Analisi del Software

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

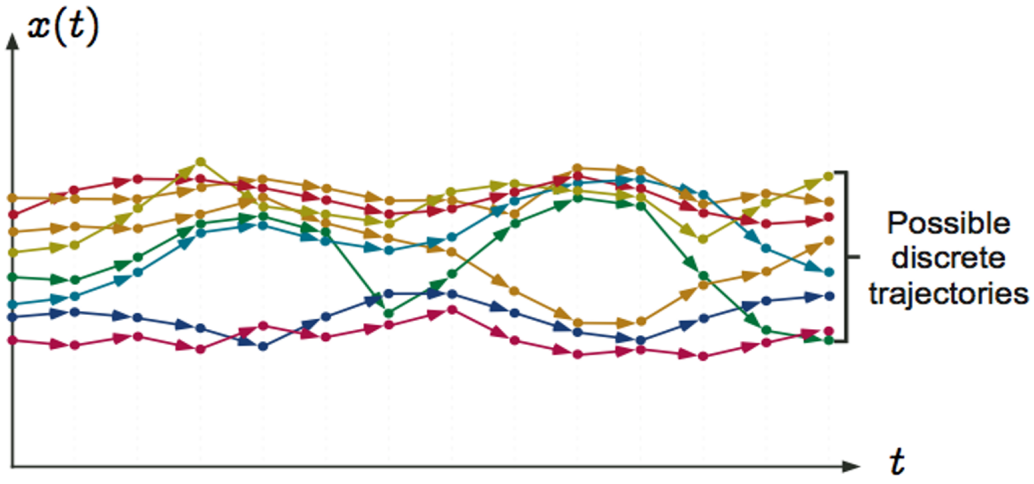
La semantica transazionale è un approccio alla comprensione del comportamento dei programmi attraverso la raccolta e l'analisi delle tracce di esecuzione. Questo approccio considera l'insieme di tutte le tracce di esecuzione possibili, partendo dagli stati iniziali del programma.

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:

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

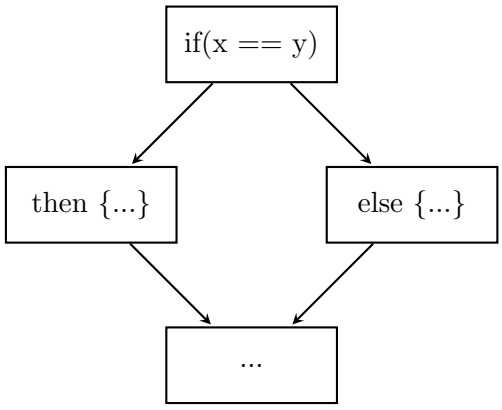
2.4 Il Control Flow Graph

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

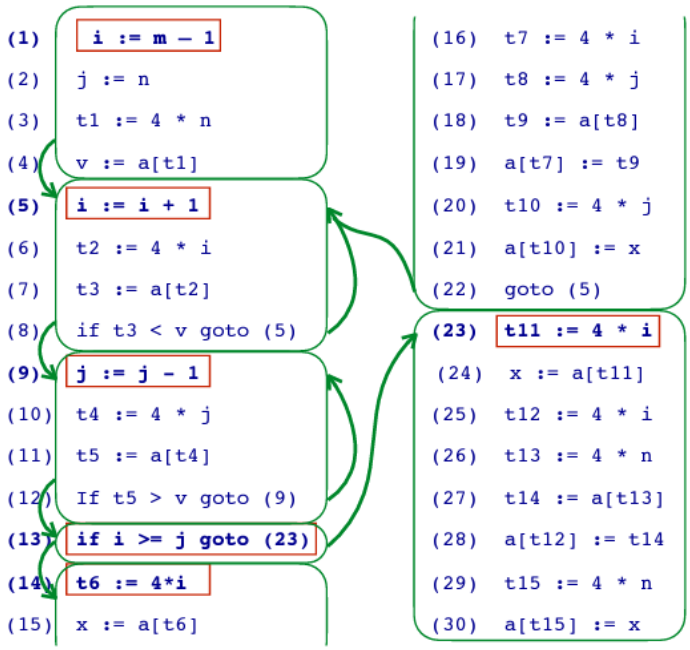
```
1. i := m - 1
2. j := n
3. t1 := 4 * n
4. v := a[t1]
5. i := m + 1
6. t2 := 4 * i
7. t3 := a[t2]
8. if t3 < v goto (5)
9. j:= j - 1
10. t4 := 4 * j
11. t5 := a[t4]
12. if t5 > v goto (9)
13. if i >= j goto (23)
14. t6 := 4 * i
15. x := a[t6]
16. t7 := 4 * i
17. t8 := 4 * j
18. t9 := a[t8]
19. a[t7] := 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:

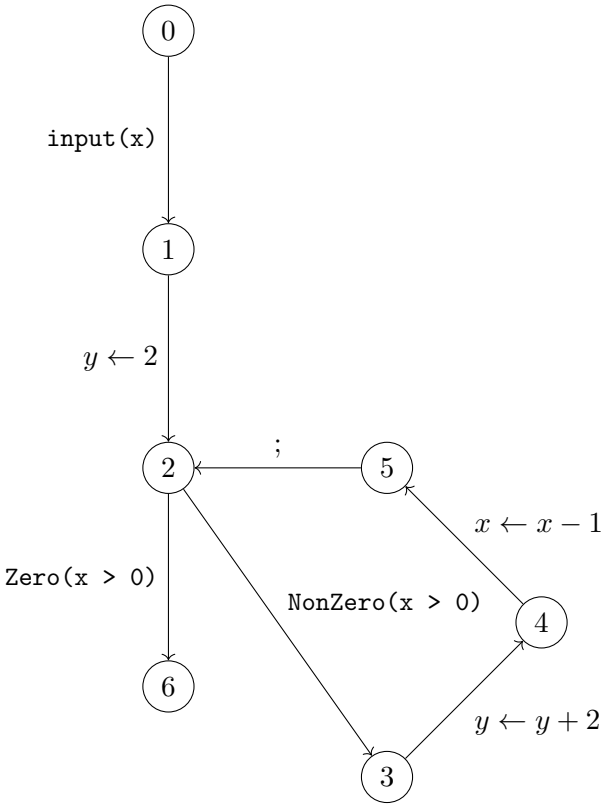
- I vertici corrispondono ai punti del programma
- Gli archi sono passi del calcolo etichettati con l'azione del programma corrispondente

- 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, $[\![P]\!]^{\#}$, 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 $[\![P]\!]^{\#}$, e l'insieme $[\![P]\!]^{\#}$ deve essere tale che possiamo dire con certezza se un oggetto appartiene a Q o meno.

Quando abbiamo questo insieme $[\![P]\!]^{\#}$, possiamo utilizzarlo per fare deduzioni su Q. Se tutti gli oggetti in $[\![P]\!]^{\#}$ 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 $[\![P]\!]^{\#}$ 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:

$$[\![P]\!] \subseteq Q \implies [\![P]\!]^{\#} \subseteq Q$$

Altrimenti, non possiamo saperlo con certezza!

3.1.1 Astrazione della semantica

Vediamo come costruire l'insieme $[\![P]\!]^{\#}$ 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

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

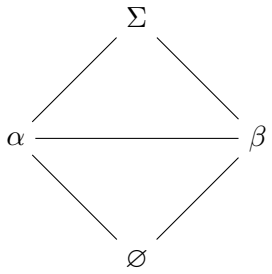
- ...

3.1.3 Proprietà

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

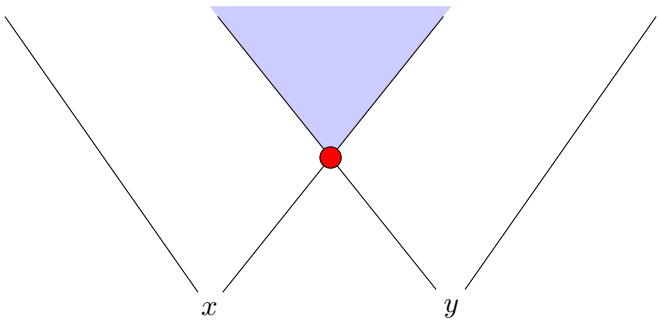
$$\langle \wp(\Sigma), \subseteq, \varnothing, \sigma, \Sigma, \cup, \cap, \neg \rangle$$

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.



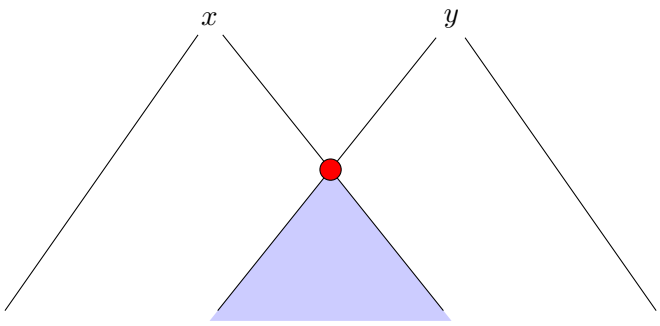
Least upper bound

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 $A \subseteq Y$.



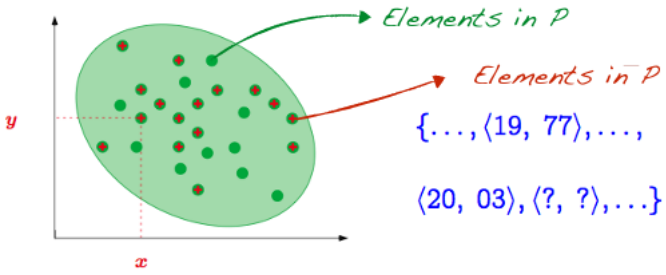
3.2 Approssimazione dei dati

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

Potremmo quindi avere:

- $d \subseteq d^{\#}$ ovvero **over approximation**.
- $d \supseteq d^{\#}$ ovvero **under approximation**.

3.2.1 Approssimazione dal basso

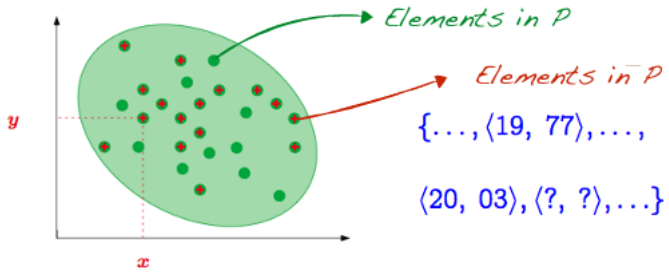


Per rispondere alla domanda $\langle x, y\rangle \in P$ utilizziamo un'astrazione $\bar{P}$, tale che $P \supseteq \bar{P}$.

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

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 $D^{\#}$ 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 $f^{\#}$ manipola quindi i segni.

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

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

- $x \subseteq \mathbb{Z} \to x^{\#} \in D^{\#}$ è il più piccolo insieme in $D^{\#}$ 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.

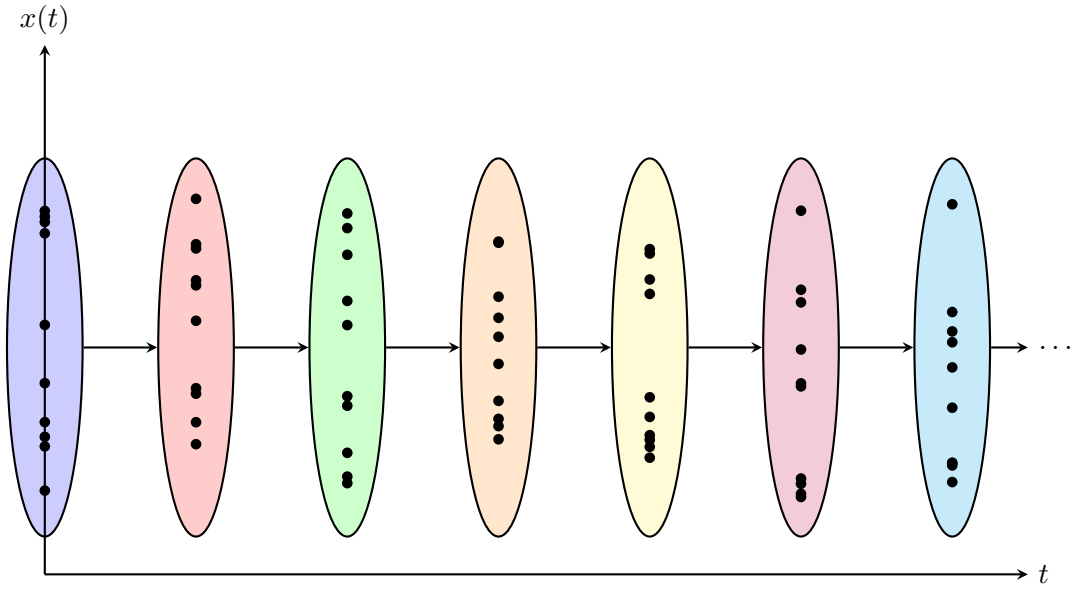
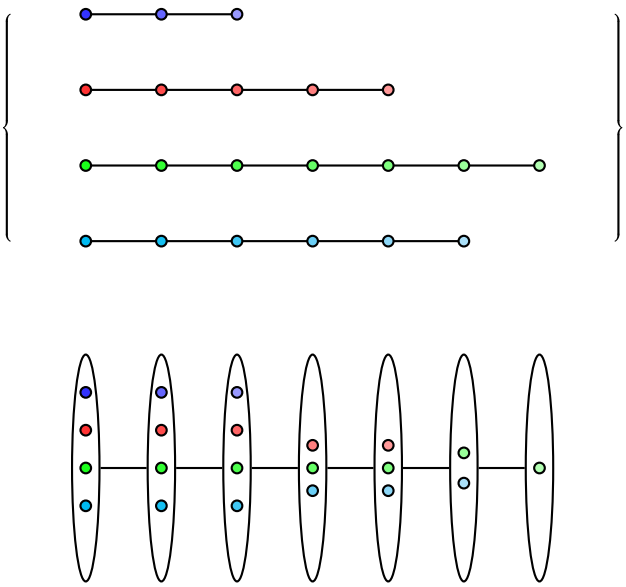


Figura 3.3.1: Traccia di una computazione di insiemi.

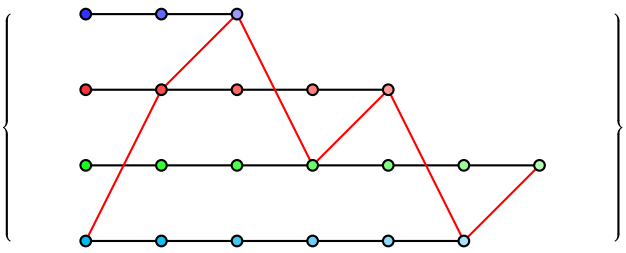
Il calcolo avviene quindi per punto fisso, partiamo quindi da un insieme di stati iniziali e andiamo via a via a collezionare tutti gli stati che raggiungiamo durante l'esecuzione, chiamato **reachability semantics** o **collecting semantics**.

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.

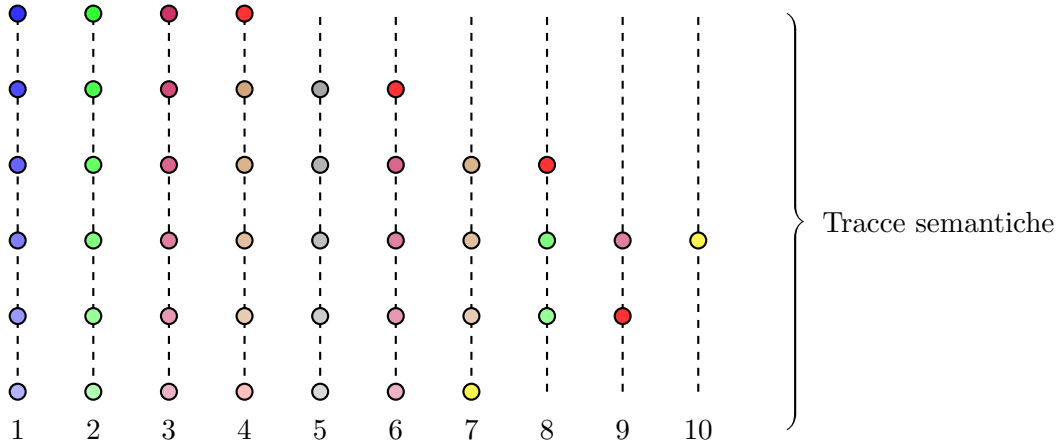


La semantica delle tracce mi colleziona l'insieme di tutte le tracce di computazione e la semantica delle collezioni invece considera per ogni passo di computazione l'insieme la proprietà raggiunta degli stati raggiunti. Abbiamo perso informazione rispetto alle tracce che rappresentiamo, in questo passaggio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

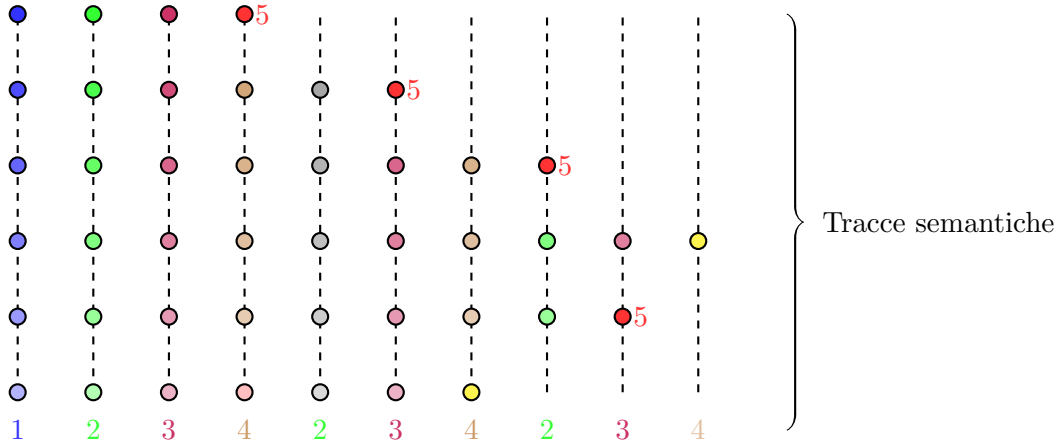


Abbiamo buttato via l'informazione che riguardava l'esatta transizione tra gli stati, aggiungendo tracce spurie.

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.

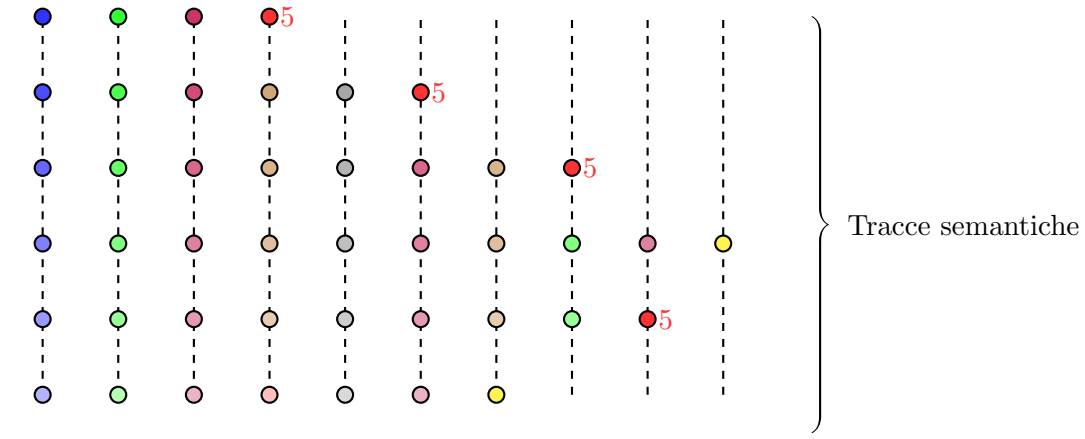


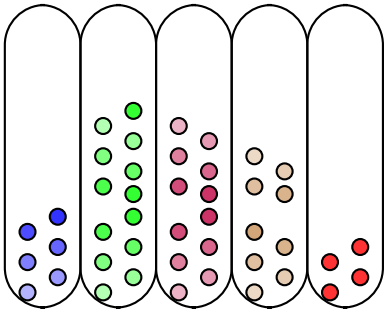
Ad ogni passo di computazione eseguiamo un istruzione in un **punto di programma**, possiamo quindi guardare il punto di programma che stiamo osservando.



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.

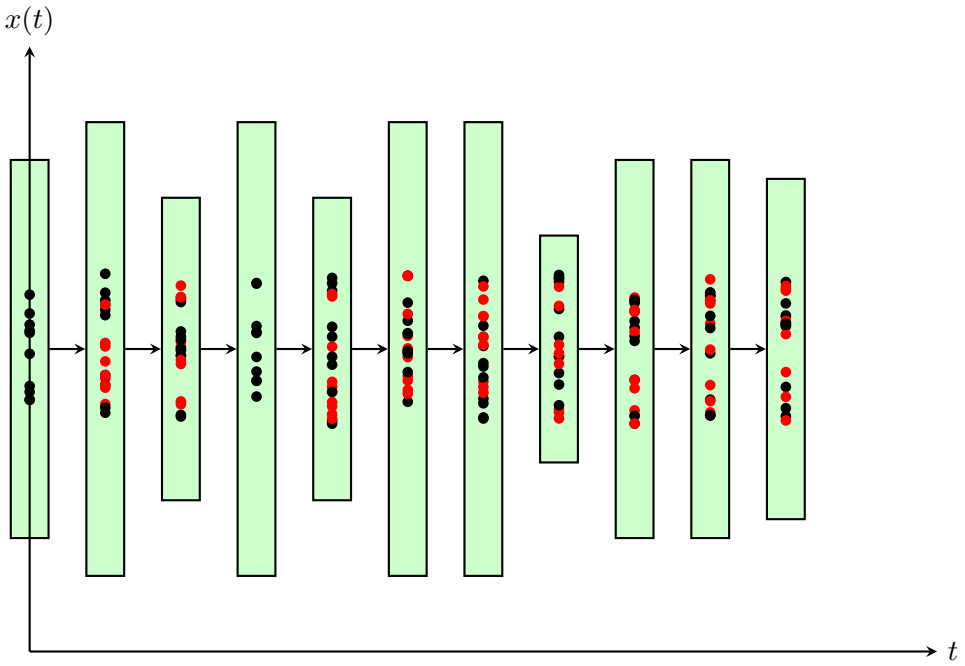


Figura 3.3.2: Traccia di una computazione sulle proprietà.

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).

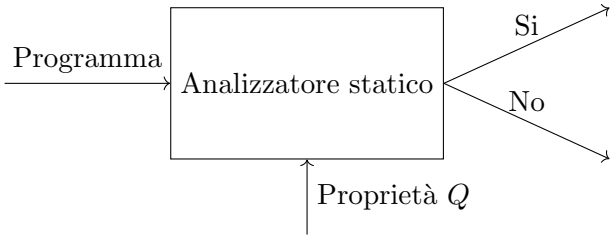
L'idea genera è quella di:

- 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 statica di Data Flow basata su CFG

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 $i \leftarrow n - 2$
2 S_5 : **if** $i < 0$ **goto** S_1
3 $j \leftarrow 0$
4 S_4 : **if** $j > i$ **goto** S_2
5 $t_1 \leftarrow j \cdot 4$
6 $t_2 \leftarrow \&A$
7 $t_3 \leftarrow t_2 + t_1$
8 $t_4 \leftarrow *t_3$
9 $t_5 \leftarrow j + 1$
10 $t_6 \leftarrow t_5 \cdot 4$
11 $t_7 \leftarrow \&A$
12 $t_8 \leftarrow t_7 + t_6$
13 $t_9 \leftarrow *t_8$
14 **if** $t_4 \leq t_9$ **goto** S_3
15 $t_{10} \leftarrow j \cdot 4$
16 $t_{11} \leftarrow \&A$
17 $t_{12} \leftarrow t_{11} + t_{10}$
18 `temp` $\leftarrow *t_{12}$
19 $t_{13} \leftarrow j + 1$
20 $t_{14} \leftarrow t_{13} \cdot 4$
21 $t_{15} \leftarrow \&A$
22 $t_{16} \leftarrow t_{15} + t_{14}$
23 $t_{17} \leftarrow *t_{16}$
24 $t_{18} \leftarrow j \cdot 4$
25 $t_{19} \leftarrow \&A$
26 $t_{20} \leftarrow t_{19} + t_{18}$
27 $*t_{20} \leftarrow t_{17}$
28 $t_{21} \leftarrow j + 1$
29 $t_{22} \leftarrow t_{21} \cdot 4$
30 $t_{23} \leftarrow \&A$
31 $t_{24} \leftarrow t_{23} + t_{22}$
32 $*t_{24} \leftarrow$ `temp`
33 $S_3 : j \leftarrow j + 1$
34 **goto** S_4
35 $S_2 : i \leftarrow i - 1$
36 **goto** S_5
37 S_1 : **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.

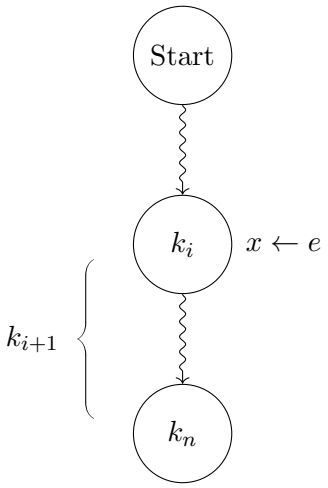
Se dimostriamo che l'espressione calcolata al punto A arriva anche al punto B, possiamo modificare il codice, ma per poterlo fare la risposta deve essere certa, senza alcun margine di errore. Non dobbiamo quindi classificare come ridondanti espressioni che non lo sono, in quanto potremmo modificare il 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.

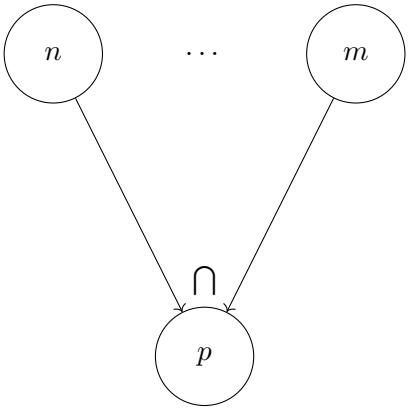


Sia $\pi = k_1, \ldots, k_n$ dal punto v al punto p. L'espressione è disponibile in x al punto p se:

- 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.

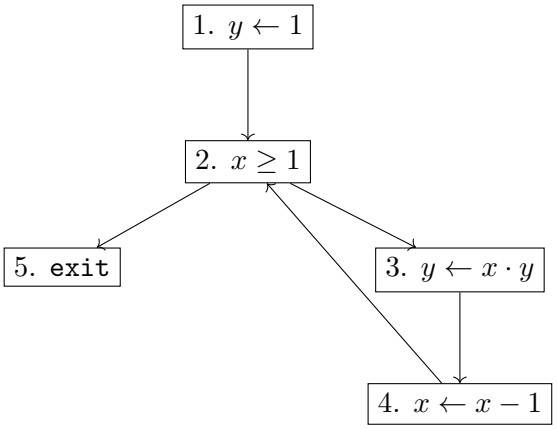
Algorithm 5: Available expressions

```
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 $[\![\cdot]\!]^{\#}\mathcal{A}$. Tale funzione è definita sulla semantica astratta, per induzione strutturale del `CFG`.

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

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

- $[\![x \leftarrow e]\!]^{\#}\mathcal{A} = \begin{cases} \mathcal{A} \setminus \texttt{Occ}(x) & x \in \texttt{Var}(e) \\ (\mathcal{A} \setminus \texttt{Occ}(x)) \cup \{x \leftarrow e\} & x \notin \texttt{Var}(e) \end{cases}$
  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.
- $[\![k_0, K_1, \ldots, k_n]\!]^{\#}\mathcal{A} = [\![k_0]\!]^{\#} \circ [\![k_1]\!]^{\#} \circ \cdots \circ [\![k_n]\!]^{\#}\mathcal{A}$

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.

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

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.

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

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	$\varnothing$	$\{c\}$	$\{a, c\}$	$\{a, c\}$	...

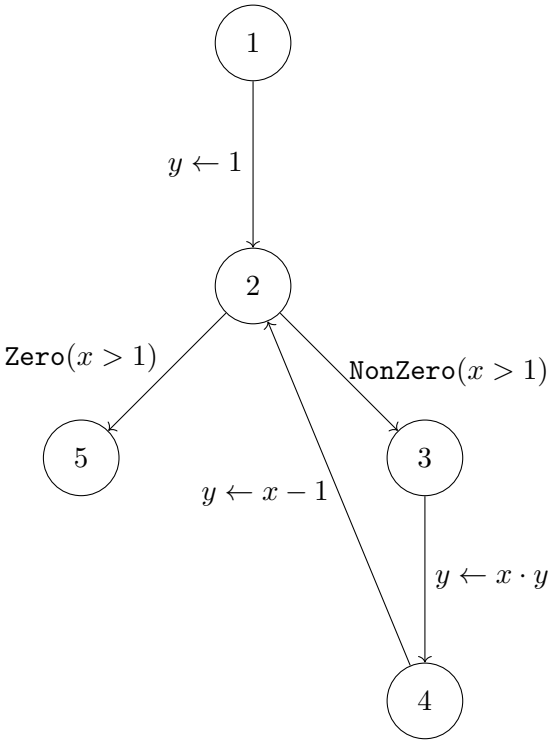
Il fix point viene trovato al punto 3, poiché i valori non cambiano più.

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:

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

ovvero

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

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.

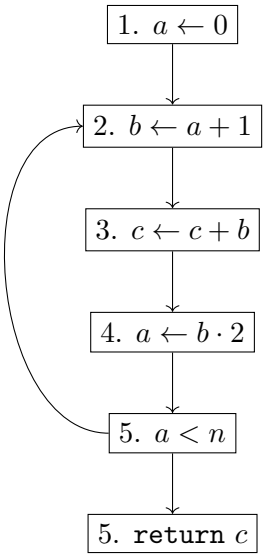
Esempio

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.



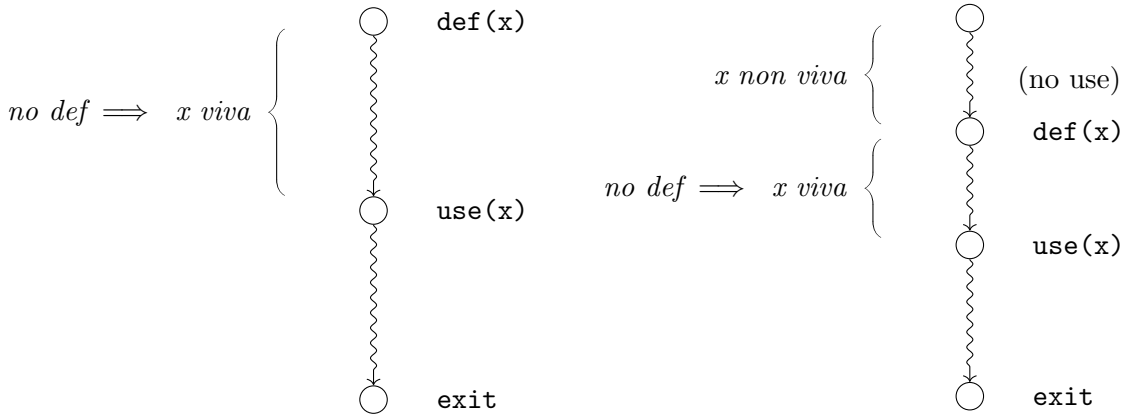
4.5.1 Analisi di liveness e l'approccio semantico

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.

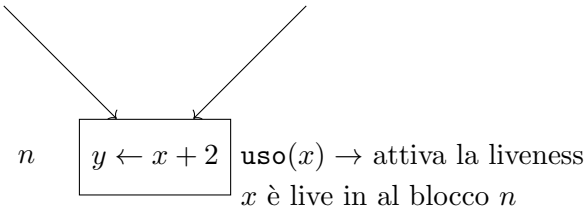


La costruzione dipende da quello che avviene prima di un certo punto di programma.

4.5.2 Calcolo della liveness

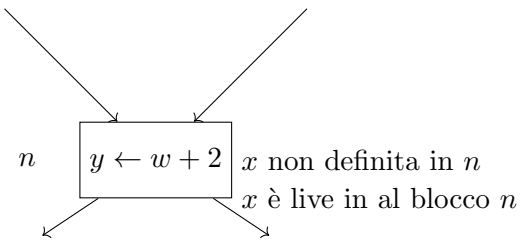
Cerchiamo di capire le condizioni che ci permettono di capire come l'esecuzione di un blocco influenza la liveness delle variabili, e se definite o possibile, ovvero se si combina per intersezione o unione.

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.

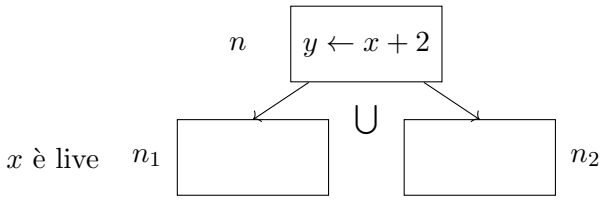


2. **Informazione preservata**: supponiamo di trovarci in un blocco n dove non abbiamo una definizione di x ma abbiamo x live in uscita da n, quindi l'informazione ha raggiunto il punto di programma attraverso i cammini. Quindi x si propaga in input allo stesso blocco.

   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.

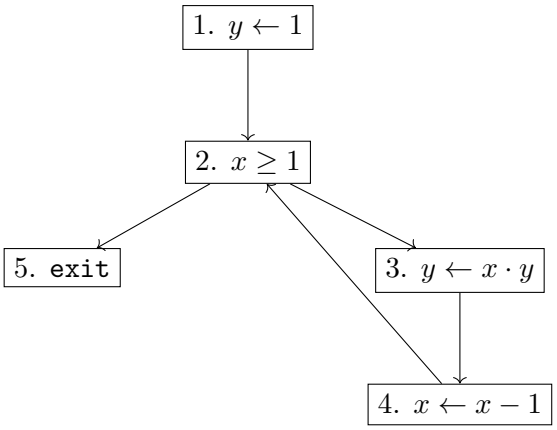
4.5.4 Precisione dell'analisi di *liveness*

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.

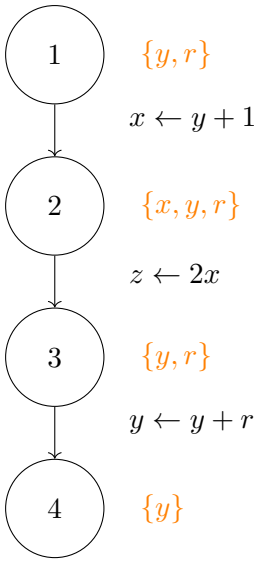


Figura 4.6.1: Prima iterazione dell'analisi liveness

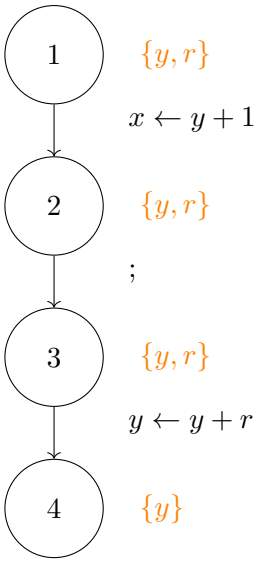


Figura 4.6.2: Seconda iterazione dell'analisi liveness

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

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

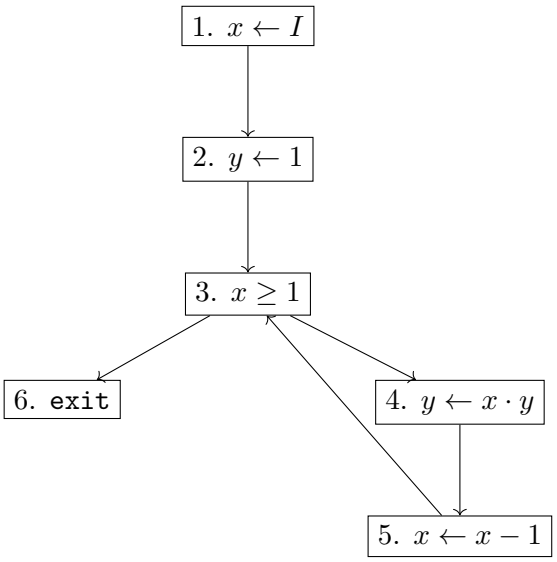
L'analisi è quindi distributiva, non cambiano quindi tutte le proprietà che abbiamo visto per le analisi precedenti.

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

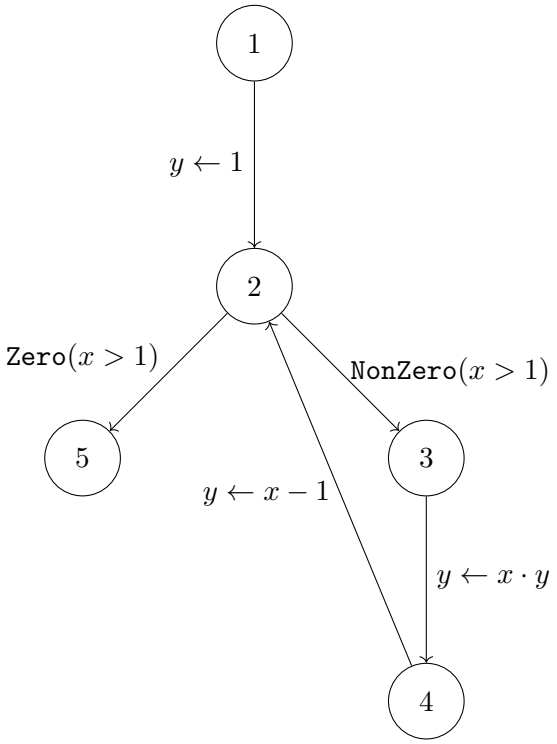
TLiveIn	1	2	3	...
6	$\varnothing$	$\varnothing$	$\varnothing$	...
5	$\varnothing$	$\{x\}$	$\{x\}$	...
4	$\varnothing$	$\varnothing$	$\{x\}$	...
3	$\{x\}$	$\{x\}$	$\{x\}$	...
2	$\varnothing$	$\{x\}$	$\{x\}$	...
1	$\varnothing$	$\varnothing$	$\{I\}$	...

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;
```



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

	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.

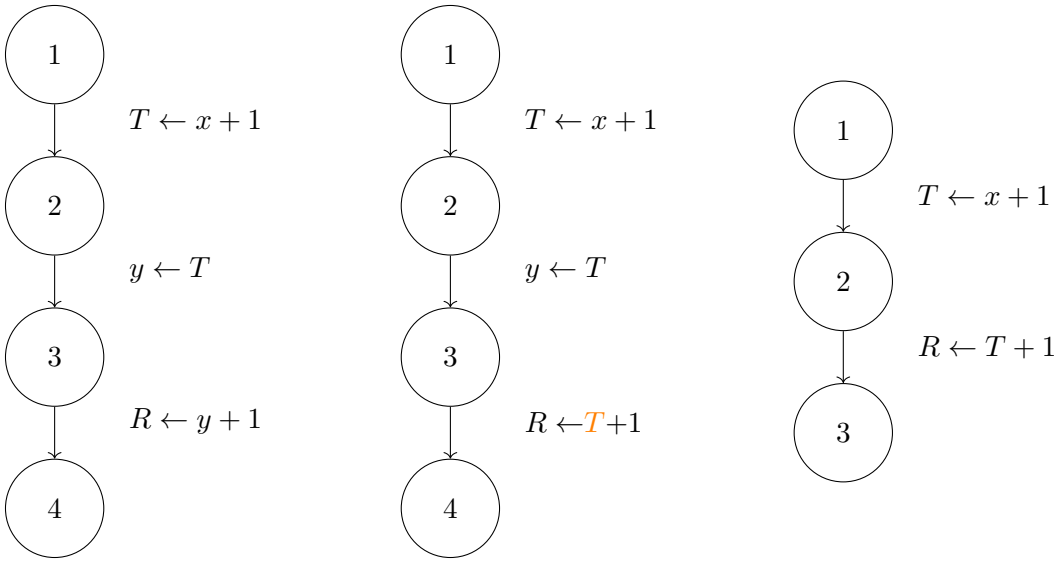
Osserviamo che la true liveness cattura falsi allarmi che un'applicazione ripetuta della liveness non saremmo in grado di catturare.

4.7 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.

È chiaro che stiamo prendendo in considerazione il programma nella direzione di esecuzione.

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.

$$e \tag{4.23}$$