Analisi del Software

Alessio Gjergji

Indice

1 Analisi **2**
1.1 Teorema di Rice . . . 3
1.1.1 Definizione Formale . . . 3
1.1.2 Conseguenze . . . 3
1.2 Approssimazione . . . 4
1.2.1 Confronto tra analisi statica e analisi dinamica nella verifica delle specifiche dei programmi . . . 5
1.2.2 Classificazione delle tecniche di analisi in base alle caratteristiche rinunciate per superare la non decidibilità . . . 6

2 Modelliamo programmi **7**
2.1 Un semplice linguaggio imperativo `IMP` . . . 7
2.1.1 La semantica di `IMP` . . . 7
2.1.2 Lo stato della memoria . . . 8
2.1.3 Semantica delle transizioni di stato . . . 8
2.1.4 Semantica delle espressioni . . . 9
2.1.5 Semantica dei comandi . . . 9
2.2 Semantica Transazionale . . . 10
2.3 Semantica come punto fisso . . . 12
2.3.1 Punto fisso inferiore . . . 12
2.3.2 Punto fisso superiore . . . 13
2.3.3 Semantica dei comandi come punto fisso . . . 14

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

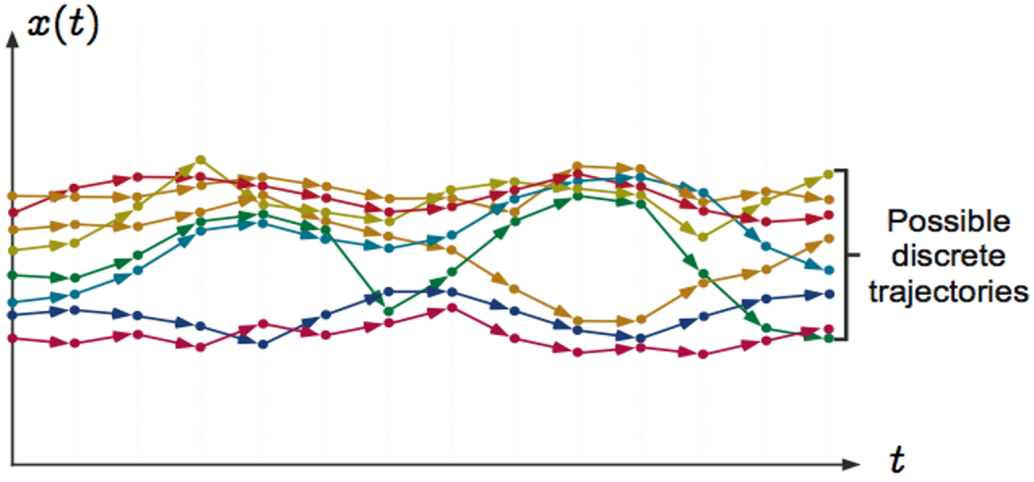
La semantica transazionale è un approccio alla comprensione del comportamento dei programmi attraverso la raccolta e l'analisi delle tracce di esecuzione. Questo approccio considera l'insieme di tutte le tracce di esecuzione possibili, partendo dagli stati iniziali del programma.

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:

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