esatta transizione tra gli stati, aggiungendo tracce spurie.

La domanda che sporge spontanea è se ci stiamo muovendo nella direzione della decidibilità; di fatto no. Vediamo quindi un'altra rappresentazione che ci permette di comprendere la situazione.

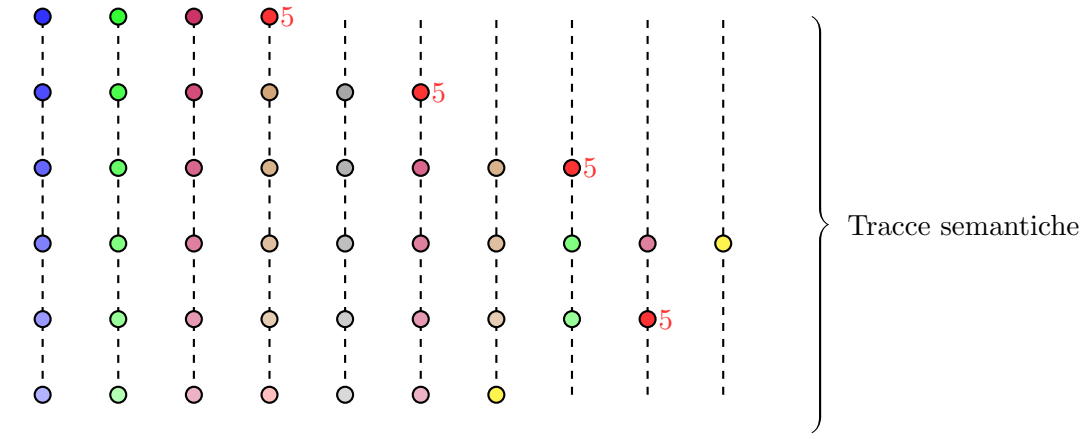


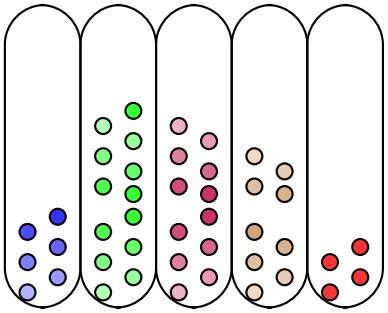
Ad ogni passo di computazione eseguiamo un istruzione in un **punto di programma**, possiamo quindi guardare il punto di programma che stiamo osservando.



Quello che osserviamo è che 5 è uno stato terminale, e che i punti 2 e 3 sono il corpo del ciclo, andando avanti nel tempo torniamo a visitate dei punti di programma.

Spostiamo la discretizzazione del punto di vista della traccia dal tempo ai punti di programma.





Quello che viene fatto è quello di collezionare gli elementi nei punti di programma che sono stati eseguiti. Di fatto per ogni punto di programma, l'insieme degli stati è sempre incrementale rispetto al punto di programma.

Potenzialmente anche questa rappresentazione sarà non terminante, poiché stiamo guardando ancora il mondo concreto, quindi gli stati raggiungibili sono ancora potenzialmente infiniti. Soprattutto in presenza di un ciclo `while` che calcola valori differenti ad ogni iterazione.

```
1 x ← 0
2 while x ≥ 0 do
3     x ← x + 1
```

L'insieme in questo caso continuerà ad espandersi all'infinito, poiché non c'è un limite e quindi non è possibile trovare un punto fisso. Il tentativo di raggiungere la terminazione è quello di trovare la stabilità di tali insiemi.

In alcuni casi la decidibilità è raggiungibile, ma nella maggior pate dei casi non è possibile.

```
1 x ← 0
2 while x ≥ 0 do
3     x ← x
```

In caso appena riportato l'insieme degli stati è sempre lo stesso, quindi è possibile trovare un punto fisso.

3.3.3 Computazioni sulle proprietà

Nella collecting semantics abbiamo quindi esecuzioni spurie, dovute al fatto che collezioniamo insiemi di stati, ma sono solo tra stati raggiungibili. Il fatto che manteniamo gli stati raggiungibili fa si che non vi sia perdita di informazione, ma dall'altra parte abbiamo esecuzioni potenzialmente infinite.

Dobbiamo ulteriormente raffinare la collecting semantics, per poter raggiungere la terminazione. Per farlo abbiamo bisogno dell'approssimazione, non più calcolando sugli insiemi di stati raggiungibili, ma su proprietà degli stati raggiungibili. Spostiamo quindi l'attenzione sugli sulle proprietà aggiungendo ulteriore rumore.

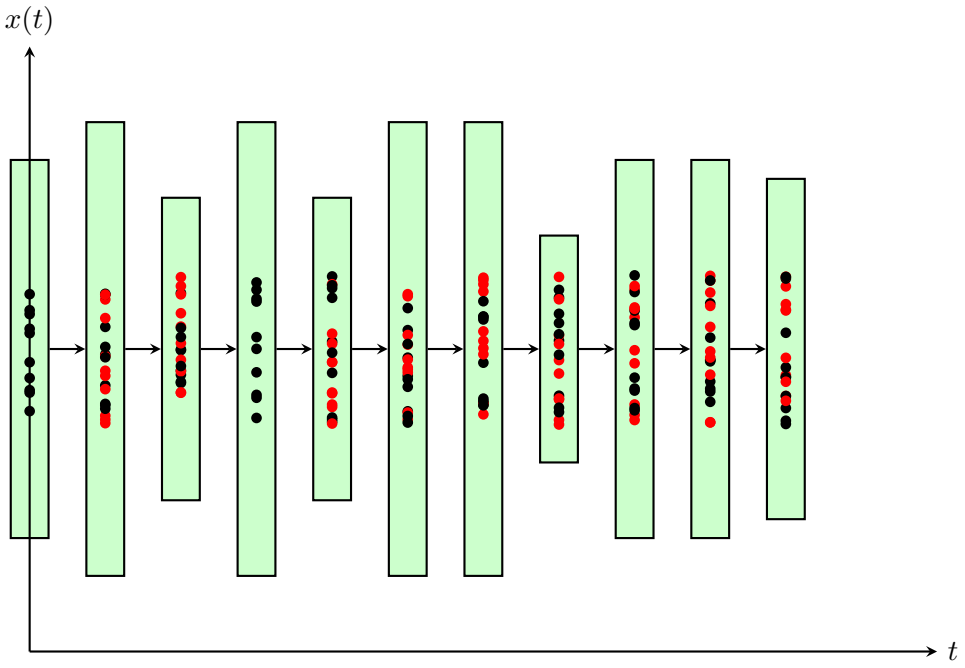


Figura 3.3.2: Traccia di una computazione sulle proprietà.

Non abbiamo solamente computazioni spurie dovute al fatto che ci muoviamo tra insiemi, ma abbiamo computazioni spurie che partono da stati che non vengono mai raggiunti nel concreto (*rappresentati dai pallini rossi nell'immagine* 3.3.2).

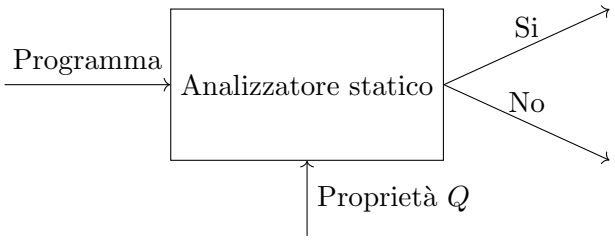
L'idea genera è quella di:

- Passare da l'insieme di tracce distinte ad una taccia di insiemi rappresentate (*che aggiunge rumore, mediante tracce spurie*).
- A questo punto è possibile approssimare la computazione guardando proprietà, utilizzando quindi la semantics collecting sulle proprietà, che possono agevolare la terminazione.

Capitolo 4

Analisi distributive

Recuperiamo il concetto di analizzatore statico, che è un software che prende in input un programma, una proprietà $\mathcal{Q}$ e ne restituisce un risultato.



L'obiettivo è costruire un analizzatore preciso, ovvero che restituisca sempre una risposta precisa, ma questa risposta viene data su un'approssimazione della semantica di $\mathcal{P}$. Su $\mathcal{P}$, in modo decidibile, possiamo dare solamente risposte approssimate.

Attraverso tale analisi ricaviamo informazioni sul punto di programma, spostando l'informazione localmente, e andiamo a caratterizzare qual è la collezione di valori raggiunti dal punto di programma. L'analizzatore riesce quindi a dare la risposta.

4.1 Idea dell'analisi statica

La semantica del linguaggio di programmazione è una funzione che prende in input un programma scritto in un linguaggio di programmazione e lo associa in un insieme di denotazioni che descrivono il significato del programma scritto in $\mathcal{L}$

$$[\![\cdot]\!] : \mathcal{L} \to \wp(\mathcal{D})$$

Con $\mathcal{L}$ denotiamo l'insieme dei programmi scritti nel linguaggio $\mathcal{L}$ e con $\wp(\mathcal{D})$ denotiamo l'insieme delle sue computazioni, ovvero l'insieme delle tracce di computazione, ovvero l'insieme di tutte le semantiche dei programmi.

La proprietà in generale, è rappresentata da un sottoinsieme di $\wp(\mathcal{D})$. Ovvero una collezione di semantiche che soddisfano la proprietà invariante.

$$\mathcal{Q} \subseteq \wp(\mathcal{D})$$

Dire che $\mathcal{Q} \subseteq \wp(\mathcal{D})$ equivale a dire che $\mathcal{Q}$ è l'insieme di tutti i programmi che soddisfano una fissata proprietà, ovvero la proprietà rappresentata.

4.2 Analisi statiche

- **Control Flow Analysis**: si trattano di proprietà analizzabili dalla sintassi del programma, su cui non entreremo nel merito.
- **Data Flow Analysis**: guarda come l'informazione fluisce dentro il programma, durante l'esecuzione, quindi riguarda i dati. In tale analisi non si entra nel merito del contenuto dei dati, perciò si riesce ad approssimare abbastanza bene sulla sintassi. Non si guarda quindi lo stato della memoria, ma la **relazione sintattica** tra gli elementi del programma, raggiungendo quindi un accettabile grado di precisione.
  - **Available Expressions**: le classiche analisi ottimizzanti dei compilatori. Riesce a capire se un'espressione è disponibile in un punto di programma, evitando quindi di ricalcolarla.
  - **Copy Propagation**: permette di capire se delle variabili sono copie di altre variabili, quindi se sono equivalenti.
  - **Liveness Analysis**: riguarda le variabili, quindi se una variabile è viva o morta, prima di un utilizzo, quindi se è utilizzata in un punto di programma.
  - **Reaching Definitions**: definizioni raggiungibili in un punto di programma.

Nella definizione intuitiva abbiamo sempre trattato elementi sintattici che raggiungono o hanno effetto su un determinato punto di programma. Per questo sono analisi che si approssimano bene sulla sintassi.

Le analisi che hanno necessità di entrare nel merito della semantica, che utilizzano altri strumenti per analizzare il programma, oltre al `CFG`, entrando nella memoria per analizzare il dato sono le **analisi distributive**.

4.2.1 Analisi statica sul `CFG`

Tale analisi si basa su un algoritmo ricorsivo di calcolo della proprietà desiderata sul `CFG`, visitandolo.

Supponiamo di avere un `CFG` per ogni procedura, da questa supposizione si ricava che tale analisi può essere svolta su vari livelli:

- **Locale**: analisi all'interno di un blocco, inteso come collezione massimale di istruzioni senza branching interni, quindi con un'unica entrata e un'unica uscita.

- **Intra-procedurali**: analisi sui singoli `CFG` in maniera indipendente.
- **Inter-procedurali**: analisi che considera l'intero programma, si tiene conto anche delle relazioni, ovvero dei ritorni, tra procedure. I `CFG` non sono isolati ma sono collegati tra loro.

Ciò che faremo in relazione alla analisi statica, consiste nel caratterizzare come l'informazione di interesse viene trasformata dentro il programma. L'arco denota solamente il passaggio di controllo, mentre il nodo rappresenta il blocco, dove vi è la modifica dello stato. Dobbiamo quindi capire come l'informazione di interesse viene trasformata, nella sua versione approssimata.

Questa analisi avviene in due fasi:

1. Caratterizzare l'informazione di interesse che entra nel blocco. Combinando le informazioni che arrivano dai blocchi predecessori. Nel caso in cui il blocco abbia più predecessori, l'informazione di interesse è la combinazione delle informazioni di interesse che arrivano dai vari predecessori. Si parla di combinazione perché dipenderà dal tipo di analisi.
2. Caratterizzare l'informazione di interesse che esce dal blocco come manipolazione dell'informazione in entrata. Il calcolo avviene per punto fisso, partendo da un'ipotetica informazione iniziale tendenzialmente vuota e vengono visitati tutti i nodi del control flow graph, finché l'informazione non si stabilizza, ovvero non viene modificata. A quel punto abbiamo trovato l'invariante dell'informazione di interesse in quel punto di entrata o di uscita del blocco.

Esempio di ottimizzazioni

Algorithm 1: Bubble sort

```
1 for i ← n − 2 to 0 do
2     for j ← 0 to i do
3         if A[j] > A[j + 1] then
4             t ← A[j]
5             A[j] ← A[j + 1]
6             A[j + 1] ← t
```

Il codice intermedio generato è il seguente:

Algorithm 2: Bubble sort

```
1  i ← n − 2
2  S5 : if i < 0 goto S1
3  j ← 0
4  S4 : if j > i goto S2
5  t1 ← j · 4
6  t2 ← &A
7  t3 ← t2 + t1
8  t4 ← ∗t3
9  t5 ← j + 1
10 t6 ← t5 · 4
11 t7 ← &A
12 t8 ← t7 + t6
13 t9 ← ∗t8
14  if t4 ≤ t9 goto S3
15 t10 ← j · 4
16 t11 ← &A
17 t12 ← t11 + t10
18 temp ← ∗t12
19 t13 ← j + 1
20 t14 ← t13 · 4
21 t15 ← &A
22 t16 ← t15 + t14
23 t17 ← ∗t16
24 t18 ← j · 4
25 t19 ← &A
26 t20 ← t19 + t18
27 ∗t20 ← t17
28 t21 ← j + 1
29 t22 ← t21 · 4
30 t23 ← &A
31 t24 ← t23 + t22
32 ∗t24 ← temp
33 S3 : j ← j + 1
34  goto S4
35 S2 : i ← i − 1
36  goto S5
37 S1 : return
```

In questo caso ci sono diverse espressioni ridondanti, ad esempio $j \cdot 4$ oppure $j + 1$. Ci chiediamo se è necessario rivalutare tali espressioni oppure possiamo utilizzare il risultato calcolato precedentemente.

Applicando varie ottimizzazioni, si ottiene il seguente codice:

Algorithm 3: Bubble sort ottimizzato

```
1  i ← n − 2
2  t27 ← i · 4
3  t28 ← &A
4  t29 ← t28 + t27
5  t30 ← ∗t29
6  S5 : if i < 0 goto S1
7  t25 ← t28
8  t26 ← t30
9  S5 : if t25 > t29 goto S2
10 t4 ← ∗t25
11 t9 ← ∗t26
12 if t4 ≤ t9 goto S3
13 temp = ∗t25
14 t17 ← ∗t25
15 ∗t25 ← t17
16 ∗t26 ← temp
17 S3 : t25 ← t25 + 4
18 t26 ← t26 + 4
19 goto S4
20 S2 : t29 ← t29 − 4
21 goto S5
22 S1 : return
```

Il codice risultante è molto più compatto, e più efficiente, in quanto non vengono più eseguite espressioni ridondanti.

4.3 Available Expressions

Supponiamo di avere a disposizione tale codice intermedio:

```
1 z ← 1
2 y ← M[5]
3 A : x1 ← y + z
4 ...
5 B : x2 ← y + z
```

Nei blocchi A e B abbiamo la stessa espressione, la domanda che dovrebbe sorgerci è se è necessario rivalutare l'espressione $y + z$ nel blocco B oppure possiamo utilizzare il risultato calcolato nel blocco A. Non è però detto che si arrivi al punto B dopo aver eseguito il blocco A, o che le variabili y e z non siano state modificate nel frattempo.

Non è così banale capire se è necessario o meno rivalutare l'espressione, tipicamente queste analisi sono fatte per ottimizzare il codice. Tipicamente nella conversione da codice ad alto

livello a codice intermedio, e può essere che in tali conversioni si generino delle espressioni ridondanti, che possono eseguire eccessivi accessi alla memoria, quindi si vuole evitare di rivalutare l'espressione.

Le available expressions si pongono il quesito di capire se un'espressione viene ricalcolata in in modo identico in un altro punto di programma.

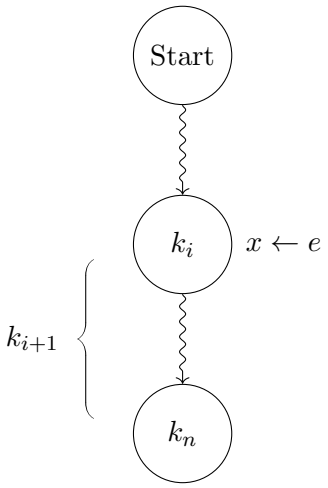
Se dimostriamo che l'espressione calcolata al punto A arriva anche al punto B, possiamo modificare il codice, ma per poterlo fare la risposta deve essere certa, senza alcun margine di errore. Non dobbiamo quindi classificare come ridondanti espressioni che non lo sono, in quanto potremmo modificare il codice in modo errato. È ammesso non catturare tutti i calcoli ridondanti perché alla peggio il calcolo sarà meno efficiente, ma non avrò modificato la semantica del programma.

4.3.1 Definizione formale di available expressions

Espressione disponibile in una variabile x al punto p

Un'espressione e è disponibile in una variabile x al punto p se:

- e deve essere stata valutata in un punto q precedente a p e salvata in una variabile x.
- Sia x che tutte le variabili usate in e non devono essere state modificate tra la valutazione di e e il punto p.

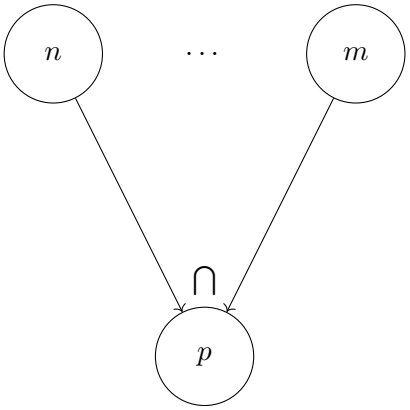


Sia $\pi = k_1, \ldots, k_n$ dal punto v al punto p. L'espressione è disponibile in x al punto p se:

- π che contiene $k_i = x \leftarrow e$.
- Nessuno tra gli archi $k_{i+1}, \ldots, k_n$ è etichettato con un assegnamento ad una variabile in $x \cup \texttt{Var}(e)$.

$$\texttt{Avail}(n) = \begin{cases} \varnothing & \text{se } n_0 = n \\ \bigcap_{m \in \texttt{Pred}(n)} \texttt{AvailOut}(m) \end{cases} \tag{4.1}$$

Ogni cammino che arriva al punto di programma che stiamo analizzando deve avere quella espressione come disponibile.



Per calcolare ciò che è disponibile in uscita da un blocco, dobbiamo caratterizzare ciò che è disponibile in uscita calcolando come il blocco manipola l'informazione disponibile in ingresso.

$$\texttt{AvailOut}(n) = \texttt{Gen}(m) \cup (\texttt{AvailIn}(n) \setminus \texttt{Kill}(m)) \tag{4.2}$$

Il blocco può manipolare espressioni e di conseguenza può generare nuove espressioni disponibili o può uccidere espressioni disponibili in ingresso.

Vediamo la presenza di **ricorsione** nella definizione di available expressions, poiché abbiamo un `AvailOut` che dipende da un `AvailIn` e viceversa, quindi il calcolo di punto fisso.

Associamo ad ogni nodo un'informazione chiamata $\texttt{AvailIn}(n)$ che contiene le espressioni disponibili in ingresso al nodo n. Il processo inizia con l'inizializzazione, in questo caso supponiamo che all'inizio il primo blocco non abbiamo disponibile nessuna espressione. Gli altri blocchi saranno di conseguenza popolati con l'elemento neutro dell'operazione di intersezione, ovvero che tutto sia disponibile.

$$\texttt{AvailIn}(n) = \begin{cases} \varnothing & \text{se } n_0 = n \\ \{x \leftarrow e \mid x := e \text{ è } nel \texttt{ CFG}\} \end{cases} \tag{4.3}$$

Combinando tali formule otteniamo la seguente equazione ricorsiva di punto fisso che posso calcolare sul `CFG`:

$$\texttt{AvailIn}(n) = \bigcap_{m \in \texttt{Pred}(n)} \texttt{Gen}(m) \cup (\texttt{AvailIn}(n) \setminus \texttt{Kill}(m)) \tag{4.4}$$

Espressione generata

$$\texttt{Gen}(n) = \{x \leftarrow e \mid x := e \in b, x \notin \texttt{Var}(e)\} \tag{4.5}$$

Espressione uccisa

$$\texttt{Kill}(n) = \{x \leftarrow e \mid \exists y := e' \in n, y \in \texttt{Var}(e) \vee x = y\} \tag{4.6}$$

4.3.2 Algoritmo di available expressions

La generazione dell'informazione iniziale altro non è che la costruzione di `Gen` e `Kill` che dipende dalla sintassi del blocco e non dal calcolo.

Algorithm 4: Generazione dell'informazione iniziale

```
1  foreach block b do
2      Init (b)
3  Function Init(b):
4      DEExpr (b) ← ∅
5      ExprKill (b) ← ∅
6      for i ← 1 to k do
7          if y ∉ ExprKill(b) ∧ z ∉ ExprKill(b) then
8              add x ← (y ∘ z) to DEExpr (b)
9          add x to DEExpr (b)
10     return
```

Una volta che abbiamo l'informazione iniziale, possiamo calcolare l'available expressions.

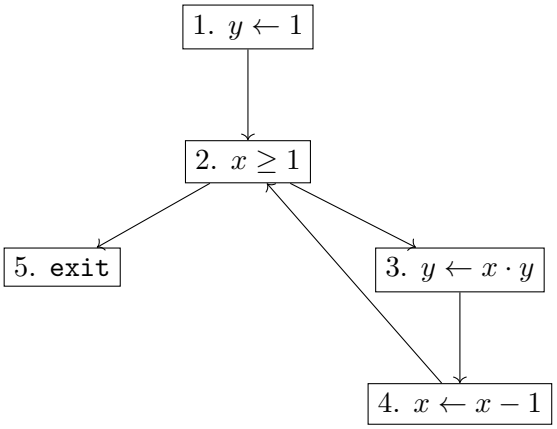
Algorithm 5: Available expressions

```
1  for i ← 1 to N − 1 do
2      AvailIn(i) ← {AllExpr}
3      AvailIn(0) ← ∅
4  changed ← true
5  while changed do
6      changed ← false
7      for i ← 1 to N − 1 do
8          old ← AvailIn(i)
9          AvailIn(i)
10         if old ≠ AvailIn(i) then
11             changed ← true
```

4.3.3 Esempio

Consideriamo il seguente programma:

```
1 y ← 1;
2 while x ≥ 1 do
3 |   y ← y · x;
4 |   x ← x − 1;
```



gli assegnamenti sono:

- $y \leftarrow 1$
- $y \leftarrow y \cdot x$
- $x \leftarrow x - 1$

Osserviamo che in $y \leftarrow y \cdot x$ abbiamo la variabile modificata y anche tra le variabili dell'espressione, perciò non sarà disponibile. Lo stesso varrà per $x \leftarrow x - 1$.
L'insieme di tutti gli assegnamenti è:

$$\texttt{AllExpr} = \{y \leftarrow 1\}$$

Costruiamo per ogni blocco l'insieme di espressioni generate e uccise:

- b_1: $\texttt{Gen}(1) = \{y \leftarrow 1\}$, $\texttt{Kill}(1) = \{y \leftarrow 1\}$
- b_2: $\texttt{Gen}(2) = \varnothing$, $\texttt{Kill}(2) = \varnothing$
- b_3: $\texttt{Gen}(3) = \varnothing$, $\texttt{Kill}(3) = \{y \leftarrow 1\}$
- b_4: $\texttt{Gen}(4) = \varnothing$, $\texttt{Kill}(4) = \varnothing$

Seguendo la formula per il calcolo dell'available expressions (Eq. 4.4), e l'algoritmo 5, otteniamo:

	1	**2**	**3**	**4**	...
1	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
2	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	$\varnothing$	...
3	$y \leftarrow 1$	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	...
4	$y \leftarrow 1$	$\varnothing$	$\varnothing$	$\varnothing$	...
5	$y \leftarrow 1$	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	...

Quindi alla quarta iterazione abbiamo che non ci sono cambiamenti, quindi l'algoritmo termina e troviamo il punto fisso.

4.3.4 Analisi del control flow graph

Riassumendo l'analisi del control flow graph, abbiamo che l'analisi può essere fatta in modi, forward o backward.

Forward

L'idea del forward è che l'informazione si propaga dai blocchi iniziali a quelli finali.

$$\texttt{FAvailIn}(n) = \begin{cases} \texttt{InitInf} & n = n_0 \\ \bigoplus_{m \in \texttt{pred}(n)} \texttt{FAvailOut}(m) \end{cases} \tag{4.7}$$

$$\texttt{FAvailOut}(n) = \texttt{Gen}(n) \cup (\texttt{FAvailIn}(n) \setminus \texttt{Kill}(n)) \tag{4.8}$$

La combinazione delle due espressioni ci permette di calcolare l'insieme di espressioni disponibili in un blocco.

$$\texttt{FAvailIn}(n) = \bigoplus_{m \in \texttt{pred}(n)} \texttt{Gen}(m) \cup (\texttt{FAvailIn}(m) \setminus \texttt{Kill}(m)) \tag{4.9}$$

La combinazione deriva dal fatto che si può differenziare in unione o intersezione:

- **Unione**: se vogliamo che l'analisi sia **possibile**;
- **Intersezione**: se vogliamo che l'analisi sia **definite**.

Backward

L'idea del backward è che l'analisi parte dal blocco di uscita e risale il grafo di controllo, quindi:

$$\texttt{BAvailOut}(n) = \begin{cases} \texttt{InitInf} & n = n_f \\ \bigoplus_{m \in \texttt{succ}(n)} \texttt{FAvailIn}(m) \end{cases} \tag{4.10}$$

$$\texttt{BAvailOut}(n) = \texttt{Gen}(n) \cup (\texttt{BAvailOut}(n) \setminus \texttt{Kill}(n)) \tag{4.11}$$

La combinazione delle due espressioni ci permette di calcolare l'insieme di espressioni disponibili in un blocco.

$$\texttt{BAout}(n) = \bigoplus_{m \in \texttt{succ}(n)} \texttt{Gen}(m) \cup (\texttt{BAvailIn}(m) \setminus \texttt{Kill}(m)) \tag{4.12}$$

La combinazione deriva dal fatto che si può differenziare in unione o intersezione:

- **Unione**: se vogliamo che l'analisi sia **possibile**;
- **Intersezione**: se vogliamo che l'analisi sia **definite**.

4.4 Framework per l'analisi

Formalizziamo il problema dell'analisi statica attraverso la semantica. L'idea è quella di riscrivere in funzione della semantica fornire un algoritmo di risoluzione del problema. Ciò permette di avere un algoritmo generico che può essere applicato a diversi problemi. La semantica è quindi un parametro di tale algoritmo, rendendo la struttura di analisi più modulare.

Il Framework ci permette inoltre di spostare l'attenzione dalle analisi di dataflow, molto più vicine alla sintassi, poiché verificano il modo in cui fluiscono i dati, in semantiche che guardano la semantica del programma, ovvero il significato del programma.

I passi per la costruzione del framework sono:

1. Formalizzare l'informazione astratta che vogliamo osservare;
2. Definizione l'*abstract edge effect*, ovvero l'effetto che un arco ha sull'informazione astratta (*transfer function*);

Una volta definite queste due componenti, costruiamo un sistema di disequazioni (*una disequazione per ogni punto di programma*), cercando la miglior soluzione possibile. La ricerca della miglior soluzione possibile può seguire due approcci:

- **Naive**: baso la soluzione sullo step precedente.
- **Round Robin**: baso la soluzione sugli step precedenti e sul calcolo attuato nello step attuale, per accelerare il processo di convergenza.

La soluzione verrà utilizzata per ottenere la soluzione del sistema che approssima la soluzione **MOP** (*Merge Over all Paths*).

L'obiettivo è la soluzione più precisa possibile, quindi la soluzione **MOP**. Tuttavia, quello che possiamo calcolare è la soluzione **MFP** (*soluzione del sistema di disequazioni*). Quando le due soluzioni coincidono avremo che la soluzione è la più precisa possibile, se non coincidono nel processo di calcolo è stata aggiunta ulteriore perdita di informazione.

4.4.1 Framework sull'available expression

Caratterizziamo l'informazione astratta che vogliamo osservare, ovvero l'insieme di espressioni disponibili in un punto di programma. Per rappresentare tale espressione utilizziamo gli insegnamenti $x \leftarrow e$, dove $x \notin \texttt{Var}$ e $e \in \texttt{Exp}$. Tale insieme è denominato `Assign`, a questo punto il dominio astratto sarà $\wp(\texttt{Assign}), A \subseteq \texttt{Assign}$.

Caratterizziamo quindi la funzione di trasferimento $[\![\cdot]\!]^\sharp\mathcal{A}$. Tale funzione è definita sulla semantica astratta, per induzione strutturale del `CFG`.

$$k = (u \,\texttt{lab}\, v) \qquad \cdot \xrightarrow{\texttt{lab}} \cdot \qquad \text{ovvero } [\![\texttt{lab}]\!]\mathcal{A}$$

- $[\![;]\!]^\sharp\mathcal{A} = \mathcal{A}$
- $[\![\texttt{NonZero}(e)]\!]^\sharp\mathcal{A} = [\![\texttt{Zero}(e)]\!]^\sharp\mathcal{A} = \mathcal{A}$

- $[\![x \leftarrow e]\!]^\sharp \mathcal{A} = \begin{cases} \mathcal{A} \setminus \texttt{Occ}(x) & x \in \texttt{Var}(e) \\ (\mathcal{A} \setminus \texttt{Occ}(x)) \cup \{x \leftarrow e\} & x \notin \texttt{Var}(e) \end{cases}$
  dove $\texttt{Occ}(x) = \{x \leftarrow e \in \mathcal{A} \mid y = x \vee x \in \texttt{Var}(e)\}$, ovvero qualunque assegnamento dove compare x, o a destra o a sinistra del simbolo di assegnamento.
- $[\![k_0, K_1, \ldots, k_n]\!]^\sharp \mathcal{A} = [\![k_0]\!]^\sharp \circ [\![k_1]\!]^\sharp \circ \cdots \circ [\![k_n]\!]^\sharp \mathcal{A}$

4.4.2 Espressione definitivamente disponibile

Un assegnamento $x \leftarrow e$ è definitivamente disponibile in un punto di programma v se è disponibile in tutti i cammini che partono dall'*entry point* e arrivano in v. Siccome la soluzione cercata è definitiva, l'operatore di combinazione è l'intersezione.

$$\mathcal{A}^\star[v] = \bigcap \{[\![\pi]\!]^\sharp \varnothing \mid \pi : \texttt{start} \longrightarrow^\star v\} \tag{4.13}$$

Dove π rappresenta tutti i cammini. La terminazione di tale calcolo è garantita dal fatto che la funzione di trasferimento è monotona e che $\wp(\texttt{Assign})$ non ha catene ascendenti infinite. In questo particolare caso $\wp(\texttt{Assign})$ è finito, quindi il calcolo termina. Spesso nella dataflow analysis si lavora con domini finiti, quindi automaticamente non si hanno catene ascendenti infinite e il fatto che sia monotona è garantito dalla definizione della semantica di riferimento.

La perdita di informazione è data dal fatto che consideriamo cammini non eseguibili, dovuto al fatto che utilizziamo il `CFG`. Prendiamo quindi tutti i cammini che contengono i cammini eseguibili. In generale i cammini eseguibile sono un sottoinsieme stretto dei cammini del `CFG`, l'identificazione dei cammini eseguibili è un problema non decidibile, mentre l'identificazione dei cammini del `CFG` è un problema decidibile.

L'altra perdita d'informazione è data dal fatto che non consideriamo il valore delle variabili, potremmo considerare variabili modificate quando in realtà non lo sono.

4.4.3 Computazione della soluzione

La soluzione è data dalla soluzione del sistema di disequazioni. Prima di tutto consideriamo ciò che è disponibile al nodo iniziale:

$$\mathcal{A}[\texttt{start}] \subseteq \varnothing$$

In generale utilizziamo contenimento invece di uguaglianza, in quanto si tratta di disequazioni. Costruiamo disequazioni in modo che la combinazione di ciò che arriva allo stesso nodo avviene per intersezione, sovrastimando ciò che avviene nel nodo.

$$\mathcal{A}[v] \subseteq [\![lab]\!]^\sharp(\mathcal{A}[u]) \qquad \forall k = (u, lab, v)$$

Quindi l'informazione si propaga attraverso ciò che viene manipolato dall'etichetta, ovvero dall'operazione che su quell'arco viene eseguita.

Consideriamo $\mathcal{D} = \wp(\{a, b, c\})$

- $x_1 \subseteq \{a\} \cup x_3$

- $x_2 \subseteq x_3 \cap \{a, b\}$
- $x_3 \subseteq x_1 \cup \{c\}$

	0	**1**	**2**	**3**	...
x_1	$\varnothing$	$\{a\}$	$\{a, c\}$	$\{a, c\}$	...
x_2	$\varnothing$	$\varnothing$	$\{a\}$	$\{a\}$	...
x_3	$\varnothing$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	...

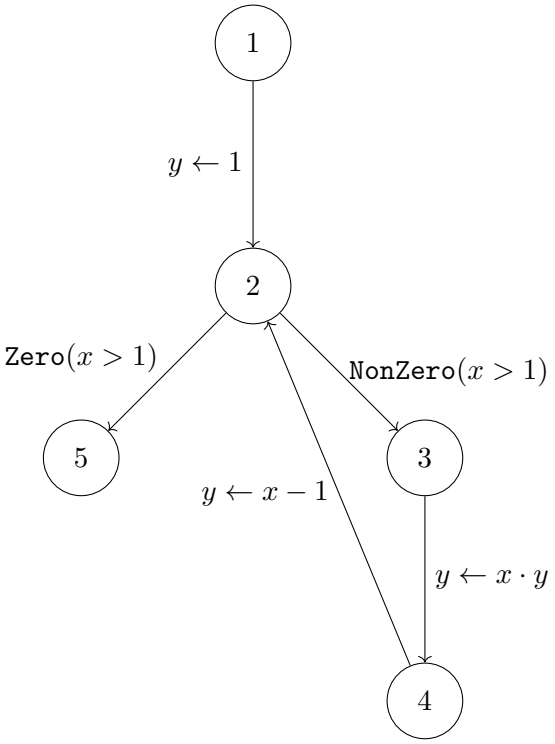
Il fix point viene trovato al punto 3, poiché i valori non cambiano più.

Quando calcoliamo tale soluzione come migliore soluzione del sistema di disequazioni in realtà stiamo calcolando la soluzione del corrispondente sistema di equazioni.

> Se f è monotona su un reticolo completo, allora il sistema ha miglior soluzione e questa soluzione coincide con la soluzione del sistema di disequazioni.

4.4.4 Calcolo della soluzione sul fattoriale

Prendiamo in considerazione la funzione fattoriale.



Procediamo con l'algoritmo:

- $\mathcal{A}[1] \subseteq \varnothing$
- $\mathcal{A}[2] \subseteq [\![y \leftarrow 1]\!]^\sharp \mathcal{A}[1] = [\![y \leftarrow 1]\!]^\sharp \cap \varnothing = [\![y \leftarrow 1]\!]^\sharp$
- $\mathcal{A}[2] \subseteq [\![x \leftarrow x - 1]\!]^\sharp \mathcal{A}[4]$
- $\mathcal{A}[3] \subseteq [\![\texttt{NonZero}(x > 1)]\!]^\sharp \mathcal{A}[2]$
- $\mathcal{A}[4] \subseteq [\![y \leftarrow x \cdot y]\!]^\sharp \mathcal{A}[3]$
- $\mathcal{A}[5] \subseteq [\![y \leftarrow x - 1]\!]^\sharp \mathcal{A}[4]$

ovvero

- $\mathcal{A}[1] \subseteq \varnothing$
- $\mathcal{A}[2] \subseteq \{y \leftarrow 1\} \cap [\![y \leftarrow x - 1]\!]^\sharp \mathcal{A}[4]$
- $\mathcal{A}[3] \subseteq [\![\texttt{NonZero}(x > 1)]\!]^\sharp \mathcal{A}[2]$
- $\mathcal{A}[4] \subseteq [\![y \leftarrow x \cdot y]\!]^\sharp \mathcal{A}[3]$
- $\mathcal{A}[5] \subseteq [\![y \leftarrow x - 1]\!]^\sharp \mathcal{A}[4]$

Risolviamo quindi il sistema di equazioni:

- $\mathcal{A}[1] = \varnothing$
- $\mathcal{A}[2] = \{y \leftarrow 1\} \cap (\mathcal{A}[4] \setminus \texttt{Occ}(x))$
- $\mathcal{A}[3] = \{y \leftarrow 1\} \cap (\mathcal{A}[4] \setminus \texttt{Occ}(x))$
- $\mathcal{A}[4] = \mathcal{A}[3] \setminus \texttt{Occ}(y)$
- $\mathcal{A}[5] = \mathcal{A}[2]$

	0	**1**	**2**	**3**	...
$\mathcal{A}[1]$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
$\mathcal{A}[2]$	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	$\varnothing$	...
$\mathcal{A}[3]$	$y \leftarrow 1$	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	...
$\mathcal{A}[4]$	$y \leftarrow 1$	$\varnothing$	$\varnothing$	$\varnothing$	...
$\mathcal{A}[5]$	$y \leftarrow 1$	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	...

Troviamo il punto fisso al punto 3.

Potremmo accelerare la convergenza dell'algoritmo utilizzando l'algoritmo di **Round Robin**.

	0	**1**	**2**	**3**	...
$\mathcal{A}[1]$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
$\mathcal{A}[2]$	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	$\varnothing$	...
$\mathcal{A}[3]$	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	$\varnothing$	...
$\mathcal{A}[4]$	$y \leftarrow 1$	$\varnothing$	$\varnothing$	$\varnothing$	...
$\mathcal{A}[5]$	$y \leftarrow 1$	$y \leftarrow 1$	$\varnothing$	$\varnothing$	...

Raggiungiamo il punto fisso al punto 2.

4.5 Analisi di liveness

Si tratta di un'analisi che cerca di individuare le variabili così dette "vive". L'idea di fondo è che due variabili vive contemporaneamente, devono occupare porzioni di memoria distinte, poiché il loro valore è utilizzato durante l'esecuzione e non possono essere allocate nella stessa locazione di memoria.

L'informazione è importante nei vari contesti, in particolare nell'ottimizzazione, infatti se due variabili non sono vive nella stessa porzione di codice, allora possono essere allocate nella stessa locazione di memoria. Lo stesso principio può essere applicato nel software watermarking, tecnica di protezione del software che garantisce di scrivere informazioni segrete all'interno del codice sorgente, solitamente utilizzata per identificare la proprietà intellettuale del software.

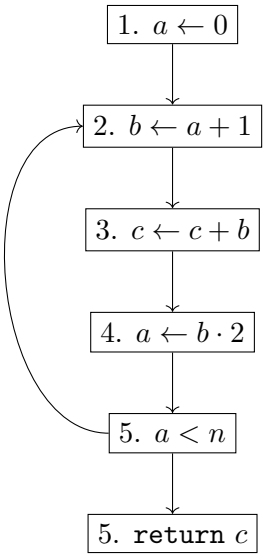
Esempio

Supponiamo di avere il seguente programma:

Algorithm 6: Liveness analysis

```
1 a ← 0
2 repeat
3 |   b ← a + 1
4 |   c ← c + b
5 |   a ← b · 2
6 until a < n
7 return c
```

Intuitivamente nell'arco $1 \to 2$ la variabile a è viva, poiché viene utilizzata per l'assegnamento di b. Nell'arco $2 \to 3$ la variabile b è viva, poiché viene utilizzata per l'assegnamento di c.
Andando avanti ci accorgiamo che la variabile a viene ridefinita nel blocco 4, ciò significa che la variabile a non sarà importante prima del blocco 4, fino all'utilizzo precedente.



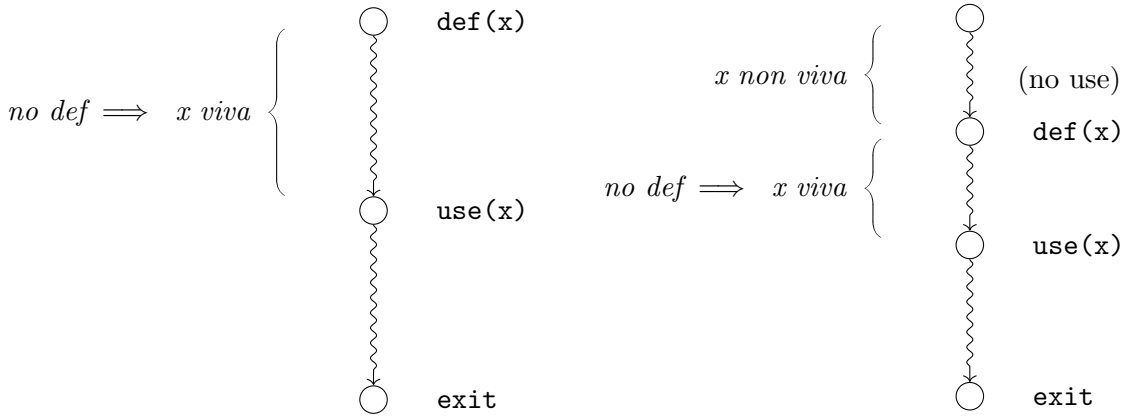
4.5.1 Analisi di liveness e l'approccio semantico

Rispetto all'analisi di available expressions, l'analisi di liveness guarda da un punto di programma, al passato, ovvero ciò che determina il fatto che una variabile sia viva è l'essere utilizzato. L'essere utilizzato la rende viva nei cammini che raggiungono quel nodo, quindi si tratta di un'analisi **backward**.

Dobbiamo ora definire in maniera formale il concetto di variabili usate (*accesso al valore*) e definite (*aggiornamento della locazione di memoria*). Siccome si trattano di concetti relativamente sintattici, possiamo definirli in maniera sintattica sulle etichette del linguaggio.

`lab`	Usate	Definite
`;`	$\varnothing$	$\varnothing$
`Zero(e)`	`var(e)`	$\varnothing$
`NonZero(e)`	`var(e)`	$\varnothing$
$x \leftarrow e$	`var(e)`	$\{x\}$
`input(x)`	$\varnothing$	$\{x\}$

Quindi x è viva (*live*) all'uscita di un blocco, se il suo valore verrà utilizzato in futuro, e non è viva (*dead*) se viene riassegnata prima del suo utilizzo.

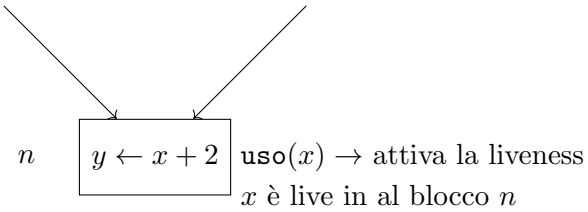


La costruzione dipende da quello che avviene prima di un certo punto di programma.

4.5.2 Calcolo della liveness

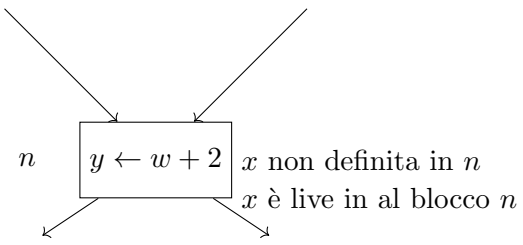
Cerchiamo di capire le condizioni che ci permettono di capire come l'esecuzione di un blocco influenza la liveness delle variabili, e se definite o possibile, ovvero se si combina per intersezione o unione.

1. **Informazione generata**: supponiamo di trovarci in un blocco n dove abbiamo un uso di x, l'utilizzo di x attiva la liveness di x. Possiamo quindi dire che x è viva in input al blocco n.

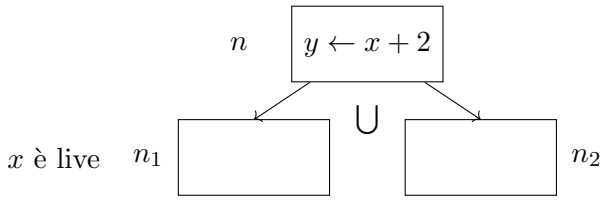


2. **Informazione preservata**: supponiamo di trovarci in un blocco n dove non abbiamo una definizione di x ma abbiamo x live in uscita da n, quindi l'informazione ha raggiunto il punto di programma attraverso i cammini. Quindi x si propaga in input allo stesso blocco.

   Se y fosse *liveout* in uscita al blocco n, non sarebbe live in ingesso, e questa sarebbe la *kill*.



3. Supponiamo di avere x live in uno dei nodi, in questo caso nel nodo n_1, in uscita del nodo n diremo che esiste un uso raggiunto dal nodo n_1, quindi lo raggiungo sicuramente in n. Quindi possiamo dire che x è *liveout* in n. Combiniamo quindi le informazioni per unione, e quindi uniamo possibili cammini non eseguiti ed è qui che avviene la possibile perdita di precisione.



4.5.3 Analisi liveness e l'approccio algoritmico

Indichiamo con `LiveIn[n]` le variabili live in ingesso al nodo n, mentre con `LiveOut[n]` le variabili live in uscita dal nodo n.

$$\texttt{LiveOut(n)} = \begin{cases} \varnothing & \text{se } n \text{ è un nodo di uscita} \\ \bigcup_{p \in \texttt{succ(n)}} \texttt{LiveIn(p)} & \text{altrimenti} \end{cases} \tag{4.14}$$

Che è quello che estraiamo dal caso 3.

$$\texttt{LiveIn(n)} = \texttt{Gen(n)} \cup (\texttt{LiveOut(n)} \setminus \texttt{Kill(n)}) \tag{4.15}$$

Che è quello che estraiamo dal caso 1 e 2.

La definizione precisa di `Gen(n)` e `Kill(n)` é:

$$\texttt{Gen(n)} = \{x \mid \exists e \in n . x = \texttt{var(e)}\} \tag{4.16}$$

$$\texttt{Kill(n)} = \{x \mid \exists e \leftarrow e \in n\} \tag{4.17}$$

Equazione di punto fisso

Combinando le equazioni, otteniamo l'equazione di punto fisso:

$$\texttt{LiveOut(n)} = \bigcup_{p \in \texttt{succ(n)}} \texttt{Gen(p)} \cup (\texttt{LiveIn(p)} \setminus \texttt{Kill(p)}) \tag{4.18}$$

La soluzione algoritmica è quindi la seguente:

- Costruzione del *control flow graph*
- Raccoglimento delle informazioni iniziali sul `CFG`:
  - $\texttt{Gen(n)} = \{x \mid \exists e \in n . x = \texttt{var(e)}\}$
  - $\texttt{Kill(n)} = \{x \mid \exists e \leftarrow e \in n\}$
- Iterazione dell'equazione di punto fisso fino a convergenza.

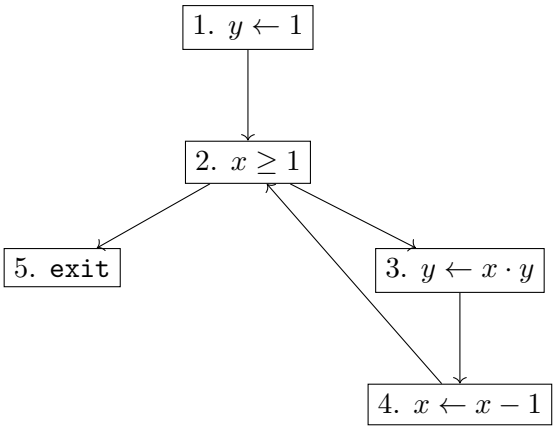
4.5.4 Precisione dell'analisi di *liveness*

Considerando cammini non eseguibili fa si che che l'analisi aggiunga potenzialmente falsi positivi, ovvero variabili che non sono effettivamente live. La perdita di precisione avviene anche per il fatto che ragioniamo sulla sintassi cercando proprietà semantiche. La variabile x potrebbe essere accessibile attraverso altri nomi, perdendo quindi una definizione.

Esempio di liveness con approccio algoritmico

Consideriamo il seguente programma:

```
1 y ← 1;
2 while x ≥ 1 do
3     y ← y · x;
4     x ← x − 1;
```



Costruiamo per ogni blocco l'insieme di espressioni generate e uccise:

- b_1: $\texttt{Gen}(1) = \varnothing$, $\texttt{Kill}(1) = \{x\}$
- b_2: $\texttt{Gen}(2) = \varnothing$, $\texttt{Kill}(2) = \{y\}$
- b_3: $\texttt{Gen}(3) = \{x\}$, $\texttt{Kill}(3) = \varnothing$
- b_4: $\texttt{Gen}(4) = \{x, y\}$, $\texttt{Kill}(4) = \{y\}$
- b_5: $\texttt{Gen}(5) = \{x\}$, $\texttt{Kill}(5) = \{x\}$

Seguendo la formula per il calcolo del punto fisso della liveness costruiamo le due tabelle di `LiveIn` e `LiveOut`, e otteniamo quindi:

`LiveOut`	**1**	**2**	**3**	**4**	...
5	$\varnothing$	$\varnothing$	$\{x, y\}$	$\{x, y\}$	...
4	$\varnothing$	$\{x\}$	$\{x\}$	$\{x, y\}$	...
3	$\varnothing$	$\{x, y\}$	$\{x, y\}$	$\{x, y\}$	...
2	$\varnothing$	$\{x\}$	$\{x, y\}$	$\{x, y\}$	...
1	$\varnothing$	$\varnothing$	$\{x\}$	$\{x\}$	...

`LiveIn`	**1**	**2**	**3**	**4**	...
5	$\{x\}$	$\{x\}$	$\{x, y\}$	$\{x, y\}$	...
4	$\{x, y\}$	$\{x, y\}$	$\{x, y\}$	$\{x, y\}$	...
3	$\{x\}$	$\{x, y\}$	$\{x, y\}$	$\{x, y\}$	...
2	$\varnothing$	$\{x\}$	$\{x\}$	$\{x\}$	...
1	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...

Al punto 4 della tabella `LiveOut` abbiamo un punto fisso, perciò ricalcoleremo gli stessi valori nel punto 4 della tabella `LiveIn`.

4.6 Analisi True Liveness

Nell'analisi liveness abbiamo osservato che rimangono dei falsi positivi, non legati ad aspetti non eseguibili o semantica non osservabile, ma legati ad aspetti sintattici, quindi catturabili.

Esempio di falsi positivi nell'analisi non dovuti all'approssimazione

Supponiamo di avere il seguente cammino:

Dall'analisi liveness otteniamo che z non è mai live in tutto il programma, quindi l'istruzione che coinvolge z può essere eliminata. Tuttavia, considerando il programma ottimizzato, reiterando l'analisi liveness otteniamo che x non è mai live in tutto il programma, quindi l'istruzione $x \leftarrow y + 1$ andrebbe eliminata.

È chiaro che si potrebbe risolvere applicando iterativamente l'analisi liveness, ma questo non è efficiente. Possiamo però sfruttare in modo ricorsivo quanto viene calcolato dall'analisi.

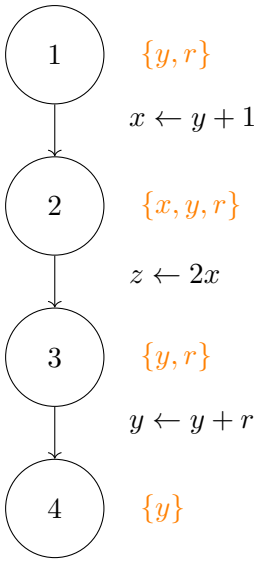


Figura 4.6.1: Prima iterazione dell'analisi liveness

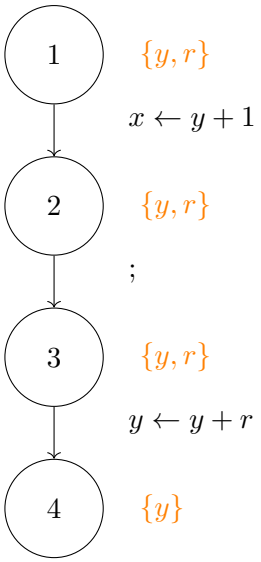


Figura 4.6.2: Seconda iterazione dell'analisi liveness

Quando inseriamo x all'interno delle variabili live, lo facciamo in funzione dell'utilizzo che avviene all'interno di un assegnamento che sappiamo che non sia live. Possiamo quindi raffinare l'analisi andando a decidere se inserire una variabile usata tra le live in funzione di quanto abbiamo calcolato come variabili live fino a quel punto di programma. Il concetto di **truly liveness** è quindi il vero uso della variabile all'interno del programma.

True liveness

Una variabile x è **truly live** su $\pi : N \to$ `exit` se π non contiene definizioni di x e contiene un vero uso della variabile prima che venga ridefinita.

Il significato di vero uso dipende dall'informazione che arriva ad un certo arco, quindi dobbiamo tener conto dell'informazione dell'arco.

$$k = (u, \texttt{lab}, v)$$

Ovvero se ciò che arriva all'arco v dipende da variabili veramente usate.

`lab`	y truly used in u
`;`	`false`
`Zero(e)`	$y \in$ `Var(e)`
`NonZero(e)`	$y \in$ `Var(e)`
$x \leftarrow e$	$y \in$ `Var(e)` $\wedge\, y \neq x$ truly used in u
`input(x)`	`false`

Usando tale definizione sull'esempio precedente otteniamo il risultato corretto solamente con una iterazione dell'analisi true liveness.

4.6.1 Analisi true liveness e l'approccio algoritmico

$$\texttt{TLiveOut}(n) = \begin{cases} x & n = \texttt{exit} \\ \bigcup_{p \in \texttt{succ}(n)} \texttt{TLiveIn}(p) & \end{cases} \tag{4.19}$$

Dove:

$$\texttt{TLiveIn}(n) = \texttt{TGen}(n)(\texttt{TLiveOut}(n) \setminus \texttt{TKill}(n)) \tag{4.20}$$

Ciò che essenzialmente varia è il modo di generare le variabili.

$$\texttt{TGen}(n) = \{x \mid \exists e \in n\texttt{(no assign)} x \in \texttt{Var(e)} \vee (\exists y \leftarrow e . x \in \texttt{Var(e)} \wedge y \text{ è true live in } n)\} \tag{4.21}$$

$$\texttt{TKill}(n) = \{x \mid x \in n\} \tag{4.22}$$

4.6.2 Analisi true liveness e l'approccio semantico

- $[\![;]\!]^\sharp \mathcal{L} = \mathcal{L}$
- $[\![\texttt{NonZero}(e)]\!]^\sharp \mathcal{L} = [\![\texttt{Zero}(e)]\!]^\sharp \mathcal{L} = \mathcal{L} \cup \texttt{Var(e)}$
- $[\![\texttt{input(x)}]\!]^\sharp \mathcal{L} = \mathcal{L} \setminus \{x\}$
- $[\![y \leftarrow e]\!]^\sharp \mathcal{L} = \begin{cases} (\mathcal{L} \setminus \{x\}) \cup \texttt{Var(e)} & x \in \mathcal{L} \\ \mathcal{L} \setminus \{x\} & \textit{altrimenti} \end{cases}$

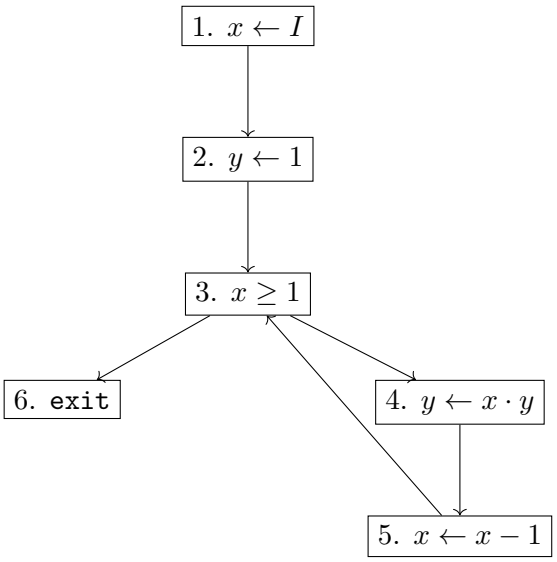
L'analisi è quindi distributiva, non cambiano quindi tutte le proprietà che abbiamo visto per le analisi precedenti.

Esempio di analisi true liveness

Approccio algoritmico

Consideriamo il seguente programma:

```
1 x ← I;
2 y ← 1;
3 while x ≥ 1 do
4 |   y ← y · x;
5 |   x ← x − 1;
```



Costruiamo per ogni blocco l'insieme di espressioni generate e uccise:

- b_1: $\texttt{Gen}(1) = I$ se x è *liveout*, $\texttt{Kill}(1) = \{x\}$
- b_2: $\texttt{Gen}(2) = \varnothing$, $\texttt{Kill}(2) = \{y\}$
- b_3: $\texttt{Gen}(3) = \{x\}$, $\texttt{Kill}(3) = \varnothing$
- b_4: $\texttt{Gen}(4) = \{x, y\}$ se y è *liveout*, $\texttt{Kill}(4) = \{y\}$ se y è *liveout*
- b_5: $\texttt{Gen}(5) = \{x\}$, $\texttt{Kill}(5) = \{x\}$
- b_6: $\texttt{Gen}(6) = \varnothing$, $\texttt{Kill}(6) = \varnothing$

Seguendo la formula per il calcolo del punto fisso della liveness costruiamo le due tabelle di `LiveIn` e `LiveOut`, e otteniamo quindi:

TLiveOut	1	2	3	4	...
6	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
5	$\varnothing$	$\{x\}$	$\{x\}$	$\{x\}$	...
4	$\varnothing$	$\varnothing$	$\{x\}$	$\{x\}$	...
3	$\varnothing$	$\varnothing$	$\varnothing$	$\{x\}$	...
2	$\varnothing$	$\{x\}$	$\{x\}$	$\{x\}$	...
1	$\varnothing$	$\varnothing$	$\{x\}$	$\{x\}$	...

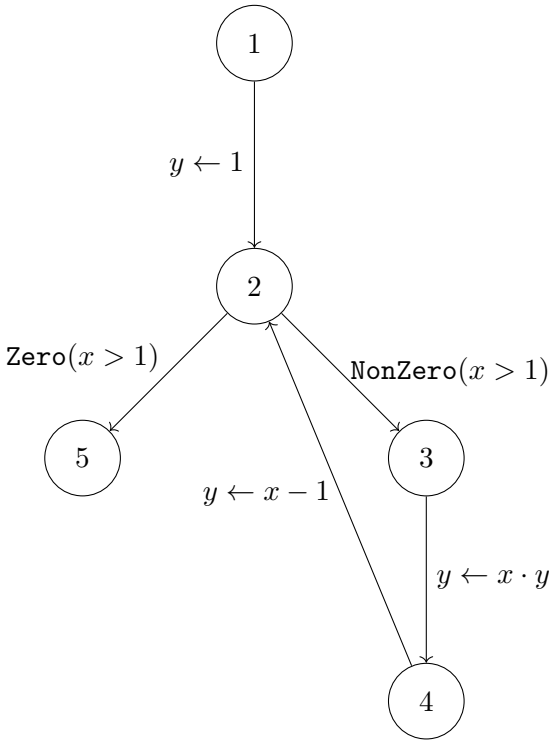
TLiveIn	1	2	3	...
6	$\varnothing$	$\varnothing$	$\varnothing$	...
5	$\varnothing$	$\{x\}$	$\{x\}$	...
4	$\varnothing$	$\varnothing$	$\{x\}$	...
3	$\{x\}$	$\{x\}$	$\{x\}$	...
2	$\varnothing$	$\{x\}$	$\{x\}$	...
1	$\varnothing$	$\varnothing$	$\{I\}$	...

Poiché la variabile y non viene mai utilizzata nelle espressioni, vediamo che non sarà mai *live*.

Approccio semantico

Riportiamo il programma programma precedentemente analizzato.

```
1 x ← I;
2 y ← 1;
3 while x ≥ 1 do
4 |   y ← y · x;
5 |   x ← x − 1;
```



- $\mathcal{L}(6) \supseteq \varnothing$
- $\mathcal{L}(5) \supseteq [\![x \leftarrow x - 1]\!]^\sharp \mathcal{L}(3) = \mathcal{L}(3) \setminus \{x\} \cup \{x \mid x \in \mathcal{L}(3)\}$
- $\mathcal{L}(4) \supseteq [\![y \leftarrow x \cdot y]\!]^\sharp \mathcal{L}(5) = \mathcal{L}(5) \setminus \{y\} \cup \{x, y \mid x \in \mathcal{L}(5)\}$
- $\mathcal{L}(3) \supseteq \mathcal{L}(6) \cup \mathcal{L}(4) \cup \{x\}$
- $\mathcal{L}(2) \supseteq \mathcal{L}(3) \cup \mathcal{L}(6) \setminus \{y\}$
- $\mathcal{L}(1) \supseteq \mathcal{L}(2) \setminus \{x\} \cup \{I \mid x \in \mathcal{L}(2)\}$

	1	**2**	**3**	...
6	$\varnothing$	$\varnothing$	$\varnothing$	...
5	$\varnothing$	$\varnothing$	$\{x\}$	...
4	$\varnothing$	$\varnothing$	$\{x\}$	...
3	$\varnothing$	$\{x\}$	$\{x\}$	...
2	$\varnothing$	$\{x\}$	$\{x\}$	...
1	$\varnothing$	$\varnothing$	$\{I\}$	...

Siamo arrivati allo stesso risultato dell'approccio algoritmo.

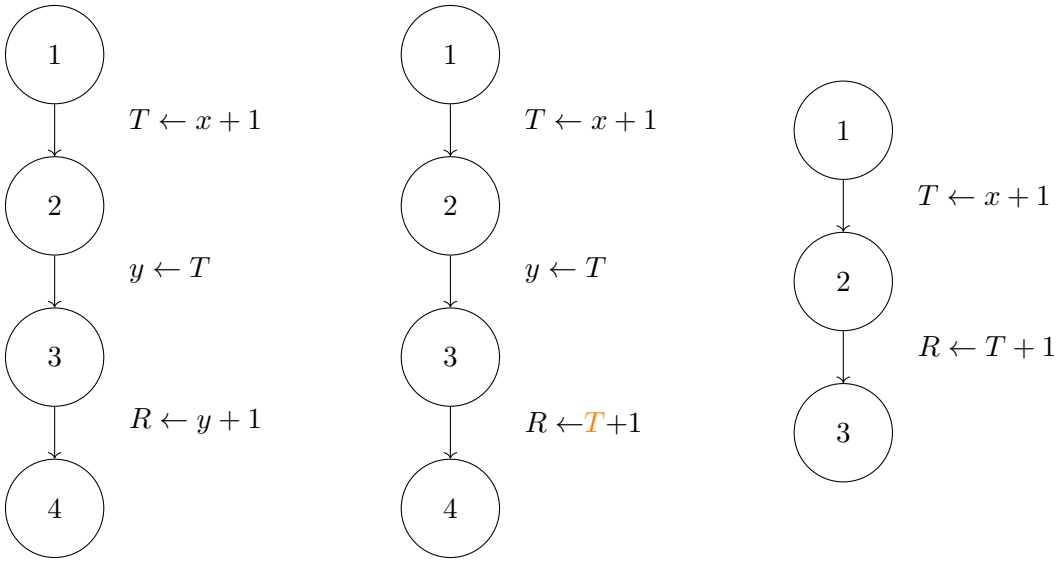
Osserviamo che la true liveness cattura falsi allarmi che un'applicazione ripetuta della liveness non saremmo in grado di catturare.

4.7 Analisi Copy propagation

La propagazione delle coppie riguarda l'idea di analizzare quali variabili durante l'esecuzione uno sono la copia dell'altra. L'idea è quella di ottimizzare andando ad utilizzare una sola delle due.

Per pensare all'utilità di tale analisi, pensiamo al contesto in cui vengono utilizzate, ovvero per l'ottimizzazione del codice intermedio.

Vediamo un'esempio di applicazione di tale ottimizzazione.



Notiamo che la variabile y è la copia di T e quindi possiamo sostituire y con T. L'idea è quindi di sostituire la variabile T all'interno dell'uso di y. Quello ottenuto è un codice in cui, applicando la *liveness* eliminiamo l'assegnamento $y \leftarrow T$.

4.7.1 Costruzione dell'informazione

Vediamo come costruire l'informazione che ci serve per propagare queste copie. Vorremmo che l'analisi mantenga ad ogni punto di programma l'insieme delle variabili che contengono una copia di una variabile x fissata. Se otteniamo questo, allora gni occorrenza di una copia di x può essere sostituita con x, ovviamente più è grande tale insieme, più copie possiamo sostituire e quindi ad ottimizzare il programma.

Dal punto di vista dell'analisi, è poco verosimile sapere a priori sapere a priori qual è una variabile per la quale è utile cercare le copie. Dobbiamo quindi estendere l'idea propagando la copia di tutte le variabili, raccogliendo l'insieme delle variabili che sono una la copia dell'altra, determinando quindi per ogni punto di programma quali variabili sono copie di altre.

L'informazione osservata è ora un insieme di coppie di variabili: $\texttt{Var} \to \texttt{Var}$, quindi $(x, y) \implies$ x è copia di y. Dal punto di vista della distruzione, distruggiamo le coppie quando una delle

due variabili viene sovrascritta, quindi dal punto di vista della distruzione c'è commutatività, mentre per la generazione non c'è commutatività, poiché c'è una direzione di generazione.

Dobbiamo definiamo $\texttt{Copy}_{out}$ e $\texttt{Copy}_{in} \subseteq \texttt{Var} \to \texttt{Var}$, dove $\texttt{Copy}_{out}$ è l'insieme delle coppie di variabili disponibili in uscita, mentre $\texttt{Copy}_{in}$ è l'insieme delle coppie di variabili disponibili in ingresso ad ogni blocco.

È chiaro che stiamo prendendo in considerazione il programma nella direzione di esecuzione. Poiché che c'è un assegnamento tra una variabile e un'altra e utilizziamo l'informazione per sostituire la variabile assegnata con l'analisi di cui è copia, perciò è **forward**. Rispetto al modo di combinazione, notiamo che collezioniamo variabili che sicuramente sono copie di altre, quindi ci dobbiamo assicurare che qualunque cammino che porti al punto di sostituzione, abbia tale copia, perciò l'analisi è di tipo **definite**.

4.7.2 Analisi di Copy Propagation e l'approccio algoritmico

Dobbiamo definire l'informazione generata e distrutta.

$$\texttt{Gen}(b) = \{(x, y) \mid \exists x := y \in b\} \tag{4.23}$$

$$\texttt{Kill}(b) = \{(x, y) \mid (\exists x := e \in b \vee \exists y := e \in b) \wedge x \neq y\} \tag{4.24}$$

L'informazione iniziale sull'entry point è ciò che vale sempre, ovvero:

$$x = \{(x, x) \mid x \in \texttt{Var}\}$$

Perché è sempre vero che una variabile è copia di se stessa.

Possiamo quindi costruire lle equazioni:

$$\texttt{Copy}_{in}(n) = \begin{cases} x & \text{se } n = \text{entry} \\ \bigcap_{p \in \text{pred}(n)} \texttt{Copy}_{out}(p) & \text{altrimenti} \end{cases} \tag{4.25}$$

$$\texttt{Copy}_{out}(n) = \texttt{Gen}(n) \cup (\texttt{Copy}_{in}(n) \setminus \texttt{Kill}(n)) \tag{4.26}$$

Notiamo che l'equazione è stata costruita partendo dalle informazioni date precedentemente, ovvero che viene utilizzata l'intersezione dato che l'analisi è *definite*. Il fatto che sia un'analisi analisi **forward** proviene dall'informazione iniziale sulla entry e il fatto che venga fatta l'intersezione sui **predecessori**.

4.7.3 Analisi di Copy Propagation e l'approccio semantico

Il dominio di osservazione è l'insieme delle coppie di variabili, quindi $\texttt{Var} \times \texttt{Var}$, e l'informazione iniziale è esattamente lo stesso insieme $x = \{(x, x) \mid x \in \texttt{Var}\}$. Le informazioni mancanti sono le informazione di calcolo vero e proprio, ovvero la semantica, quindi l'**abstract edge effect**. Definita in modo induttivo sulla struttura sintattica del linguaggio $\mathcal{C} \subseteq \texttt{Var} \times \texttt{Var}$. Quindi:

- $[\![;]\!]^\sharp \mathcal{C} = \mathcal{C}$

- $[\![\texttt{NonZero}(e)]\!]^\sharp\mathcal{C} = [\![\texttt{Zero}(e)]\!]^\sharp\mathcal{C} = \mathcal{C}$
- $[\![\texttt{input(x)}]\!]^\sharp\mathcal{C} = \mathcal{C} \setminus \{x\}$
- $[\![x \leftarrow e]\!]^\sharp\mathcal{C} = \begin{cases} (\mathcal{C} \setminus \texttt{Copy}(x)) \cup \{(x,y)\} & e \equiv y \\ \mathcal{C} \setminus \texttt{Copy}(x) & \text{altrimenti} \end{cases}$

Dove:

$$\texttt{Copy}(x) = \{(x,y) \mid y \in \texttt{Var} \wedge x \neq y\} \cup \{(y,x) \mid y \in \texttt{Var} \wedge x \neq y\}$$

Ricordiamo che la soluzione migliore che vogliamo ottenere è la **MOP**, quindi:

$$\mathcal{C}^\star[v] = \bigcap\{[\![\pi]\!]^\sharp x \mid \pi : \texttt{entry} \to c\}$$

Dove la semantica del cammino π è calcolata in modo forward: $[\![\pi]\!]^\sharp = [\![k_n]\!]^\sharp \circ \cdots \circ [\![k_1]\!]^\sharp x$. La semantica è monotona e distributiva, quindi $\mathcal{C}^\star[v]$ può essere calcolata come soluzione del sistema di disequazioni.

Forniamo quindi l'informazione di trasferimento locale ad ogni punto programma, ignorando il calcolo sull'intero cammino caratterizzando l'informazione locale su ogni arco.

- $\mathcal{C}[\texttt{entry}] \subseteq x$
- $\mathcal{C}[v] \subseteq [\![\texttt{lab}]\!]^\sharp\mathcal{C}[u]$

Notiamo la presenza del contenimento dato che l'analisi è **definite** e il target dell'arco è scritto in funzione della sorgente $\cdot \xrightarrow{\texttt{lab}} \cdot$ è dato che l'analisi è **forward**.

La semantica distributiva fa si che la soluzione del sistema di disequazioni ovvero la **MFP**, sia uguale alla soluzione **MOP**.

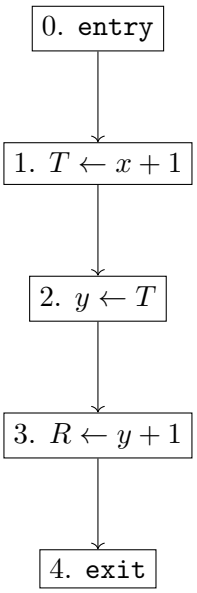
Esempio di analisi di Copy Propagation

Approccio algoritmico

Consideriamo il seguente programma:

```
1 T ← x + 1;
2 y ← T;
3 R ← y + 1;
```

Dove la notazione $(zz) = \{(z, z) \mid z \in \texttt{Var}\}$, mentre $(zw) = \{(z, w) \mid z, w \in \texttt{Var}\}$.



Costruiamo per ogni blocco l'insieme di espressioni generate e uccise:

- b_1: $\texttt{Gen}(1) = \varnothing$, $\texttt{Kill}(1) = \texttt{Copy}(T)$
- b_2: $\texttt{Gen}(2) = (yT)$, $\texttt{Kill}(2) = \texttt{Copy}(y)$
- b_3: $\texttt{Gen}(3) = \varnothing$, $\texttt{Kill}(3) = \texttt{Copy}(R)$

Ricordiamo che la coppia (zw) denota l'insieme di tutte le possibili coppie di variabili. Seguendo la formula per il calcolo del punto fisso della Copy Propagation costruiamo le due tabelle di $\texttt{Copy}_{in}$ e $\texttt{Copy}_{out}$, e otteniamo quindi:

$\texttt{Copy}_{in}$	**1**	**2**	**3**	**4**	...
0	(zz)	(zz)	(zz)	(zz)	...
1	(zw)	(zz)	(zz)	(zz)	...
2	(zw)	$(xy)(xR)(yR)$	(zz)	(zz)	...
3	(zw)	$(yT)(xT)(xR)(RT)$	$(yT)(xR)$	(yT)	...
4	(zw)	$(xy)(xT)(yT)$	$(yT)(xT)$	(yT)	...

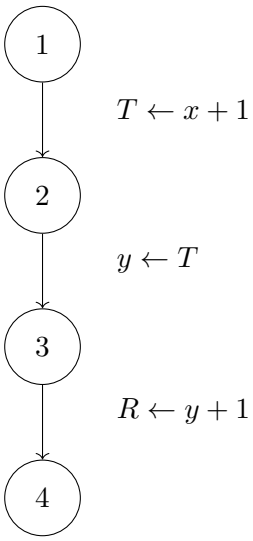
$\texttt{Copy}_{out}$	**1**	**2**	**3**	...
0	(zz)	(zz)	(zz)	...
1	$(xy)(xR)$	(zz)	(zz)	...
2	(yR)	$(yT)(xR)$	(yT)	...
3	$(yT)(xT)(xR)(RT)$	$(yT)(xT)$	(yT)	...

Notiamo che effettivamente dal nodo 3 in po y e T sono uno la copia dell'altro, quindi T può essere copiata in y.

Approccio semantico

Riportiamo il programma programma precedentemente analizzato.

```
1 T ← x + 1;
2 y ← T;
3 R ← y + 1;
```



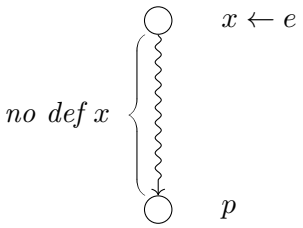
- $\mathcal{C}(1) \subseteq (zz)$
- $\mathcal{C}(2) \subseteq [\![T \leftarrow x + 1]\!]\mathcal{C}(1) = \mathcal{C}(1) \setminus \texttt{Copy}(T)$
- $\mathcal{C}(3) \subseteq [\![y \leftarrow T]\!]\mathcal{C}(2) = \mathcal{C}(2) \setminus \texttt{Copy}(y) \cup (yT)$
- $\mathcal{C}(4) \subseteq [\![R \leftarrow y + 1]\!]\mathcal{C}(3) = \mathcal{C}(3) \setminus \texttt{Copy}(R)$

	1	**2**	...
1	(zz)	(zz)	...
2	(zw)	(zz)	...
3	(zw)	(yT)	...
4	(zw)	(yT)	...

Siamo arrivati allo stesso risultato dell'approccio algoritmo, ma eseguendo una solo iterazione.

4.8 Analisi Reaching Definitions

Tale analisi ha come obiettivo quello di individuare, dato un punto di programma, quali sono gli assegnamenti che non vengono sovrascritti prima di raggiungere quel punto, quindi senza ridefinizioni intermedie.



Una definizione viene uccisa quando in un punto intermedio si va a riscrivere una definizione per una stessa variabile.

Si tratta di un analisi molto semplice, utilizzata in vari contesti. Un esempio di applicazione consiste nel *loop invariant code motion*, ovvero spostare all'esterno del ciclo del codice che non viene modificato all'interno del ciclo.

L'analisi è chiaramente *forward*, in quanto si parte da un punto di programma e si arriva ad un altro punto di programma, poiché arriviamo alla definizione, generiamo l'informazione e da lì in poi l'informazione vale finché non si trova una ridefinizione.

Il fatto che si considerino tutte le definizioni che raggiungono un punto di programma, rende l'analisi *possibile*, quindi colleziona per unione.

L'informazione osservata sarà le coppie tra variabili e punti di programma, ovvero $\texttt{Var} \times \texttt{pp}$. Quando scriveremo (x, p) intendiamo che la variabile x è definita al punto p.

Supponendo che il programma non abbia assegnamenti precedenti, l'informazione iniziale sarà l'insieme vuoto.

4.8.1 Analisi di Reaching Definitions e l'approccio algoritmo

Definiamo come l'informazione vine generata e distrutta:

$$\texttt{Gen}(n) = \{(x, n) \mid \exists x := e \in n \vee \texttt{input}(x) \in n\} \tag{4.27}$$

$$\texttt{Kill}(n) = \{(x, p) \mid \exists x := e \in n \vee \texttt{input}(x) \in n, p \in \texttt{pp}\} \tag{4.28}$$

Possiamo quindi definire le funzioni di punto fisso $\texttt{RD}_{in}$ e $\texttt{RD}_{out}$:,

$$\texttt{RD}_{in}(n) = \begin{cases} \varnothing & n = \texttt{entry} \\ \bigcup_{p \in pred(n)} \texttt{RD}_{out}(p) & \text{altrimenti} \end{cases} \tag{4.29}$$

Notiamo che l'equazione è stata costruita partendo dalle informazioni date precedentemente, ovvero che viene utilizzata l'unione dato che l'analisi è *possibile*. Il fatto che sia un'analisi analisi **forward** proviene dall'informazione iniziale sulla entry e il fatto che venga fatta l'intersezione sui **predecessori**.

$$\texttt{RD}_{out}(n) = \texttt{Gen}(n) \cup (\texttt{RD}_{in}(n) \setminus \texttt{Kill}(n)) \tag{4.30}$$

4.8.2 Analisi di Reaching Definitions e l'approccio semantico

Descriviamo l'abstract edge effect, ovvero l'elemento caratteristico dei singoli archi del `CFG`.

- $[\![;]\!]^\sharp\mathcal{R} = \mathcal{R}$
- $[\![\texttt{NonZero}(e)]\!]^\sharp\mathcal{R} = [\![\texttt{Zero}(e)]\!]^\sharp\mathcal{R} = \mathcal{R}$
- $[\![\texttt{input(x)}]\!]^\sharp\mathcal{R} = \mathcal{R} \setminus \texttt{def}(x) \cup \{\langle xu\rangle \mid \langle u, x \leftarrow e, v\rangle \ \textit{arco eseguito}\}$
- $[\![x \leftarrow e]\!]^\sharp\mathcal{R} = \mathcal{R} \setminus \texttt{def}(x) \cup \{\langle xu\rangle \mid \langle u, x \leftarrow e, v\rangle \ \textit{arco eseguito}\}$

Dove:

$$\texttt{Def}(x) = \{(x, p) \mid p \in \texttt{pp}\}$$

Come sempre la soluzione **MOP**, che chiamiamo $\mathcal{R}[v]$ è definita come:

$$\mathcal{R}^\star[v] = \bigcup\{[\![\pi]\!]^\sharp\varnothing \mid \pi : \texttt{entry} \to v\}$$

La semantica è monotona e distributiva, quindi $\mathcal{R}^\star[v]$ può essere calcolata come soluzione del sistema di disequazioni.

Forniamo quindi l'informazione di trasferimento locale ad ogni punto programma, ignorando il calcolo sull'intero cammino caratterizzando l'informazione locale su ogni arco.

- $\mathcal{R}[\texttt{entry}] \supseteq x$
- $\mathcal{R}[v] \supseteq [\![\texttt{lab}]\!]^\sharp\mathcal{R}[u]$

Notiamo la presenza del *supset* dato che l'analisi è **possibile** e il target dell'arco è scritto in funzione della sorgente $\cdot \xrightarrow{\texttt{lab}} \cdot$ è dato che l'analisi è **forward**.

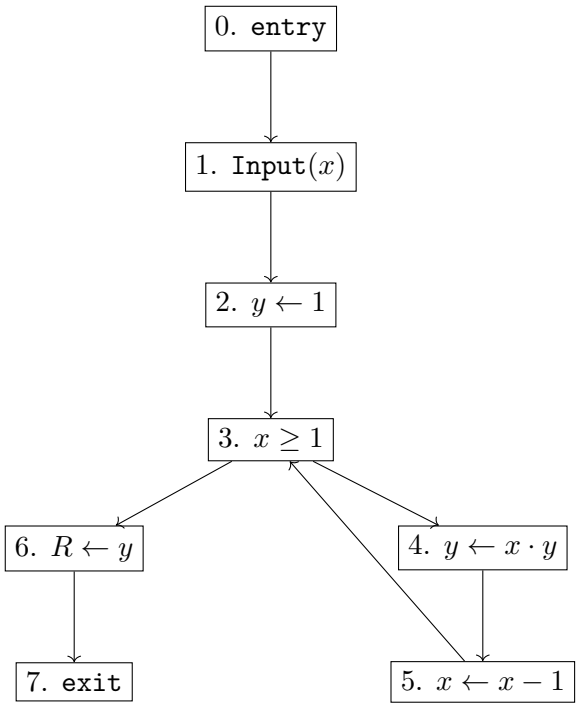
La semantica distributiva fa si che la soluzione del sistema di disequazioni ovvero la **MFP**, sia uguale alla soluzione **MOP**.

Esempio di analisi di Reaching Definitions

Approccio algoritmico

Consideriamo il seguente programma:

```
1 Input(x);
2 y ← 1;
3 while x ≥ 1 do
4 |   y ← y · x;
5 |   x ← x − 1;
6 R ← y;
```



Costruiamo per ogni blocco l'insieme di espressioni generate e uccise:

- b_1: $\texttt{Gen}(1) = (x1)$, $\texttt{Kill}(1) = (xn)$
- b_2: $\texttt{Gen}(2) = (y2)$, $\texttt{Kill}(2) = (yn)$
- b_3: $\texttt{Gen}(3) = \texttt{Kill}(3) = \varnothing$
- b_4: $\texttt{Gen}(4) = (y4)$, $\texttt{Kill}(4) = (yn)$
- b_5: $\texttt{Gen}(5) = (x5)$, $\texttt{Kill}(5) = (xn)$
- b_6: $\texttt{Gen}(6) = (R6)$, $\texttt{Kill}(6) = (Rn)$

Seguendo la formula per il calcolo del punto fisso della Reaching Definitions, costruiamo le due tabelle di $\texttt{RD}_{in}$ e $\texttt{RD}_{out}$, e otteniamo quindi:

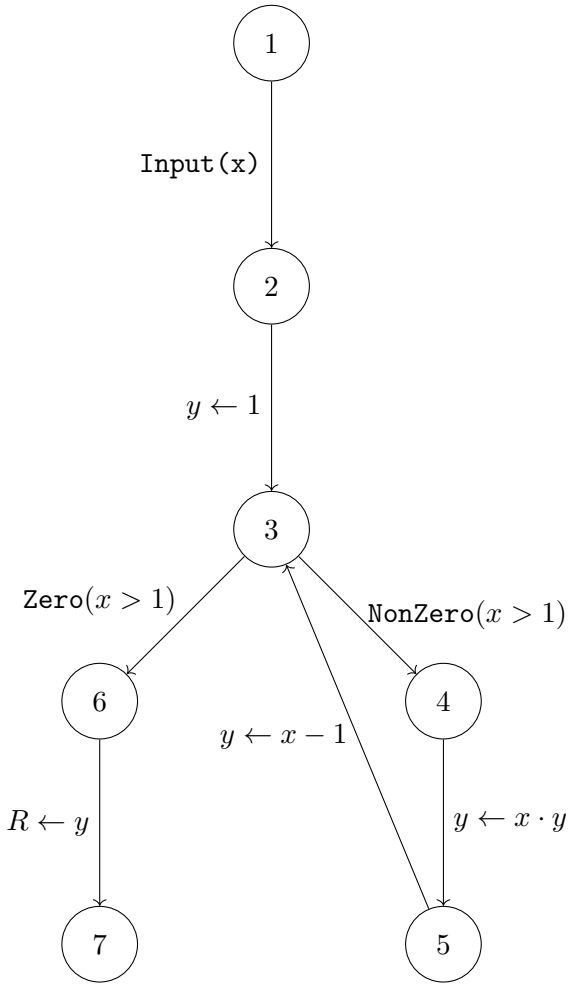
$\texttt{RD}_{out}$	**1**	**2**	**3**	**4**	**5**	...
0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
1	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
2	$\varnothing$	$(x1)$	$(x1)$	$(x1)$	$(x1)$	...
3	$\varnothing$	$(y2)(x5)$	$(y2)(x5)(x1)(y4)$	$(y2)(x5)(x1)(y4)$	$(y2)(x5)(x1)(y4)$	...
4	$\varnothing$	$\varnothing$	$(y2)(x5)$	$(y2)(x5)(x1)(y4)$	$(y2)(x5)(x1)(y4)$	...
5	$\varnothing$	$(y4)$	$(y4)$	$(y4)(x5)$	$(y4)(x5)(x1)$	...
6	$\varnothing$	$\varnothing$	$(y2)(x5)$	$(y2)(x5)(x1)(y4)$	$(y2)(x5)(x1)(y4)$	...
7	$\varnothing$	$(R6)$	$(R6)$	$(R6)(y2)(x5)$	$(R6)(y2)(x5)(x1)(y4)$	...

$\texttt{RD}_{in}$	**1**	**2**	**3**	**4**	...
0	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	...
1	$(x1)$	$(x1)$	$(x1)$	$(x1)$	...
2	$(y2)$	$(y2)(x1)$	$(y2)(x1)$	$(y2)(x1)$	...
3	$\varnothing$	$(y2)(x5)$	$(y2)(x5)(x1)(y4)$	$(y2)(x5)(x1)(y4)$	...
4	$(y4)$	$(y4)$	$(y4)(x5)$	$(x5)(x1)(y4)$	...
5	$(x5)$	$(x5)(y4)$	$(x5)(y4)$	$(y4)(x5)$	...
6	$(R6)$	$(R6)$	$(R6)(y2)(x5)$	$(R6)$	...

Approccio semantico

Riportiamo il programma programma precedentemente analizzato.

```
1 Input(x);
2 y ← 1;
3 while x ≥ 1 do
4 |   y ← y · x;
5 |   x ← x − 1;
6 R ← y;
```



- $\mathcal{R}(1) \supseteq \varnothing$
- $\mathcal{R}(2) \supseteq [\![\texttt{input(x)}]\!]^\sharp \varnothing = (x1)$
- $\mathcal{L}(3) \supseteq [\![y \leftarrow 1]\!]^\sharp \mathcal{R}(2) = \mathcal{R}(2) \setminus \texttt{def}(y) \cup (y2) \cup (\mathcal{R}(5) \setminus \texttt{def}(x) \cup (x5))$
- $\mathcal{R}(4) \supseteq \mathcal{R}(3)$
- $\mathcal{R}(5) \supseteq [\![y \leftarrow x \cdot y]\!]^\sharp \mathcal{R}(4) = \mathcal{R}(4) \setminus \texttt{def}(y) \cup (y4)$
- $\mathcal{R}(6) \supseteq \mathcal{R}(3)$
- $\mathcal{R}(7) \supseteq [\![R \leftarrow y]\!]^\sharp \mathcal{R}(6) = \mathcal{R}(6) \setminus \texttt{def}(R) \cup (R6)$

	1	**2**	**3**	...
1	$\varnothing$	$\varnothing$	$\varnothing$	...
2	$\varnothing$	$(x1)$	$(x1)$	...
3	$\varnothing$	$(x1)(y2)(x5)$	$(x1)(y2)(y4)(x5)$	...
4	$\varnothing$	$(x1)(y2)(x5)$	$(x1)(y2)(y4)(x5)$	...
5	$\varnothing$	$(x1)(x5)(y4)$	$(x1)(x2)(y4)$	...
6	$\varnothing$	$(x1)(y2)(x5)$	$(x1)(y2)(y4)(x5)$	...
7	$\varnothing$	$(x1)(y2)(x5)$	$(x1)(y2)(y4)(x5)$	...

Capitolo 5

Interpretazione astratta

5.1 Introduzione

Nelle analisi precedenti l'approccio all'analisi statica è basato sull'informazione che vogliamo osservare dell'esecuzione del programma, in particolare per proprietà relativamente semplici, molto vicine alla sintassi che non descrivono informazioni sul contenuto delle variabili, quindi in relazione agli elementi del programma piuttosto che al contenuto di tali elementi.

Per questo tipo di analisi, o in modo algoritmico fornendo direttamente l'equazione da risolvere, o in modo semantico, fornendo una semantica di elaborazione dell'informazione, costruita in modo induttivo sulla sintassi del linguaggio, riusciamo a fornire una soluzione sull'informazione ricercata.

Il problema si pone quando vogliamo guardare ciò che accade al programma durante l'esecuzione, siamo quindi interessati a guardare all'interno delle stato della macchina, e non più alla relazione tra gli elementi del programma. Questo diventa relativamente un problema, poiché fornendo una semantica, tale semantica non risulterebbe distributiva, di conseguenza ci sarà necessariamente una perdita di informazione legata al fatto che risolvendo l'analisi localmente su punti di programma del `CFG` perdiamo informazione rispetto alla semantica desiderata, ovvero quella che considera tutti i cammini di esecuzione, ovvero la **MOP**. Si tratta del prezzo da pagare per rendere decidibile un calcolo su un informazione che si avvicina sempre di più all'informazione concreta, ovvero l'evoluzione dello stato della macchina.

Guardare all'interno dello stato della macchina significa avvicinarsi sempre di più all'informazione concreta, ma richiede un prezzo che è quello di staccarci dalla soluzione **MOP**, ottenendo attraverso la soluzione del calcolo di un sistema di disequazioni, una soluzione approssimata.

Nel momento in cui siamo interessati a guardare all'interno dello stato della macchina, allora abbiamo bisogno, per mantenere delle garanzie sul calcolo della semantica astratta, di un'infrastruttura chiamata **interpretazione astratta**.

Interpretazione astratta

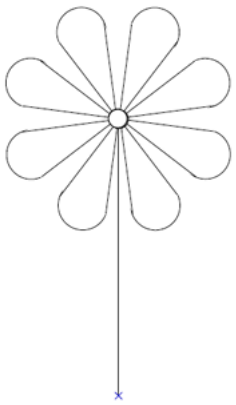
L'interpretazione astratta è un framework formale che permette di descrivere dettagliatamente la relazione tra il momento concreto, di cui vogliamo dire qualcosa, e il momento astratto, su cui possiamo dire qualcosa, garantendo informazioni certe, anche se approssimate, su alcuni aspetti di interesse del mondo concreto.

L'obiettivo è di verificare in modo automatico proprietà di interesse dei programmi e l'astrazione è il procedimento utilizzato per arginare il problema della non decidibilità della semantica concreta.

5.1.1 L'idea di base dell'interpretazione astratta

L'idea di base è che un qualunque oggetto concreto sia formato da due elementi, un origine e un insieme finito di punti.

Supponiamo di definire nel nostro dominio concreto un oggetto fiore e vediamo come è possibile ottenerlo attraverso l'applicazione di operazioni concrete e partendo da operazioni concrete.

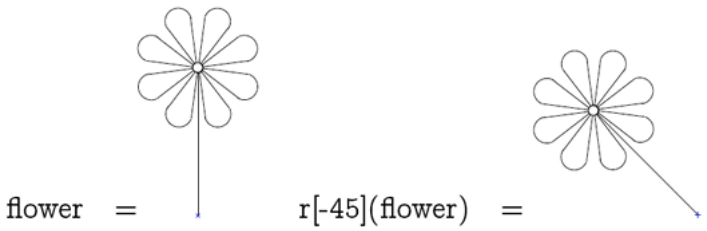


Tali oggetti non sono altro che dei numeri interni, quindi immaginiamo il fiore come una sequenza di operazioni che applichiamo ad elementi più piccoli, ovvero elementi del dominio stesso.

5.1.2 Operazioni concrete

Le operazioni che possiamo applicare sono le seguenti:

- Costante: operazione che fornisce un petalo.
- Rotazione: $r[a](o)$ è un'operazione che ruota di a gradi l'oggetto o.



- Unione: $o_1 \cup o_2$ è un'operazione che unisce due oggetti o_1 e o_2, sovrapponendo le origini.

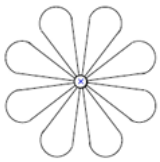
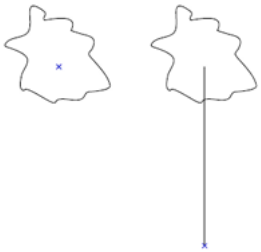


Figura 5.1.1: `corolla` = `petal` $\cup\, r[45]$`petal` $\cup\, r[90]$`petal` $\cup\, r[135]$`petal` $\cup\, r[180]$`petal` $\cup\, r[225]$`petal` $\cup\, r[270]$`petal` $\cup\, r[315]$`petal`

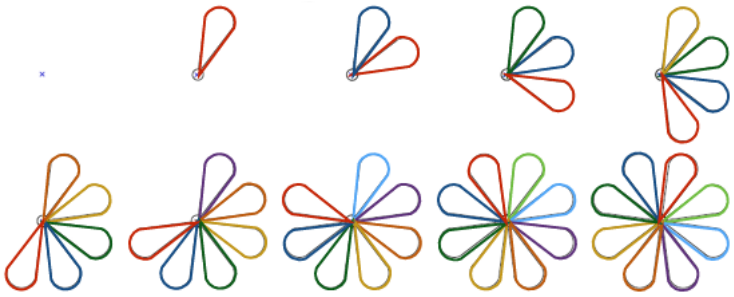
- *stem*(o): è un'operazione che aggiunge un gambo all'oggetto o, il gambo viene sovrapposto all'origine e la nuova origine è quella del gambo.



Per la costruzione della corolla possiamo trovare un operatore monotono che applicato iterativamente fino al raggiungimento di un punto fisso, ci permette di ottenere la corolla. Possiamo costruire oggetti concreti per punto fisso a partire da oggetti più semplici, che è ciò che avviene con la semantica.

$$\texttt{corolla} = lfp^{\subseteq}F$$

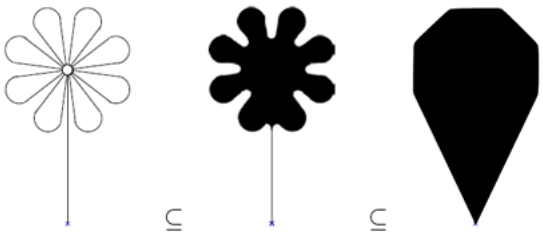
$$F(X) = \texttt{petal} \cup r[45](X)$$



Abbiamo quindi introdotto un dominio concreto di oggetti su cui abbiamo definito delle operazioni.

5.1.3 Approssimazione verso l'alto

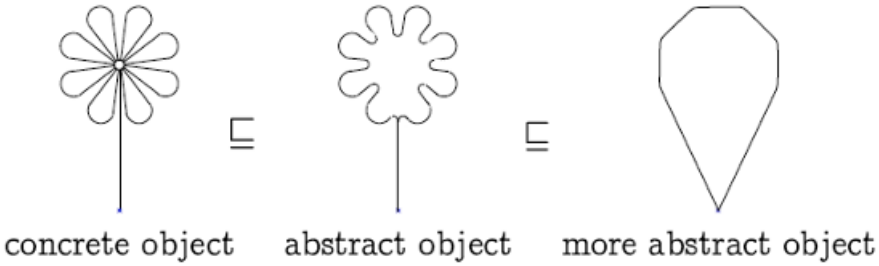
Operare su tali oggetti può comportare situazioni di non decidibilità, l'idea è quindi di approssimare tali oggetti concreti. Definiamo la relazione di approssimazione, ovvero cosa vuol dire essere meno precisi. Nel caso degli oggetti, quello che è essere meno precisi è avere la stessa origine, ma che contengono più pixel, ovvero aggiungere rumore rispetto all'informazione originale.



In questo particolare caso abbiamo perso parte del dettaglio della figura, in particolare la corolla. Aver più pixel aggiunge quindi rumore.

5.1.4 Oggetti astratti

L'oggetto astratto è una rappresentazione di un oggetto concreto che vogliamo fornire in forma approssimata. Decidiamo di rappresentare l'insieme di pixel dal suo contorno. Questo è l'ordine che inseriamo nel dominio di computazione, propagando l'ordinamento, che nel dominio concreto è dato dal contenimento, sul dominio astratto. Un oggetto è quindi più astratto se è la rappresentazione di un oggetto concreto più grande.



L'idea è che il contorno di un certo oggetto venga disegnato con una pennarello di spessore variabile, la variabilità di tale spessore è data dal livello di astrazione. Disegnando un un pennarello più spesso, si perde dettaglio, mentre disegnando con un pennarello più sottile, si guadagna dettaglio.

Dominio astratto

Il dominio astratto è l'insieme di tutti gli oggetti astratti, più le operazioni astratte che permettono di costruire oggetti astratti (*che approssimano le operazioni concrete*).

Astrazione

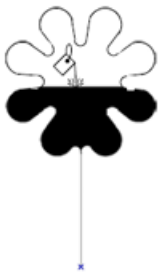
L'astrazione è una funzione α che mappa ogni oggetto concreto in una approssimazione rappresentata da un oggetto astratto $\alpha(o)$ in modo univoco.

Il tutto è costruito in modo tale da poter confrontare oggetti astratti tra di loro, scegliendo di guardare l'informazione approssimata allargando il diametro del pennarello.

5.1.5 Concretizzazione

Nel mondo dell'interpretazione astratta corrisponde ad assegnare il significato concreto dell'oggetto astratto che stiamo osservando.

Nel caso degli oggetti rappresentati precedentemente, la concretizzazione è possibile vederla come l'idea di riempire il contorno dell'oggetto astratto.



Si tratta di una funzione γ che mappa ogni oggetto astratto $\bar{o}$ in un oggetto concreto, in modo univoco $\gamma(\bar{o})$.

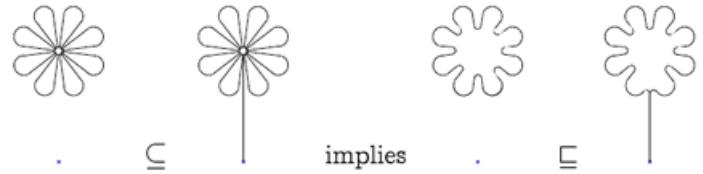
Prendendo in considerazione l'insieme $\{2, 6\}$ l'astrazione di tale insieme, ovvero l'applicazione α, può essere visto come la funzione che mappa l'insieme $\{2, 6\}$ (*fiore concreto*) in un oggetto astratto che rappresenta l'insieme $2\mathbb{Z}$ (*fiore astratto*). L'applicazione della funzione γ su tale oggetto astratto, fa si che si ottenga $\{0, 2, 4, 6, \dots\}$ (*fiore riempito*).

5.1.6 Connessione di Galois

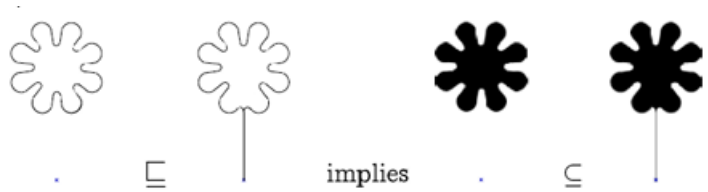
Queste due funzioni di concretizzazione e astrazione nell'interpretazione astratta sono modellate dal concetto di **connessione di Galois**.

Dice infatti che è una coppia di funzioni, tali che:

- α è monotona



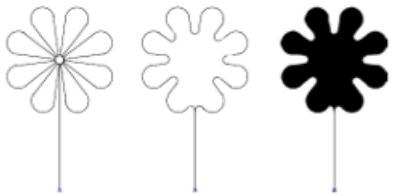
- γ è monotona



La monotonia fa si che preservino l'ordine tra i due domini.

In più le connessioni di Galois devono estendere quando si esegue il processo di astrazione, quindi il processo di astrazione può aggiungere rumore:

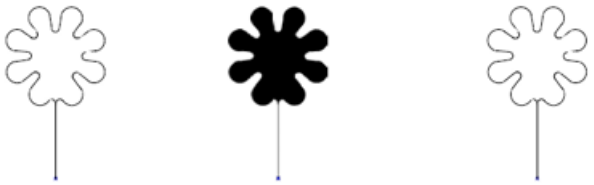
$$\textit{per ogni oggetto concreto } x, \gamma \circ \alpha(x) \subseteq x$$



Inoltre se prima concretizziamo e poi astraiamo, otteniamo un oggetto astratto più piccolo di quello di partenza, questo perché nel mondo concreto potrebbero esserci oggetti inutili che non hanno significato.

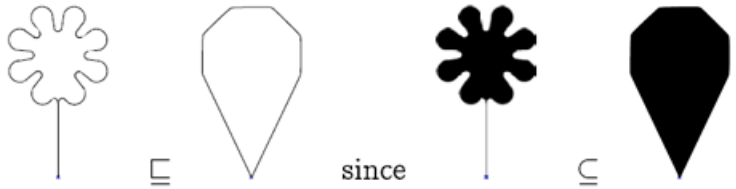
$$\textit{per ogni oggetto astratto } y, \alpha \circ \gamma(y) \sqsubseteq y$$

Tipicamente si ha che $\alpha \circ \gamma(y) = y$.



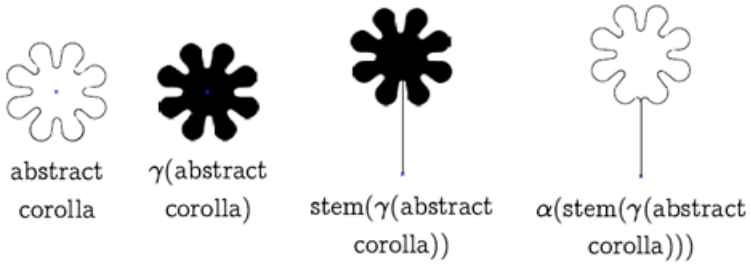
5.1.7 Ordinamento astratto

L'ordinamento astratto è il trasferimento dell'ordinamento concreto sul dominio astratto. Quindi un oggetto astratto è più piccolo di un altro oggetto astratto se la corrisponde concretizzazione è più piccola.



In origine ciò che volevamo fare era calcolare delle operazioni sul dominio concreto con l'assunzione che tali operazioni non fossero sempre decidibili. L'obiettivo è trasferire tali operazioni dal mondo concreto al mondo astratto, in modo tale da poterle calcolare in modo sempre decidibile. Il modo più immediato per farlo è quello di passare attraverso le operazioni concrete. Nell'esempio del gambo abbiamo in fatti che:

$$\overline{\texttt{stem}} = \alpha(\texttt{stem}(\gamma(y)))$$



Non potevamo aggiungere direttamente il gambo alla corolla astratta poiché avremmo avuto un livello di dettaglio diverso dal risultato finale che non avrebbe coinciso con lo spessore

del pennarello scelto per osservare l'oggetto concreto. È necessario definire un'operazione ad hoc per poter lavorare su oggetti astratti, preservando quindi l'informazione che abbiamo sull'oggetto astratto, ovvero che il suo contorno è stato delimitato con un certo spessore e su tutte le operazioni è possibile eseguire tale passaggio.

5.1.8 Punto fisso astratto

Esattamente come le operazioni precedenti, anche il calcolo di punto fisso può essere trasferito dal mondo concreto a quello astratto.

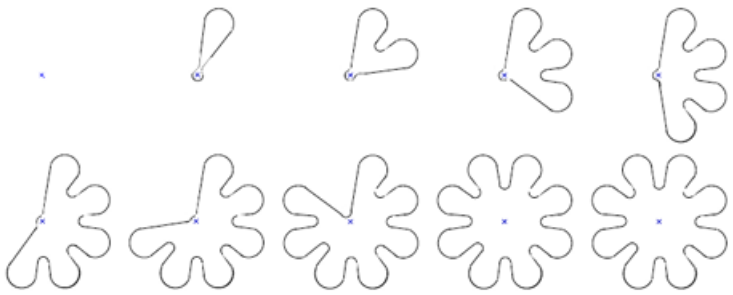
Possiamo eseguire direttamente un'astrazione del punto fisso:

$$\texttt{abstract corolla} = \alpha(\texttt{concrete corolla}) = \alpha(lfp^{\subseteq}F)$$

Dove $F(X) = \texttt{petal} \cup r[45](X)$. Sappiamo però che possa ereditare problemi di decidibilità e di convergenza, poiché il punto fisso se esiste in presenza di funzioni monotone, può divergere. È chiaro che poiché ci spostiamo nel mondo astratto per controllare questo tipo di divergenza, non avrebbe senso calcolare il punto fisso concreto e poi astrarlo. L'obiettivo è quello di costruire un operatore astratto il cui punto fisso nel mondo ideale che sia esattamente l'astrazione del punto fisso concreto, però con il fatto il punto fisso astratto è calcolato su un mondo approssimato e quindi potenzialmente convergente.

$$\alpha(lfp^{\subseteq}F) = lfp^{\sqsubseteq}\overline{F}$$

In generale si tratta della situazione ideale, in cui abbiamo costruito esattamente l'operatore di punto fisso che sul nostro dominio riesce a raggiungere precisamente la proprietà del punto fisso concreto. In generale quello che succede è che il punto fisso astratto sarà un'approssimazione ulteriore della proprietà del punto fisso concreto.



5.2 Costruzione formale dell'interpretazione astratta

Vediamo ora gli strumenti formali che utilizziamo per modellare e descrivere quelle che sono le basi dell'interpretazione astratta.

Il concetto principale utilizzato per descrivere l'interpretazione astratta, ovvero il legame che c'è tra mondo concreto, nel quale vogliamo calcolare una semantica e nel quale vorremmo dare risposte, e il mondo astratto, nel quale potenzialmente possiamo calcolare la semantica,

e nel quale possiamo dare delle risposte, sono le **connessioni di Galois**. Per farlo dobbiamo costruire questo mondo astratto nel rispetto di alcuni vincoli, in modo tale da dare determinate garanzie sul passaggio dal mondo concreto al mondo astratto.

Non si tratta dell'unico strumento formale, ma alcuni di questi sono completamente equivalenti.

Connessione di Galois

La connessione di Galois `GC` è una coppia di funzioni monotone α e γ definite tra un dominio concreto $\mathcal{C}$ e un dominio astratto $\mathcal{A}$.

In generale $\mathcal{A}$ e $\mathcal{C}$ possono essere dei *poset*, per semplicità di formalizzazione consideriamo $\mathcal{A}$ e $\mathcal{C}$ come reticoli completi, dove è quindi definito un *least upper bound* e un *greatest lower bound*, oltre all'ordine parziale. Gli ordinamenti sono definiti come segue:

- $(\mathcal{A}, \leq)$;
- $(\mathcal{C}, \leq)$;

Le due funzioni sono definite come segue:

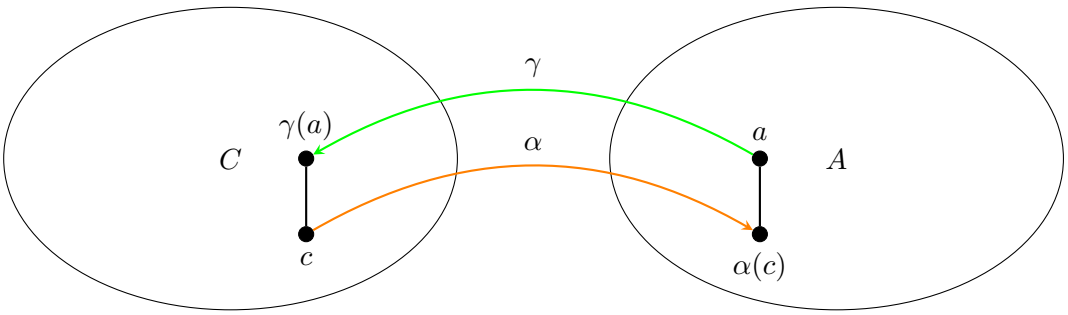
- $\alpha : \mathcal{C} \to \mathcal{A}$ è detta di **astrazione**
- $\gamma : \mathcal{A} \to \mathcal{C}$ è detta di **concretizzazione**

α e γ formano una connessione di Galois se sono monotone, ovvero:

$$\forall x, y \in \mathcal{C}.x \leq_{\mathcal{C}} y \implies \alpha(x) \leq_{\mathcal{A}} \alpha(y)$$

$$\forall x, y \in \mathcal{A}.x \leq_{\mathcal{A}} y \implies \gamma(x) \leq_{\mathcal{C}} \gamma(y)$$

e $\alpha(c) \leq_{\mathcal{A}} a \iff c \leq_{\mathcal{C}} \gamma(a)$.



Dire che l'astrazione di un elemento è più piccolo di un certo oggetto astratto equivale a dire che la concretizzazione di tale oggetto astratto è più grande dell'oggetto astratto da cui siamo partiti.

In generale α è chiamata aggiunta a destra, mentre γ è chiamata aggiunta a sinistra.

La monotonia garantisce di preservare l'ordine, mentre la condizione $\alpha(c) \leq_{\mathcal{A}} a \iff c \leq_{\mathcal{C}} \gamma(a)$ garantisce l'esistenza, nel mondo astratto, della migliore approssimazione possibile per ogni elemento concreto.

Come notazione, la connessione di Galois è rappresentata come segue:

$$(\mathcal{C}, \leq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \leq_{\mathcal{A}})$$

Inoltre la condizione $\alpha(c) \leq_A a \iff c \leq_C \gamma(a)$ equivale a dire che:

$$c \leq_{\mathcal{C}} \gamma \circ a(c) \iff \alpha \circ \gamma(a) \leq_{\mathcal{A}} a$$

questo perché non ponendo vincoli sul dominio astratto, può essere che vi siano elementi assolutamente inutili per rappresentare la proprietà che ci interessa osservare e tali elementi hanno un significato che coincide al significato di altri elementi astratti. Avendo più elementi astratti con lo stesso significato e supponendo che a abbia lo stesso significato di $\alpha(c)$, applicando la funzione γ entrambi gli elementi vengono mappati sullo stesso elemento concreto, e tornando indietro con α si ottiene si ottiene $\alpha(c)$.

Le inserzioni di Galois, invece, differiscono dalle connessioni di Galois nel fatto che $c \leq_{\mathcal{C}} \gamma \circ a(c)$ diventa un'uguaglianza, quindi:

$$c \leq_{\mathcal{C}} \gamma \circ a(c) \iff \alpha \circ \gamma(a) = a$$

Hanno come caratteristica di non avere nel dominio astratto A nessun elemento inutile, quindi ogni elemento astratto ha uno specifico significato concreto che ci interessa osservare. Catturano meglio perché catturano tutti gli elementi significativi del dominio.

La rappresentazione dell'inserzione di Galois è la seguente:

$$(\mathcal{C}, \leq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\rightarrow}} (\mathcal{A}, \leq_{\mathcal{A}})$$

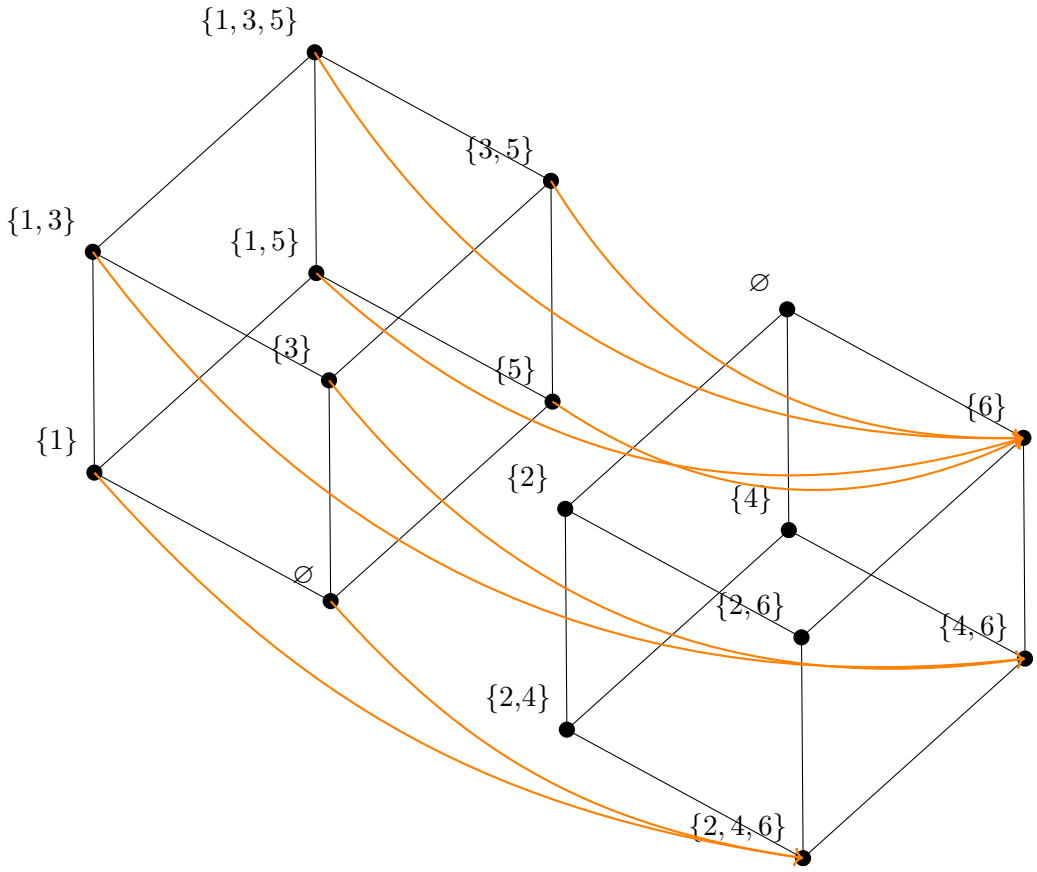
Poiché α è una funzione suriettiva.

Ogni connessione di Galois può essere ridotta ad una inserzione di Galois, eliminando gli elementi astratti inutili, ovvero quelli per cui esiste un elemento più piccolo che ha lo stesso significato.

5.2.1 Esempio di inserzione e connessione di Galois

Vogliamo vedere cosa vuol dire quando abbiamo a disposizione una connessione di Galois che non è effettivamente un'inserzione di Galois, con elementi inutili nel nostro dominio astratto, ridurre la connessione di Galois ad una inserzione di Galois.

Supponiamo di disporre di questi due domini, dove il dominio concreto è $\wp(\mathcal{A})$ e come dominio astratto abbiamo $\wp(\mathcal{B})^d$, dove $\wp(\mathcal{B})^d$ è ordinato per inclusione inversa.



L'obiettivo è costruire una funzione di concretizzazione che permetta di vedere l'insieme a destra come un'astrazione del dominio a sinistra.

$$\forall \mathcal{A}' \subseteq \mathcal{A}. \quad \alpha(\mathcal{A}') = \{b \in \mathcal{B} \mid \mathcal{A}' \subseteq b\}$$

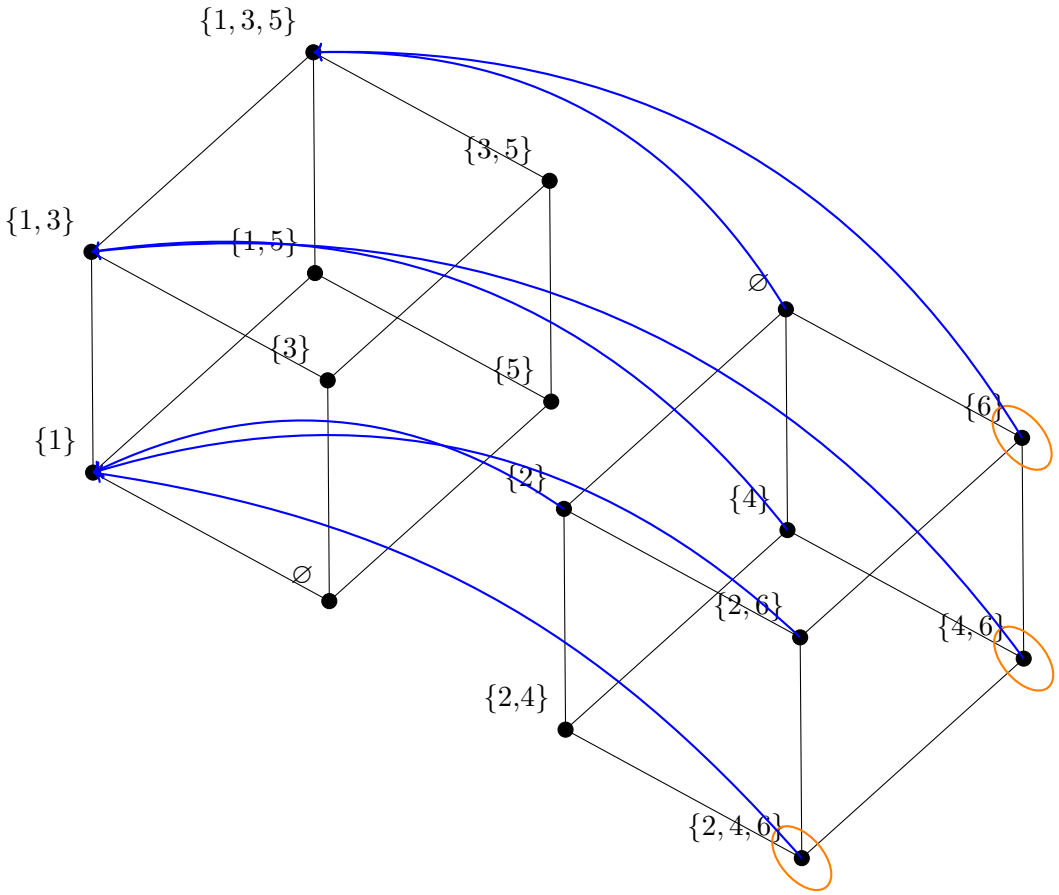
Dove dal punto di vista formale si tratta di una funzione monotona che mappa un insieme nell'insieme più grande di elementi in cui tutti gli elementi sono più grandi dell'insieme di partenza.

Da quello che osserviamo solo tre elementi di elementi sono immagine di α secondo la definizione.

Tutti gli altri elementi sono inutili all'interno del significato di tale astrazione. Fatto che possiamo verificare calcolando la funzione γ che è la funzione di astrazione.

$$\forall \mathcal{B}' \subseteq \mathcal{B}. \quad \gamma(\mathcal{B}') = \{a \in \mathcal{A} \mid a \subseteq \mathcal{B}'\}$$

La funzione γ preso un elemento astratto restituisce il suo significato concreto, in questo caso l'insieme massimale di elementi che sono più piccoli.



Quindi emergono 3 elementi, ovvero $\{1,3,5\}$, $\{1,3\}$ e $\{1\}$, che sono i più concreti tra quelli che hanno la stessa immagine astratta, ovvero $\{4\}$, $\{4,6\}$ e $\{6\}$.

Verifichiamo che α e γ rispettano le condizioni sull'essere connessioni di Galois.

Partendo dall'insieme vuoto e applicando α otteniamo l'insieme $\{4\}$ che è l'immagine astratta di $\{2,4,6\}$, applicando γ otteniamo l'insieme $\{1\}$. Quindi $\gamma \circ \alpha(\varnothing) \geq \varnothing$, e provandolo per tutti gli elementi otteniamo lo stesso risultato generico:

$$\gamma \circ \alpha(c) \geq c$$

e anche la condizione opposta:

$$\alpha \circ \gamma(a) \leq a$$

Quindi α e γ sono connessioni di Galois.

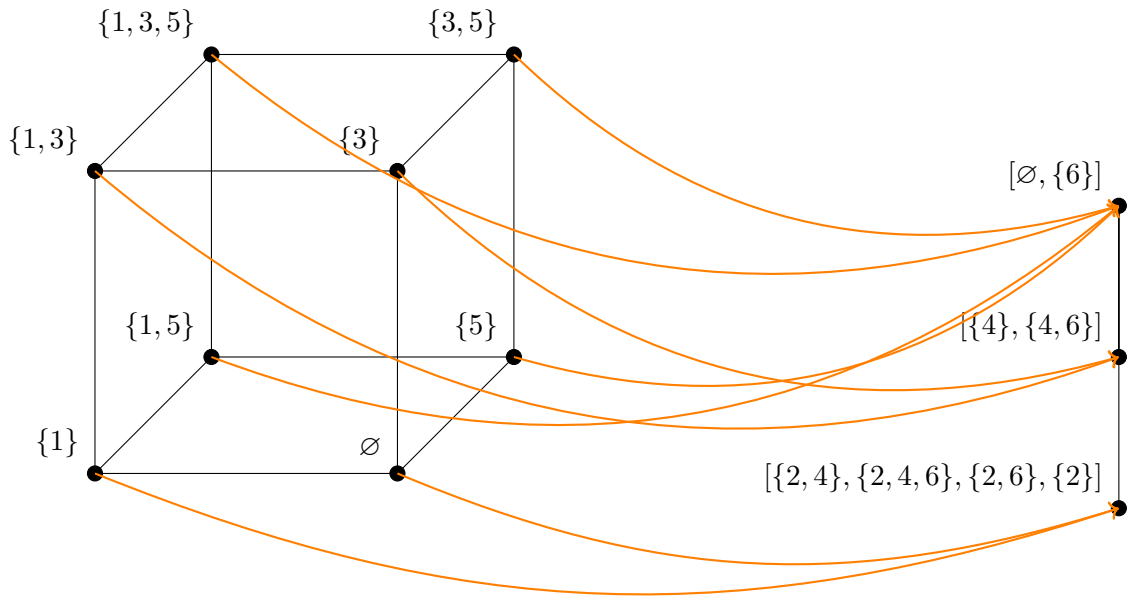
Per riuscire a eliminare gli elementi inutili possiamo applicare il concetto relativo all'inserzione di Galois, quindi:

$$\mathcal{B}' \equiv \mathcal{B}'' \iff \gamma(\mathcal{B}') = \gamma(\mathcal{B}'') \iff \forall a \in \mathcal{A}.\quad a\mathcal{R}\mathcal{B}' \iff a\mathcal{R}\mathcal{B}''$$

La differenza è che $\alpha \circ \gamma(a) = a$, quindi vediamo che modificando il dominio astratto e non le astrazioni otteniamo un'inserzione di Galois, osserviamo quindi gli elementi che hanno lo stesso significato. quindi:

- $\varnothing$ e $\{6\}$ hanno lo stesso significato, quindi vengono mappati in un unico elemento, che è $\{6\}$;
- $\{4\}$ e $\{4,6\}$ hanno lo stesso significato, quindi vengono mappati in un unico elemento, che è $\{4,6\}$;
- $\{2,4\}$, $\{2,4,6\}$, $\{2,6\}$ e $\{2\}$ hanno lo stesso significato e vengono mappati in un unico elemento, che è $\{2,4,6\}$.

Accorpando gli elementi astratti che hanno lo stesso significato. Quindi otteniamo il seguente diagramma:



Definiamo quindi una relazione di equivalenza tale per cui, se prendiamo anziché gli elementi astratti, le classi di equivalenza rispetto a tale relazione, otteniamo un'inserzione di Galois, senza variare le funzioni α e γ, riducendo il dominio astratto.

Vediamo come questi formalismi (*in particolare l'inserzione di Galois*) sono equivalenti ad altri due formalismi, che però spostano l'osservazione dal dominio astratto, ovvero $\mathcal{A}$, che ha una propria rappresentazione, a quello del dominio degli oggetti concreti che vogliamo osservare. Invece di guardare un mondo astratto rappresentato in un modo diverso, osserviamo il loro significato sul mondo concreto. Specifichiamo gli oggetti che osserviamo con precisione, ma nel mondo concreto e quindi modello con una funzione, chiamata **funzione di chiusura superiore**, o un sottodominio, chiamato **Moore family**, ovvero l'astrazione del dominio concreto.

5.3 Operatore di chiusura superiore (UCO)

Introduciamo questo nuovo concetto, che è una tipologia di funzioni che possiamo definire su un dominio. Supponiamo di avere un dominio $\mathcal{P}$ con un suo ordinamento $\leq_{\mathcal{P}}$ (*in generale su reticoli completi*).

Allora una funzione $\rho : \mathcal{P} \to \mathcal{P}$ è un UCO se la funzione ρ è:

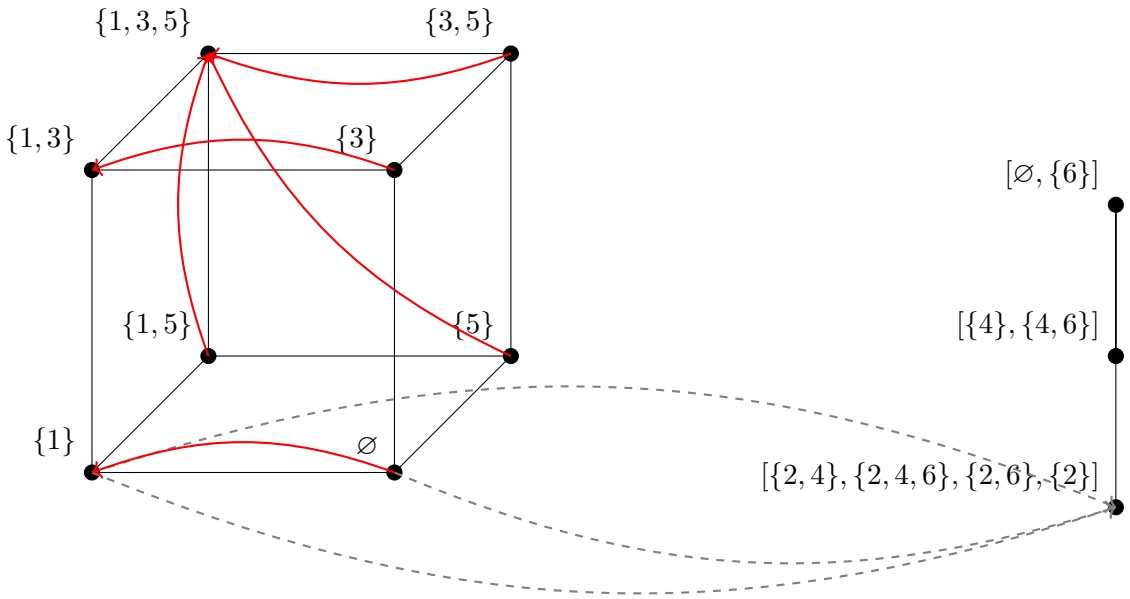
- **Monotona**: $\forall x, y \in \mathcal{P} \quad x \leq_{\mathcal{P}} y \implies \rho(x) \leq_{\mathcal{P}} \rho(y)$;
- **Estensiva**: $\forall x \in \mathcal{P} \quad x \leq_{\mathcal{P}} \rho(x)$;
- **Idempotente**: $\forall x \in \mathcal{P} \quad \rho(\rho(x)) = \rho(x)$.

Ovvero che ρ può perdere informazione, ma tale informazione viene persa tutta in un'unica applicazione, ogni successiva applicazione non aggiunge ulteriore imprecisione.

$\gamma \circ \alpha$ è un UCO nelle connessioni di Galois

Questo perché sappiamo che $\gamma \circ \alpha$ è monotona, poiché si tratta di composizione di funzioni monotone, è estensiva perché nelle proprietà delle connessioni di Galois abbiamo che $\gamma \circ \alpha(c) \geq c$ ed è idempotente, ovvero perché $\gamma \circ \alpha \circ \gamma \circ \alpha(c) = \gamma \circ \mathtt{id} \circ \alpha(c) = \gamma \circ \alpha(c)$, poiché $\gamma \circ \alpha \equiv \mathtt{id}$.

5.3.1 Esempio di UCO



$$\gamma \circ \alpha(\mathcal{A}') = \gamma(\{b \in \mathcal{B} \mid \mathcal{A}' \mathcal{R} b\}) = \{a \in \mathcal{A} \mid a \mathcal{R} \{b \in \mathcal{R} b\}\} = \{a \in \mathcal{A} \mid \forall b \in \mathcal{B}. \mathcal{A}' \mathcal{R} b \implies a \mathcal{R} b\}$$

Se partiamo dall'insieme vuoto $\varnothing$ e applichiamo $\gamma \circ \alpha$ otteniamo l'insieme $\{1\}$, se applichiamo nuovamente $\gamma \circ \alpha$ otteniamo l'insieme $\{1\}$, ovvero l'insieme di partenza. Quindi è **idempotente**, e tale risultato è l'`UCO`.

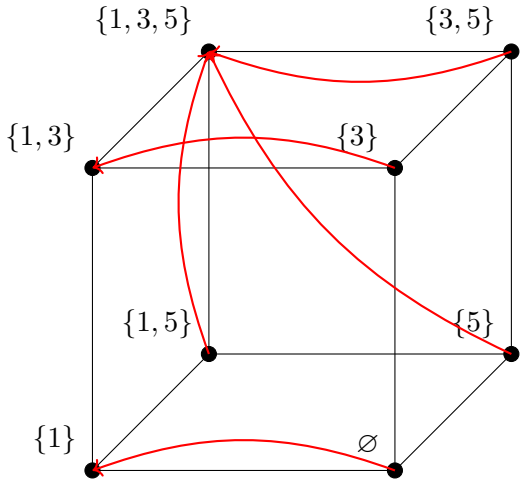
L'`UCO` che definiamo sul dominio concreto permette di associare ad ogni elemento concreto l'osservazione precisa che abbiamo scelto di guardare e che meglio approssima l'oggetto da cui siamo partiti.

Le immagini del nostro `UCO`, che sono i punti fissi, sono i significati degli oggetti astratti, ovvero gli elementi concreti che abbiamo scelto di osservare con precisione, ed è da qui che cerchiamo di approssimare gli altri elementi.

I punti in questione sono quindi i significati degli oggetti astratti, ovvero esattamente gli elementi concreti che abbiamo deciso di osservare con precisione. Gli elementi in cui cercare l'approssimazione di tutti gli altri elementi concreti.

> L'`UCO` altro non fa che associare ad ogni elemento concreto la sua migliore approssimazione. Possiamo ignorare la rappresentazione astratta degli elementi che abbiamo scelto di osservare con precisione e consideriamo solamente la trasformazione che associa ad ogni elemento concreto la sua migliore approssimazione, ovvero che associa ad ogni elemento concreto la proprietà astratta di interesse che lo caratterizza.

Ciò che ci permette di fare il passaggio agli operatori di chiusura superiore è di dimenticarci del dominio astratto.



5.4 Moore family

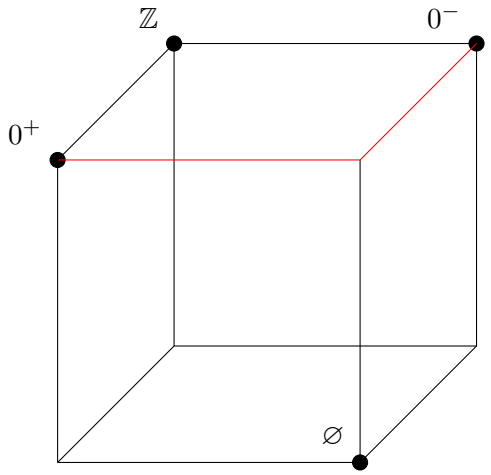
Siano $\mathcal{P}, \leq_{\mathcal{P}}$ un reticolo completo, allora $x \subseteq \mathcal{P}$ è un **insieme di Moore** se $x = \mathcal{M}(x)$, ovvero **chiusura di Moore di x**. dove:

$$\mathcal{M}(x) = \left\{ \bigwedge_{\mathcal{P}} \mathcal{S} \mid \mathcal{S} \subseteq x \right\}$$

Dove x è chiuso per $\bigwedge_{\mathcal{P}}$, ovvero per ogni sottoinsieme $\mathcal{S} \subseteq x$.

In altri termini:

$$Sia \qquad \mathcal{S} \subseteq x \implies \bigwedge_{\mathcal{P}} \mathcal{S} \in x$$



Considerando il sottoreticolo di $\wp(\mathcal{Z})$ e prendendo come elementi di x: $0^+, 0^-, \mathbb{Z}, \varnothing$; allora x non è Moore family di $\wp(\mathcal{Z})$, perché prendendo la congiunzione logica `AND` tra due elementi di 0^+ e 0^- otteniamo un elemento che non appartiene a x (*tale elemento è lo zero*).

$$\exists \mathcal{S} = \{0^+, 0^-\} \, t.c. \bigwedge_{\mathcal{P}} \mathcal{S} = 0 \notin x$$

Nella nostra concezione di dominio astratto, l'insieme dei punti fissi di un `UCO` $\rho : \mathcal{P} \to \mathcal{P}$ formano una Moore family di $\mathcal{P}$. Di fatto ogni Moore family di un dominio concreto rappresenta una proprietà di un dominio astratto.

> I punti fissi di ρ sono una Moore family del dominio concreto, ovvero di $\wp(\mathcal{A})$.

5.4.1 Considerazioni

Disponiamo di tre modi differenti per rappresentare un dominio astratto:

- **UCO**: $\rho : \mathcal{P} \to \mathcal{P}$;
- **Moore family**: $x \subseteq \mathcal{P}$;
- **Inserzioni di Galois**: $(\mathcal{C}, \leq_{\mathcal{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \leq_{\mathcal{A}})$.

Ha senso parlare delle tre rappresentazioni, perché ognuno può essere utilizzato in contesti diversi. Le connessioni di Galois sono utili per rappresentare dominio astratto, quindi dipende dalla rappresentazione degli elementi, ed è utile durante l'implementazione, in modo da rappresentare in modo semplice le proprietà di oggetti astratti.

Gli altri due metodi di modellazione del dominio astratto sul dominio concreto sono indipendenti dalla rappresentazione di elementi perché si parla di elementi astratti attraverso i loro significati, indipendenti dall'astrazione. Chiaramente meno adatti ai contesti applicativi, ma più adatti in contesti di verifica di proprietà sui domini astratti, o comunque ragionamenti formali sulle caratteristiche del dominio astratto e dell'analisi.

Quindi in base al contesto si decide la metodologia di rappresentazione adatta.

5.4.2 Relazione tra Moore family e inserzioni di Galois

- Supponiamo di partire da un'inserzione di Galois.

  $$\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A} \iff \mathcal{A} \text{ è } \textit{isomorfo a una famiglia di Moore di } \mathcal{C}$$

  Quindi

  $$\exists \mathcal{X} \textit{famiglia di Moore di } \mathcal{C} \textit{ t.c. } \exists \iota : \mathcal{A} \to \mathcal{X}$$

  Esiste quindi una funzione biunivoca tra $\mathcal{A}$ e $\mathcal{X}$.

- Supponiamo di partire da un `UCO`.

  $$\rho \in \texttt{UCO}(\mathcal{C}) \implies \exists \textit{ isomorfismo tra } \iota : \rho(\mathcal{C}) \to \mathcal{A}$$

  Dove $\mathcal{A}$ e una qualunque rappresentazione degli elementi di $\rho(\mathcal{C})$.

  Quindi $\mathcal{C} \underset{\iota^{-1}}{\overset{\iota \circ \rho}{\leftrightarrows}} \mathcal{A}$.

$$\iota^{-1} \circ \iota \circ \rho = \rho$$

Questo perché $\iota^{-1} \circ \iota = \texttt{id}$ e $\iota \circ \rho = \alpha$.

$$\iota \circ \rho \circ \iota^{-1}(\alpha) = \iota \circ \rho \circ \rho(\alpha) = \iota \circ \rho(\alpha) = \alpha$$

Dove $\iota^{-1}(\alpha) \in \rho$

5.4.3 Relazione tra inserzioni di Galois e `UCO`

La relazione è immediata, poiché se $\mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$, allora $\gamma \circ \alpha \in \texttt{UCO}$

5.4.4 Relazione tra `UCO` e Moore family

La relazione è immediata, poiché se $\rho \in \texttt{UCO}(\mathcal{C})$, allora $\rho(\mathcal{C})$, che è l'insieme dei punti fissi è famiglia di Moore di $\mathcal{C}$.

5.5 Reticolo delle interpretazioni astratte

Supponiamo di avere un reticolo completo

$$\langle \mathcal{C}, \leq_{\mathcal{P}}, \wedge, \vee, \top, \bot \rangle$$

Abbiamo inoltre l'insieme di tutti i domini astratti $\mathcal{A}_i \in \texttt{UCO}(\mathcal{C})$.

$$\langle \texttt{UCO}(\mathcal{C}), \sqsubseteq, \sqcap, \sqcup, \lambda x.\top, \lambda x.x \rangle$$

UCO ordinato per precisione relativa è un reticolo completo.

L'ordinamento di precisione relativa permette di confrontare il grado di precisione dei domini astratti, dove un elemento è più preciso se contiene più elementi.

$$\mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \iff \mathcal{A}_1 \textit{ corrisponde ad un insieme di elementi più grande}$$

$$\rho \sqsubseteq \eta \iff \forall y \in \mathcal{C} \,.\, \rho(y) \leq \eta(y) \iff \rho(\mathcal{C}) \subseteq \eta(\mathcal{C})$$

Con questo ordinamento relativo possiamo definire un operatore di greatest lower bound, chiamato prodotto ridotto, $\sqcap\mathcal{A}_i$. Viene preso il più piccolo dominio astratto tra tutti i domini che contengono $\mathcal{A}_i$. Ricordiamo che non è banalmente l'unione, l'unione può non essere una Moore family.

L'operatore di least upper bound, chiamato prodotto esteso, $\sqcup\mathcal{A}_i$. Viene preso il più grande dominio astratto contenuto in tutti i domini astratti. Quindi:

$$\bigcap \mathcal{A}_i = \sqcap\mathcal{A}_i$$

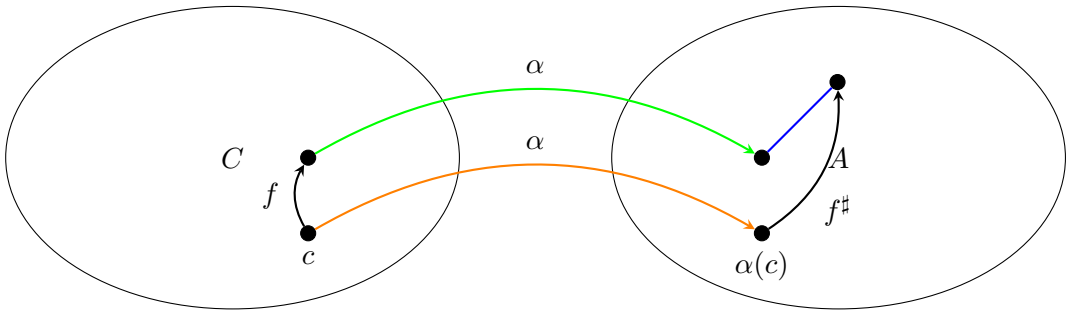
L'operatore $\lambda x.\top$ è il dominio astratto più astratto, ovvero quello che non osserva nulla, tutti gli elementi concreti sono mappati nel top.

L'operatore $\lambda x.x$ è il dominio astratto più concreto, ovvero quello che osserva tutto, tutti gli elementi concreti sono mappati in se stessi, quindi l'identità.

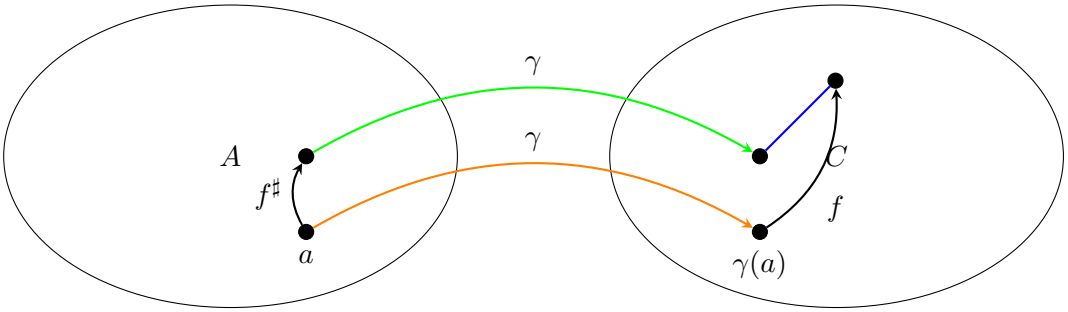
5.6 Computazioni astratte

La computazione astratta tratta di come si trasferisce un calcolo dal dominio concreto al dominio astratto. L'obiettivo è quello di trasferire il calcolo, partiamo da una funzione definita su oggetti concreti: $f : c \to c$, che sono le nostre operazioni su elementi di $\mathcal{C}$, che vogliamo trasferire in operazioni su elementi di $\mathcal{A}$: $f^\sharp : a \to a$.

Diciamo che $f^\sharp$ è corretta per f se $\alpha \circ f(c) \leq_{\mathcal{A}} f^\sharp \circ \alpha(c)$.



Il calcolo astratto può perdere informazione rispetto alla proprietà del calcolo concreto. Il calcolo astratto è corretto se approssima la proprietà del calcolo concreto. In modo analogo possiamo caratterizzare la correttezza confrontato i risultati sul mondo concreto.



Il significato del risultato del calcolo astratto approssima il risultato concreto. Quindi $f \circ \gamma(a) \leq_{\mathcal{C}} \gamma \circ f^\sharp(a)$.

Dato $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}, f : c \to c$, allora $f^\sharp : a \to a$ soddisfa la relazione

$$\alpha \circ f(c) \leq_{\mathcal{A}} f^\sharp \circ \alpha(c)$$

se e solo se soddisfa

$$f \circ \gamma(a) \leq_{\mathcal{C}} \gamma \circ f^\sharp(a)$$

In particolare se partiamo dalla condizione che per ogni x appartenente al dominio concreto, $\alpha \circ f(x) \leq_{\mathcal{A}} f^\sharp \circ \alpha(x) \iff \alpha \circ f \circ \gamma(x) \leq_{\mathcal{A}} f^\sharp(x)$. dove $\alpha \circ f \circ \gamma(x)$ è la funzione **best correct approximation**.

$$f^\sharp \text{ è } \textit{sound} \iff \textit{approssima } f^\sharp \equiv \alpha \circ f \circ \gamma : \mathcal{A} \to \mathcal{A}$$

Tale costruzione va bene per gli operatori, ma non è accettabile come trasferimento per l'intera semantica, perché il nostro obiettivo è non passare da f.

5.6.1 Soundness sulle chiusure

$\alpha f(x) \leq_{\mathcal{A}} \alpha \circ f \circ \alpha(x)$ per monotonia di γ otteniamo $\gamma \circ \alpha \circ f(x) \leq_{\mathcal{C}} \gamma \circ \alpha \circ f \circ \gamma \circ \alpha(x)$. Osserviamo che abbiamo sempre $\gamma \circ \alpha$ e questo possiamo ridefinirlo come $\rho \in$ `UCO` perché abbiamo già detto che $\gamma \circ \alpha$ è `UCO` del dominio concreto.

Quindi la relazione di correttezza ottenuta confrontato i due calcoli nel mondo astratto si riscrive in termini di chiusure esattamente in questo modo:

$$\rho \circ f(x) \leq_{\mathcal{C}} \rho \circ f \circ \rho(x)$$

Considerando $f \circ \gamma(a) \leq_{\mathcal{C}} \gamma \circ f^\sharp(a)$, allora $f \circ \gamma(a) \leq \gamma \circ \alpha \circ f \circ \gamma(a)$. Essendo γ, α connessioni di Galois, allora ogni elemento astratto è l'astrazione di un elemento concreto, $a = \alpha(c)$, quindi:$f \circ \gamma \circ \alpha(c) \leq \gamma \circ \alpha \circ f \circ \gamma \circ \alpha(c)$

Otteniamo quindi:

$$f \circ \rho(c) \leq_{\mathcal{C}} \rho \circ f \circ \rho(c)$$

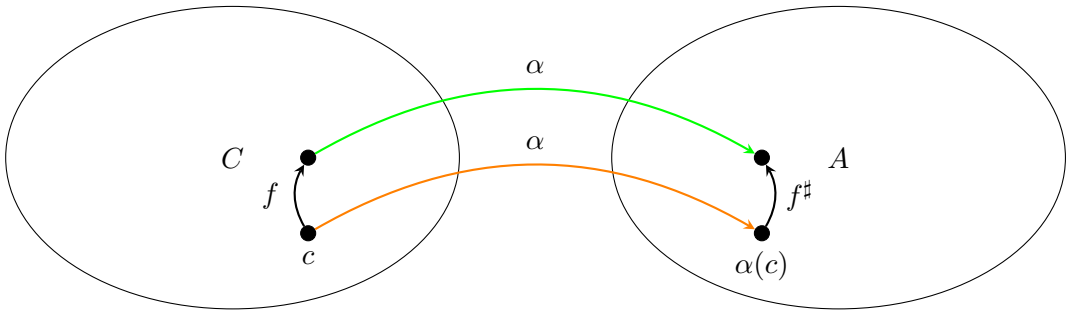
La relazione è leggermente diversa, ma è equivalente.

> Ragionare con le inserzioni di Galois garantisce per costruzione la correttezza. Le analisi costruite nel contesto di interpretazione astratta sono dette corrette per costruzione.

5.6.2 Completeness sulle chiusure

Backward completeness - Relazione di precisione sul dominio $\mathcal{A}$

Un'analisi è precisa quando eseguire il calcolo nel mondo astratto ed eseguirlo nel mondo concreto non fa differenza. L'informazione scartata durante la costruzione del mondo astratto, non era informazione utile per il calcolo che interessa effettuare. Quello che non osserviamo non è rilevante per il calcolo che vogliamo effettuare.



$$\alpha \circ f(x) = f^\sharp \circ \alpha(x)$$

Se sostituiamo $f^\sharp$ con la `BCA` otteniamo:

$$\alpha \circ f(x) = \alpha \circ f \circ \gamma \circ \alpha(x)$$

Possiamo applicare γ a sinistra e a destra:

$$\gamma \circ \alpha \circ f(x) = \gamma \circ \alpha \circ f \circ \gamma \circ \alpha(x)$$

Quindi

$$\rho \circ f(x) = \rho \circ f \circ \rho(x)$$

La proprietà del risultato ottenuto calcolando sugli elementi astratti è la stessa del risultato ottenuto calcolando sugli elementi concreti. Astrarre l'input non genera nessuna approssimazione.

Forward completeness - Relazione di precisione sul dominio $\mathcal{C}$

Il significato del calcolo astratto coincide con il calcolo concreto; eseguendo le operazioni precedenti:

$$f^\sharp = \alpha \circ f \circ \gamma \qquad f \circ \rho(x) = \rho \circ f \circ \rho(x)$$

Approssimare il risultato del calcolo astratto non genera imprecisione.

Esempio di backward completeness

Supponiamo che $\mathcal{C} = \wp(\mathbb{Z})$, $\mathcal{A} = \mathit{Segni}$ $(\texttt{Pos}, \texttt{Neg}, \bot, \top)$ e che $f = +$.

Allora $f\rho \circ f \circ \rho \to \texttt{Pos} +^\sharp \texttt{Neg} =?$ ed è evidente che abbiamo perso informazione.

Se invece facciamo il calcolo completo allora $\rho \circ f \to \texttt{Segni}(5+(-7)) = \texttt{Neg}$, quindi è evidente che non siamo regolari. In questo caso il problema è nell'input. Aver calcolato la somma sui segni ha fatto si che non ci fosse più l'informazione di cui abbiamo bisogno per calcolare in modo abbastanza preciso l'operatore di somma. L'astrazione dell'input ha fatto perdere l'informazione necessaria affinché la somma restituisse un risultato preciso.

Non c'è modo di cambiare l'osservazione in output perché la risposta diventi più precisa. Questo significa che da una parte abbiamo la backward completeness e inoltre dall'altra il fatto che non possa cambiare il modo di osservare il risultato per ottenere precisione mi dice che la proprietà è anche forward complete.

Cambiando $f \circ \rho \to \texttt{Neg} +^\sharp \texttt{Pos} =?$ comunque non otteniamo precisione.

Abbiamo perso talmente tanta informazione sull'astrazione dell'input che comunque non ho speranze di avere un risultato che abbia significato.

La perdita di precisione è dovuta all'astrazione dell'input e perciò si parla di backward completeness.

Quindi $\rho \circ f \circ \rho \neq \rho \circ f$ e quindi non siamo backward completi, ma siamo forward completi perché $\rho \circ f \circ \rho = f \circ \rho$.

Esempio di forward completeness

Supponiamo che il nostro dominio concreto sia sempre $\mathcal{C} = \wp(\mathcal{Z})$ e che il dominio astratto sia $\mathcal{A} = \texttt{Costanti}$. Osserviamo se un valore è una costante n. Consideriamo la funzione f come funzione che esegue la scelta non deterministica tra due valori, quindi:

$$f = \boxdot \qquad e_1 \boxdot e_2 = \textit{ prende non deterministicamente uno dei due valori}$$

Quindi il calcolo di $\rho \circ f \circ \rho$ è:

$$\rho \circ (\texttt{Const}_1 \boxdot \texttt{Const}_2) =?$$

Di fatto non possiamo sapere quale delle due costanti è stata scelta, di conseguenza la funzione darà come risultato $\top$.

Il calcolo di $f \circ \rho$ è:

$$\texttt{Const}_1 \boxdot \texttt{Const}_2 = \{\texttt{Const}_1, \texttt{Const}_2\} \in \wp(\mathbb{Z})$$

Quindi non siamo forward completi, perché sul risultato ρ non è in grado di essere abbastanza precisa, in questo caso basterebbe rendere più preciso, l'insieme di cardinalità 2 alla chiusura ρ.

Quindi l'approssimazione del risultato fa si che si perda informazione, e non l'approssimazione dell'input.

Anche applicando $\rho \circ f$ si ottiene:

$$\rho \circ (n_1 \boxdot n_2) =?$$

Quindi $\rho \circ f \circ \rho = \rho \circ f$ e quindi siamo backward completi, ma non siamo forward completi perché $\rho \circ f \circ \rho \neq f \circ \rho$.

5.6.3 Esempio di non completezza backward e forward

In questo caso, sia raffinando l'input che l'output in modi diversi, si ottiene il raggiungimento di completezza, quindi **precisione**.

Supponiamo di essere in un contesto di programmi che manipolano sia interi, che reali, che stringhe.

$$\mathcal{C} = \wp(\mathbb{Z} \cup \mathbb{R} \cup \texttt{String})$$

La proprietà che vogliamo osservare è il tipo, quindi abbiamo un insieme `Type` che nel nostro specifico caso può distinguere tra interi, float e stringhe.

$$\mathcal{A} = \texttt{Type} = \{\texttt{Int}, \texttt{Float}, \texttt{String}\} \cup \{\top, \bot\}$$

Di questi insiemi guardiamo solo l'informazione di tipo e di fronte all'insieme misto diremo che non vi è abbastanza informazione per dire il tipo, quindi $\top$.

L'operatore che consideriamo è la somma, quindi $f = +$, che ha proprietà aritmetica, la somma di un numero ad una stringa, la stringa viene convertita in numero, quindi avviene una conversione implicita, ovvero nel può lungo prefisso numerico che ha la stringa.

Supponiamo di sommare un intero ad una stringa:

$$2 + \texttt{Ciao} = 2 + 0 = 2$$

Consideriamo ora la funzione di astrazione ρ che considera solo il tipo:

$$\rho \circ f \circ \rho = \rho \circ (\texttt{Int} +^\sharp \texttt{String}) = \rho \circ (\{\texttt{Int}, \texttt{Float}\}) = \top$$

La stringa potrebbe avere un prefisso intero e quindi corrispondere ad un intero, o reale e quindi corrispondere ad un reale, oppure non avere un prefisso numerico. Tale informazione, però, non è presente nell'astrazione.

Se proviamo a calcolare attraverso la computazione concreta e poi l'astrazione:

$$\rho \circ f = \begin{cases} \texttt{Int} +^\sharp \texttt{Int} = \texttt{Int} & \textit{Se il prefisso della stringa è intero} \\ \texttt{Int} +^\sharp \texttt{Float} = \texttt{Float} & \textit{Se il prefisso della stringa è reale} \end{cases}$$

Se proviamo procediamo con l'astrazione dell'input e poi con l'applicazione di f otteniamo:

$$f \circ \rho = f \circ (\texttt{Int} +^\sharp \texttt{String}) = \{\texttt{Int}, \texttt{Float}\}$$

Quindi $\rho \circ f \circ \rho \neq \rho \circ f$ non è nè backward completo, nè forward completi perché $\rho \circ f \circ \rho \neq f \circ \rho$.

Il problema della non completezza è dato sia dalla perdita di informazione nell'input, sia nella perdita di informazione nell'output.

Nel caso della backward completezza il problema è dato dall'osservazione dell'input, che non è stata in grado di distinguere tra stringhe che danno come risultato un intero e stringhe che danno come risultato un reale. Per la backward completezza potremmo pensare di raffinare la ρ per essere maggiormente precisi nell'osservazione dell'input.

Potremmo considerare un dominio astratto differente:

$$\mathcal{A} = \texttt{Type}' = \{\texttt{Int}, \texttt{Float}, \texttt{StringInt}, \texttt{StringFloat}\} \cup \{\top, \bot\}$$

Se consideriamo ρ costruita secondo il dominio astratto $\texttt{Type}'$:

$$\rho \circ f \circ \rho = \begin{cases} \rho \circ (\{\texttt{Int}, \texttt{StringInt}\}) = \texttt{Int} \\ \rho \circ (\{\texttt{Int}, \texttt{StringFloat}\}) = \texttt{Float} \end{cases}$$

Se proviamo a calcolare attraverso la computazione concreta e poi l'astrazione:

$$\rho \circ f = \begin{cases} \texttt{Int} +^\sharp \texttt{Int} = \texttt{Int} & \textit{Se il prefisso della stringa è intero} \\ \texttt{Int} +^\sharp \texttt{Float} = \texttt{Float} & \textit{Se il prefisso della stringa è reale} \end{cases}$$

Otteniamo backward completezza, perché l'astrazione dell'input è stata in grado di distinguere tra stringhe che danno come risultato un intero e stringhe che danno come risultato un reale, infatti $\rho \circ f \circ \rho = f \circ \rho$.

Per la forward completezza potremmo pensare di avere ρ preciso per l'output, quindi pensiamo ad un insieme di tipi che riesca a dar valore alla combinazione di intero e float, quindi il tipo generico `Num`:

$$\mathcal{A} = \texttt{Type}'' = \{\texttt{Int}, \texttt{Float}, \texttt{Num}, \texttt{String}\} \cup \{\top, \bot\}$$

Se consideriamo ρ costruita secondo il dominio astratto $\texttt{Type}''$:

$$\rho \circ f \circ \rho = \rho \circ (\texttt{Int} +^\sharp \texttt{String}) = \rho \circ (\{\texttt{Int}, \texttt{Float}\}) = \texttt{Num}$$

Se proviamo procediamo con l'astrazione dell'input e poi con l'applicazione di f otteniamo:

$$f \circ \rho = f \circ (\texttt{Int} +^\sharp \texttt{String}) = \{\texttt{Int}, \texttt{Float}\} = \texttt{Num}$$

Otteniamo forward completezza, perché l'astrazione dell'output è stata in grado di distinguere tra stringhe che danno come risultato un intero e stringhe che danno come risultato un reale, infatti $\rho \circ f \circ \rho = f \circ \rho$.

Raggiungimento di completezza

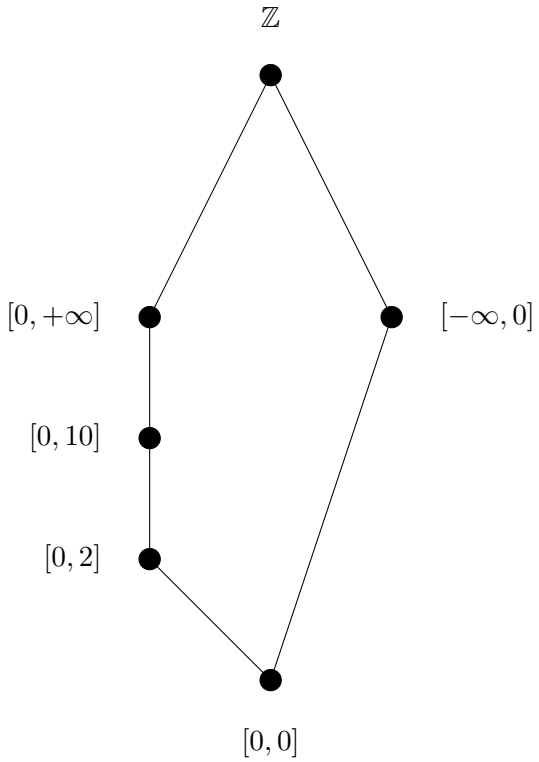
Quindi abbiamo a disposizione diverse strade per raggiungere completezza, o aggiungendo maggior precisione nell'osservazione del tipo di input o aggiungendo maggior flessibilità nel tipo dell'output. Nel primo caso abbiamo una completezza backward, quindi una precisione che è indipendente dall'osservazione astratta degli input e nel secondo caso abbiamo una completezza forward, quindi una precisione che è indipendente dall'osservazione dell'output.

Nel caso dell'analisi statica, ha senso parlare di completezza backward, perché si confrontano le proprietà astratte del calcolo concreto e del calcolo astratto.

La completezza è una proprietà del dominio astratto rispetto ad un'operazione. Tipicamente si dice se un dominio è corretto per una certa operazione o meno. Nel momento in cui possiamo parlare in modo indipendente dalla scelta dell'approssimazione dell'operazione, automaticamente dipendiamo solo dal dominio astratto e dalla funzione concreta da dover approssimare.

Esempio

Riportiamo il dominio concreto:



Come prima istanza di astrazione consideriamo i punti

- $\mathbb{Z}$;
- $[0, +\infty]$;
- $[0, 10]$;

Che sarà ρ_1. Mentre come seconda istanza di astrazione consideriamo i punti:

- $\mathbb{Z}$;
- $[0, 2]$;
- $[0, 0]$;

Che sarà ρ_2.

Prendiamo come funzione operatore che approssima il quadrato su questo dominio concreto. Quindi ρ_1 non è backward completa per f perché:

$$\rho_1 \circ f \circ \rho_1([0, 2]) = \rho_1 \circ f \circ ([0, 10]) = \rho_1 \circ ([0, +\infty]) = [0, +\infty]$$

Mentre:

$$\rho_1 \circ f([0, 2]) = \rho_1 \circ ([0, 10]) = [0, 10]$$

Quindi ρ_1 non è backward completo poiché $\rho_1 \circ f \circ \rho_1 \neq \rho_1 \circ f$.

Consideriamo:

$$f \circ \rho_1([0, 2]) = f \circ ([0, 10]) = [0, +\infty]$$

Quindi ρ_1 è forward completo per f perché $f \circ \rho_1 \circ f = f \circ \rho_1$.

> Intuitivamente, la forward completezza deriva dal fatto che ogni elemento all'interno della chiusura ρ_1 ha come immagine di f un elemento all'interno della chiusura ρ_1, mentre il fatto che non sia backward completo deriva dal fatto che ci sono punti non all'interno di ρ_1 che vengono mappati in elementi di ρ_1.

Consideriamo ora ρ_2:

$$\rho_2 \circ f \circ \rho_2([0, 2]) = \rho_2 \circ f([0, 10]) = \rho_2([0, 10]) = \mathbb{Z}$$

Mentre:

$$\rho_2 \circ f([0,2]) = \rho_2([0,10]) = [0,10]$$

Quindi ρ_2 non è backward completo per f perché $\rho_2 \circ f \circ \rho_2 \neq \rho_2 \circ f$.

Consideriamo:

$$f \circ \rho_2([0,2]) = f([0,10]) = \mathbb{Z}$$

Quindi ρ_2 è forward completo per f perché $f \circ \rho_2 \circ f = f \circ \rho_2$.

5.7 Trasferimento della semantica

Il calcolo della semantica si basa sul fatto che disponiamo di un'operatore costruito sul nostro dominio concreto, ovvero il nostro reticolo completo.

$$\mathcal{F} : \mathcal{C} \to \mathcal{C}$$

Disponiamo di un operatore α che mappa un elemento del dominio concreto in un elemento del dominio astratto.

$$\alpha : \mathcal{C} \to \mathcal{A}$$

L'obiettivo è quello di calcolare la proprietà del **least fixed point** di $\mathcal{F}$, senza però dover calcolare il least fixed point, poiché questo è un problema indecidibile, essendo che può divergere.

Tipicamente viene fatto il trasferimento del punto fisso, ovvero si cerca un operatore $\bar{\mathcal{F}} : \mathcal{A} \to \mathcal{A}$ tale che:

$$\texttt{lfp}\,\bar{\mathcal{F}} = \alpha \circ \texttt{lfp}\,\mathcal{F}$$

Il least fix point di $\bar{\mathcal{F}}$ sia uguale al least fixed point di $\mathcal{F}$ composto con α.

La soluzione ideale sarebbe quella di prendere come least fixed point di $\bar{\mathcal{F}}$ l'applicazione $\alpha \circ \mathcal{F} \circ \gamma$, purtroppo questo non è possibile perché:

$$\bar{\mathcal{F}} = \alpha \circ \mathcal{F} \circ \gamma \not\Rightarrow \texttt{lfp}\,\bar{\mathcal{F}} = \alpha \circ \texttt{lfp}\,\mathcal{F}$$

Esistono delle condizioni specifiche e aggiuntive per garantire questo trasferimento in modo esatto.

Dobbiamo quindi accettare un'approssimazione:

$$\exists \bar{\mathcal{F}} \qquad t.c.\ \alpha \circ \texttt{lfp}\mathcal{F} \sqsupseteq \texttt{lfp}\bar{\mathcal{F}}$$

Anche il punto fisso astratto può avere problemi di terminazione, in particolare termina se:

- Il dominio e finito;
- Il dominio non ha catene ascendenti infinite.

Se $\mathcal{A}$ è un dominio infinito e ha catene ascendenti infinite, non abbiamo garanzie di terminazione. Per garantire terminazione abbiamo bisogno di un operatore astratto $\bar{\mathcal{F}}$ su $\mathcal{A}$ e un calcolo approssimato del calcolo del punto fisso, bastato su un operatore di Widening, e tale operatore va definito ad hoc per ogni dominio astratto.

Capitolo 6

Analisi non distributive

Le analisi distributive sono analisi statiche che si basano su ciò che viene calcolato, entrando quindi nel merito del valore che viene attribuito alle variabili cercando proprietà su tali valori.

6.1 Propagazione delle costanti

La propagazione delle costanti è un'analisi che ha come obiettivo quello di determinare se una variabile ha sempre lo stesso valore in un certo punto di programma.

Dato un punto di programma p, determina se una variabile nel punto di programma p è sempre un valore costante. Tale analisi collegata alla **valutazione parziale**, che in un certo senso, che è collegata al concetto di **specializzazione**, discusso nell'ambito dei linguaggi di programmazione.

Cerchiamo quindi di capire se in un certo punto di programma una variabile ha sempre lo stesso valore e in questo caso è possibile utilizzare tale valore per valutare parzialmente il programma preso in considerazione, vedendo quindi se alcuni dei calcoli possono essere elaborati in funzione di un valore costante che il programma in quel punto di programma assume.

In un caso pratico, se in un certo punto di programma una variabile che è utilizzata come condizione di un *branching*, assume sempre lo stesso valore, talvolta è possibile eliminare il *branching* e eseguire solo una delle due parti del *branching*.

```
1 x ← 1;
2 ...;
3 if x > 0 then
4 |  e;
5 else
6 |  e';
```

```
1 x ← 1;
2 ...;
3 e;
```

In questo senso l'informazione può essere utilizzata in vari ambiti permettendoci inoltre di capire quali valori possono assumere le variabili in un certo punto di programma. Come *side effect* di tale analisi, possiamo inoltre capire se un punto di programma è raggiungibile o meno.

Esempio

Supponiamo di avere il seguente programma:

```
1 a := 1; b := 2; c := 3; d := 3; e := 0;
2 while B do
3     b := 2 · a; d := d + 1; e := e − a;
4     c := e + d; a := b − a;
```

Dove B indica che la condizione sul ciclo non è nota staticamente. Verifichiamo lo stato delle variabili dopo ogni punto di programma:

	a	b	c	d	e
1	1	2	3	3	0
2	1	2	3	3	0
3	1	2	3	4	−1
4	1	2	3	4	−1
1	1	2	3	3	0
2	1	3	3	?	?
3	1	3	2	?	?
4	2	3	?	?	?

Dopo la prima iterazione, al punto di programma 2, viene collezionato ciò che viene fatto al punto di programma 4, visto che dal punto 4 si ritorna al punto 2 data la presenza del ciclo.

Collezionando valori vediamo che variano sono i valori di d e e, quindi non possiamo dire nulla sui loro valori. Calcolando il valore di d, viene eseguita una somma con un valore non conosciuto, quindi ? non può essere a sua volta conosciuto.

Concludiamo quindi che non possiamo dire nulla sul valore di c, d e e, mentre possiamo dire che a e b sono sempre uguali a 1 e 2 rispettivamente. A differenza dell'analisi astratta, nel caso concreto il valore di c sarebbe sempre 3, di fatto costante.

6.1.1 Costruzione dell'analisi

Il primo passo per costruire l'analisi è quello di definire il dominio delle informazioni astratte, ovvero delle proprietà che vogliamo osservare con precisione.

Nel dominio concreto $\mathcal{C} = \wp(\mathbb{Z})$, lavoriamo su valori interi, insiemi di interi. Tra questi insiemi di interi, l'obiettivo è osservare con precisione i singoletti, quindi:

$$\mathcal{A} = \{n \mid n \in \mathbb{Z}\} \cup \{\bot, \top\} = \mathbb{Z}^\top$$

Dal momento in cui una variabile colleziona più di un valore possibile, l'informazione non è più precisa, quindi non è più possibile dire nulla sul fatto che possa essere costante o meno.

L'inserzione di Galois tra i due domini $\mathcal{C}$ e $\mathcal{A}$ è data dalle funzioni:

$$\alpha(x) = \begin{cases} \top & \text{se } x = \varnothing \\ n & \text{se } S = \{n\} \quad n \in \mathbb{Z} \\ \top & \text{altrimenti} \end{cases}$$

$$\gamma(a) = \begin{cases} \varnothing & \text{se } a = \top \\ \{n\} & \text{se } a = n \quad n \in \mathbb{Z} \\ \mathbb{Z} & \text{altrimenti} \end{cases}$$

dove $x \in \wp(\mathbb{Z})$ e $a \in \mathcal{A}$ e con α che è la funzione di astrazione e γ che è la funzione di concretizzazione, entrambe funzioni monotone; si può dimostrare che formano una *inserzione di Galois*.

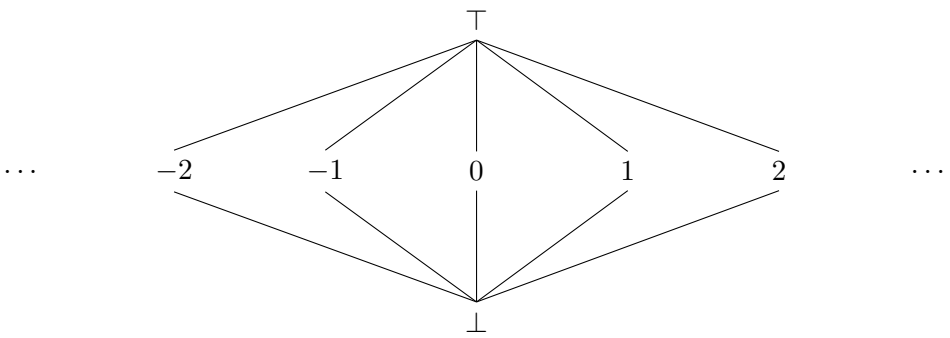


Figura 6.1.1: Rappresentazione grafica del dominio delle costanti $\mathbb{Z}^\top$

Dati $x, y \in \mathbb{Z}^\top$ se e solo se $x \sqsubseteq y$ oppure $y = \top$ o $x = \bot$. È chiaro che si tratti di un **astrazione dei valori**, ma la semantica opera su stati, quindi l'obiettivo è di estendere il dominio delle costanti a quello degli stati, su cui calcoleremo la semantica astratta.

Se lo stato è una funzione da variabili a valori, $\sigma : \texttt{Var} \to \mathbb{Z}$; nel caso collecting, $S : \texttt{Var} \to \wp(\mathbb{Z})$. Lo stato astratto sarà quindi una funzione $\mathbb{D}$ all'astrazione di parti di $\mathbb{Z}$, ovvero

$$\mathbb{D} : \texttt{Var} \to \mathbb{Z}^\top$$

Definiamo quindi il dominio degli stati astratti nelle costanti come

$$\mathbb{D} = (\texttt{Var} \to \mathbb{Z}^\top) \cup \{\bot\}$$

Dove $\bot$ è un elemento che rappresenta i punti non raggiungibili, che sarà ovviamente minore di ogni memoria del dominio. Definiamo $\mathbb{D}_\bot \stackrel{def}{=} \forall x.x \mapsto \bot$, ovvero la memoria che associa ad ogni variabile il valore $\bot$, dove quindi nessuna variabile è stata inizializzata.

Questo è quindi il dominio dove definiamo la semantica astratta, tale dominio è *pointwise*, ovvero il confronto tra due stati astratti è dato dal confronto tra i valori delle variabili.

$$\mathcal{D}_1 \sqsubseteq \mathcal{D}_2 \iff \forall x \in \texttt{Var}.\mathcal{D}_1(x) \sqsubseteq_{const} \mathcal{D}_2(x)$$

Confrontiamo due funzioni, confrontando l'immagine delle due funzioni per ogni singola variabile. Tale ordinamento permette di definire il *least upper bound* come:

$$\texttt{Lub} : \bigsqcup_i \mathcal{D}_i(x) = \begin{cases} z & \text{se } \forall i.\mathcal{D}_i(x) = z \qquad z \in \mathbb{Z} \\ \top & \text{altrimenti} \end{cases}$$

Con questo ordinamento, $\mathbb{D}$ è un reticolo completo, ed è quindi il dominio astratto per le costanti.

A questo punto possiamo iniziare a capire cosa vogliamo calcolare nella nostra analisi, esattamente come abbiamo fatto nell'analisi di *data flow*, vogliamo fornire una semantica astratta dei programmi nel dominio astratto $\mathbb{D}$, quindi una funzione

$$\forall \mathcal{D} \in \mathbb{D}.[\![\cdot]\!]^\sharp \mathcal{D}$$

Che dice se ogni variabile è costante o meno. L'obiettivo è avere questa informazione per ogni punto di programma, quindi cerchiamo la soluzione `MOP`, ovvero la soluzione *merge over all paths*, definendo quindi:

$$\texttt{MOP} : \mathcal{D}^\star[v] = \bigsqcup\{[\![\pi]\!]^\sharp \mathcal{D}_\bot \mid \pi : \text{start} \to v\}$$

Ovvero il *least upper bound* (*poiché la collecting semantics è di tipo possible*) delle semantiche di tutti i cammini a partire dall'informazione iniziale $\mathcal{D}_\bot$ dei cammini che vanno dal nodo di inizio al nodo v. È chiaro che se guardiamo tale analisi dal punto di vista delle analisi di data flow, l'analisi è forward, ed è di tipo possibile perché colleziona valori.

Dobbiamo quindi definire la semantica approssimata sugli stati astratti $[\![\cdot]\!]^\sharp$, considerando il fatto che i nostri archi sono della forma $(u, \texttt{lab}, v)$, considerando che la semantica è fornita dall'etichetta dell'arco, a partire dalla memoria $\mathcal{D}$ astratta che raggiunge il nodo u, quindi $[\![\texttt{lab}]\!]^\sharp : \mathcal{D}$.

6.1.2 Semantica delle espressioni

La prima cosa che conviene fare è andare a trasferire il calcolo delle operazioni concrete nel dominio astratto, quindi definiamo la funzione di trasferimento astratta. Per farlo definiamo l'operazione generica $\square$ su interi e la sua versione astratta $\square^\sharp$ su $\mathbb{Z}^\top$.

$$a, b \in \mathbb{Z}^\top \quad . \quad a \square^\sharp b = \begin{cases} \top & \text{se } a = \top \vee b = \top \\ a \square b & \text{altrimenti} \end{cases}$$

Si tratta esattamente della **best correct approximation** dell'operazione $\square$, perché prendere nel concreto l'operazione $\square$ equivale a farla per tutti gli elementi dei due insiemi, equivalente quindi al singoletto, per poi tornare nel dominio astratto e osservare il signoletto.

Una volta definita l'operazione astratta possiamo definire la semantica delle espressioni, $[\![e]\!]^\sharp\mathcal{D}$ ovvero la semantica approssimata a partire da una memoria $[\![e]\!]^\sharp : \mathcal{D} \to \mathbb{Z}^\top$. Nella analisi di data flow non è stata definita poiché non è mai interessato il valore restituito dalla valutazione di un'espressione, ma solo la struttura sintattica del programma. Nelle analisi di tipo distributive, dove il valore delle variabili è parte dell'analisi, è necessario definire la semantica delle espressioni perché la valutazione restituisce un valore appartenente al dominio astratto, quindi è necessario per l'analisi.

Ragioniamo quindi induttivamente sulla struttura della semantica astratta delle espressioni, definendo la semantica:

- $c \in \mathbb{Z}.[\![c]\!]^\sharp\mathcal{D} = c$
- $x \in \texttt{Var}.[\![x]\!]^\sharp\mathcal{D} = \mathcal{D}(x) \in \mathbb{Z}^\top$
- $[\![\square e]\!]^\sharp\mathcal{D} = \square^\sharp[\![e]\!]^\sharp\mathcal{D}$
- $[\![e_1 \square e_2]\!]^\sharp\mathcal{D} = [\![e_1]\!]^\sharp\mathcal{D}\square^\sharp[\![e_2]\!]^\sharp\mathcal{D}$

Esempio di valutazione di un'espressione astratta

Consideriamo la seguente espressione:

$$\mathcal{D} = [x \mapsto 2, y \mapsto \top]$$

$$[\![x + 7]\!]^\sharp\mathcal{D} = [\![x]\!]^\sharp\mathcal{D} +^\sharp [\![7]\!]^\sharp\mathcal{D} = \mathcal{D}(x) +^\sharp 7 = 2 +^\sharp 7 = 9$$

Consideriamo ora la seguente espressione:

$$[\![x + y]\!]^\sharp\mathcal{D} = [\![x]\!]^\sharp\mathcal{D} +^\sharp [\![y]\!]^\sharp\mathcal{D} = \mathcal{D}(x) +^\sharp \mathcal{D}(y) = 2 +^\sharp \top = \top$$

6.1.3 Semantica dei comandi

La semantica dei comandi ovviamente si baserà sulla semantica delle espressioni nel caso delle guardie e dell'assegnamento. Ragioniamo quindi induttivamente sulla struttura dei comandi, definendo la semantica astratta dei comandi:

- $[\![;]\!]^\sharp\mathcal{D} = \mathcal{D}$
- Se il ramo non è percorribile allora utilizzeremo il valore $\bot$, se invece lo è lasceremo la memoria invariata. Quindi non sarà percorribile quando nessun valore che rende vero il test è presente tra i possibili valori che abbiamo calcolato per e. Poiché per e è possibile

calcolare o è un valore costante o è $\top$, allora non è percorribile quando la valutazione di e è esattamente 0.

$$[\![\texttt{NonZero(e)}]\!]^\sharp\mathcal{D} = \begin{cases} \bot & \text{se } [\![e]\!]^\sharp\mathcal{D} = 0 \\ \mathcal{D} & \text{altrimenti } (\exists n \neq 0.[\![e]\!]^\sharp\mathcal{D} = n \vee [\![e]\!]^\sharp\mathcal{D} = \top) \end{cases}$$

- Abbiamo un caso analogo per il ramo `Zero`, sicuramente il rmo non sarà percorribile quando 0 non è contenuto all'interno di e.

$$[\![\texttt{Zero(e)}]\!]^\sharp\mathcal{D} = \begin{cases} \bot & \text{se } [\![e]\!]^\sharp\mathcal{D} \not\sqsupseteq 0 \\ \mathcal{D} & \text{altrimenti } ([\![e]\!]^\sharp\mathcal{D} = 0 \vee [\![e]\!]^\sharp\mathcal{D} = \top) \end{cases}$$

- In caso di assegnamento andiamo a calcolare il valore astratto associato all'espressione e aggiorniamo la memoria astratta con tale valore.

$$\mathcal{D}[x \mapsto [\![e]\!]^\sharp\mathcal{D}]$$

- Con l'input sappiamo il valore che x assume solamente a tempo di esecuzione, nonostante si sappia il fatto che sicuramente assumerà un unico valore.

$$[\![\texttt{input x}]\!]^\sharp\mathcal{D} = \mathcal{D}[x \mapsto \top]$$

Essendo una semantica forward, dato un cammino $\pi = k_0 \dots k_n$, sappiamo che $[\![\pi]\!]^\sharp = [\![k_n]\!]^\sharp \circ \cdots \circ [\![k_0]\!]^\sharp\mathcal{D}_\bot$, la semantica approssimata del cammino, sarà la composizione delle semantica approssimata dei singoli comandi a partire da una memoria iniziale $\mathcal{D}_\bot$, dove la semantica di un arco è esattamente la semantica approssimata della sua etichetta $[\![\texttt{lab}_n]\!]^\sharp$.

Si può dimostrare che la semantica astratta $[\![\cdot]\!]^\sharp$ non è distributiva, quindi calcolando la soluzione `MFP` questa non sarà uguale alla soluzione `MOP`, ma la contiene strettamente.

$$\texttt{MFP} \sqsupseteq \texttt{MOP}$$

In ogni caso, la soluzione `MFP` è l'unica che riusciamo a costruire e l'analisi costruisce tale soluzione. Per farlo trova la soluzione del sistema di disequazioni, costruito esattamente come nel caso della analisi di data flow.

- Sul nodo di partenza la memoria astratta dove non abbiamo informazioni sulle variabili, quindi:

$$\mathcal{D}[\texttt{entry}] \sqsupseteq \mathcal{D}_\top$$

- Per i nodi successivi, essendo una semantica forward, $\mathcal{D}[v]$ contiene la semantica dell'etichetta a partire dalla memoria astratta a partire dal nodo sorgente.

$$\mathcal{D}[v] \sqsupseteq [\![\texttt{lab}]\!]^\sharp\mathcal{D}[u]$$

In ogni caso la semantica $[\![\cdot]\!]^\sharp$ è monotona e il dominio è `ACC`, data l'altezza finita, e quindi la soluzione `MFP` esiste ed è calcolabile.

Esempio di non distributività

Consideriamo le seguenti memorie astratte:

$$\mathcal{D}_1 = \{x \mapsto 2, y \mapsto 3\} \qquad \mathcal{D}_2 = \{x \mapsto 3, y \mapsto 2\}$$

e vediamo cosa succede nel calcolo:

$$[\![x \leftarrow x + y]\!]^\sharp \mathcal{D}_1 \bigsqcup [\![x \leftarrow x + y]\!]^\sharp \mathcal{D}_2$$

A sinistra abbiamo: Quindi

$$\mathcal{D}_1[x \mapsto 5] \qquad \mathcal{D}_1 = \{x \mapsto 5, y \mapsto 3\}$$

A destra abbiamo:

$$\mathcal{D}_2[x \mapsto 5] \qquad \mathcal{D}_2 = \{x \mapsto 5, y \mapsto 2\}$$

Se facciamo il least upper bound otteniamo

$$\mathcal{D}_1 \bigsqcup \mathcal{D}_2 = \{x \mapsto 5, y \mapsto 3\} \bigsqcup \{x \mapsto 5, y \mapsto 2\} = \{x \mapsto 5, y \mapsto \top\}$$

Calcolando la semantica combinando le due memorie astratte otteniamo:

$$[\![x \leftarrow x + y]\!]^\sharp (\mathcal{D}_1 \bigsqcup \mathcal{D}_2) = [\![x \leftarrow x + y]\!]^\sharp \{x \mapsto \top, y \mapsto \top\} = \{x \mapsto \top, y \mapsto \top\}$$

quindi:

Non distributività della semantica

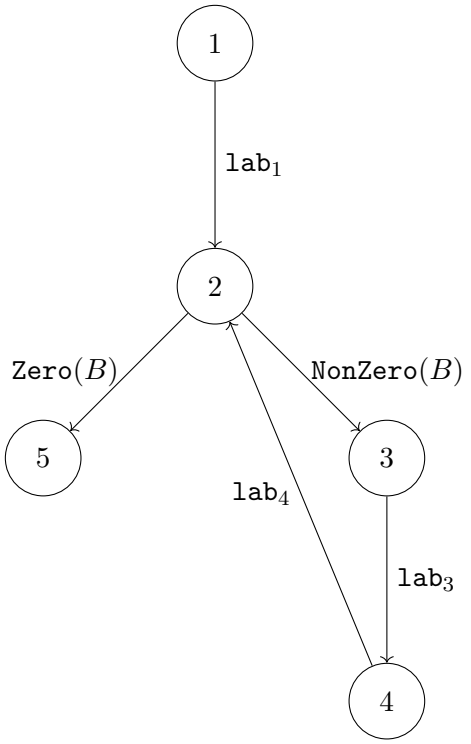
Non possiamo calcolare precisamente la combinazione della semantica dei cammini come semantica delle combinazioni locali delle memorie raggiunte.

$$[\![x \leftarrow x + y]\!]^\sharp \mathcal{D}_1 \bigsqcup [\![x \leftarrow x + y]\!]^\sharp \mathcal{D}_2 \not\sqsubseteq [\![x \leftarrow x + y]\!]^\sharp (\mathcal{D}_1 \bigsqcup \mathcal{D}_2)$$

Esempio di analisi della propagazione delle costanti

Riportiamo il programma programma precedentemente analizzato.

```
1 a := 1; b := 2; c := 3; d := 3; e := 0;
2 while B do
3 |   b := 2 · a; d := d + 1; e := e − a;
4 |   c := e + d; a := b − a;
```



Dobbiamo costruire il sistema di disequazioni:

- $\mathcal{D}(1) \sqsupseteq \mathcal{D}_\perp$
- $\mathcal{D}(2) \sqsupseteq [\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto 3, \mathtt{e} \mapsto 0] \sqcup \mathcal{D}(4)[a \mapsto \mathcal{D}(4)(a) -^\sharp \mathcal{D}(4)(b), c \mapsto \mathcal{D}(4)(e) +^\sharp \mathcal{D}(4)(d)]$
- $\mathcal{D}(3) \sqsupseteq \mathcal{D}(2)$
- $\mathcal{D}(4) \sqsupseteq \mathcal{D}(3)[b \mapsto 2 \cdot^\sharp \mathcal{D}(3)(a), d \mapsto \mathcal{D}(3)(d) +^\sharp 1, e \mapsto \mathcal{D}(3)(e) -^\sharp \mathcal{D}(3)(a)]$
- $\mathcal{D}(5) \sqsupseteq \mathcal{D}(2)$

Nel least upper bound accorpiamo i vari archi che arrivano allo stesso punto, quindi la minima soluzione del sistema di disequazioni diventa uguale alla soluzione del sistema di equazioni.

- $\mathcal{D}(1) = \mathcal{D}_\perp$
- $\mathcal{D}(2) = [\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto 3, \mathtt{e} \mapsto 0] \sqcup \mathcal{D}(4)[a \mapsto \mathcal{D}(4)(b) -^\sharp \mathcal{D}(4)(a), c \mapsto \mathcal{D}(4)(e) +^\sharp \mathcal{D}(4)(d)]$
- $\mathcal{D}(3) = \mathcal{D}(2)$
- $\mathcal{D}(4) = \mathcal{D}(3)[b \mapsto 2 \cdot^\sharp \mathcal{D}(3)(a), d \mapsto \mathcal{D}(3)(d) +^\sharp 1, e \mapsto \mathcal{D}(3)(e) -^\sharp \mathcal{D}(3)(a)]$
- $\mathcal{D}(5) = \mathcal{D}(2)$

Procediamo con la soluzione del sistema di equazioni:

	0	1	2
$\mathcal{D}(1)$	$\mathcal{D}_\bot$	$\mathcal{D}_\bot$	$\mathcal{D}_\bot$
$\mathcal{D}(2)$	$\varnothing$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto 3, \mathtt{e} \mapsto 0$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto \top, \mathtt{e} \mapsto \top$
$\mathcal{D}(3)$	$\varnothing$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto 3, \mathtt{e} \mapsto 0$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto \top, \mathtt{e} \mapsto \top$
$\mathcal{D}(4)$	$\varnothing$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto 4, \mathtt{e} \mapsto -1$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto \top, \mathtt{e} \mapsto \top$
$\mathcal{D}(5)$	$\varnothing$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto 3, \mathtt{e} \mapsto 0$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto 3, \mathtt{d} \mapsto \top, \mathtt{e} \mapsto \top$

	3
$\mathcal{D}(1)$	$\mathcal{D}_\bot$
$\mathcal{D}(2)$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto \top, \mathtt{d} \mapsto \top, \mathtt{e} \mapsto \top$
$\mathcal{D}(3)$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto \top, \mathtt{d} \mapsto \top, \mathtt{e} \mapsto \top$
$\mathcal{D}(4)$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto \top, \mathtt{d} \mapsto \top, \mathtt{e} \mapsto \top$
$\mathcal{D}(5)$	$\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{c} \mapsto \top, \mathtt{d} \mapsto \top, \mathtt{e} \mapsto \top$

Raggiungiamo il punto fisso alla terza iterazione. Di fatto in funzione dell'analisi osserviamo che l'informazione persa è quella relativa alla variabile c rispetto al calcolo concreto. Infatti nel calcolo concreto il suo valore è costante, ma nel calcolo astratto dipende da variabili che nel calcolo astratto sono $\top$.

6.1.4 Migliorie dell'analisi

Possiamo pensare di migliorare l'analisi sapendo che la presenza di guardie la cui informazione calcolata è rappresentabile all'interno della nostra osservazione; per capirlo osserviamo il seguente esempio:

```
1 if x = 7 then
2 └ e1;
```

All'interno del ramo e_1 è possibile considerare l'informazione relativa al fatto che il valore di x è 7. Tale informazione è rappresentabile nel dominio delle costanti e utilizzarlo per il calcolo successivo.

Ovviamente il concetto riguarda le guardie la cui informazione è osservabile all'interno del ramo di esecuzione. In questo caso rappresentiamo la semantica con tali migliorie nel seguente modo:

$$[\![\texttt{NonZero(e)}]\!]^\sharp\mathcal{D} = \begin{cases} \bot & \text{se } [\![e]\!]^\sharp\mathcal{D} = 0 \\ \mathcal{D}_1 & \text{altrimenti } (\exists n \neq 0. [\![e]\!]^\sharp\mathcal{D} = n \vee [\![e]\!]^\sharp\mathcal{D} = \top) \end{cases}$$

Dove $\mathcal{D}_1 = \mathcal{D}[x \mapsto \mathcal{D}(x) \sqcap [\![e]\!]^\sharp\mathcal{D}] = [\![e]\!]^\sharp\mathcal{D}$ Poiché si tratta di un'informazione di uguaglianza, quello che sappiamo è che nel raggio di azione della guardia, il valore di x è esattamente il valore della semantica.

6.2 Analisi degli intervalli

In generale quando ci interessa qualche proprietà più sofisticate perdiamo le condizioni di finitezza del dominio e talvolta l'`ACC`. In particolare nell'analisi degli intervalli, dove il dominio è infinito e non abbiamo l'`ACC`, quindi abbiamo catene ascendenti **infinite**.

Nel dominio degli intervalli ogni elemento è rappresentato da un oggetto nella forma $[l, u] = \{n \in \mathbb{Z} \mid l \leq n \leq u\}$, dove l è il limite inferiore e u è il limite superiore. Tra questi valori abbiamo anche intervalli dove $l = -\infty$ e $u = +\infty$.

$$[l, +\infty] = \{n \in \mathbb{Z} \mid l \leq n\}$$

$$[-\infty, u] = \{n \in \mathbb{Z} \mid n \leq u\}$$

Prendendo in considerazione un qualsiasi elemento $[l, u]$ del dominio, siamo in grado di trovare una quantità infinita di elementi che sono maggiori di $[l, u]$. Il calcolo del punto fisso lavora collezionando i valori che le variabili possono assumere durante l'esecuzione del programma e lavorare in un dominio non `ACC` significa che tale catena può divergere perché possiamo continuare ad aggiungere elementi senza trovare il punto fisso.

Si tratta di un caso, nella sua semplicità, che contiene tutte le difficoltà dell'analisi statica. Vediamone ora la costruzione nel dettaglio.

6.2.1 Costruzione dell'analisi

L'analisi degli intervalli non ha come obiettivo osservare l'insieme concreto dei valori raggiungibili durante l'esecuzione, ma solamente il range di valori che una variabile può acquisire in un certo punto del programma. I punti di programma che sono solitamente interessanti sono quelli all'interno di un ciclo, essendo questi i punti in cui ci può essere un'evoluzione del valore. Gli intervalli approssimano riempiendo i buchi tra i valori, e per questo sono insiemi convessi. Un chiaro esempio di convessità è il seguente.

Supponiamo che x sia una variabile intera e che assume i seguenti valori durante l'esecuzione del programma:

$$x \mapsto \{1, 5, 20, 35\}$$

Nella nostra analisi, è chiaro che nell'analisi assumerà il seguente valore:

$$x \mapsto [1, 35]$$

Prendendo in considerazione elementi che in realtà non sono presenti nel dominio concreto, rendendo di fatto l'insieme convesso.

La prima cosa da fare è definire il dominio degli elementi che vogliamo osservare con precisione.

$$\mathbb{I} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

Nel dominio concreto abbiamo l'insieme di valori su $\mathbb{Z}$ che le variabili possono assumere, ovvero $\wp(\mathbb{Z})$. Il secondo passo è assicurarci che il dominio $\mathbb{I}$ sia effettivamente un dominio astratto, ovvero che esista un'inserzione di Galois tra il dominio concreto e quello astratto.

L'inserzione di Galois tra i due domini $\mathcal{C}$ e $\mathcal{A}$ è data dalle funzioni:

$$\alpha(x) = [\texttt{min}_\infty(x), \texttt{max}_\infty(x)] \in \mathbb{I}$$

Dove le due funzioni $\texttt{min}_\infty$ e $\texttt{max}_\infty$ sono definite nel seguente modo:

$$\texttt{min}_\infty(x) = \begin{cases} n \in x & \text{se } \forall n \in x.m \leq n \\ -\infty & \text{altrimenti} \end{cases}$$

$$\texttt{max}_\infty(x) = \begin{cases} n \in x & \text{se } \forall n \in x.m \geq n \\ +\infty & \text{altrimenti} \end{cases}$$

Mentre la funzione γ è definita nel seguente modo:

$$\gamma([l, u]) = \{n \in \mathbb{Z} \mid l \leq_\infty n \leq_\infty u\}$$

Dove il pedice ∞ indica che:

$$\forall m. \quad -\infty \leq_\infty \qquad m \leq_\infty +\infty$$

Quello che si può dimostrare è che α e γ costituiscono un'inserzione di Galois tra i due domini.

$$\wp(\mathbb{Z}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{I}$$

Dato un qualunque insieme, esiste la miglior approssimazione possibile, quindi esiste sempre il più piccolo intervallo che contiene l'insieme di partenza.

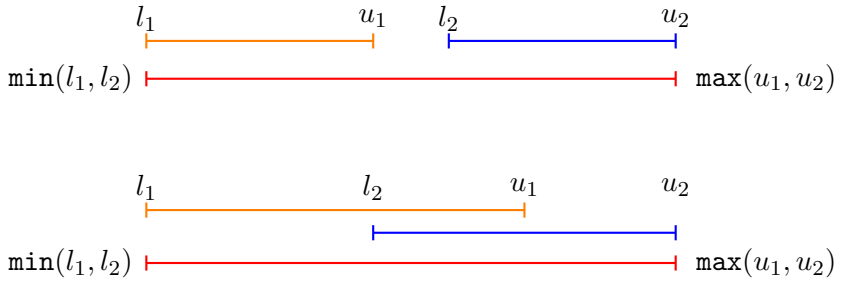
6.2.2 Operazioni del reticolo

- Ordinamento

$$[l_1, u_1] \sqsubseteq [l_2, u_2] \iff l_1 \geq l_2 \wedge u_1 \leq u_2$$

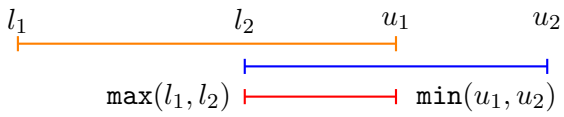
- Least upper bound: ovvero il più piccolo intervallo che contiene entrambi gli intervalli.

$$[l_1, u_1] \sqcup [l_2, u_2] = [\texttt{min}(l_1, l_2), \texttt{max}(u_1, u_2)]$$



- Greatest lower bound: ovvero il più grande intervallo contenuto in entrambi gli intervalli.

$$[l_1, u_1] \sqcap [l_2, u_2] = [\texttt{max}(l_1, l_2), \texttt{min}(u_1, u_2)]$$



La descrizione di tali operazioni completa la descrizione del dominio, poiché siamo in presenza di un reticolo completo.

6.2.3 Semantica delle espressioni

Trattiamo ora del trasferimento del calcolo delle operazioni concrete nel dominio astratto, quindi definiamo la funzione di trasferimento astratta. Tutte le operazioni possono essere trasferite, possiamo quindi fornire in modo operativo sugli intervalli la **best correct approximation** delle operazioni concrete. Definiamo alcune di queste operazioni:

- $[l_1, u_1] +^\sharp [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$ dove ricordiamo che $-\infty + n = -\infty$ e $+\infty + n = +\infty$.
- $-^\sharp [l, u] = [-u, -l]$.
- $[l_1, u_1] \cdot^\sharp [l_2, u_2] = [\texttt{min}(l_1 \cdot l_2, l_1 \cdot u_2, u_1 \cdot l_2, u_1 \cdot u_2), \texttt{max}(l_1 \cdot l_2, l_1 \cdot u_2, u_1 \cdot l_2, u_1 \cdot u_2)]$.

Nelle operazioni di confronto, per dire che due intervalli sono uguali, dobbiamo essere sicuri che i loro valori nel dominio concreto lo siano, perciò due intervalli sono uguali se e solo se gli intervalli sono composti da un solo elemento e sono uguali, ovvero:

$$[l_1, u_1] =^\sharp [l_2, u_2] = \begin{cases} [1,1] & \text{se } l_1 = u_1 = l_2 = u_2 \\ [0,0] & \text{se } u_i < l_2 \vee u_2 < l_1 \quad \textit{intervalli disgiunti} \\ [0,1] & \text{altrimenti} \end{cases}$$

Ciò significa che siamo assolutamente certi che due intervalli sono uguali se e solo se sono composti da un solo elemento e sono uguali. Sono sicuramente diversi se sono disgiunti, poiché nel corso dell'esecuzione hanno assunto valori totalmente diversi, rendendo impossibile per qualunque combinazione di valori che essi siano uguali. In tutti gli altri casi non possiamo essere certi che siano uguali, anche se i due intervalli sono uguali, poiché i valori nel dominio concreto potrebbero essere diversi.

Definiamo quindi il dominio degli stati astratti negli intervalli come:

$$\mathbb{D} = (\texttt{Var} \to \mathbb{I}) \cup \{\bot\}$$

Dove $\bot$ è indica uno stato non percorribile, oppure un errore.

Dato $\mathcal{D} \in \mathbb{D}$, definiamo la funzione di trasferimento astratta α come la funzione che per ogni x appartenente alle variabili, associa ad x, l'applicazione della funzione di trasferimento astratta α all'intervallo associato ad x in $\mathcal{D}$.

$$\alpha(\mathcal{D}) \implies \forall x \in \texttt{Var} \quad x \mapsto \alpha(\mathcal{D})(x)$$

Estendendo quindi la funzione di trasferimento astratta α dalla funzione che opera sugli intervalli, alla funzione che opera sugli stati che vogliono associare intervalli ai valori.

L'ordinamento rimane sempre quello definito per la propagazione delle costanti, quindi quando dobbiamo definire due stati astratti si confronteranno i valori per ogni variabile. Quindi:

$$\mathcal{D}_1 \sqsubseteq \mathcal{D}_2 \iff \forall x \in \texttt{Var} \quad \mathcal{D}_1(x) \sqsubseteq_{intervalli} \mathcal{D}_2(x)$$

L'obiettivo è avere questa informazione per ogni punto di programma, quindi cerchiamo la soluzione `MOP`, ovvero la soluzione *merge over all paths*, definendo quindi:

$$\texttt{MOP} : \mathcal{I}^\star[v] = \bigsqcup \{ [\![\pi]\!]^\sharp \mathcal{D}_\bot \mid \pi : \text{start} \to v \}$$

Ovvero il *least upper bound* (*poiché la collecting semantics è di tipo possible*) delle semantiche di tutti i cammini a partire dall'informazione iniziale $\mathcal{D}_\bot$ dei cammini che vanno dal nodo di inizio al nodo v. È chiaro che se guardiamo tale analisi dal punto di vista delle analisi di data flow, l'analisi è forward, ed è di tipo possibile perché colleziona valori.

6.2.4 Semantica dei comandi

Descriviamo la semantica astratta dei comandi, sorvolando sulla semantica delle espressioni, poiché è analoga alla definizione della semantica astratta delle espressioni della propagazione delle costanti (6.1.2).

La semantica dei comandi ovviamente si baserà sulla semantica delle espressioni nel caso delle guardie e dell'assegnamento. Ragioniamo quindi induttivamente sulla struttura dei comandi, definendo la semantica astratta dei comandi:

- $[\![\texttt{;}]\!]^\sharp \mathcal{D} = \mathcal{D}$
- $[\![x \leftarrow e]\!]^\sharp \mathcal{D} = \mathcal{D}[x \mapsto [\![e]\!]^\sharp \mathcal{D}]$
- $[\![\texttt{NonZero}(e)]\!]^\sharp \mathcal{D} = \begin{cases} \bot & \text{se } [0,0] = [\![e]\!]^\sharp \\ \mathcal{D} & \text{altrimenti} \end{cases}$
- $[\![\texttt{Zero}(e)]\!]^\sharp \mathcal{D} = \begin{cases} \bot & \text{se } [0,0] \not\sqsubseteq [\![e]\!]^\sharp \\ \mathcal{D} & \text{se } [0,0] \sqsubseteq [\![e]\!]^\sharp \end{cases}$
- $[\![\texttt{input}(x)]\!]^\sharp \mathcal{D} = \mathcal{D}[x \mapsto \top]$

Si può dimostrare che la semantica astratta $[\![\cdot]\!]^\sharp$ non è distributiva, quindi calcolando la soluzione `MFP` questa non sarà uguale alla soluzione `MOP`, ma la contiene strettamente.

$$\texttt{MFP} \sqsupseteq \texttt{MOP}$$

In ogni caso, la soluzione `MFP` è l'unica che riusciamo a costruire e l'analisi costruisce tale soluzione. Per farlo trova la soluzione del sistema di disequazioni, costruito esattamente come nel caso della analisi di data flow.

- Sul nodo di partenza la memoria astratta dove non abbiamo informazioni sulle variabili, quindi:
$$\mathcal{I}[\texttt{entry}] \sqsupseteq \mathcal{D}_\top$$
- Per i nodi successivi, essendo una semantica forward, $\mathcal{D}[v]$ contiene la semantica dell'etichetta a partire dalla memoria astratta a partire dal nodo sorgente.
$$\mathcal{I}[v] \sqsupseteq [\![\texttt{lab}]\!]^\sharp \mathcal{I}[u]$$

In ogni caso la semantica $[\![\cdot]\!]^\sharp$ è monotona, purtroppo non `ACC`, quindi contiene catene ascendenti infinite, ma non abbiamo la garanzia di terminazione.

Teorema 6.2.1 **Teorema di Knaster-Tarski**
Sia $\mathcal{L}$ un reticolo completo, e sia $f : \mathcal{L} \to \mathcal{L}$ una funzione monotona. Allora f ha un punto fisso, ovvero un elemento $x \in \mathcal{L}$ tale che $f(x) = x$.

La monotonia, per il teorema di Knaster-Tarski, ci garantisce che esiste un punto fisso, ma non ci garantisce che sia raggiungibile in un numero finito di passi.

6.2.5 Migliorie all'analisi

Possiamo pensare di migliorare l'analisi sapendo che la presenza di guardie la cui informazione calcolata è rappresentabile all'interno della nostra osservazione.

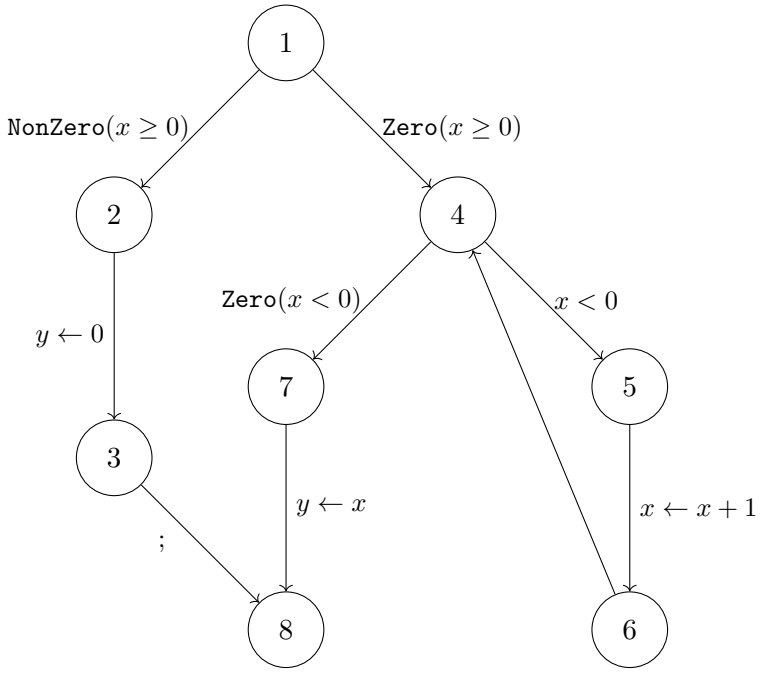
- $$[\![\texttt{NonZero}(e)]\!]^\sharp \mathcal{D} = \begin{cases} \bot & \text{se } [0,0] = [\![e]\!]^\sharp \\ \mathcal{D}_1 & \text{altrimenti} \end{cases}$$
  Dove $\mathcal{D}_1$ è può essere:
$$\mathcal{D}_1 = \begin{cases} \mathcal{D}[x \mapsto \mathcal{D}(x) \sqcap [\![e_1]\!]^\sharp] & \text{se } e \equiv (x = e_1) \\ \mathcal{D}[x \mapsto \mathcal{D}(x) \sqcap [-\infty, -u-1]] & \text{se } e \equiv (e_1 < x) \text{ e } [\![e_1]\!]^\sharp = [l,u] \\ \mathcal{D}[x \mapsto \mathcal{D}(x) \sqcap [l+1, +\infty]] & \text{se } e \equiv (x > e_1) \text{ e } [\![e_1]\!]^\sharp = [l,u] \end{cases}$$
- Il caso di `Zero` è analogo.

Ovviamente, la condizione vale se e è del tipo $x \,\square\, e_1$, dove il simbolo $\square$ può essere $\{=, <, >\}$.

Esempio di analisi degli intervalli

Consideriamo il seguente programma:

```
1 if x ≥ 0 then
2     y ← 0 ;
3 else
4     while x < 0 do
5         x ← x + 1 ;
6         y ← x ;
```



Dobbiamo costruire il sistema di disequazioni, che possiamo riportare già in forma di equazioni:

- $\mathcal{I}(1) = \mathcal{I}_\bot$
- $\mathcal{I}(2) = \mathcal{I}(1) \sqcap \mathcal{I}_\bot[x \mapsto [0, +\infty]] = \mathcal{I}(1) \sqcap [x \mapsto [0, +\infty]]$
- $\mathcal{I}(3) = \mathcal{I}(2)[y \mapsto [0, 0]]$
- $\mathcal{I}(4) = \mathcal{I}(1) \sqcap [x \mapsto [-\infty, -1]] \sqcup \mathcal{I}(6)$
- $\mathcal{I}(5) = \mathcal{I}(4) \sqcap [x \mapsto [-\infty, -1]]$
- $\mathcal{I}(6) = \mathcal{I}(5)[x \mapsto \mathcal{I}(5)(x) +^\sharp [1, 1]]$
- $\mathcal{I}(7) = \mathcal{I}(4) \sqcap [x \mapsto [0, +\infty]]$
- $\mathcal{I}(8) = \mathcal{I}(7)[y \mapsto \mathcal{I}(7)(x)] \sqcup \mathcal{I}(3)$

Possiamo quindi calcolare il punto fisso:

	0	1	2
$\mathcal{I}(1)$	$\mathcal{I}_\bot$	$\mathcal{I}_\bot$	$\mathcal{I}_\bot$
$\mathcal{I}(2)$	$\varnothing$	$[x \leftarrow [0, +\infty], y \leftarrow \top]$	$[x \leftarrow [0, +\infty], y \leftarrow \top]$
$\mathcal{I}(3)$	$\varnothing$	$[x \leftarrow [0, +\infty], y \leftarrow [0, 0]]$	$[x \leftarrow [0, +\infty], y \leftarrow [0, 0]]$
$\mathcal{I}(4)$	$\varnothing$	$[x \leftarrow [-\infty, -1], y \leftarrow \top]$	$[x \leftarrow [-\infty, 0], y \leftarrow \top]$
$\mathcal{I}(5)$	$\varnothing$	$[x \leftarrow [-\infty, -1], y \leftarrow \top]$	$[x \leftarrow [-\infty, -1], y \leftarrow \top]$
$\mathcal{I}(6)$	$\varnothing$	$[x \leftarrow [-\infty, 0], y \leftarrow \top]$	$[x \leftarrow [-\infty, 0], y \leftarrow \top]$
$\mathcal{I}(7)$	$\varnothing$	$\bot$	$[x \leftarrow [0, 0], y \leftarrow \top]$
$\mathcal{I}(8)$	$\varnothing$	$\bot$	$[x \leftarrow [0, +\infty], y \leftarrow [0, 0]]$

Dove raggiungiamo il punto fisso in 2 iterazioni.

6.3 Widening

Il *widening* è un'operazione che accelera le computazioni a punto fisso garantendo la convergenza. L'operatore di widening consente di eseguire "salti" per raggiungere un punto fisso approssimato.

Dato un dominio $\mathbb{P}$ su cui è definito un poset, il widening è una funzione:

$$\nabla : \mathbb{P} \times \mathbb{P} \to \mathcal{P}$$

dove:

$$\forall x, y \in \mathbb{P}. \quad x \sqsubseteq (x \nabla y) \wedge y \sqsubseteq (x \nabla y)$$

L'applicazione dell'operatore deve mantenere un certo ordine. Inoltre, quando definiamo ad hoc l'operatore di widening, le catene ascendenti devono essere convergenti.

Supponiamo di avere una catena del tipo:

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \ldots$$

La costruzione di una nuova catena, chiamata y, è definita come:

$$y_0 = x_0 \quad y_1 = x_1 \nabla y_0 \quad y_2 = x_2 \nabla y_1 \quad \ldots \quad y_n = x_n \nabla y_{n-1} \quad \ldots$$

L'elemento precedente della nuova catena deve essere combinato con il nuovo elemento della catena precedente, dove y_i è una catena non strettamente crescente ma che converge a un punto fisso.

La definizione del concetto di widening potrebbe risultare poco esplicativa in questo contesto; tuttavia, se si considerano le disequazioni nella forma

$$x_i \sqsubseteq f_i(x_1, \ldots, x_n)$$

e si applica la definizione di contenimento,

$$x_i \sqsubseteq f_i(x_1, \ldots, x_n) \iff x_i = x_i \sqcup f_i(x_1, \ldots, x_n)$$

sostituendo l'operatore di widening all'operatore di least upper bound, si ottiene

$$x_i = x_i \nabla f_i(x_1, \ldots, x_n) \tag{6.1}$$

Questo sistema di disequazioni assicura la costruzione di una catena convergente al punto fisso, garantendo nel contempo la terminazione (*sotto l'assunzione delle proprietà specificate*).

L'idea di fondo è quella di fornire un operatore che sostituiamo al least upper bound. Il least upper bound, per costruzione, essendo la più piccola soluzione, fa si che si possa divergere poiché prende in considerazione tutti gli elementi della catena. Il widening, invece, è un operatore che si comporta come un least upper bound ma che non prende in considerazione tutti gli elementi della catena, ma una sottocatena che converge al punto fisso. Perciò l'operazione di widening è un'operazione che viene utilizzata in combinazione con l'operazione di `narrowing` per garantire la terminazione. Che a partire dalla soluzione del widening, ovvero una soluzione estremamente approssimata, cerca di capire se il calcolo aveva informazioni che in realtà possono essere sfruttate per affinare la soluzione.

6.3.1 Widening per gli intervalli

Per gli intervalli, l'operatore di widening è definito come:

$$\bot \nabla \mathcal{D} = \mathcal{D} \nabla \bot = \mathcal{D}$$

Il widening di una memoria astratta con il bottom è sempre la memoria astratta stessa. Siamo costretti a fornire i due casi poiché il widening non è un'operazione commutativa. In un certo senso la posizione dice qual è l'oggetto calcolato precedentemente e quello successivo. C'è quindi l'idea di trattare l'elemento a destra come il nuovo valore e quello a sinistra come il valore precedente, dando quindi l'idea di direzione del calcolo che il widening vuole sfruttare.

Definiamo ora su due memorie qualunque $\mathcal{D}_1$ e $\mathcal{D}_2$ l'operatore di widening come:

$$\mathcal{D}_1 \nabla \mathcal{D}_2(x) = \mathcal{D}_1(x) \nabla \mathcal{D}_2(x)$$

Di fatto viene definito puntualmente, di fatto il widening sugli stati astratti, come per le altre operazioni, non è altro che l'applicazione dell'operatore sul dominio dei valori astratti, permettendoci quindi di definirlo sul dominio astratto degli intervalli.

Dati due intervalli $[l_1, u_1]$ e $[u_2, u_2] \in \mathbb{I}$, il widening è definito come:

$$[l_1, u_1] \nabla [l_2, u_2] = [l, u]$$

Definito come:

$$l = \begin{cases} l_1 & \text{se } l_1 \leq l_2 \\ -\infty & \text{altrimenti} \end{cases}$$

L'oggetto a destra è quindi il nuovo oggetto calcolato, quindi se l_2 è più piccolo o uguale di l_1, ricordandoci che dobbiamo mantenere conservatività, allora il nuovo valore calcolato è l_1. Se prendessimo l_2 come nuovo valore calcolato, potremmo gli elementi calcolati precedentemente. Nel caso in cui l_2 sia maggiore di l_1, allora il nuovo valore calcolato è $-\infty$. Questo perché il calcolo ha portato all'aumento degli elementi che dobbiamo includere, per far il "salto", il widening, butta via l'informazione assumendo che il valore calcolato vada verso $-\infty$.

Per quanto riguarda l'upper bound, invece, abbiamo:

$$u = \begin{cases} u_1 & \text{se } u_1 \geq u_2 \\ +\infty & \text{altrimenti} \end{cases}$$

Se u_1 è maggiore o uguale di u_2, allora il nuovo valore calcolato è u_1, altrimenti è $+\infty$. Anche in questo caso, se u_2 è maggiore di u_1, allora il calcolo ha portato all'aumento degli elementi che dobbiamo includere, per far il "salto", il widening, butta via l'informazione assumendo che il valore calcolato vada verso $+\infty$.

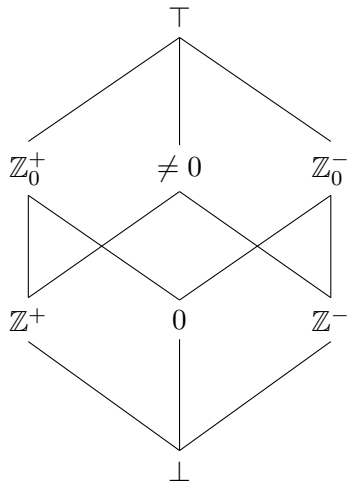
6.4 Analisi dei segni

Usiamo il dominio dei segni per sintetizzare il processo di costruzione di un'analisi nell'ambito non distributivo.

6.4.1 Costruzione dell'analisi

Nella fase di costruzione dell'analisi procediamo per individuare gli elementi da osservare.

Supponiamo di voler osservare i segni andando a distinguere tra zero, i positivi e i negativi.



A questo punto dobbiamo descrivere le operazioni astratte sugli oggetti che abbiamo deciso di osservare, fornendo quindi la semantica delle espressioni. In generale:

$$[\![e_1 \,\square e_2\,]\!]^\sharp \mathcal{D} = [\![e_1]\!]^\sharp \mathcal{D} \square [\![e_2\,]\!]^\sharp\, \mathcal{D}$$

Dove l'operatore $\square$ è un'operazione concreta. Scendiamo induttivamente all'interno della struttura dell'espressione, valutando la semantica di e_1 e e_2 e applicando l'operazione astratta $\square^\sharp$ ridefinite sugli elementi astratti.

Consideriamo la moltiplicazione nel dominio dei segni:

$*$	$\top$	$\mathbb{Z}_0^-$	$\mathbb{Z}_0^+$	$\mathbb{Z}^-$	$\mathbb{Z}^+$	0
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	0
$\mathbb{Z}_0^-$	$\top$	$\mathbb{Z}_0^+$	$\mathbb{Z}_0^-$	$\mathbb{Z}_0^+$	$\mathbb{Z}_0^-$	0
$\mathbb{Z}_0^+$	$\top$	$\mathbb{Z}_0^-$	$\mathbb{Z}_0^+$	$\mathbb{Z}_0^-$	$\mathbb{Z}_0^+$	0
$\mathbb{Z}^-$	$\top$	$\mathbb{Z}_0^+$	$\mathbb{Z}_0^-$	$\mathbb{Z}^+$	$\mathbb{Z}^-$	0
$\mathbb{Z}^+$	$\top$	$\mathbb{Z}_0^-$	$\mathbb{Z}_0^+$	$\mathbb{Z}^-$	$\mathbb{Z}^+$	0
0	0	0	0	0	0	0

Figura 6.4.1: Operazioni astratte relative alla moltiplicazione.

Ovviamente tale procedura può essere estesa a tutte le operazioni.

A questo punto dobbiamo descrivere la semantica dei comandi, che per semplicità forniamo sull'*abstract edge effect*, quindi sulle etichette del *control flow graph*. Nel caso dei segni, supponiamo che $\mathcal{D}$ sia lo stato astratto, quindi:

- $[\![;]\!]^\sharp\mathcal{D} = \mathcal{D}$
- $[\![x \leftarrow e]\!]^\sharp\mathcal{D} = \mathcal{D}[x \mapsto [\![e]\!]^\sharp\mathcal{D}]$
- $[\![\texttt{input}(x)]\!]^\sharp\mathcal{D} = \mathcal{D}[x \mapsto \top]$
- Il ramo non è percorribile se il valore astratto di 0 è esattamente la valutazione astratta di e, altrimenti prendiamo $\mathcal{D}$ intersecato ai valori dell'espressione e.

$$[\![\texttt{NonZero}(x)]\!]^\sharp\mathcal{D} = \begin{cases} \bot & \text{se } \alpha(0) = [\![e]\!]^\sharp\mathcal{D} \\ \mathcal{D} \sqcap [\![e]\!]^\sharp\mathcal{D} & \text{altrimenti} \end{cases}$$

- Il ramo non è percorribile se il valore astratto di 0 non è contenuto all'interno della valutazione astratta di e, altrimenti prendiamo $\mathcal{D}$ intersecato ai valori della negazione dell'espressione e.

$$[\![\texttt{Zero}(x)]\!]^\sharp\mathcal{D} = \begin{cases} \bot & \text{se } \alpha(0) \not\sqsubseteq [\![e]\!]^\sharp\mathcal{D} \\ \mathcal{D} \sqcap [\![\neg e]\!]^\sharp\mathcal{D} & \text{altrimenti} \end{cases}$$

Dove all'analisi abbiamo fornito la logica di miglioramento.

Definiamo ora il calcolo dell'analisi, definendo il sistema di disequazioni. Nelle analisi non distributive e basate su semantiche collecting, interpretiamo il programma, quindi si tratta di analisi forward e possibile.

$$\mathcal{S}[\texttt{entry}] \sqsupseteq \mathcal{D}_\top$$

Che equivale a dire che non abbiamo informazioni sulle variabili allo stato iniziale, quindi l'informazione è $\top$ su tutte le variabili.

$$\mathcal{S}[v] \sqsupseteq [\![\texttt{lab}]\!]^\sharp \mathcal{S}[u] \qquad \forall \mathcal{R} = (u, \texttt{lab}, v)$$

Il fatto che sia *possible* è dato dall'ordinamento che è di tipo *contenimento*. Il valore che calcoliamo deve sovrastimare il valore calcolato, di fatto prendiamo il *least upper bound* tra il valore precedente e il valore calcolato.

Quello che facciamo di fatto è sostituire l'operatore di contenimento con l'operatore di uguaglianza, perché tutti i punti che ricevono più archi li forniamo direttamente come il least upper bound degli archi che arrivano, diventando equivalente al sistema di equazioni.