

Fondamenti di analisi e verifica del software

Riassunto del corso

Creato da:

Davide Zampieri

Indice

1	Introduzione	1
1.1	Soddisfazione delle proprietà	1
1.2	Esempi	1
1.2.1	Buffer overflow	1
1.2.2	Pointer analysis	2
1.3	Alcuni ingredienti	2
1.3.1	Astrazione matematica	2
1.3.2	Semantica (vista come astrazione)	2
1.3.3	Interpretazione astratta	3
1.3.4	Control Flow Graph	3
2	Modellare i programmi	4
2.1	Semantica delle tracce	4
2.2	Semantica del punto fisso	4
2.2.1	Sistema di transizione	5
2.3	Come costruire il punto fisso	5
2.4	Control-Flow-Graph (CFG)	6
2.4.1	Basic blocks	6
2.4.2	Generazione del CFG	6
2.4.3	Notazione dei CFG	7
2.4.4	Linguaggio e semantica di IMP-CFG	7
3	Approssimare il significato dei programmi	8
3.1	Interpretazione astratta	8
3.2	Un esempio di oggetto: il fiore	8
3.2.1	Operazioni	9
3.2.2	Sovra-approssimazione	9
3.2.3	Oggetti e domini astratti	9
3.2.4	Funzioni di astrazione e concretizzazione	9
3.2.5	Connessioni di Galois	10
3.2.6	Ordinamento e operazioni nel mondo astratto	10
3.3	Astrazione delle semantiche	10
3.3.1	Collecting semantics	11
3.3.2	Esempio di funzionamento	11
3.3.3	Altre notazioni	12

4 Interpretazione astratta	13
4.1 Introduzione	13
4.1.1 Astrazione in base ad una proprietà	13
4.1.2 Oggetti e proprietà dei programmi	13
4.1.3 Direzione dell'astrazione	14
4.1.4 Approssimazione minima	14
4.1.5 Migliore approssimazione	15
4.1.6 Esempio sulla rappresentazione del segno	15
4.2 Connessioni di Galois	16
4.2.1 Esempio	16
4.3 Inserzioni di Galois	16
4.3.1 Esempio	17
4.4 Operatori di chiusura superiore (uco)	17
4.4.1 Esempio	17
4.5 Famiglie di Moore	18
4.5.1 Esempio	18
4.6 Esempio grafico	18
4.7 Reticolo delle astrazioni	19
4.8 Computazioni astratte e concrete	20
4.8.1 Correttezza (soundness)	20
4.8.2 Completezza (precision)	20
4.8.3 Forward vs Backward	21
4.8.4 Esempio su intervalli di interi	21
4.9 Estrapolazione del punto fisso	22
5 Analisi statica	23
5.1 Introduzione	23
5.1.1 Analisi sul CFG	23
5.2 Available Expressions	24
5.2.1 Costruire le equazioni	24
5.2.2 Risolvere il problema	24
5.2.3 Esempio	25
5.3 Framework monotono	25
5.3.1 Abstract edge effect	25
5.3.2 Fonti di imprecisione	26
5.3.3 Esempio	26
5.3.4 MOP vs MFP	27
5.3.5 Problemi distributivi e non distributivi	27
5.3.6 Sommario	28
5.3.7 Soluzione IDEAL	29
5.3.8 MOP vs MFP vs IDEAL	29
5.4 Analisi di data-flow	29
5.5 Analisi delle variabili live	29
5.5.1 Variabile live	29
5.5.2 Esempio su un programma	30
5.5.3 Notazione di base	30
5.5.4 Liveness su archi e su blocchi	31
5.5.5 Proprietà dell'analisi	31
5.5.6 Costruire le equazioni	32
5.5.7 Risolvere il problema	32

5.5.8 Esempio	32
5.5.9 Fonti di imprecisione	33
5.5.10 Abstract edge effect	33
5.5.11 Esempio	33
5.5.12 Rafforzare l'analisi	34
5.5.13 Modificare le equazioni	34
5.5.14 Esempio (con le modifiche)	35
5.5.15 Modificare l'abstract edge effect	35
5.5.16 Esempio (con le modifiche)	35
5.6 Propagazione delle copie	36
5.6.1 Analisi di copy propagation	36
5.6.2 Costruire le equazioni	36
5.6.3 Esempio	37
5.6.4 Abstract edge effect	37
5.6.5 Esempio e fonti di imprecisione	37
5.7 Reaching definitions	38
5.7.1 Costruire le equazioni	38
5.7.2 Ottimizzazione code motion	38
5.7.3 Esempio	39
5.7.4 Abstract edge effect	39
5.7.5 Esempio	39
5.8 Struttura delle analisi di data-flow	40
5.8.1 Riepilogo	40
5.8.2 Risoluzione degli esercizi	40
5.9 Analisi non distributive	41
5.10 Propagazione delle costanti	41
5.10.1 Esempio	41
5.10.2 Costruzione dell'analisi	42
5.10.3 Dominio astratto	42
5.10.4 Insieme dei valori assunti	43
5.10.5 Abstract edge effect	43
5.10.6 Esempio	44
5.10.7 MOP vs MFP	45
5.10.8 Rafforzare l'analisi	45
5.11 Analisi degli intervalli	46
5.11.1 Dominio astratto degli intervalli	46
5.11.2 Astrarre gli stati concreti	47
5.11.3 Abstract edge effect	47
5.11.4 Rafforzare l'analisi	47
5.11.5 Esempio	48
5.11.6 Widening	48
5.11.7 Definizione sugli intervalli	48
5.11.8 Narrowing	49
5.11.9 Esempio	49
5.11.10 Un esempio completo	50
5.12 Altre analisi su domini (non) relazionali	50
5.13 Risoluzione degli esercizi	50

6 Analisi dinamica	51
6.1 Introduzione	51
6.2 Testing	51
6.2.1 Obiettivi del testing	51
6.2.2 Tipologie di errori	51
6.2.3 Progressione di un fallimento	52
6.3 Aspetti teorici	52
6.3.1 Teoremi	53
6.4 Processo di testing	54
6.4.1 Generazione dei casi di test	54
6.4.2 Approccio code-based	54
6.5 Coverage testing	54
6.5.1 Statement coverage	55
6.5.2 Branch coverage	55
6.5.3 Condition coverage	55
6.5.4 Path coverage	56
6.6 Monitoring	56
6.6.1 Execution monitor	56
6.6.2 Proprietà di safety	57
6.6.3 Monitorare un linguaggio formale	57
6.6.4 Sfide del monitoring	58
6.6.5 Integrazione del monitor nell'applicazione	58
6.6.6 Violazione e validazione	59
6.6.7 Outline enforcement	59
7 Model checking	60
7.1 Introduzione	60
7.1.1 Fasi e aspetti critici	60
7.2 Scelta del modello del sistema	61
7.2.1 Strutture di Kripke	62
7.2.2 Esempio completo	62
7.2.3 Granularità	63
7.2.4 Dai programmi alle strutture di Kripke	63
7.3 Logiche temporali	64
7.3.1 CTL*	64
7.3.2 Sintassi e semantica	65
7.3.3 Altre logiche temporali	66
7.3.4 Algoritmi per CTL	66
8 Slicing	67
8.1 Intuizione	67
8.2 Motivazioni	67
8.3 Tipi di program slicing	68
8.4 Forme di program slicing	68
8.5 Esempi	69
8.5.1 Standard slicing	69
8.5.2 KL slicing	69
8.5.3 IC slicing	70

A Domande di riepilogo	71
A.1 Semantica e approssimazioni (capitoli 1-4)	71
A.2 Analisi di data-flow (capitolo 5)	73
A.3 Analisi statica non distributiva (capitolo 5)	75
B Esempi di domande d'esame	77
B.1 Domande sull'analisi statica	77
B.2 Domande sull'analisi dinamica	79
Credits	81

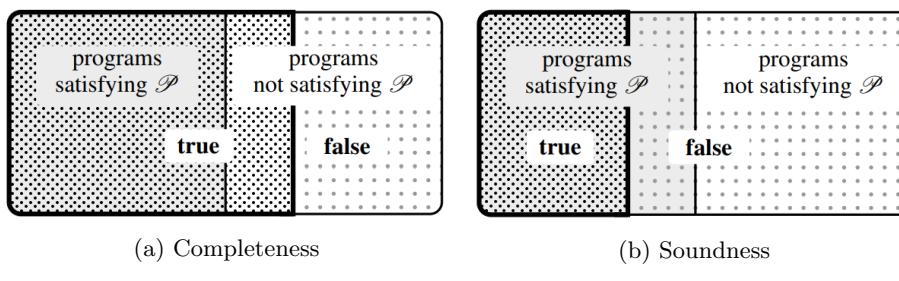
Capitolo 1

Introduzione

1.1 Soddisfazione delle proprietà

Le *strategie* per la stima della soddisfazione di una certa proprietà sono:

- *Completeness* (inaccuratezza ottimistica), in cui si accettano anche alcuni programmi che non possiedono la proprietà (come nel caso del *testing*).
- *Soundness* (inaccuratezza pessimistica), in cui si suppone che il sistema non rispetti le specifiche (come nel caso dell'*analisi statica* dove se non si riesce a dare una risposta allora si suppone che non valga la proprietà).
- *Proprietà semplificate*, in cui si semplifica una proprietà troppo difficile (non verificabile) in una proprietà più semplice che può essere verificata (il problema sta nel fatto che non verifica la proprietà originale).



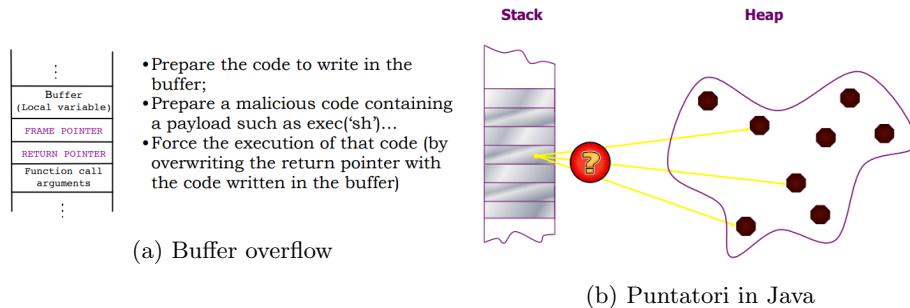
1.2 Esempi

1.2.1 Buffer overflow

Consiste nel far straripare un buffer, dal momento che in C non c'è il controllo sulle dimensioni degli array. Siccome nello stack si scrive verso il basso, se si arriva a sovrascrivere il *return pointer* si può aprire una shell per poi andare ad eseguire codice malevolo.

1.2.2 Pointer analysis

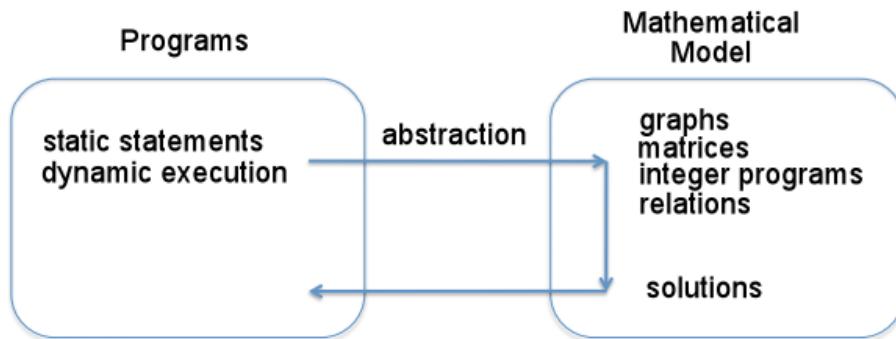
Consiste nel guardare quando i puntatori sono legati alla stessa locazione di memoria. Questo in *Java* può essere difficile dal momento che ogni variabile è considerata come un puntatore nascosto.



1.3 Alcuni ingredienti

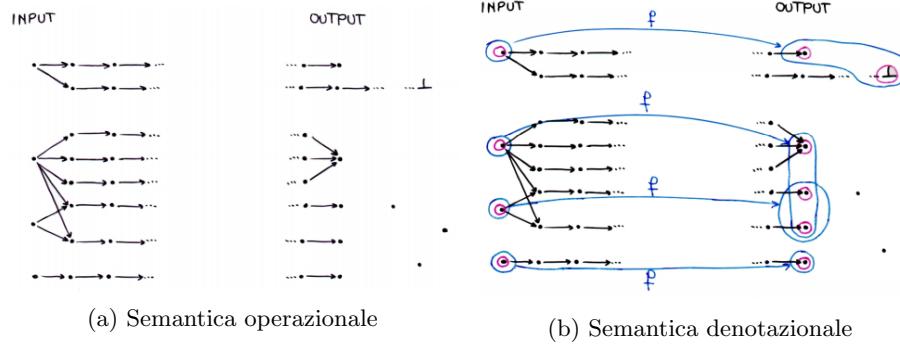
1.3.1 Astrazione matematica

Consiste nello scartare i dettagli e nel creare un'interfaccia per manipolare meglio gli oggetti. Si possono quindi creare degli *oggetti astratti* (ad es. grafi, matrici o relazioni) per cercare poi le *soluzioni* di tali oggetti. Le proprietà verificate nell'astrazione dovranno valere anche nel codice. Ovviamente nel creare il modello matematico e nel definire le proprietà astratte perderò dei dettagli.



1.3.2 Semantica (vista come astrazione)

- *Semantica operazionale*, in cui si descrivono le operazioni come sequenza di sotto-operazioni (operazioni diverse possono confluire nello stesso output).
- *Semantica denotazionale*, in cui si associa con delle sequenze ogni input al relativo output togliendo le sotto-operazioni (ad ogni input/output vengono associate tutte le tracce che hanno quell'input e quell'output).

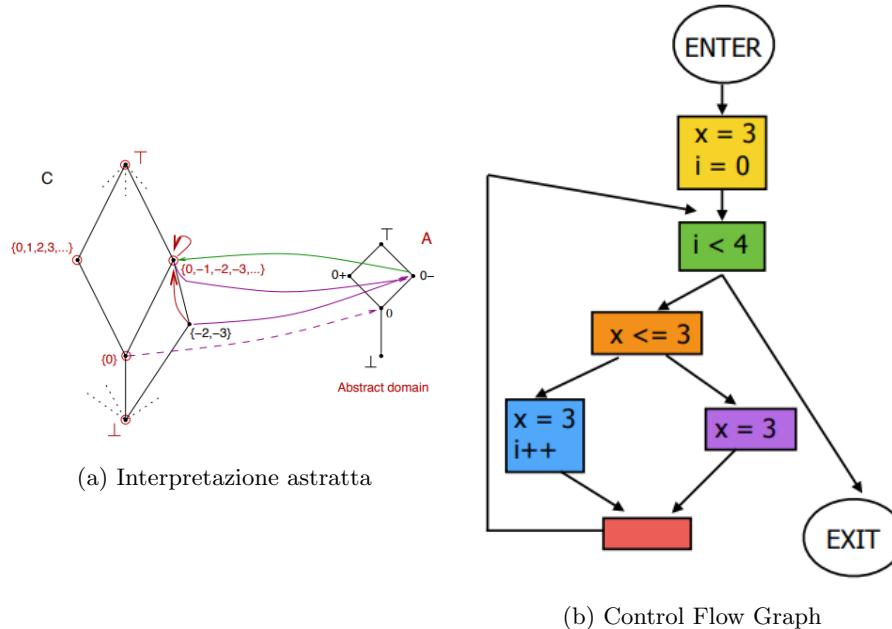


1.3.3 Interpretazione astratta

Quando, ad esempio, voglio verificare che in un programma si manipolino solo numeri interi, la proprietà che deve essere soddisfatta è quella dell'insieme delle parti. Nella figura (a) sottostante viene mostrato come si può costruire un grafo che controlla che il programma abbia solo numeri interi: si può lavorare solo sui segni ma non si è in grado di valutare il dominio (non sembra essere quindi un'astrazione corretta).

1.3.4 Control Flow Graph

Rappresenta il *flusso di controllo* di un programma, separando le informazioni di controllo dalla modifica della memoria. In sintesi, si divide il codice in base alle tipologie di operazioni che si fanno (ad es. branch o join).

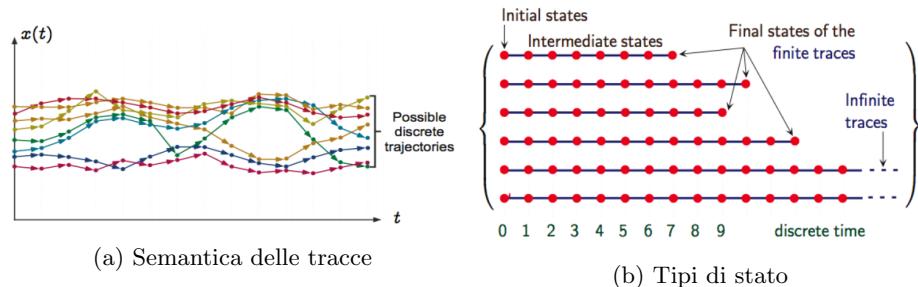


Capitolo 2

Modellare i programmi

2.1 Semantica delle tracce

La *semantica delle tracce*, o semantica delle esecuzioni, è l'insieme di tutte le possibili esecuzioni di un programma. Tali esecuzioni potrebbero essere infinite e generalmente sono deterministiche (per ogni istante di tempo si avrà un solo valore possibile). Inoltre, le tracce sono discretizzate ma è bene ricordare che sono anche potenzialmente infinite (i possibili stati iniziali per la sola variabile intera sono infatti infiniti). Infine, nella *figura (b)* si può notare che la semantica delle tracce è formata da: stati iniziali, stati intermedi e stati finali (nel caso di tracce finite). In ogni caso, potrebbero esserci tracce infinite che non terminano.



2.2 Semantica del punto fisso

La *semantica del punto fisso* è specificata da una coppia $\langle D, f \rangle$ dove:

- D è il *dominio semantico*, ovvero quello che ci interessa osservare della macchina.
- f è la *funzione di mapping*, ovvero una funzione totale, monotona (per preservare l'ordine) e iterabile (per poterla applicare a sé stessa continuamente) anche detta trasformatore semantico.

2.2.1 Sistema di transizione

La semantica operazionale di un linguaggio di programmazione associa ad ogni programma (scritto in quel linguaggio) un *sistema di transizione*, ovvero una coppia $\langle \Sigma, \tau \rangle$ dove:

- Σ è un insieme non vuoto di stati.
- $\tau \subseteq \Sigma \times \Sigma$ è la regola di transizione che permette di capire come passare allo stato successivo.

2.3 Come costruire il punto fisso

Se le tracce sono *finite*, parto dal calcolo di tutte le possibili tracce che terminano:

- Al passo zero non ho ancora nessuna traccia.
- Al primo passo prendo tutte le tracce di lunghezza 1 (stati terminali).
- Al secondo passo prendo gli stati terminali e tutti gli stati che terminano con un passo.
- Al terzo passo prenderò anche tutti gli stati che terminano con due passi, e così via.

Nelle figure sottostanti un pallino rosso rappresenta uno stato terminale, mentre un pallino blu rappresenta uno stato non terminale.

$$\begin{aligned} X^0 &= \emptyset \\ X^1 &= \{\textcircled{red}\} \\ X^2 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}\} \\ X^3 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}\} \\ &\dots \\ X^n &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots, \textcircled{blue} \xrightarrow{\tau} \dots \xrightarrow{\tau} \textcircled{red}\} \\ \text{lfp}_{\emptyset}^{\subseteq F^+} &= \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \dots \xrightarrow{\tau} \textcircled{red} \\ &\dots \\ X^m &= \{\textcircled{blue} \xrightarrow{\tau} \dots \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red} \mid m \geq 0\} \end{aligned}$$

(a) Per minimo punto fisso (lfp)

$$\begin{aligned} X^0 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots\} \\ X^1 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots\} \\ X^2 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots\} \\ X^3 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots\} \\ &\dots \\ X^\omega &= \{\textcircled{blue} \xrightarrow{\tau} \dots \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red} \mid m \geq 0\} \end{aligned}$$

(b) Per massimo punto fisso (gfp)

Invece, se le tracce sono *infinite*, si può costruire per minimo punto fisso scartando le tracce che ad ogni passo non rispettano la regola τ :

$$\begin{aligned} X^0 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots\} \Sigma^\omega \\ X^1 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots\} \\ X^2 &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots\} \\ X^n &= \{\textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red}, \dots\} \\ X^\omega &= \{\textcircled{blue} \xrightarrow{\tau} \dots \xrightarrow{\tau} \textcircled{blue} \xrightarrow{\tau} \textcircled{red} \mid m \geq 0\} \Sigma^\omega \\ \text{gfp}_{\Sigma^\omega}^{\subseteq F^\omega} &= \textcircled{blue} \xrightarrow{\tau} \Sigma^\omega \end{aligned}$$

2.4 Control-Flow-Graph (CFG)

Il *Control Flow Graph* è generato dalla sintassi del programma e permette di capire la struttura del codice. Viene utilizzato per effettuare testing, debugging e per individuare codice morto. Un CFG è un *grafo diretto* $G = \langle N, E \rangle$ dove:

- $n \in N$ sono i nodi che corrispondono ai punti di programma (basic blocks).
- $e = (n_i, n_j) \in E = N \times N$ sono gli archi, ovvero i passi di computazione (passaggi di controllo).

2.4.1 Basic blocks

I *basic blocks* sono una sequenza massimale di comandi aventi un singolo entry point, un singolo exit point e nessun branch interno. Essi si costruiscono individuando i blocchi *leader*:

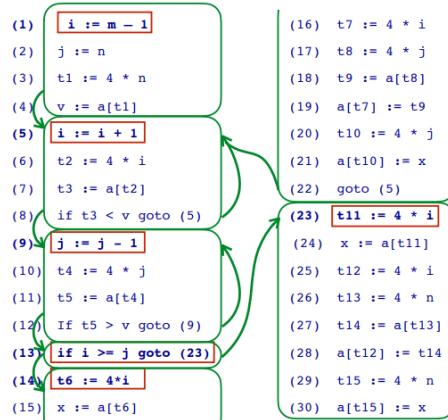
- Il primo statement del programma (entry point) è leader.
- Ogni statement che è target di un punto di branch è leader.
- Ogni statement che segue immediatamente un punto di branch è leader.

Quindi, un basic block è la sequenza di comandi che si trova tra un leader (incluso) e un altro leader (escluso).

2.4.2 Generazione del CFG

Dopo aver diviso il codice in basic blocks, essi verranno collegati dagli archi in corrispondenza di:

- Salti (*goto*) non condizionali.
- *Branch condizionali* (che produrranno archi multipli).
- *Flussi di programma* (quando non ci sono branch alla fine dei blocchi).



2.4.3 Notazione dei CFG

Dato un CFG $G = \langle N, E \rangle$ e un arco $(n_i, n_j) \in E$:

- n_i è detto *predecessore* di n_j .
- n_j è detto *successore* di n_i .

Inoltre, per ogni nodo $n \in N$:

- $Pred(n)$ è l'insieme dei predecessori del nodo n .
- $Succ(n)$ è l'insieme dei successori del nodo n .
- È detto *branch node* un nodo che ha più di un successore.
- È detto *join node* un nodo che ha più di un predecessore.

2.4.4 Linguaggio e semantica di IMP-CFG

In questo linguaggio (leggermente diverso da IMP) gli *archi* vengono etichettati con l'*effetto del comando* e sono della forma $k = (u, lab, v)$ dove u è il nodo sorgente, v è il nodo di destinazione e lab è l'etichetta. Una *computazione* è quindi un percorso (insieme di archi del CFG) che va da un nodo di partenza u a un nodo terminale v :

$$\pi = k_1 \dots k_n, k_i = (u_i, lab_i, u_{i+1}), i = 1 \dots n-1, u = u_1, v = u_n$$

La *trasformazione dello stato* è data invece dalla composizione degli effetti degli archi:

$$[\pi] = [k_n] \circ \dots \circ [k_1]$$

test:	$\text{NonZero}(e)$ or $\text{Zero}(e)$	$[\cdot](m) = m$
assignment:	$x \leftarrow e$	$[\text{NonZero}(e)](m) = m \quad \text{if } [\text{NonZero}(e)](m) = \text{true}$
empty statement:	$;$	$[\text{Zero}(e)](m) = m \quad \text{if } [\text{Zero}(e)](m) = \text{false}$
input:	$\text{input}(x)$	$[x \leftarrow e](m) = m[x \mapsto [\text{NonZero}(e)](m)]$
		$[\text{input}(x)](m) = m[x \mapsto m(x)]$

(a) Linguaggio IMP-CFG

(b) Semantica di IMP-CFG

Capitolo 3

Approssimare il significato dei programmi

3.1 Interpretazione astratta

Data una proprietà Q , voglio renderla decidibile (ma ho una semantica concreta che non è decidibile). L'*idea* di base è quella di trovare un'approssimazione $\llbracket P \rrbracket^\#$ della semantica $\llbracket P \rrbracket$ tale per cui valgano:

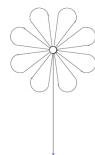
- *Correttezza*: $\llbracket P \rrbracket \subseteq \llbracket P \rrbracket^\#$.
- *Decidibilità*: $\llbracket P \rrbracket^\# \subseteq Q$.

Se entrambe le proprietà sono soddisfatte (la semantica concreta deve essere contenuta in quella astratta la quale deve essere decidibile), allora vale che $\llbracket P \rrbracket^\# \subseteq Q \Rightarrow \llbracket P \rrbracket \subseteq Q$; altrimenti non so dire nulla. La *semantica* sarà specificata da una coppia $\langle D, f \rangle$ dove D è un dominio semantico ordinato e f è una funzione con una soluzione a punto fisso. Quello che dobbiamo studiare sarà quindi:

- L'astrazione degli oggetti e la relazione tra rappresentazione astratta e concreta.
- L'astrazione del calcolo del punto fisso nel caso della rappresentazione astratta.

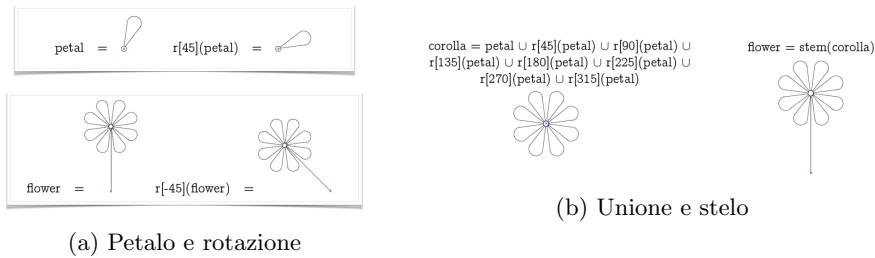
3.2 Un esempio di oggetto: il fiore

Un oggetto può essere rappresentato come una coppia formata da un'*origine* e un *insieme finito di pixel neri* (su sfondo bianco).



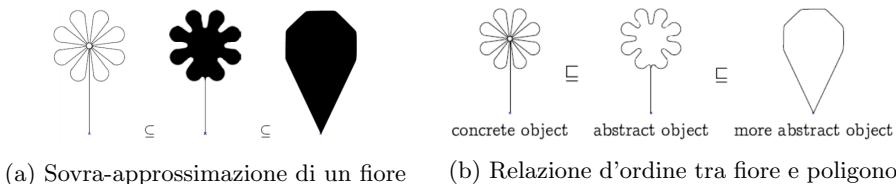
3.2.1 Operazioni

- *Costante petal*: l'oggetto è formato solo dall'origine e dai pixel del petalo.
- *Rotazione $r[a](o)$* : l'oggetto o viene ruotato di a gradi lasciando l'origine fissa.
- *Unione $o_1 \cup o_2$* : giustappone gli oggetti unendo le origini.
- *Funzione stem*: aggiunge uno stelo nell'origine e sposta l'origine alla radice.



3.2.2 Sovra-approssimazione

La *sovra-approssimazione* di un oggetto è un oggetto avente la stessa origine ma un insieme di pixel più ampio. Ad esempio, un fiore lo posso descrivere come un poligono (aggiungo del rumore). Possiamo però notare che c'è una *relazione d'ordine* tra il fiore e il poligono, ovvero che il mondo astratto (poligono) ha meno dettagli ma contiene l'oggetto concreto (fiore).



3.2.3 Oggetti e domini astratti

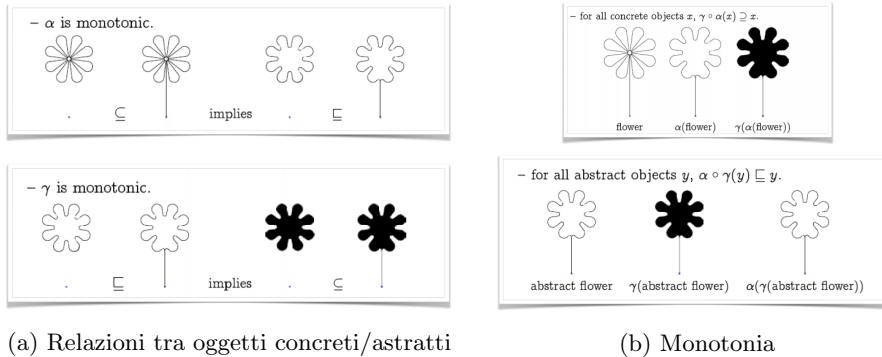
Un *oggetto astratto* è una rappresentazione matematica dell'approssimazione di un oggetto concreto (il concreto è contenuto nell'astratto). Un *dominio astratto*, invece, è l'insieme degli oggetti astratti e delle funzioni astratte che rappresentano le operazioni concrete.

3.2.4 Funzioni di astrazione e concretizzazione

L'*astrazione* avviene per mezzo di una funzione α che mappa ogni oggetto concreto o nella sua rappresentazione astratta $\alpha(o)$. La *concretizzazione*, invece, avviene per mezzo di una funzione γ che mappa ogni oggetto astratto \bar{o} nell'oggetto concreto $\gamma(\bar{o})$ che rappresenta.

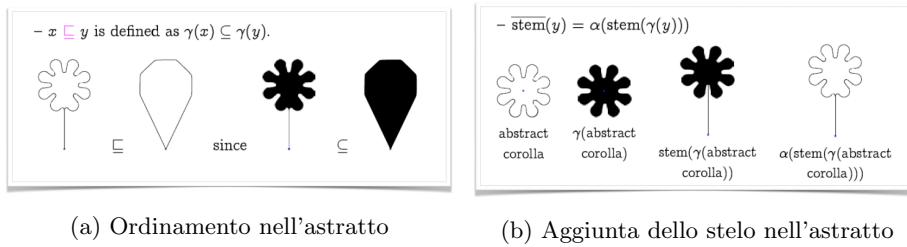
3.2.5 Connessioni di Galois

Le funzioni α e γ sono *monotone*, ovvero se due oggetti concreti (o astratti) sono in relazione tra loro allora lo devono essere anche nella rispettiva rappresentazione astratta (o concreta). Ciò significa che se parto dal concreto, astraggo e poi ri-concretizzo devo ottenere qualcosa di più grande rispetto al concreto (fare $\gamma(\alpha(x))$ cambia il risultato rispetto all'inizio). Viceversa, nell'astrarre dopo una concretizzazione non perdo nulla perché avevo già scartato tutti i dettagli (fare $\alpha(\gamma(y))$ lascia invariato il risultato rispetto all'inizio).



3.2.6 Ordinamento e operazioni nel mondo astratto

Un oggetto astratto è più piccolo (\sqsubseteq) di un altro oggetto astratto se ha meno errore. La rappresentazione astratta sarà quindi più precisa e contenuta (\sqsubseteq) in quella meno precisa. Con l'aggiunta di questo *ordinamento* posso trasferire le *operazioni* nel mondo astratto. Ad esempio, per l'aggiunta dello stelo (funzione *stem*) prendo l'oggetto astratto, lo concretizzo, applico l'operazione concreta e poi astraggo di nuovo. Per le altre operazioni la tecnica è analoga.

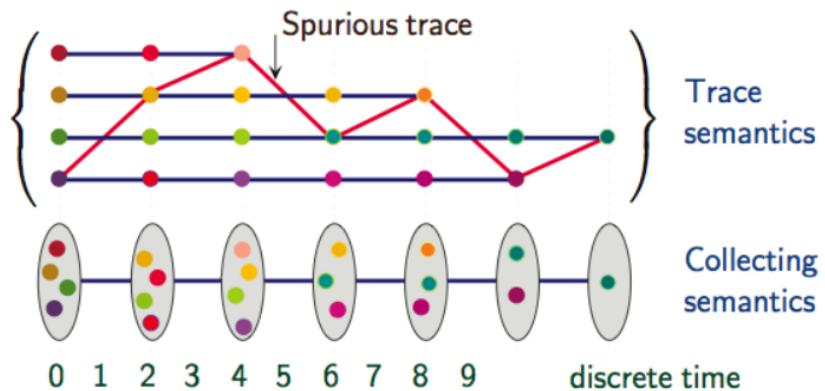


3.3 Astrazione delle semantiche

Finora abbiamo definito la *semantica delle tracce*, nel caso discreto, come l'insieme di tutte le possibili esecuzioni di un programma. Siccome tali esecuzioni sono potenzialmente infinite, quello che ci serve è una semantica ancora più astratta che useremo per approssimare la realtà concreta (non decidibile per definizione).

3.3.1 Collecting semantics

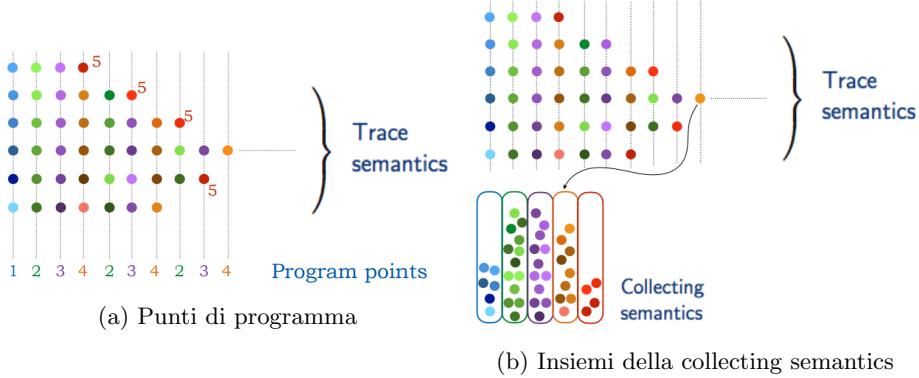
Nella collecting semantics, la semantica delle tracce (insieme di tracce) è vista come una *collezione* di tutte le possibili esecuzioni (traccia di insiemi). L'*idea* alla base della collecting semantics è quella di rimuovere la sequenza temporale e mantenere solo le informazioni date dal punto di programma considerato. In sostanza, si tratta di una collezione degli stati che possono apparire su alcune tracce nei diversi punti di programma. Trattandosi di un'*astrazione*, non è più possibile risalire alle tracce di esecuzione del programma. Ad esempio, nella *figura* sottostante, se voglio controllare l'esistenza della traccia rossa, nella semantica delle tracce so che tale evoluzione temporale non è possibile mentre nella collecting semantics non posso saperlo.



3.3.2 Esempio di funzionamento

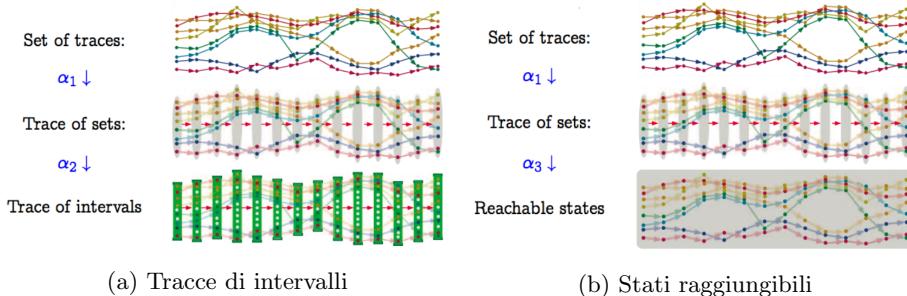
La collecting semantics differisce dalla semantica delle tracce se nel codice sono presenti dei *cicli*, in quanto per iterazioni successive di tali cicli avrà che la collecting semantics porrà i relativi stati nello stesso insieme (si perde l'ordine temporale degli stati). In sostanza si collezionano gli stati in base a *proprietà simili*. Se paradossalmente avessi degli stati che differiscono tutti gli uni dagli altri, allora raggiungerei un'equivalenza con la semantica delle tracce. Vediamo ora un *esempio di funzionamento*:

- Al punto di programma 1 (tempo 0), corrispondente al primo insieme di sinistra, si raggruppano tutti gli stati che compaiono (anche per input differenti).
- Al punto di programma 2 comincia un ciclo, perciò il secondo insieme raggrupperà stati con proprietà uguali ad ogni iterazione.
- Dal punto di programma 4 si torna a collezionare oggetti nell'insieme creato al punto di programma 2.
- Nel quinto insieme si raggruppano gli stati terminali.



3.3.3 Altre notazioni

Nella *notazione ad intervalli* si applica l'idea di osservare per ogni punto di programma quali sono gli stati che vi ci passano con l'obiettivo di osservare una proprietà che si mantiene nel tempo. Nell'osservazione degli *stati raggiungibili*, invece, non si modella più quando e dove uno stato compare, ma solo se esso viene raggiunto dall'assegnamento di una certa variabile che si sta analizzando.



Capitolo 4

Interpretazione astratta

4.1 Introduzione

4.1.1 Astrazione in base ad una proprietà

Come abbiamo già detto, l'*astrazione* è un processo molto comune in informatica e consiste nel rappresentare oggetti concreti con descrizioni in cui si considerano solo alcune *proprietà comuni*. Infatti, se considerassimo tutte le proprietà avremmo indecidibilità poiché saremmo ancora nel piano concreto. Definiremo quindi un insieme $A \subseteq \wp(\Sigma)$ come l'insieme degli elementi che vogliamo descrivere in modo preciso con l'astrazione (senza perdita di precisione). Gli elementi al di fuori di A dovranno però essere rappresentati da altri elementi di A (perdita di precisione).

4.1.2 Oggetti e proprietà dei programmi

Quando si analizza un programma, si devono considerare degli *oggetti* che rappresentano gli stati del nostro programma. Tali oggetti possono essere:

- Valori (booleani, interi) \mathcal{V} .
- Nomi delle variabili \mathbb{X} .
- Ambienti $\mathbb{X} \mapsto \mathcal{V}$.
- Stacks.

Abbiamo parlato di *proprietà* invarianti come di oggetti in A che hanno tali proprietà. Ora dobbiamo passare da oggetti singoli ad un insieme di oggetti. Possiamo anche vedere un insieme di proprietà $\wp(\Sigma)$ di oggetti in Σ (insieme di tutti gli stati) come un *reticolo completo*.

	$\langle \wp(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap, \neg \rangle$
Examples:	
* Odd naturals $\{1, 3, 5, \dots, 2n + 1, \dots\}$	* \subseteq is the logical implication between properties
* Even integers $\{2z \mid z \in \mathbb{Z}\}$	* Σ is true
* Values of integer variables $\{z \in \mathbb{Z} \mid \text{minint} \leq z \leq \text{maxint}\}$	* \emptyset is false
* Invariance property: of a program with states Σ	* \cup is disjunction
$I \in \wp(\Sigma)$	* \cap is conjunction
(a) Esempi di proprietà	* \neg is negation
	(b) Reticolo delle proprietà

4.1.3 Direzione dell'astrazione

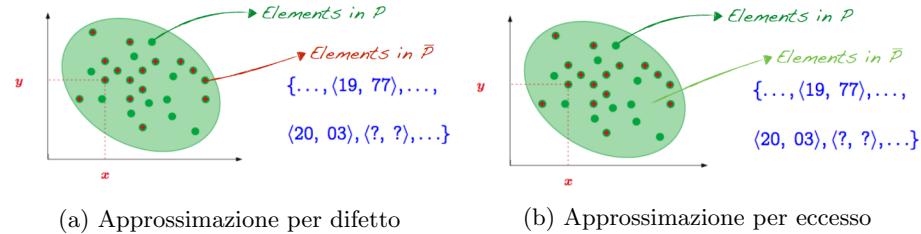
Quando approssimo una proprietà concreta P in una proprietà astratta \bar{P} posso avere due casi: approssimazione per difetto $\bar{P} \subseteq P$ o approssimazione per eccesso $\bar{P} \supseteq P$ (in cui aggiungo del rumore). Nel caso dell'*approssimazione per difetto*, per stimare se un oggetto ha una proprietà P non decidibile devo stimare un sottoinsieme della proprietà originale P :

- Se \bar{P} non vale per un generico punto, allora non so dire nulla.
- Se \bar{P} vale, allora vale anche P .

Nel caso dell'*approssimazione per eccesso*, invece, ho una proprietà \bar{P} che sovrasta P e quindi posso solo dire se un generico punto non ha la proprietà:

- Se \bar{P} non vale, allora sicuramente non vale nemmeno P .
- Se \bar{P} vale, allora non so dire nulla.

Infine ricordiamo che quando si approssima attraverso una rappresentazione astratta si crea un insieme di oggetti *più grande*.



4.1.4 Approssimazione minima

Assumiamo che una proprietà concreta P debba essere approssimata per eccesso da una proprietà \bar{P} tale che $\bar{P} \supseteq P$. L'*approssimazione minima* sarà l'approssimazione ottimale per P . Notiamo infatti che non è detto che esista una sola approssimazione per P . Inoltre, tali proprietà astratte potrebbero anche essere confrontabili, e quindi si dovrà scegliere quella che aggiunge la minor quantità di rumore.

4.1.5 Migliore approssimazione

Una buona scelta per l'astrazione di una proprietà $P \in A$ consiste nella *migliore approssimazione* delle sue sovra-approssimazioni $\overline{P} \supseteq P$.

$$\forall \overline{P}' \in A. (P \subseteq \overline{P}') \Rightarrow (\overline{P} \subseteq \overline{P}')$$

Si può dimostrare che la migliore approssimazione è il *glb* (greatest lower bound) di tutte le approssimazioni, ovvero l'intersezione tra tutte le diverse rappresentazioni astratte.

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

4.1.6 Esempio sulla rappresentazione del segno

Quello che vogliamo rappresentare sono le proprietà delle moltiplicazioni e delle addizioni tra interi. In particolare, la proprietà che dobbiamo studiare è il *segno*. Quindi gli oggetti del dominio astratto saranno:

- Gli interi positivi, zero compreso (notazione +).
- Gli interi negativi, zero compreso (notazione -).
- Il singoletto {0} (notazione 0).

Si può dimostrare che il dominio astratto del segno è *isomorfo* all'insieme delle parti di \mathbb{Z} . Da ciò deriviamo che ogni oggetto può essere approssimato come appartenenza ad un insieme in cui si tiene solo il segno (ma si perde il valore). Nella figura (b) sottostante viene mostrata l'interpretazione dell'addizione e della moltiplicazione nel nuovo dominio astratto (la notazione \mathbb{Z} indica che non so dire nulla).

* Concrete domain: $\wp(\mathbb{Z})$

* Abstract domain: $Sign = \{\mathbb{Z}, +, -, 0, \emptyset\}$

* It is contained in $\wp(\mathbb{Z})$ up to isomorphism

* It is a meet-subsemilattice of $\wp(\mathbb{Z})$ (it is closed under intersection)

A set X in $\wp(\mathbb{Z})$ is approximated by the smallest set of integers which contains X and is represented in $Sign$

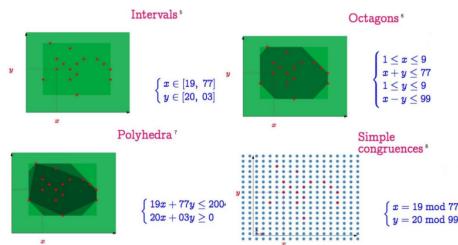
$\{-5, 4\} \mapsto \mathbb{Z}$
 $\{3\} \mapsto + \equiv \mathbb{Z}^+$
 $\{-5\} \mapsto - \equiv \mathbb{Z}^-$
 $\{-5, -6\} \mapsto - \equiv \mathbb{Z}^-$
 \dots

\oplus	\mathbb{Z}	$+$	$-$	0	\emptyset	\otimes	\mathbb{Z}	$+$	$-$	0	\emptyset
\mathbb{Z}	0	\mathbb{Z}									
$+$	\mathbb{Z}	$+$	\mathbb{Z}	$+$	$+$	$+$	\mathbb{Z}	$+$	$-$	0	$+$
$-$	\mathbb{Z}	\mathbb{Z}	$-$	$-$	$-$	$-$	\mathbb{Z}	$-$	$-$	0	$-$
0	\mathbb{Z}	$+$	$-$	0	0	0	\mathbb{Z}	0	0	0	0
\emptyset	\mathbb{Z}	$+$	$-$	0	\emptyset	\emptyset	\mathbb{Z}	$+$	$-$	0	\emptyset

(b) Approssimazione degli operatori

(a) Dominio concreto e astratto

Infine, possiamo avere anche *altri tipi* di rappresentazioni astratte come per esempio: rettangoli (o intervalli), ottagoni, poliedri, studio delle congruenze.



4.2 Connessioni di Galois

Per garantire l'esistenza della migliore approssimazione, occorre descrivere matematicamente la *relazione* tra il mondo concreto e quello astratto. Abbiamo già visto che esistono due funzioni α e γ , dette funzioni di astrazione e concretizzazione, che garantiscono l'esistenza della migliore approssimazione (sugli insiemi ordinati A e C). Possiamo dire che esse formano una *connessione di Galois* se:

- α e γ stesse sono monotone.
- $\forall a \in A, c \in C. \alpha(c) \leq_A a \iff c \leq_C \gamma(a)$.

Una connessione di Galois (GC) si scrive come:

$$(C, \leq_C) \xrightarrow[\alpha]{\gamma} (A, \leq_A)$$

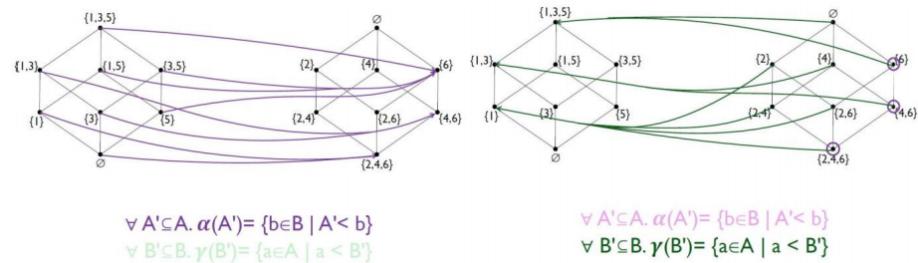
dove la funzione α è detta *aggiunta sinistra*, mentre la funzione γ è detta *aggiunta destra*. Inoltre, la connessione di Galois tra due domini è *univoca* (se due oggetti hanno la stessa α allora hanno anche la stessa γ e viceversa) e quindi le funzioni α e γ sono identificabili attraverso:

$$\alpha(c) = \bigwedge \{a \in A \mid c \leq_C \gamma(a)\}$$

$$\gamma(a) = \bigvee \{c \in C \mid \alpha(c) \leq_A a\}$$

4.2.1 Esempio

Supponiamo di avere due insiemi A e B in relazione tra loro mediante $R \subseteq A \times B$. Indichiamo con A' l'approssimazione di A , cioè l'astrazione che contiene tutti gli elementi di B che sono in relazione con tutti gli elementi di A' . Analogamente, indichiamo con B' l'approssimazione di B , cioè l'astrazione che contiene tutti gli elementi di A che sono in relazione con tutti gli elementi di B' . Nella figura sottostante viene mostrata una rappresentazione grafica in cui si utilizzano gli insiemi $A = \{1, 3, 5\}$ e $B = \{2, 4, 6\}$ e come relazione l'essere *minore di*.



4.3 Inserzioni di Galois

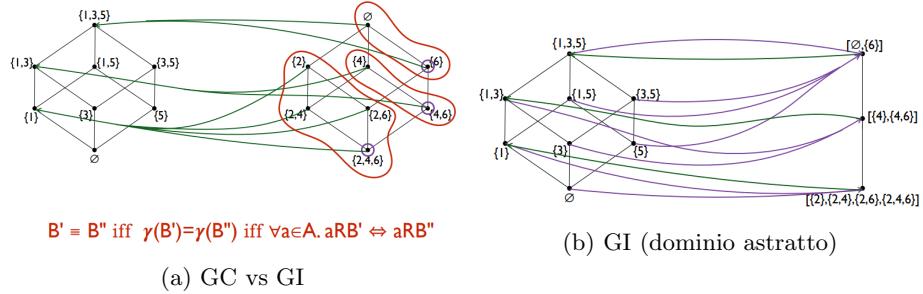
Una connessione di Galois in cui vale che $\alpha\gamma(a) = a$ (identità) si dice *inserzione di Galois*. In questo caso l'astrazione è *suriettiva* per cui ho che più elementi vanno nello stesso valore concreto; e inoltre, la concretizzazione è *iniettiva*

perché un valore concreto è anche la sua concretizzazione. Una inserzione di Galois (GI) si scrive come:

$$(C, \leq_C) \xrightleftharpoons[\alpha]{\gamma} (A, \leq_A)$$

4.3.1 Esempio

Se nella concretizzazione di un valore astratto ho un solo valore concreto rappresentato, posso rimuovere i valori astratti ridondanti (per la suriettività) e unirli insieme in gruppi creando le inserzioni di Galois.



4.4 Operatori di chiusura superiore (uco)

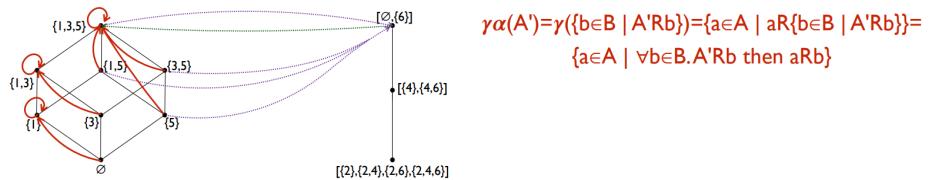
Rappresentano un’ulteriore modalità di rappresentazione astratta. Una funzione $\rho : C \rightarrow C$ è un upper closure operator (*uco*) se soddisfa le seguenti proprietà:

- Estensività, $\forall x \in C. \rho(x) \geq x$
- Monotonia, $\forall x, y \in C. x \leq y \Rightarrow \rho(x) \leq \rho(y)$
- Idempotenza, $\forall x \in C. \rho\rho(x) = \rho(x)$

Nelle connessioni di Galois (o nelle inserzioni di Galois) $\gamma\alpha$ è un uco.

4.4.1 Esempio

Le *GI* associano ad ogni elemento concreto la proprietà astratta rappresentata fuori dal concreto, mentre gli *uco* associano ad ogni elemento concreto la proprietà astratta come suo significato dentro il concreto. Nella *figura* sottostante viene mostrato un esempio di uco (a sinistra) ottenuto partendo da una GI (a destra).



4.5 Famiglie di Moore

Sia L un reticolo completo. $X \subseteq L$ è una *famiglia di Moore* di L se:

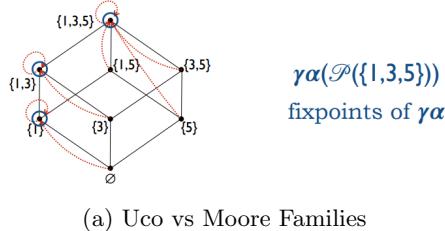
$$X = \mathcal{M}(X) = \left\{ \bigwedge S \mid S \subseteq X \right\}$$

dove $\bigwedge \emptyset = \top \in \mathcal{M}(X)$. Da questa definizione segue che essere una famiglia di Moore garantisce l'esistenza della *migliore approssimazione*.

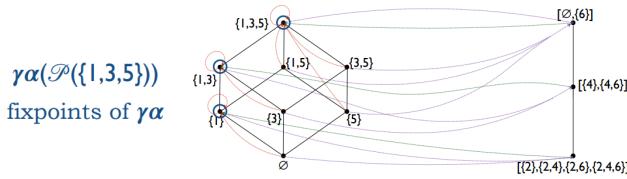
4.5.1 Esempio

È possibile ricavare le seguenti relazioni:

- $C \xrightarrow[\alpha]{\gamma} A$ è una *GI* se A è isomorfo ad una *Moore family* di C (c'è dipendenza dalla rappresentazione degli elementi).
- Se $(C, \leq_C) \xrightarrow[\alpha]{\gamma} (A, \leq_A)$ è una *GI*, allora $\rho = \gamma\alpha$ è un *uco* (c'è indipendenza dalla rappresentazione degli elementi).
- I punti fissi di un *uco* ρ sono una *Moore family* di C (ogni elemento concreto viene astratto nel punto fisso più vicino).



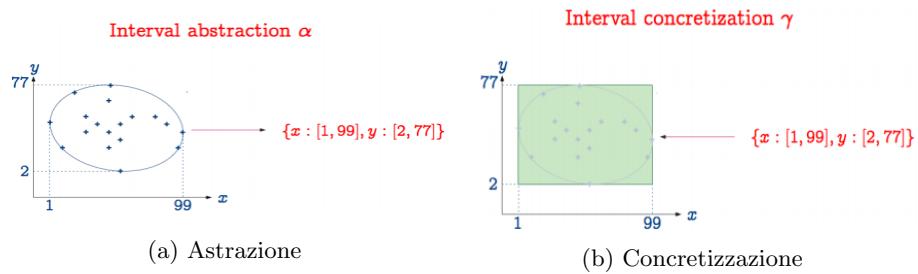
(a) Uco vs Moore Families



(b) Esempio di riepilogo

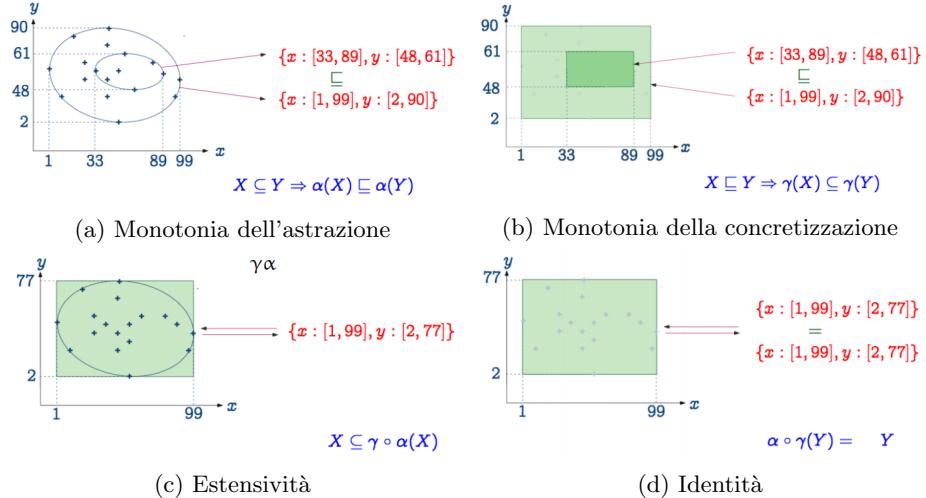
4.6 Esempio grafico

Facciamo ora un esempio di astrazione e concretizzazione di punti sui numeri reali. Nelle figure i + rappresentano i punti concreti. L'*astrazione* crea quindi degli intervalli in cui ho che $x = [1, 99]$ e $y = [2, 77]$. Il *significato concreto* è rappresentato dal rettangolo verde in cui considero tutti i possibili punti posti in quegli intervalli. In pratica dagli intervalli, avendo perso la posizione esatta dei punti, passo ad un'area.



L'astrazione è *monotona*, ovvero gli intervalli più piccoli sono contenuti in quelli più grandi. La stessa cosa vale anche per la concretizzazione.

Inoltre, la composizione $\gamma\alpha$ è *estensiva* (uco), ovvero mi dice che i punti + sono contenuti nel più piccolo rettangolo che li contiene. Se inverto gli operatori ottengo invece l'*identità*, ovvero se applico l'astrazione dopo la concretizzazione $(\alpha\gamma)$ rimango negli intervalli e quindi non ricavo informazione.



4.7 Reticolo delle astrazioni

Se $\langle C, \leq, \wedge, \vee, \perp, \top \rangle$ è un reticolo completo, allora

$$\langle uco(C), \sqsubseteq, \sqcap, \sqcup, \lambda x. \top, \lambda x. x \rangle$$

è un reticolo completo in cui vengono soddisfatte le seguenti proprietà:

- *Confronto* tra domini astratti (grado di precisione) $A_1 \sqsubseteq A_2$

$$\rho \sqsubseteq \eta \Leftrightarrow \forall y \in C. \rho(y) \leq \eta(y) \Leftrightarrow \eta(C) \subseteq \rho(C)$$

- *GLB* di astrazioni $\prod_i A_i$

$$\left(\prod_{i \in I} \rho_i \right) (x) = \bigwedge_{i \in I} \rho_i(x)$$

- *LUB* di astrazioni $\bigsqcup_i A_i$

$$\left(\bigsqcup_{i \in I} \rho_i \right) (x) = x \Leftrightarrow \forall i \in I. \rho_i(x) = x$$

- Astrazione *più astratta* (imprecisa), ovvero il dominio $\{\top\}$

$$\lambda x. \top$$

- Astrazione *più concreta* (precisa), ovvero il dominio identità C

$$\lambda x. x$$

4.8 Computazioni astratte e concrete

Finora abbiamo visto come costruire i domini astratti, ma il nostro obiettivo è quello di spostare la *computazione* da un mondo concreto non decidibile ad un mondo astratto decidibile.

4.8.1 Correttezza (soundness)

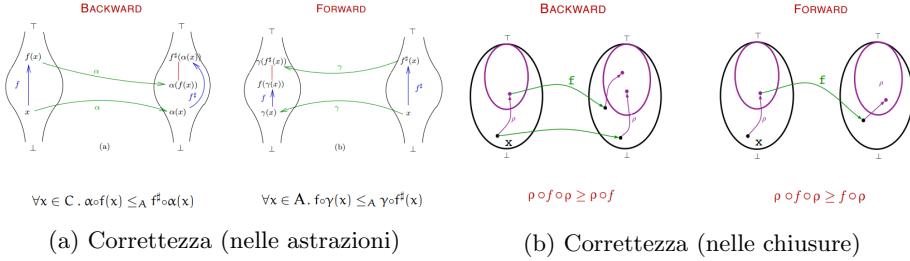
Consideriamo una GI $C \xleftarrow[\alpha]{\gamma} A$, una funzione concreta $f : C \rightarrow C$ e una funzione astratta $f^\sharp : A \rightarrow A$. Possiamo dire che f^\sharp è un'approssimazione corretta (sound) di f in A se:

$$\forall x \in C. \alpha(f(x)) \leq f^\sharp(\alpha(x)) \Leftrightarrow \forall x \in A. f(\gamma(x)) \leq \gamma(f^\sharp(x))$$

Come conseguenza abbiamo che:

$$\forall x \in C. \alpha(f(x)) \leq f^\sharp(\alpha(x)) \Leftrightarrow \forall x \in A. \alpha(f(\gamma(x))) \leq f^\sharp(x)$$

e quindi $\alpha \circ f \circ \gamma : A \rightarrow A$ è la *best correct approximation* (bca) di f in A .



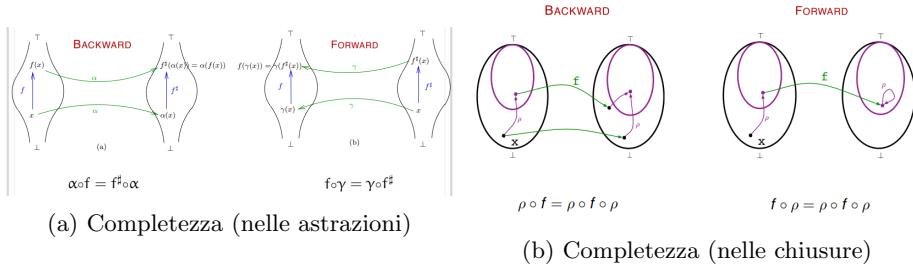
4.8.2 Completezza (precision)

Quando rafforziamo le condizioni di correttezza richiedendo l'uguaglianza otteniamo due nozioni di *completezza* incomparabili. Infatti, se consideriamo una GI $C \xleftarrow[\alpha]{\gamma} A$, una funzione concreta $f : C \rightarrow C$ e una funzione astratta $f^\sharp : A \rightarrow A$, allora potremo dire che:

- f^\sharp è *backward-complete* se $\forall x \in C. \alpha(f(x)) = f^\sharp(\alpha(x))$.
- f^\sharp è *forward-complete* se $\forall x \in A. f(\gamma(x)) = \gamma(f^\sharp(x))$.

Le definizioni di completezza possono essere date anche usando gli *uco*:

- $\rho \in uco(C)$ è *backward-complete* se $\rho \circ f = \rho \circ f \circ \rho$.
- $\rho \in uco(C)$ è *forward-complete* se $f \circ \rho = \rho \circ f \circ \rho$.



4.8.3 Forward vs Backward

I due tipi di completezza rappresentano una situazione ideale in cui non si verifica alcuna perdita di precisione durante l'astrazione. In particolare:

- La *Backward-completezza* considera l'astrazione sull'output delle operazioni (non si accumula alcuna perdita di precisione astraendo in ρ gli argomenti di f).
- La *Forward-completezza* considera l'astrazione sull'input delle operazioni (non si accumula alcuna perdita di precisione approssimando il risultato della funzione f calcolata in ρ).

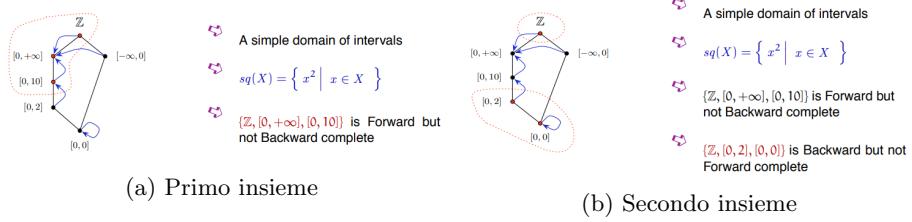
4.8.4 Esempio su intervalli di interi

Consideriamo un dominio composto da certi *intervalli di interi* (con distinzione tra positivi e negativi). I pallini rossi sono gli intervalli che formano la *rappresentazione astratta* degli intervalli concreti. La funzione f che dovremo calcolare è il quadrato. La *proprietà astratta* sarà quindi rappresentata da tutti i valori positivi, poiché i valori negativi al quadrato confluiscono in quelli positivi. Il *primo insieme* di pallini rossi considerato $\{\mathbb{Z}, [0, +\infty], [0, 10]\}$ è Forward ma non Backward completo, in particolare:

- È *forward-complete* perché per ogni elemento vale che $f \circ \rho = \rho \circ f \circ \rho$, ovvero se seguo f in avanti cado sempre dentro l'insieme.
- Non è *backward-complete* perché $\rho \circ f([0, 2]) \neq \rho \circ f \circ \rho([0, 2])$, ovvero esiste almeno un elemento che entra dentro ma che si trova fuori dall'insieme (se seguo f all'indietro cado fuori dall'insieme).

Il *secondo insieme* di pallini rossi considerato $\{\mathbb{Z}, [0, 2], [0, 0]\}$ è Backward ma non Forward completo, in particolare:

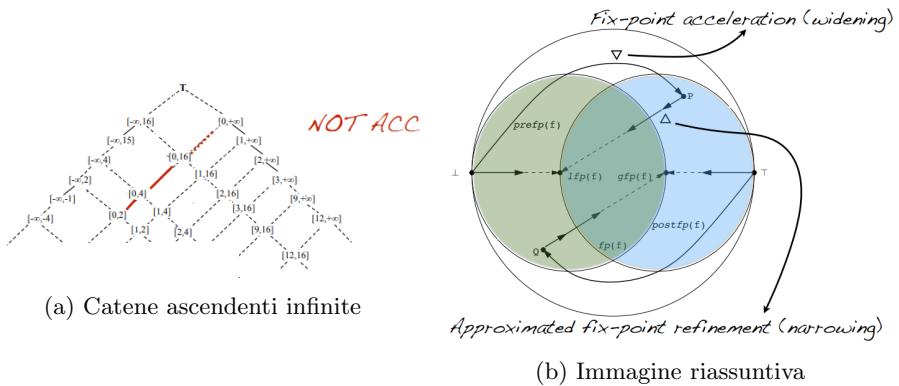
- È *backward-complete* perché se seguo f all'indietro dai pallini rossi non esco mai dall'insieme (in questo insieme solo $[0, 0]$ può seguire f all'indietro e infatti rimane dentro).
- Non è *forward-complete* perché esiste almeno un elemento tale per cui se seguo f in avanti da esso esco dall'insieme.



4.9 Estrapolazione del punto fisso

In generale, è impossibile calcolare la *semantica del punto fisso* mediante astrazione $\alpha(\text{lfp } F)$ senza calcolare $\text{lfp } F$. La cosa più semplice da fare sarebbe definire \overline{F} , sul dominio astratto, tale che $\alpha(\text{lfp } F) = \text{lfp } \overline{F}$. Sfortunatamente però, \overline{F} non è in genere stimabile nel concreto. Inoltre, anche se siamo nel dominio astratto, non è garantita la *convergenza* in quanto l'astrazione cerca di risolvere il problema della decidibilità ma non è detto che risolva quello della convergenza. Nella figura (a) sottostante notiamo infatti la presenza di catene ascendenti divergenti a valori infiniti. Ciò significa che non esiste un punto fisso (non è calcolabile). Analizziamo infine la figura (b):

- L'insieme di sinistra (in verde) corrisponde ai *punti pre-fissi*.
- L'insieme di destra (in azzurro) corrisponde ai *punti post-fissi*.
- Con il *widening* si accelera il calcolo del punto fisso spostandosi nel calcolo dei punti post-fissi.
- Con il *narrowing* si cerca la migliore approssimazione del punto fisso raffinando i punti post-fissi.



Capitolo 5

Analisi statica

5.1 Introduzione

L'*analisi statica* studia le proprietà del programma (senza eseguirlo). Tali proprietà devono però essere astratte, per poter far rispondere l'analizzatore in modo decidibile. L'obiettivo sarà quindi quello di dire se una certa *semantica* soddisfa o meno una proprietà astratta. L'analisi statica può essere utile per cercare di prevedere il comportamento di un programma a livello di esecuzione o per ottimizzare il programma stesso (riduzione della lunghezza, eliminazione di parti non raggiungibili).

5.1.1 Analisi sul CFG

L'analisi statica si basa sull'analisi della semantica. Un possibile approccio è quello di analizzare il *flusso di controllo* servendosi del CFG. Per l'*analisi sul CFG* viene quindi generato un grafo per ogni procedura. Poi, le analisi che vengono eseguite sono localizzate su tre livelli:

1. *Locali* al blocco, cioè eseguite all'interno di uno stesso basic block.
2. *Intra-procedurali*, che considerano il flusso di informazioni in un singolo CFG.
3. *Inter-procedurali*, che considerano il flusso di informazioni in un programma visto come un insieme di vari CFG.

Nell'analisi sul CFG si studia come l'informazione viene modificata all'interno di un blocco di istruzioni. L'informazione è caratterizzata dalla soluzione dell'*equazione di punto fisso* che viene definita per ogni blocco. In alcuni casi questa equazione si ottiene dai seguenti tre passaggi:

1. Caratterizzare l'*informazione entrante* in un blocco come la combinazione di tutto ciò che esce dai blocchi precedenti.
2. Caratterizzare l'*informazione uscente* dal blocco come l'informazione entrante modificata dalle operazioni eseguite nel blocco.
3. *Combinazione* delle definizioni precedenti nell'equazione di punto fisso.

5.2 Available Expressions

Potremmo essere interessati a controllare se un'espressione di un assegnamento non cambia nel tempo. In questo modo si potrebbe evitare di *ricalcolare* l'espressione successivamente, ovvero quando essa viene assegnata ad un'altra variabile. Dovremo quindi verificare se l'espressione e è disponibile (*available*) nella variabile x al punto di programma p . Per fare questo bisogna controllare che:

- e sia stata valutata in un punto di programma precedente a p .
- Il risultato di e sia stato assegnato a x .
- Sia x che le variabili in e non devono essere state modificate tra la valutazione e p .

```

z   <- 1;
y   <- M[5];
A:   x1 <- y+z;
      ...
B:   x2 <- y+z;

```

Consider the assignment $x \leftarrow e$ (x not in e)

Let $\pi = k_1 \dots k_n$ be a path from the entry point to the program point v . The expression e is *available* in x at v if:

- The path π contains an edge k_i , labeled with an assignment $x \leftarrow e$.
- No edge k_{i+1}, \dots, k_n is labeled with an assignment to one of the variables in $\text{Var}(e) \cup \{x\}$.

(a) Ricalcolo di un'espressione

(b) Available expression su un cammino

5.2.1 Costruire le equazioni

Le *equazioni* da costruire sono:

$$\begin{aligned} \text{AvailOut}(n) &= \text{Gen}(n) \cup (\text{AvailIn}(n) \setminus \text{Kill}(n)) \\ \text{AvailIn}(n) &= \begin{cases} \emptyset & \text{se } n = \text{entry} \\ \bigcap_{p \in \text{pred}(n)} \text{AvailOut}(p) & \text{altrimenti} \end{cases} \\ \text{Gen}(p) &= \{x \leftarrow e \in p \mid x \notin \text{Var}(e)\} \\ \text{Kill}(p) &= \{x \leftarrow e \mid \exists y \leftarrow e' \in p. y \in \text{Var}(e) \cup \{x\}\} \end{aligned}$$

Manipolandole possiamo ottenere la seguente *equazione di punto fisso*:

$$\text{AvailIn}(n) = \bigcap_{p \in \text{pred}(n)} \text{Gen}(p) \cup (\text{AvailIn}(p) \setminus \text{Kill}(p))$$

AvailIn rappresenta le espressioni disponibili all'ingresso del blocco e viene calcolato come intersezione sui predecessori di quello che generano i predecessori unito a quello che viene tramandato invariato tra predecessori meno l'informazione ridotta man mano dai vari blocchi.

5.2.2 Risolvere il problema

Per calcolare AvailIn l'analizzatore usa un algoritmo formato da tre passaggi:

1. Costruire il CFG.
2. Raccogliere l'informazione iniziale (*Gen* e *Kill*).
3. Risolvere l'equazione di punto fisso.

```
// assume block b has k operations of form x <- y op z
for each block b Init(b)

Init(b)
  DEExpr(b) <- ∅
  ExprKill(b) <- ∅
  for i <- 1 to k
    if y ∉ ExprKill(b) and z ∉ ExprKill(b)
      then add x <- (y op z) to DEExpr(b)
      add x to ExprKill(b)

// assume CFG has N blocks numbered 0 to N-1
for i <- 1 to N-1
  AvailIn(i) <- {AllExpr}
  AvailIn(0) <- ∅
  changed <- true
while (changed)
  changed <- false
  for i <- 0 to N-1
    recompute AvailIn(i)
    if AvailIn(i) changed then
      changed <- true
```

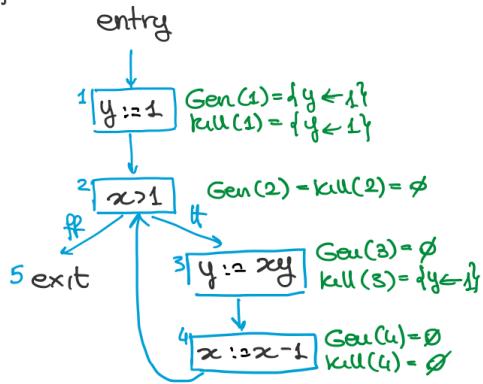
(a) Raccolta dell'informazione iniziale

(b) Risoluzione dell'equazione

5.2.3 Esempio

$y := 1;$
 $\text{while } (x > 1) \text{ do}$
 $\quad \{ y := xy; \}$
 $\quad x := x - 1; \}$

	0	1	2	3
1	∅	∅	∅	∅
2	Ass	$y \leftarrow 1$	∅	
3	Ass	$y \leftarrow 1$	∅	
4	Ass	∅	∅	
5	Ass	$y \leftarrow 1$	∅	



5.3 Framework monotono

Dobbiamo formalizzare l'*informazione astratta* che vogliamo analizzare. Inoltre, dobbiamo anche definire la funzione di trasferimento astratta (*abstract edge effect*) che rappresenta la semantica operazionale astratta. Dobbiamo poi costruire un *sistema di equazioni* la cui soluzione ci darà l'informazione astratta per ogni punto di programma.

5.3.1 Abstract edge effect

Definiamo la semantica astratta nel caso dell'analisi di *available expressions* per ogni etichetta del CFG:

$$\begin{aligned} \llbracket ; \rrbracket^{\#} A &= A \\ \llbracket \text{NonZero}(e) \rrbracket^{\#} A &= \llbracket \text{Zero}(e) \rrbracket^{\#} A = A \\ \llbracket \text{input}(x) \rrbracket^{\#} A &= A \\ \llbracket x \leftarrow e \rrbracket^{\#} A &= \begin{cases} (A \setminus \text{Occ}(x)) \cup \{x \leftarrow e\} & \text{se } x \notin \text{Var}(e) \\ A \setminus \text{Occ}(x) & \text{altrimenti} \end{cases} \end{aligned}$$

dove $\text{Occ}(x) = \{y \leftarrow e \mid x \in \text{Var}(e) \cup \{y\}\}$.

5.3.2 Fonti di imprecisione

Un assegnamento è *disponibile definitivamente* al punto di programma v se è disponibile su tutti i cammini che portano dall'entry a v . In sostanza, si intersecano tutte le semantiche dei cammini che arrivano a v partendo dall'entry. La monotonia della funzione di trasferimento garantisce la *terminazione* e quindi il raggiungimento del punto fisso (soundness). Abbiamo però un'*imprecisione* che nasce dal fatto che approssimiamo il programma usando il CFG, il quale rende possibili dei cammini che semanticamente non lo sarebbero. Inoltre, siccome si guarda alla sintassi e non al valore interno, se ho due assegnamenti allo stesso valore (ma con espressioni diverse), non riuscirò a catturare tutte le espressioni disponibili.

The set of the assignment *definitively available* at the program point v is
 $\mathcal{A}^*[v] \triangleq \bigcap \{ [\pi]^k(\varnothing) \mid \pi : start \rightarrow^* v \}$
 $\mathcal{A}^*[v]$ are called *merge-over-all-paths* (MOP) solution of the analysis problem.

Assume the assignment $x <- y+z$ is available at u , and there exists an edge $k=(u,y <- e,v)$. Assume the value of e is always the one that has y at u . In this case the transformation replacing $y+z$ by x would still be correct although $x <- y+z$ would no longer be recognized as available at v .

(a) Disponibile definitivamente

(b) Imprecisione

5.3.3 Esempio

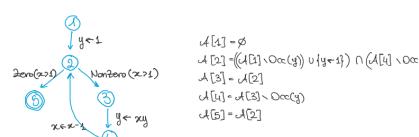
Il *calcolo della soluzione* consiste nel costruire un sistema di disequazioni (una per ogni punto di programma):

$$\begin{aligned}\mathcal{A}[entry] &\subseteq \emptyset \\ \mathcal{A}[v] &\subseteq [k]^\#(\mathcal{A}[u])\end{aligned}$$

dove $k = (u, lab, v)$. \mathcal{A} rappresenta quindi l'informazione disponibile localmente:

- All'entry non ho nulla.
- Al punto v ho quello che avevo prima più quello che sopravvive dopo l'esecuzione dell'arco k .

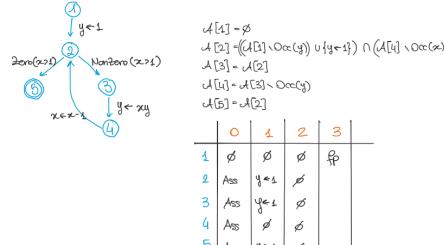
Per risolvere il sistema di disequazioni si può usare l'algoritmo *Naive* o l'algoritmo *Round-robin*. Nel caso del Round-robin la computazione di un valore utilizza tutti i valori calcolati precedentemente, e quindi anche quelli dell'iterazione corrente, poiché viene mantenuta una struttura dati a cui si accede sia in lettura che in scrittura.



* The only assignment whose left-side does not occur in the right-side is $y <- 1$.

* Solution: $\boxed{\mathcal{A}[1] = \mathcal{A}[2] = \mathcal{A}[3] = \mathcal{A}[4] = \mathcal{A}[5] = \emptyset}$

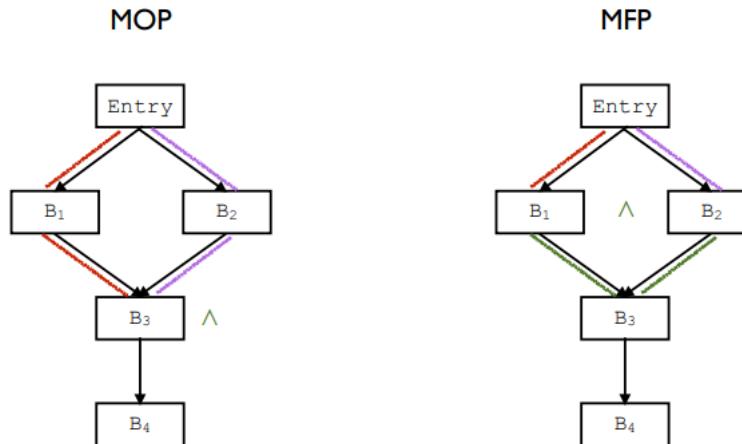
(a) Algoritmo Naive



(b) Algoritmo Round-robin

5.3.4 MOP vs MFP

La *MFP* (Maximum Fixed Point) è la soluzione che combina i valori dell'analisi quando il CFG ha dei nodi in cui convergono due o più percorsi. La soluzione MFP *approssima* la soluzione MOP. Infatti, la *MOP* (Merge Over all Paths) è una soluzione più precisa rispetto alla MFP ($MOP \supseteq MFP$) poiché combina i valori dell'analisi di *tutti i percorsi* del CFG (dopo averli attraversati tutti). La soluzione MOP è quella che vorremmo ma, in generale, essa *non è computabile* in quanto ci possono essere un numero infinito di percorsi. Le soluzioni MOP e MFP coincidono quando tutte le funzioni di trasferimento (abstract edge effect) sono *distributive* ($MOP = MFP$). Dunque se le funzioni di trasferimento sono distributive, è possibile calcolare la soluzione MOP attraverso l'algoritmo iterativo del *punto fisso*.



$$MOP[B4] = ((f_{B3} \circ f_{B1}) \wedge (f_{B3} \circ f_{B2}))(\nu_{Entry}) \leq IN[B4] = f_{B3}(f_{B1}(\nu_{Entry}) \wedge f_{B2}(\nu_{Entry}))$$

5.3.5 Problemi distributivi e non distributivi

I *problemi distributivi* sono i cosiddetti problemi "semplici", ovvero tutte quelle proprietà che ci dicono *come* un programma viene eseguito. Sono distributive le analisi di:

- *Available expressions.*
- *Live variables.*
- *Reaching definitions.*
- *Very busy expressions.*

I *problemi non distributivi*, invece, sono quelli che ci dicono *cosa* calcola un programma (ad esempio che l'output è una costante, è un valore positivo, appartiene ad un intervallo eccetera). È non distributiva l'analisi di *constant propagation*.

5.3.6 Sommario

La *funzione di trasferimento* trasforma ad ogni passo di computazione l'informazione raccolta fino a quel momento. Essa ha due proprietà:

1. Include la funzione identità $\forall x \in V. I(x) = x$.
2. È chiusa rispetto alla composizione.

Per garantire che un algoritmo iterativo abbia una soluzione, bisogna avere funzioni di trasferimento monotone (*framework monotono*). Tale condizione è formalizzata come:

$$\forall x, y \in V. \forall f \in F. x \leq y \Rightarrow f(x) \leq f(y)$$

$$\forall x, y \in V. \forall f \in F. f(x \wedge y) \leq f(x) \wedge f(y)$$

Tuttavia, alcune volte un framework soddisfa condizioni più restrittive. Infatti, un framework è detto *distributivo* se:

$$\forall x, y \in V. \forall f \in F. f(x \wedge y) = f(x) \wedge f(y)$$

L'*algoritmo iterativo* per il calcolo di un'analisi di data-flow è il seguente:

- *Input*, un data-flow framework composto da:
 1. Un *CFG* (con entry ed exit point).
 2. Un *insieme di valori* V (dominio).
 3. Un'*operazione di combinazione* \wedge (possible/definite).
 4. Un *insieme di funzioni di trasferimento* F tale che $f_B \in F$.
 5. Un *valore costante* di ingresso per entry o exit (forward/backward).
- *Output*, valori in V per $IN[B]$ e $OUT[B]$ per ogni blocco B del CFG (cioè le informazioni in ingresso a B e in uscita da B).

```

IN[Entry] = vEntry;
for (each basic block B != Entry) IN[B] = True;
while (some IN changes)
  for (each basic block B != Entry) {
    IN[B] = ∩_{P ∈ pred(B)} OUT[P];
    OUT[B] = f_B(IN[B]);
  }
  OUT[Exit] = vExit;
  for (each basic block B != Exit) OUT[B] = True;
  while (some OUT changes)
    for (each basic block B != Exit) {
      OUT[B] = ∩_{S ∈ succ(B)} IN[S];
      IN[B] = f_B(OUT[B]);
    }
  }
}

```

(a) Forward data-flow algorithm

(b) Backward data-flow algorithm

Le *proprietà* dell'algoritmo sono:

- Se l'algoritmo converge, il risultato è una possibile soluzione al sistema di equazioni.
- Se il framework è monotono, il risultato è la soluzione MFP del sistema di equazioni (ogni altra soluzione del problema sarà sicuramente meno precisa).
- Se il framework monotono è un reticolo ACC (ad altezza finita), la convergenza è garantita.

5.3.7 Soluzione IDEAL

La *soluzione IDEAL* è la soluzione migliore, ma non è computabile. Infatti a differenza della MOP, la IDEAL prende in considerazione solamente i *percorsi possibili*, ossia quelli che verranno sicuramente attraversati da qualche esecuzione. La soluzione IDEAL calcola quindi i valori alla fine di ogni possibile percorso di esecuzione e ne fa la combinazione. Il problema della non computabilità sta nel fatto che non è possibile conoscere i cammini eseguibili in maniera statica. Qualsiasi soluzione più *piccola* della IDEAL sarà quindi *meno precisa* (safe), anche se convergente, in quanto avrà considerato un numero di vincoli maggiore di quello necessario. Qualsiasi soluzione più *grande* della IDEAL sarà invece *scorretta* (unsound).

5.3.8 MOP vs MFP vs IDEAL

La soluzione MOP prende in considerazione tutti i cammini che dall'entry vanno in un determinato blocco B . La *perdita di precisione* sta nel fatto che possono essere considerati anche cammini che non verranno mai eseguiti. Nella MOP avrà quindi i cammini della IDEAL più alcuni cammini spuri ($MOP \sqsubseteq IDEAL$). Combinando queste informazioni con quanto già osservato nel confronto tra MOP e MFP si ottiene la seguente relazione:

$$MFP \sqsubseteq MOP \sqsubseteq IDEAL$$

5.4 Analisi di data-flow

Le *analisi di data-flow* sono analisi che parlano di *come* il programma calcola i valori, ovvero di come i dati evolvono attraverso gli elementi del programma a tempo di esecuzione.

5.5 Analisi delle variabili live

L'analisi delle variabili live è essenziale per l'*allocazione dei registri*. Infatti, due variabili diverse non possono essere salvate nel medesimo registro se sono entrambe in uso. L'*obiettivo* dell'analisi sarà quindi quello di individuare gli intervalli di liveness delle variabili per dire, ad esempio, se nel programma ci sono delle definizioni a vuoto che possono essere rimosse.

5.5.1 Variabile live

Una variabile x è *live* all'uscita del blocco C se verrà usata successivamente. Una variabile x è invece *non-live* (o *dead*) dopo il blocco C se viene ridefinita prima di un successivo utilizzo. Una variabile x è *live in un cammino* π ($v \rightarrow exit$) se:

- π non contiene definizioni di x .
- Esiste almeno un uso di x in π senza precedenti ridefinizioni di x .

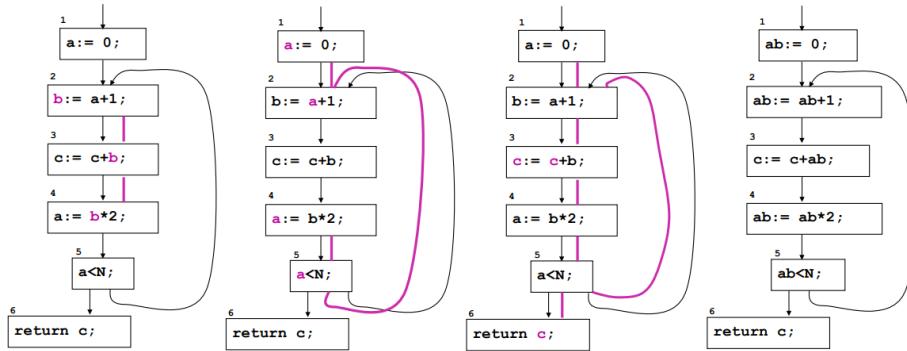
Possiamo quindi dire che una variabile x è *live* se si trova tra una definizione ed un uso. Infatti, se decomponiamo il cammino π in $\pi = \pi_1 k \pi_2$ abbiamo che: k contiene un uso di x ; e π_1 non contiene definizioni di x .

5.5.2 Esempio su un programma

Per dire se una variabile è live oppure no bisogna guardare tutto il *programma*. Ad esempio, nel nodo di uscita non ci saranno mai variabili live in quanto esse non verranno più utilizzate. Nelle *figure* sottostanti è mostrato un esempio in cui:

- La variabile b è live negli archi $2 \rightarrow 3$ e $3 \rightarrow 4$.
- La variabile a è live negli archi $4 \rightarrow 5$, $5 \rightarrow 2$ e $1 \rightarrow 2$.
- La variabile c è live in tutti gli archi.

Saranno quindi sufficienti due registri per memorizzare tutte le variabili, in quanto a e b possono essere considerate come un'unica variabile ab .



5.5.3 Notazione di base

- *Definizione*: un assegnamento ad una variabile definisce quella variabile.
- *Uso*: ogni occorrenza che accede al valore di una variabile è un uso di quella variabile.
- $Kill(n)$: è l'insieme delle variabili definite nel nodo n .
- $Gen(n)$: è l'insieme delle variabili usate nel nodo n .

Lab	Used	Defined
$;$	\emptyset	\emptyset
$\text{NonZero}(e)$	$\text{Vars}(e)$	\emptyset
$\text{Zero}(e)$	$\text{Vars}(e)$	\emptyset
$x \leftarrow e$	$\text{Vars}(e)$	$\{x\}$
$\text{input}(x)$	$\{x\}$	\emptyset

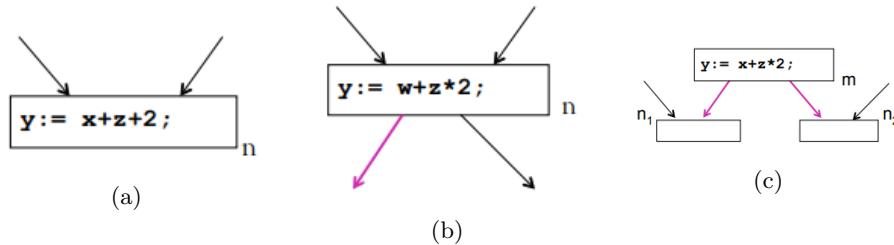
5.5.4 Liveness su archi e su blocchi

Una variabile x è *live* su un arco $e \rightarrow f$ se esiste un cammino π ($e \rightarrow m$) tale che:

- $e \rightarrow f$ è il primo arco di π .
- $x \in Gen(m)$.
- $\forall n \in \pi \wedge n \neq e \wedge n \neq m. x \notin Kill(n)$.

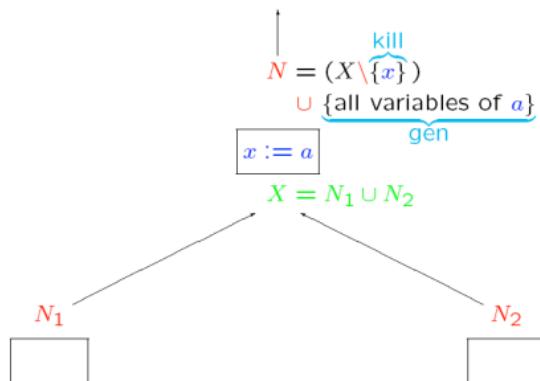
La stessa informazione può essere calcolata anche sui *blocchi*:

- (a) Se x è usata in n , allora x è *live-in* in n (cioè live in tutti gli archi in ingresso).
- (b) Se x è *live-out* in n (cioè live su un arco che esce da n) e se x non è definita in n , allora x è *live-in* in n .
- (c) Se x è *live-in* in almeno un arco uscente da m , allora x è *live-out* in m (si propaga per unione).



5.5.5 Proprietà dell'analisi

L'analisi dipende dalla *continuazione* (futuro della computazione) perché se trovo una definizione non posso sapere se esiste un uso futuro (e quindi se la variabile è live). Questo ci dice che l'analisi è di tipo *backward*. Inoltre, il fatto che l'informazione si propaga per unione ci dice che l'analisi è di tipo *possible*.



5.5.6 Costruire le equazioni

Le *equazioni* da costruire sono:

$$LiveOut(n) = \begin{cases} \emptyset & \text{se } n = \text{exit} \\ \bigcup_{p \in succ(n)} LiveIn(p) & \text{altrimenti} \end{cases}$$

$$LiveIn(n) = Gen(n) \cup (LiveOut(n) \setminus Kill(n))$$

$$Gen(p) = \{x \mid \exists e \in p. x \in Var(e)\}$$

$$Kill(p) = \{x \mid x \leftarrow e \in p\}$$

Manipolandole possiamo ottenere la seguente *equazione di punto fisso*:

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

LiveIn(n) sarà quindi l'insieme delle variabili live in ingresso ad un nodo, mentre *LiveOut(n)* sarà l'insieme delle variabili live in uscita da un nodo.

5.5.7 Risolvere il problema

Per calcolare *LiveOut* l'analizzatore usa un algoritmo formato da tre passaggi:

1. Costruire il CFG.
2. Raccogliere le informazioni iniziali (*Gen* e *Kill*) per ogni blocco.
3. Risolvere l'equazione di punto fisso.

```
// assume block b has k operations of form x <- y op z
for each block b Init(b)

Init(b)
  Use(b) <- ∅
  VarKill(b) <- ∅
  for i <- 1 to k
    if y ∉ VarKill(b)
      then add y to Use(b)
    if z ∉ VarKill(b)
      then add z to Use(b)
    add x to VarKill(b)
```

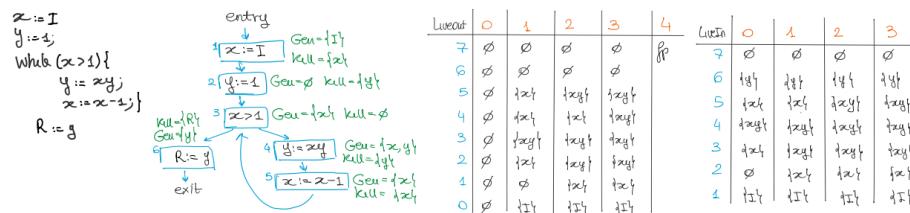
```
// assume CFG has N blocks numbered 0 to N-1
for i <- 0 to N-1
  LiveOut(i) <- ∅

changed <- true
while (changed)
  changed <- false
  for i <- 0 to N-1
    recompute LiveOut(i)
    if LiveOut(i) changed then
      changed <- true
```

(a) Raccolta dell'informazione iniziale

(b) Risoluzione dell'equazione

5.5.8 Esempio



5.5.9 Fonti di imprecisione

- Se x è accessibile attraverso altri nomi, l'analisi di liveness fallisce.
- L'analisi di liveness potrebbe trovare un cammino che porta ad un uso di una variabile su cui però la variabile non è definita (anche se tale cammino non verrà mai percorso a tempo di esecuzione).

5.5.10 Abstract edge effect

Dato un arco $k = (u, \text{lab}, v)$ del CFG, dobbiamo costruire l'informazione su u utilizzando l'informazione ricevuta su v (backward). Definiamo quindi la semantica astratta nel caso dell'analisi di *liveness* per ogni etichetta *lab*:

$$\begin{aligned} \llbracket ; \rrbracket^{\#} L &= L \\ \llbracket \text{NonZero}(e) \rrbracket^{\#} L &= \llbracket \text{Zero}(e) \rrbracket^{\#} L = L \cup \text{Var}(e) \\ \llbracket \text{input}(x) \rrbracket^{\#} L &= L \setminus \{x\} \\ \llbracket x \leftarrow e \rrbracket^{\#} L &= (L \setminus \{x\}) \cup \text{Var}(e) \end{aligned}$$

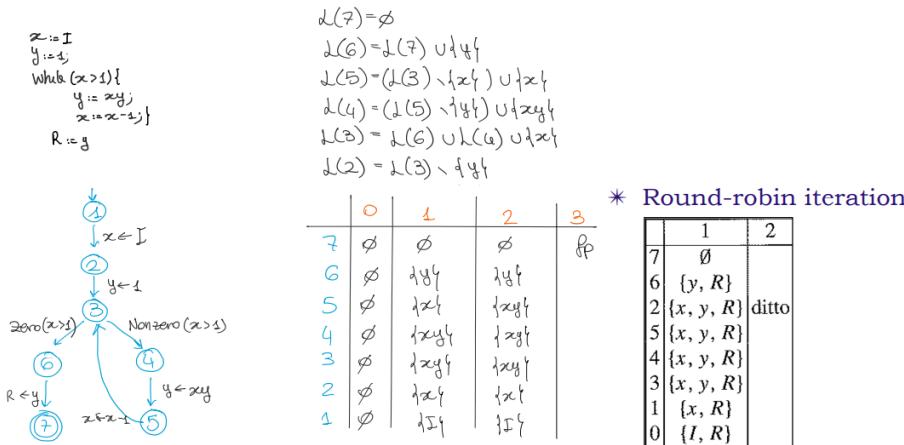
5.5.11 Esempio

Il *calcolo della soluzione* consiste nel costruire un sistema di disequazioni (una per ogni punto di programma):

$$\begin{aligned} \mathcal{L}[\text{exit}] &\supseteq X \\ \mathcal{L}[u] &\supseteq \llbracket k \rrbracket^{\#}(\mathcal{L}[v]) \end{aligned}$$

dove $k = (u, \text{lab}, v)$. Il calcolo della liveness prevede quindi che:

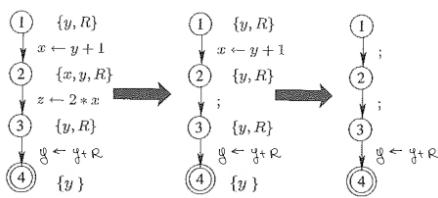
- Si parte da un'informazione iniziale X (insieme di tutte le variabili live in uscita).
- La liveness in un generico punto u è data dall'unione di tutti gli archi v che escono da u .



5.5.12 Rafforzare l'analisi

Quando sfruttiamo l'analisi per rimuovere assegnamenti a variabili non-live potremmo in realtà rendere non-live altre variabili. Nella figura (a) sottostante possiamo infatti notare che la rimozione delle definizioni di z e x (perché non-live) non permettono poi la definizione di y . In questi casi è utile definire delle condizioni più restrittive per la liveness:

- L'uso di una variabile in un assegnamento ad una variabile non-live non è un vero uso (*true use*).
- Una variabile che si trova tra una definizione e un vero uso è detta *true live*.

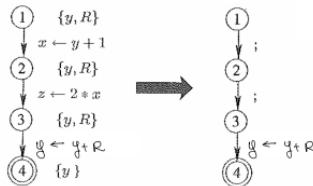


(a) Debolezza dell'analisi

Lab	y truly used
$;$	false
$\text{NonZero}(e)$	$y \in \text{Vars}(e)$
$\text{Zero}(e)$	$y \in \text{Vars}(e)$
$x \leftarrow e$	$y \in \text{Vars}(e) \wedge x$ is truly live at v
$\text{input}(x)$	false

(b) Vero uso di una variabile

Se analizziamo l'esempio di prima con le nuove informazioni, possiamo dire che x non è truly used perché z è una variabile non-live. Di conseguenza x non sarà true live e ciò permetterà la definizione di y .



5.5.13 Modificare le equazioni

Le *equazioni* dovranno essere modificate in questo modo:

$$T\text{LiveOut}(n) = \begin{cases} \emptyset & \text{se } n = \text{exit} \\ \bigcup_{p \in \text{succ}(n)} T\text{LiveIn}(p) & \text{altrimenti} \end{cases}$$

$$T\text{LiveIn}(n) = \text{Gen}(n) \cup (T\text{LiveOut}(n) \setminus \text{Kill}(n))$$

$$\text{Gen}(p) = \{x \mid (\exists y := e \in p. x \in \text{Var}(e) \wedge y \text{ true live}) \vee \exists e \in p. x \in \text{Var}(e)\}$$

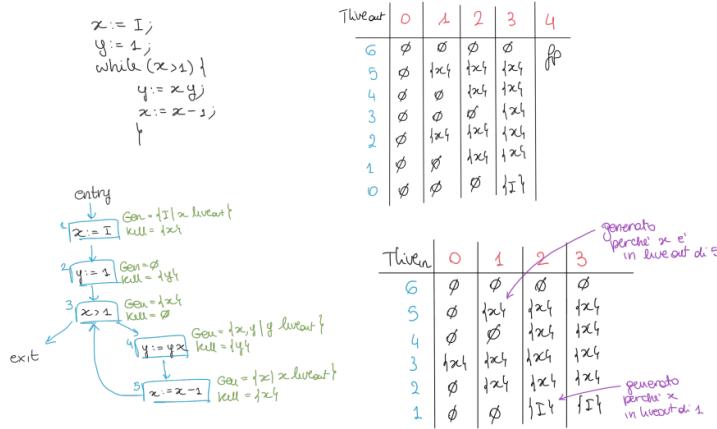
$$\text{Kill}(p) = \{x \mid \exists x := e \in p\}$$

Manipolandole possiamo ottenere la seguente *equazione di punto fisso*:

$$T\text{LiveOut}(n) = \bigcup_{p \in \text{succ}(n)} \text{Gen}(p) \cup (T\text{LiveOut}(p) \setminus \text{Kill}(p))$$

$T\text{LiveIn}(n)$ sarà quindi l'insieme delle variabili truly live all'entrata di un nodo, mentre $T\text{LiveOut}(n)$ sarà l'insieme delle variabili truly live all'uscita di un nodo.

5.5.14 Esempio (con le modifiche)

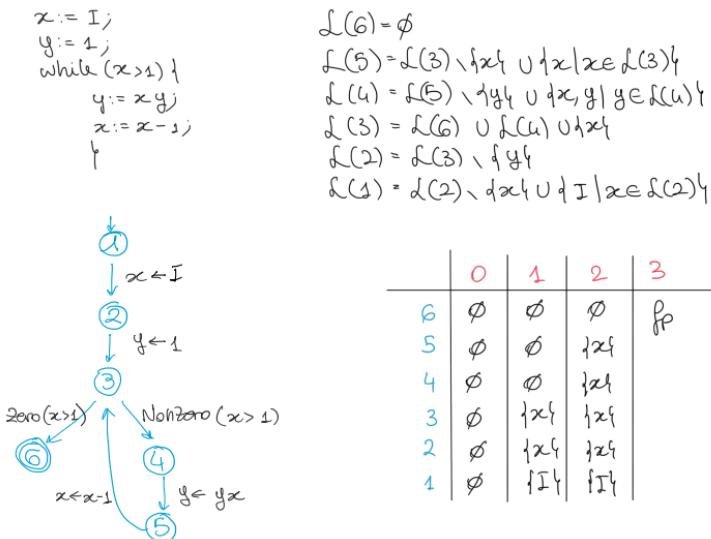


5.5.15 Modificare l'abstract edge effect

Definiamo la semantica astratta nel caso dell'analisi di *true liveness* per ogni etichetta del CFG:

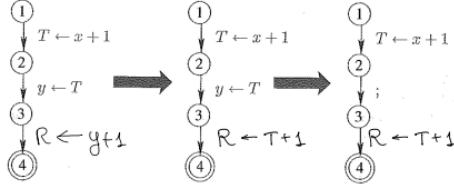
$$\begin{aligned}
 [\cdot]^\# L &= L \\
 [[NonZero(e)]]^\# L &= [[Zero(e)]]^\# L = L \cup Var(e) \\
 [[input(x)]]^\# L &= L \setminus \{x\} \\
 [[x \leftarrow e]]^\# L &= (L \setminus \{x\}) \cup ((x \in L)?Var(e) : \emptyset)
 \end{aligned}$$

5.5.16 Esempio (con le modifiche)



5.6 Propagazione delle copie

In un programma ci possono essere degli assegnamenti che *copiano* semplicemente il valore di altre variabili. Potremo quindi eliminare tali assegnamenti e *propagare* l'informazione della variabile copiata. Nella *figura* sottostante, la variabile y fa solo da placeholder per la variabile T e quindi è possibile eliminarla propagando l'uso di T invece che l'uso di y .



5.6.1 Analisi di copy propagation

L'*analizzatore* tiene traccia, ad ogni punto di programma e per una certa variabile x , dell'insieme di variabili che contengono il valore di x (copie), per poterle rimpiazzarle tutte con un uso di x . Più grande è questo insieme, maggiori sono le chance di ottimizzare il codice.

5.6.2 Costruire le equazioni

Le *equazioni* da costruire sono:

$$\begin{aligned} CopyIn(n) &= \begin{cases} \{(x, x) \mid x \in Var\} & \text{se } n = entry \\ \bigcap_{p \in pred(n)} CopyOut(p) & \text{altrimenti} \end{cases} \\ CopyOut(m) &= Gen(m) \cup (CopyIn(m) \setminus Kill(m)) \\ Gen(m) &= \{(x, y) \mid \exists x := y \in m\} \\ Kill(m) &= \{(x, y) \mid \exists x := e \in m. x \neq y\} \end{aligned}$$

$CopyIn(n)$ sarà quindi l'insieme delle copie che valgono in entrata ad un nodo, mentre $CopyOut(n)$ sarà l'insieme delle copie che valgono in uscita da un nodo. Entrambi sono sottoinsiemi dell'insieme delle copie definito come:

$$Copy(x) = \{(x, y) \mid x, y \text{ sono copie} \wedge x \neq y\}$$

Inoltre, il fatto che una copia è disponibile prima di un blocco p se è disponibile dopo ogni predecessore di p ci dice che l'analisi è *forward* e *definite* (l'informazione si propaga per intersezione).

5.6.3 Esempio

$T := x+1;$ $\mathcal{C}(1) = \{(2z) \mid z \in \text{Var}\}$ $(x z_T) = \{f(xw) \mid w + T\}$
 $y := T;$ $\mathcal{C}(2) = \{(2w) \mid z, w \in \text{Var}\}$
 $R := y+1;$ omettiamo $(2z)$ dove ci sono altre copie.

entry

1 $T := x+1$ $\text{Gen} = \emptyset$ $\text{Kill} = \{f(T, 2)\} \quad z \neq T$
2 $y := T$ $\text{Gen} = \{(T, y)\}$ $\text{Kill} = \{f(y, 2)\} \quad z \neq y$
3 $R := y+1$ $\text{Gen} = \emptyset$ $\text{Kill} = \{f(R, 2)\} \quad z \neq R$
exit

In	0	1	2	3	4
0	$(2z)$	$(2z)$	$(2z)$	$(2z)$	fp
1	$(2w)$	$(2z)$	$(2z)$	$(2z)$	
2	$(x z_T)(y z_T)(R z_T)$	$(2z)$	$(2z)$		
3	$(x z_T)(T z_T)(R z_T)(y T)(y R)$	$(y T)$	$(y T)$		
4	$(x z_T)(T z_R)(y z_R)$	$(x T)(y T)$	$(y T)$		

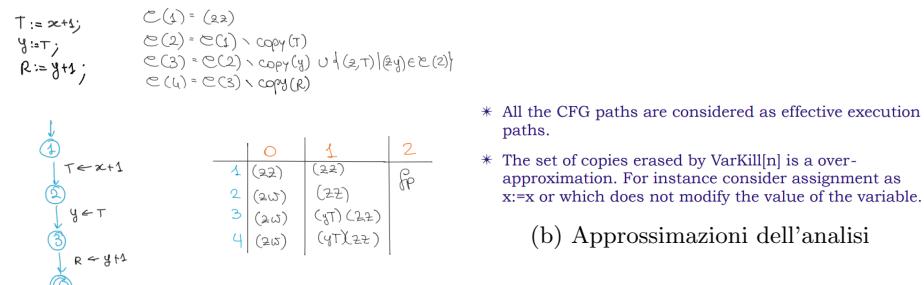
Out	0	1	2	3
0	$(2z)$	$(2z)$	$(2z)$	$(2z)$
1	$(x z_T)(y z_T)(R z_T)$	$(2z)$	$(2z)$	$(2z)$
2	$(x z_y)(T z_y)(R z_y)(y T)$	$(y T)(x R)$	$(y T)$	$(y T)$
3	$(x z_R)(T z_R)(y z_R)$	$(x T)(y T)$	$(y T)$	$(y T)$

5.6.4 Abstract edge effect

Definiamo la semantica astratta nel caso dell'analisi di *copy propagation* per ogni etichetta del CFG:

$$\begin{aligned} \llbracket ; \rrbracket^{\#C} &= C \\ \llbracket \text{NonZero}(e) \rrbracket^{\#C} &= \llbracket \text{Zero}(e) \rrbracket^{\#C} = C \\ \llbracket \text{input}(x) \rrbracket^{\#C} &= C \setminus \text{Copy}(x) \\ \llbracket x \leftarrow e \rrbracket^{\#C} &= C \setminus \text{Copy}(x) \\ \llbracket x \leftarrow y \rrbracket^{\#C} &= C \setminus \{(z, x) \mid z \in \text{Var}\} \cup \{(z, x) \mid (z, y) \in C\} \end{aligned}$$

5.6.5 Esempio e fonti di imprecisione



(a) Applicazione della semantica astratta

5.7 Reaching definitions

Dato un program point n vogliamo identificare le *definizioni* di variabili (assegnamenti) che *raggiungono* n . Tale proprietà può essere *formalizzata* come:

$$\{(x, p) \mid x \in \text{Var}, p \text{ program point}\}$$

dove (x, p) significa che x è definita al punto di programma p . Inoltre, useremo il simbolo speciale $?$ per dire che una variabile non è stata definita.

5.7.1 Costruire le equazioni

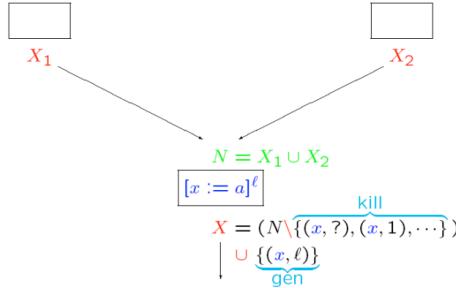
L'analisi è specificata dalle seguenti *equazioni*:

$$\begin{aligned} RdIn(n) &= \begin{cases} \{(x, ?) \mid x \in \text{Var}\} & \text{se } n = \text{entry} \\ \bigcup_{m \in \text{pred}(n)} RdOut(m) & \text{altrimenti} \end{cases} \\ RdOut(m) &= Gen(m) \cup (RdIn(m) \setminus Kill(m)) \\ Gen(m) &= \{(x, m) \mid \exists x := e \in m \vee \text{input}(x) \in m\} \\ Kill(m) &= \{(x, m') \mid \exists x := e \in m \vee \text{input}(x) \in m\} \end{aligned}$$

Manipolandole possiamo ottenere la seguente *equazione di punto fisso*:

$$RdIn(n) = \bigcup_{m \in \text{pred}(n)} Gen(m) \cup (RdIn(m) \setminus Kill(m))$$

Si può notare come l'informazione iniziale sia che all'entry nessuna variabile è stata definita. L'analisi è quindi *forward* e *possible* (l'informazione si propaga per unione).

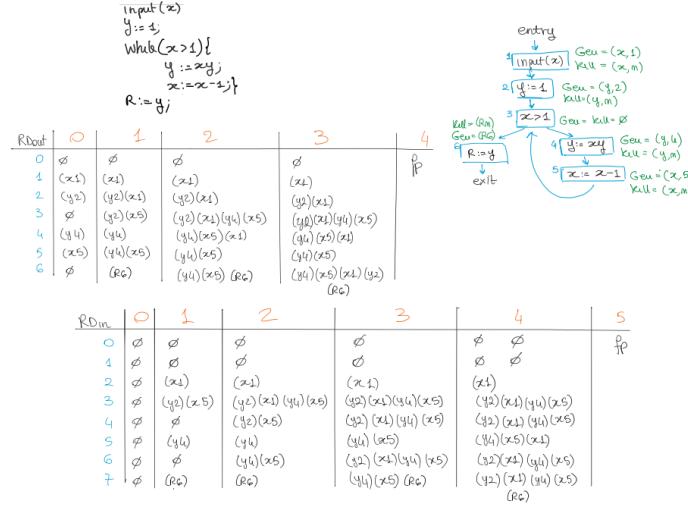


5.7.2 Ottimizzazione code motion

L'analisi di *reaching definitions* viene usata in *code motion*. In particolare so che se faccio un assegnamento dentro un ciclo senza però modificare le sue variabili all'interno del ciclo, allora posso spostarlo direttamente all'entrata del ciclo (*invariante*).

<pre>for(int i=0; i<n; i++) { x = y + z; a[i] = 2*i + x; }</pre>	<pre>x = y + z; for(int i=0; i<n; i++) { a[i] = 2*i + x; }</pre>
---	---

5.7.3 Esempio



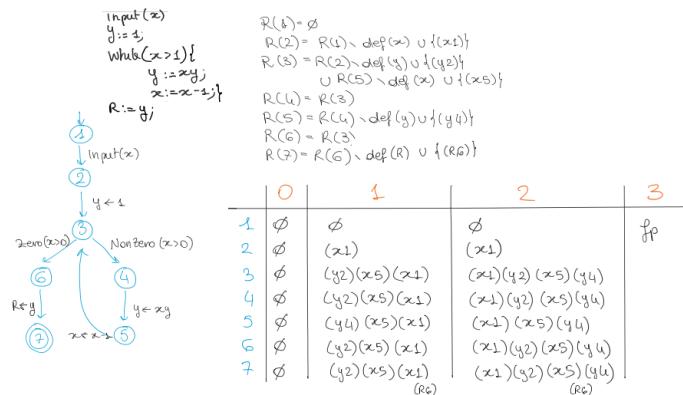
5.7.4 Abstract edge effect

Definiamo la semantica astratta nel caso dell'analisi di *reaching definitions* per ogni etichetta del CFG:

$$\begin{aligned}
[\![;\]\!]^{\#} R &= R \\
[\![NonZero(e)]\!]^{\#} R &= [\![Zero(e)]\!]^{\#} R = R \\
[\![input(x)]\!]^{\#} R &= \{(x, u) \mid (u, input(x), v) \text{ executed edge}\} \cup (R \setminus def(x)) \\
[\![x \leftarrow e]\!]^{\#} R &= \{(x, u) \mid (u, x \leftarrow e, v) \text{ executed edge}\} \cup (R \setminus def(x))
\end{aligned}$$

dove $def(x) = \{(x, p) \mid p \text{ program point}\}$.

5.7.5 Esempio



5.8 Struttura delle analisi di data-flow

Le analisi di data-flow possono essere:

- *Forward*, se le informazioni vengono costruite partendo dall'inizio del programma.
- *Backward*, se le informazioni vengono costruite partendo dalla fine del programma.

Inoltre, a seconda di come viene propagata l'informazione, le analisi di data-flow si dividono ulteriormente in:

- *Possible*, se si propaga per unione.
- *Definite*, se si propaga per intersezione.

5.8.1 Riepilogo

	POSSIBLE	DEFINITE
FORWARD	Reaching definitions	Available expressions Copy propagation
BACKWARD	Live variables	

5.8.2 Risoluzione degli esercizi

Per risolvere un esercizio su una particolare analisi di data-flow è necessario:

1. Fornire una *descrizione* intuitiva dell'analisi richiesta.
2. Fornire l'*equazione di punto fisso* (assieme a tutti i suoi elementi).
3. Fornire la *semantica astratta*.
4. Mostrare in forma tabellare il calcolo dell'analisi mediante *punto fisso*.
5. Mostrare in forma tabellare il calcolo dell'analisi mediante la *semantica*.

Per questioni di spazio, si riportano di seguito i collegamenti ai paragrafi di questo documento in cui si possono cercare le risposte ai passi appena descritti:

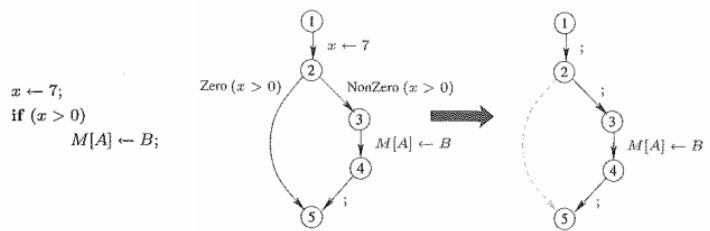
Available expressions	Descrizione, equazioni, semantica astratta Esempio di calcolo mediante punto fisso Esempio di calcolo mediante semantica
Live variables	Descrizione, equazioni, semantica astratta Esempio di calcolo mediante punto fisso Esempio di calcolo mediante semantica
Copy propagation	Descrizione, equazioni, semantica astratta Esempio di calcolo mediante punto fisso Esempio di calcolo mediante semantica
Reaching definitions	Descrizione, equazioni, semantica astratta Esempio di calcolo mediante punto fisso Esempio di calcolo mediante semantica

5.9 Analisi non distributive

Le *analisi non distributive* si occupano di cosa calcola il programma, e quindi sono più vicine alla semantica e più distanti dalla sintassi.

5.10 Propagazione delle costanti

L'analisi di *constant propagation* ha come obiettivo quello di determinare in ogni punto di programma se una variabile ha valore costante ogni volta che durante l'esecuzione si ritorna al punto di programma. Questa analisi può quindi essere utile per l'ottimizzazione del codice. Ad esempio, nella figura sottostante, il fatto che la variabile x sia costante a 7 rende sempre vero il test al punto 2, per cui si potrebbe eliminare direttamente tale test.



Inoltre, questa analisi è un caso speciale di *valutazione parziale* dei programmi. Infatti, per ogni punto di programma v si calcolano le seguenti informazioni:

- Quale valore ha una variabile quando il controllo raggiunge v ?
- Il punto v è potenzialmente raggiungibile?

5.10.1 Esempio

Siccome siamo nell'analisi delle costanti, dobbiamo verificare che i valori delle variabili rimangano sempre gli stessi. Quindi, se una variabile non è costante, metteremo un ? in quanto a priori non possiamo sapere se nel futuro essa sarà costante o meno.

- Non distributive data-flow analysis
- Example:

```
a:=1; b:=2; c:=3; d:= 3; e:=0;
1: while B {
2:   b:= 2*a; d:=d+1; e:=e-a;
3:   a:=b-a; c:=e+d
4: }
```
- Symbolic execution [Aho-Ullman, Dragoon '76]

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

Dopo aver raggiunto il *punto fisso* alla terza iterazione, facciamo alcune considerazioni. Le uniche variabili che mutano il loro valore sono d (continua a crescere) ed e (continua a decrescere). Ad esempio, d all'iterazione 2 passa da valore 4 a valore 5 e quindi le abbiamo messo valore ?. Abbiamo poi una perdita di precisione sulla variabile c perché $e+d=?+?$ non so cosa fa, anche se in realtà so che c è costante perché d cresce sempre di 1 ed e decresce sempre di 1.

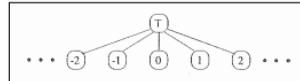
5.10.2 Costruzione dell'analisi

Dobbiamo innanzitutto costruire un *reticolo* fatto da tutti i singoletti uniti a \top . Il dominio astratto sarà quindi fatto dall'insieme di tutti i numeri interi unito \top . Dobbiamo poi astrarre il *dominio delle memorie* (da cui si estraggono i valori delle variabili). Aggiungiamo quindi il \perp per modellare le espressioni non raggiungibili. $D(x)$ sarà un singoletto se il valore di x è costante nel punto di programma considerato, oppure sarà \top nel caso in cui non si sa se è costante.

* We design a partial order for the possible values of variables

* Natural number plus \top to denote unknown variables

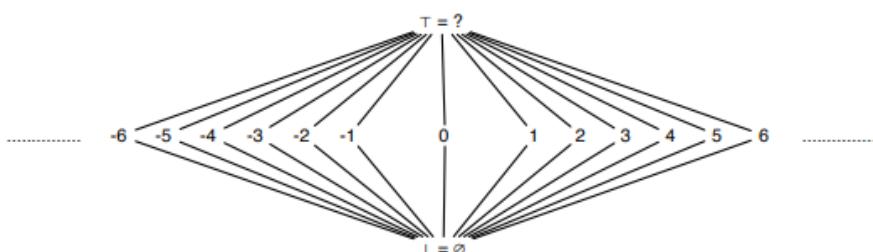
$$\mathbb{Z}^\top = \mathbb{Z} \cup \{\top\} \quad \text{and} \quad x \sqsubseteq y \text{ iff } x = y \text{ or } y = \top$$



* We construct the complete lattice of abstract variable bindings $\mathbb{D} = (\text{Var} \rightarrow \mathbb{Z}^\top)_\perp = (\text{Var} \rightarrow \mathbb{Z}^\top) \cup \{\perp\}$

5.10.3 Dominio astratto

Il dominio astratto deve essere una *famiglia di Moore*, ovvero devono esistere \perp , \top e i singoletti dei valori che mi identificano un valore costante (non confrontabili gli uni con gli altri). Nel dominio astratto rappresentato nella *figura* sottostante, $\alpha(0, 1) = \top$ perché abbiamo una variabile non costante, mentre $\alpha(5) = 5$ perché la variabile è costante.



5.10.4 Insieme dei valori assunti

Andiamo a definire il calcolo dei valori di una certa variabile x servendoci dell'*insieme dei valori assunti* X .

$$X \subseteq (\text{Var} \rightarrow \mathbb{Z}^\top)$$

$$\begin{cases} X = \emptyset & \text{then } \bigsqcup X = \perp \in \mathbb{D} \\ X \neq \emptyset & \text{then } D(x) = \bigsqcup \{ f(x) \mid f \in X \} = \begin{cases} z & \text{if } \forall f \in X. f(x) = z \\ \top & \text{otherwise} \end{cases} \end{cases}$$

Il dominio di ciò che vogliamo osservare viene quindi costruito in questo modo:

- Se X è vuoto, allora ho \perp perché l'espressione non è raggiungibile.
- Se X non è vuoto, allora il valore di x è il least upper bound di tutti i suoi valori assunti; in particolare:
 - Il valore di x sarà z , se il valore assunto in ogni punto di programma è sempre lo stesso (variabile costante).
 - Il valore di x sarà \top , altrimenti (variabile non costante).

5.10.5 Abstract edge effect

Andiamo ora a definire la *semantica astratta* per ogni arco $k = (u, lab, v)$, dove lab sono i comandi, del CFG che simula il programma. Un arco è \perp se l'espressione non è raggiungibile. Mentre se una variabile definita nel dominio D è raggiungibile, c'è bisogno di una *funzione* di valutazione astratta per calcolare il suo valore. Tale funzione dovrà gestire il caso in cui il valore di un'espressione non può essere determinato in D e quindi viene valutato a \top . Per garantire questo dobbiamo rimpiazzare gli *operatori* concreti op con una loro versione astratta $op^\#$ così definita:

$$a op^\# b = \begin{cases} \top & \text{se } a = \top \vee b = \top \\ a op b & \text{altrimenti} \end{cases}$$

La semantica astratta delle *espressioni* sarà quindi la seguente:

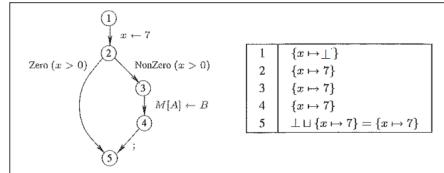
$$\begin{aligned} \llbracket c \rrbracket^\# D &= c \\ \llbracket op e \rrbracket^\# D &= op^\# \llbracket e \rrbracket^\# D \\ \llbracket e_1 op e_2 \rrbracket^\# D &= \llbracket e_1 \rrbracket^\# D op^\# \llbracket e_2 \rrbracket^\# D \\ \llbracket x \rrbracket^\# D &= D(x) \end{aligned}$$

La semantica astratta dei *comandi*, invece, è la seguente:

$$\begin{aligned} \llbracket ; \rrbracket^{\#} D &= D \\ \llbracket \text{NonZero}(e) \rrbracket^{\#} D &= \begin{cases} \perp & \text{se } 0 = \llbracket e \rrbracket^{\#} D \\ D & \text{altrimenti} \end{cases} \\ \llbracket \text{Zero}(e) \rrbracket^{\#} D &= \begin{cases} \perp & \text{se } 0 \not\subseteq \llbracket e \rrbracket^{\#} D \\ D & \text{se } 0 \sqsubseteq \llbracket e \rrbracket^{\#} D \end{cases} \\ \llbracket \text{input}(x) \rrbracket^{\#} D &= D[x \mapsto \top] \\ \llbracket x \leftarrow e \rrbracket^{\#} D &= D[x \mapsto \llbracket e \rrbracket^{\#} D] \end{aligned}$$

$$\begin{aligned} D &= \{x \mapsto 2, y \mapsto \top\} \\ \llbracket x + 7 \rrbracket^{\#} D &= \llbracket x \rrbracket^{\#} D +^{\#} \llbracket 7 \rrbracket^{\#} D \\ &= 2 +^{\#} 7 \\ &= 9 \\ \llbracket x - y \rrbracket^{\#} D &= 2 -^{\#} \top \\ &= \perp \end{aligned}$$

(a) Esempio con le espressioni



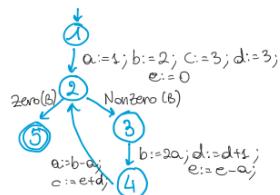
(b) Esempio con i comandi

5.10.6 Esempio

```

1 a:=1; b:=2; c:=3; d:=3; e:=0;
2 while B do
    3 b := 2a; d := d+1; e := e-a;
    4 a := b-a; c := e+d;
5 endw

```



$$\begin{aligned} C(1) &= \emptyset \\ C(2) &= [a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 3; e \mapsto 0] \\ &\cup C(4)[a \mapsto C(4)(b) - C(4)(a); \\ &\quad c \mapsto C(4)(c) + C(4)(d)] \\ C(3) &= C(2) \\ C(4) &= C(3)[b \mapsto 2C(3)(a), d \mapsto C(3)(d)+1, \\ &\quad e \mapsto C(3)(e) - C(3)(a)] \\ C(5) &= C(2) \end{aligned}$$

	0	1	2	3	4
1	\perp	\perp	\perp	\perp	\perp
2	\perp	$a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 3; e \mapsto 0$	$a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 3; e \mapsto 1$	$a \mapsto 1; b \mapsto 2; c \mapsto 1; d \mapsto 1; e \mapsto 1$	$\text{C} \mapsto \perp$
3	\perp	$a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 3; e \mapsto 0$	$a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 4; e \mapsto 1$	$a \mapsto 1; b \mapsto 2; c \mapsto 1; d \mapsto 1; e \mapsto 1$	\perp
4	\perp	$a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 3; e \mapsto -1$	$a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 4; e \mapsto 1$	$a \mapsto 1; b \mapsto 2; c \mapsto 1; d \mapsto 1; e \mapsto 1$	\perp
5	\perp	$a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 3; e \mapsto 0$	$a \mapsto 1; b \mapsto 2; c \mapsto 3; d \mapsto 1; e \mapsto 1$	$a \mapsto 1; b \mapsto 2; c \mapsto 1; d \mapsto 1; e \mapsto 1$	\perp

5.10.7 MOP vs MFP

La *soluzione MOP* in un certo punto di programma v è data dal least upper bound (\sqcup) delle semantiche di tutti i cammini che vanno dall'entry a v (calcolate partendo dall'informazione iniziale D_\perp tale che $D_\perp(x) = \perp$).

$$\mathcal{D}^*[v] = \bigsqcup \{\llbracket \pi \rrbracket^\# D_\perp \mid \pi : \text{entry} \rightarrow^* v\}$$

Tuttavia possiamo calcolare solo la *soluzione MFP*, ovvero un'approssimazione della MOP che risolve equazioni semantiche per ogni punto di programma. Nella *figura* sottostante viene infatti costruito un controesempio per dimostrare che la semantica astratta *non è distributiva* rispetto al least upper bound e quindi che $MFP \neq MOP$.

$$D_1 = \{x \mapsto 2, y \mapsto 3\} \quad \text{and} \quad D_2 = \{x \mapsto 3, y \mapsto 2\}$$

On the one hand, we have:

$$\begin{aligned} \llbracket x \leftarrow x + y \rrbracket^\# D_1 \sqcup \llbracket x \leftarrow x + y \rrbracket^\# D_2 &= \{x \mapsto 5, y \mapsto 3\} \sqcup \{x \mapsto 5, y \mapsto 2\} \\ &= \{x \mapsto 5, y \mapsto \top\} \end{aligned}$$

On the other hand, it holds that:

$$\begin{aligned} \llbracket x \leftarrow x + y \rrbracket^\# (D_1 \sqcup D_2) &= \llbracket x \leftarrow x + y \rrbracket^\# \{x \mapsto \top, y \mapsto \top\} \\ &= \{x \mapsto \top, y \mapsto \top\} \end{aligned}$$

Therefore

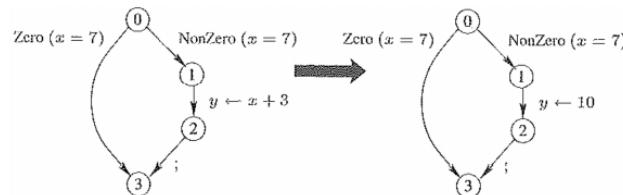
$$\llbracket x \leftarrow x + y \rrbracket^\# D_1 \sqcup \llbracket x \leftarrow x + y \rrbracket^\# D_2 \neq \llbracket x \leftarrow x + y \rrbracket^\# (D_1 \sqcup D_2)$$

violating the distributivity property.

5.10.8 Rafforzare l'analisi

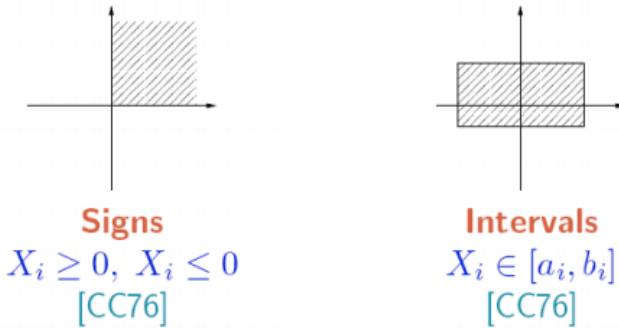
L'analisi di *constant propagation* può essere migliorata propagando anche l'informazione che l'espressione di una guardia è vera. Infatti, anche non conoscendo il valore di una variabile prima di un branch, l'analisi può derivare che quella variabile è costante sul valore che rende vera la clausola *NonZero* (a partire dal relativo arco). Per rendere effettivo questo rafforzamento, la *semantica astratta* dei comandi va aggiornata con:

$$\llbracket \text{NonZero}(x = e) \rrbracket^\# D = \begin{cases} \perp & \text{se } 0 = \llbracket x = e \rrbracket^\# D \\ D \sqcup [x \mapsto D(x) \cap \llbracket e \rrbracket^\# D] & \text{altrimenti} \end{cases}$$



5.11 Analisi degli intervalli

L'analisi della propagazione delle costanti non è altro che un'analisi di *limiti* di variabili che si riferiscono ad intervalli. Si cerca infatti di trovare le variabili che sono ridotte ad un *intervallo* di un singolo valore.



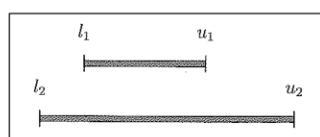
Passeremo quindi all'*analisi degli intervalli*, ovvero accetteremo il fatto che una variabile in certo punto di programma possa avere valori diversi (ma comunque entro i limiti di un intervallo) e cercheremo di calcolare il range di quella variabile.

5.11.1 Dominio astratto degli intervalli

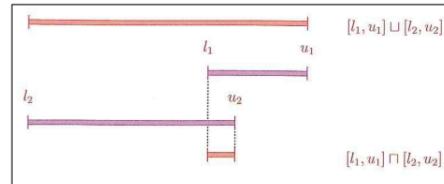
Il dominio degli intervalli è un dominio *non relazionale* perché non guarda a relazioni tra variabili diverse. Per questo motivo esso è più efficiente ma anche più impreciso in quanto, una volta decisi i limiti, si approssima il valore della variabile a quell'intervallo. Un *intervallo* è un insieme (non vuoto) di interi formato da oggetti del tipo $[l, u]$ tali che $l \in \mathbb{Z} \cup \{-\infty\}$, $u \in \mathbb{Z} \cup \{+\infty\}$ e $l \leq u$. La *relazione d'ordine* \sqsubseteq dice che un intervallo è contenuto in un altro più grande se il suo estremo inferiore è maggiore di quello più grande e il suo estremo superiore è minore di quello più grande. Infine, il *lub* tra due intervalli è l'intervallo più piccolo che contiene entrambi ($\sqcup \neq \cup$) mentre il *glb* tra due intervalli è il più grande intervallo contenuto in entrambi ($\sqcap = \cap$). Inoltre, il dominio degli intervalli *non è ACC* perché dato un intervallo si riesce sempre a trovare un intervallo più grande che non è \top .

$$[l_1, u_1] \sqcup [l_2, u_2] = [\min[l_1, l_2], \max[u_1, u_2]]$$

$$[l_1, u_1] \sqsubseteq [l_2, u_2] \quad \text{iff} \quad l_2 \leq l_1 \text{ and } u_1 \leq u_2 \quad [l_1, u_1] \sqcap [l_2, u_2] = [\max[l_1, l_2], \min[u_1, u_2]]$$



(a) Relazione d'ordine



(b) LUB e GLB

5.11.2 Astrarre gli stati concreti

Un insieme di interi è approssimato dal più piccolo intervallo $[l, u]$ che lo contiene. La corrispondente concretizzazione, però, sarà $\{n \mid l \leq n \leq u\}$ per cui viene persa l'informazione precisa di quali erano gli interi dell'insieme originale. Infine, nelle *figure* sottostanti si mostra in che modo le *operazioni* aritmetiche e booleane possono essere fatte sugli intervalli.

$[l_1, u_1] +^\sharp [l_2, u_2] = [l_1 + l_2, u_1 + u_2] \quad \text{where}$ $-\infty +_- = -\infty$ $+\infty +_- = +\infty$ $-[l, u] = [-u, -l]$ $[l_1, u_1] \cdot^\sharp [l_2, u_2] = [a, b] \quad \text{where}$ $a = \min\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}$ $b = \max\{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\}$	$[l_1, u_1] =^\sharp [l_2, u_2] = \begin{cases} [1, 1] & \text{if } l_1 = u_1 = l_2 = u_2 \\ [0, 0] & \text{if } u_1 < l_2 \vee u_2 < l_1 \\ [0, 1] & \text{otherwise} \end{cases}$ $[l_1, u_1] <^\sharp [l_2, u_2] = \begin{cases} [1, 1] & \text{if } u_1 < l_2 \\ [0, 0] & \text{if } u_2 \leq l_1 \\ [0, 1] & \text{otherwise} \end{cases}$
--	--

(a) Operazioni aritmetiche astratte

(b) Operazioni booleane astratte

5.11.3 Abstract edge effect

La semantica astratta per l'*analisi degli intervalli* è simile a quella già definita per l'analisi di constant propagation:

$$\begin{aligned} \llbracket ; \rrbracket^{\#} D &= D \\ \llbracket \text{NonZero}(e) \rrbracket^{\#} D &= \begin{cases} \perp & \text{se } [0, 0] = \llbracket e \rrbracket^{\#} D \\ D & \text{altrimenti} \end{cases} \\ \llbracket \text{Zero}(e) \rrbracket^{\#} D &= \begin{cases} \perp & \text{se } [0, 0] \not\subseteq \llbracket e \rrbracket^{\#} D \\ D & \text{se } [0, 0] \subseteq \llbracket e \rrbracket^{\#} D \end{cases} \\ \llbracket \text{input}(x) \rrbracket^{\#} D &= D[x \mapsto \top] \\ \llbracket x \leftarrow e \rrbracket^{\#} D &= D[x \mapsto \llbracket e \rrbracket^{\#} D] \end{aligned}$$

dove $\top = [-\infty, +\infty]$ e l'informazione iniziale è $\{x \mapsto [-\infty, +\infty] \mid x \in \text{Var}\}$.

5.11.4 Rafforzare l'analisi

Anche nel caso degli intervalli possiamo migliorare l'analisi propagando l'informazione data dalle *guardie*. Infatti, se seguiamo il ramo falso sappiamo che la guardia è falsa mentre se seguiamo il ramo vero sappiamo che la guardia è vera e quindi possiamo intersecare questa informazione con quella collezionata fino al punto di branch. Per rendere effettivo questo rafforzamento, la *semantica astratta* va aggiornata con:

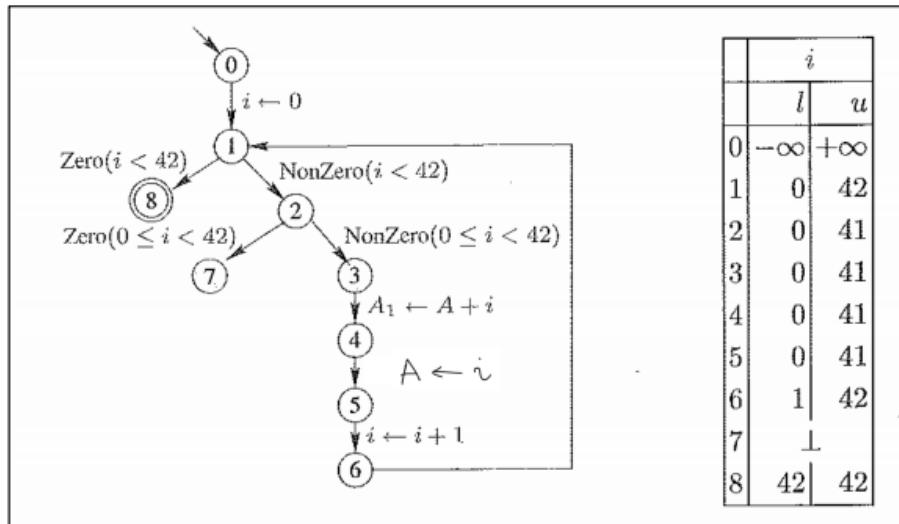
$$\begin{aligned} \llbracket \text{NonZero}(e) \rrbracket^{\#} D &= \begin{cases} \perp & \text{se } [0, 0] = \llbracket e \rrbracket^{\#} D \\ D_1 & \text{altrimenti} \end{cases} \\ \llbracket \text{Zero}(e) \rrbracket^{\#} D &= \begin{cases} \perp & \text{se } [0, 0] \not\subseteq \llbracket e \rrbracket^{\#} D \\ D_2 & \text{se } [0, 0] \subseteq \llbracket e \rrbracket^{\#} D \end{cases} \end{aligned}$$

dove

$$D_1 = \begin{cases} D[x \mapsto D(x) \cap [e_1]^\# D] & \text{se } e \equiv (x = e_1) \\ D[x \mapsto D(x) \cap [-\infty, u - 1]] & \text{se } e \equiv (x < e_1) \wedge [e_1]^\# D = [l, u] \\ D[x \mapsto D(x) \cap [l + 1, +\infty]] & \text{se } e \equiv (x > e_1) \wedge [e_1]^\# D = [l, u] \end{cases}$$

$$D_2 = \begin{cases} D[x \mapsto D(x) \cap [l, +\infty]] & \text{se } e \equiv (x < e_1) \wedge [e_1]^\# D = [l, u] \\ D[x \mapsto D(x) \cap [-\infty, u]] & \text{se } e \equiv (x > e_1) \wedge [e_1]^\# D = [l, u] \\ D & \text{se } e \equiv (x = e_1) \end{cases}$$

5.11.5 Esempio



5.11.6 Widening

Nell'esempio precedente il *punto fisso* viene raggiunto dopo ben 43 iterazioni, ma esistono anche programmi che possono non arrivare mai al punto fisso (in quanto il dominio degli intervalli non è ACC). L'idea è quindi quella di *accelerare* la convergenza tramite il *widening*, anche a costo di una possibile perdita di precisione. Considerando il sistema di equazioni sui punti di programma $x_i = x_i \sqcup f_i(x_1, \dots, x_n)$, rimpiazzando l'operatore di *lub* con il *widening* si ottiene il nuovo sistema di equazioni $x_i = x_i \nabla f_i(x_1, \dots, x_n)$, il quale calcola un punto fisso ma non necessariamente il minimo ($v_1 \sqcup v_2 \sqsubseteq v_1 \nabla v_2$).

5.11.7 Definizione sugli intervalli

Il *widening* su intervalli consiste nell'accelerare a $-\infty$ se c'è un trend di decrescita oppure a $+\infty$ se c'è un trend di crescita. L'operatore di *widening* *non è commutativo*, ossia l'operando di sinistra è il vecchio valore mentre quello di destra è il nuovo valore.

$$D\nabla\perp = \perp\nabla D = D$$

$$(D_1\nabla D_2)(x) = D_1(x)\nabla D_2(x) = [l_1, u_1]\nabla[l_2, u_2] = [l, u]$$

dove

$$l = \begin{cases} l_1 & \text{se } l_1 \leq l_2 \\ -\infty & \text{altrimenti} \end{cases}$$

$$u = \begin{cases} u_1 & \text{se } u_1 \geq u_2 \\ +\infty & \text{altrimenti} \end{cases}$$

5.11.8 Narrowing

Con il *narrowing* si cerca di recuperare la precisione che è stata persa con il *widening*. Considerando il sistema di equazioni $y_i = x_i \sqcap f_i(x_1, \dots, x_n)$, rimpiazzando l'operatore di *glb* con il *narrowing* si ottiene un nuovo sistema di equazioni in cui vale che $a_1 \sqcap a_2 \sqsubseteq a_1 \Delta a_2 \sqsubseteq a_1$. Anche l'operatore di *narrowing* non è *commutativo*.

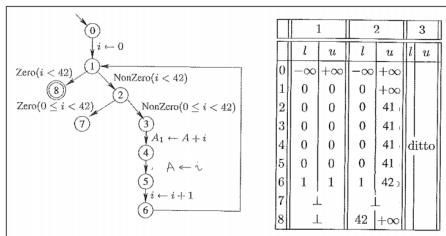
$$(D_1 \Delta D_2)(x) = D_1(x) \Delta D_2(x) = [l_1, u_1] \Delta [l_2, u_2] = [l, u]$$

dove

$$l = \begin{cases} l_2 & \text{se } l_1 = -\infty \\ l_1 & \text{altrimenti} \end{cases}$$

$$u = \begin{cases} u_2 & \text{se } u_1 = +\infty \\ u_1 & \text{altrimenti} \end{cases}$$

5.11.9 Esempio



(a) Applicazione del widening

	0	1	2			
	l	u	l	u	l	u
0	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
1	0	$+\infty$	0	$+\infty$	0	42
2	0	$+\infty$	0	41	0	41
3	0	$+\infty$	0	41	0	41
4	0	$+\infty$	0	41	0	41
5	0	$+\infty$	0	41	0	41
6	1	$+\infty$	1	42	1	42
7	42	$+\infty$	\perp	\perp	\perp	\perp
8	42	$+\infty$	42	$+\infty$	42	42

(b) Applicazione del narrowing

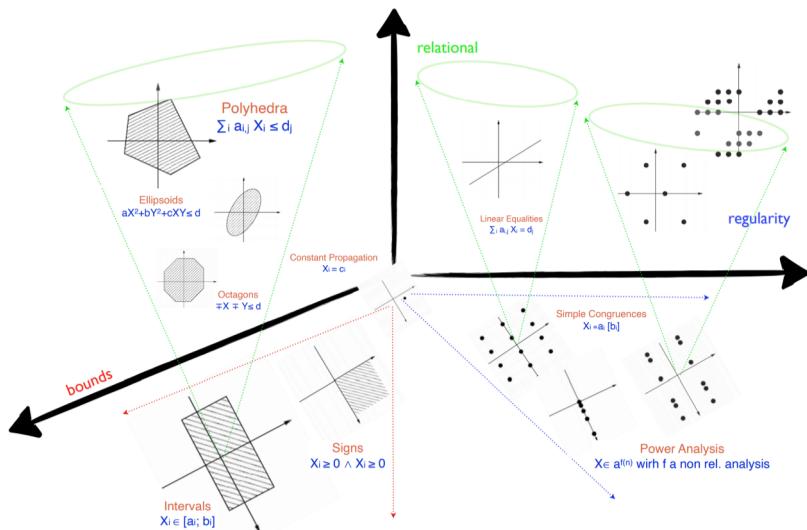
5.11.10 Un esempio completo

$x := 1;$ 1: while $x < 10000$ do 2: $x := x + 1$ 3: od; 4:	$\begin{cases} X_1 = [1, 1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1, 1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$	$x := 1;$ 1: while $x < 10000$ do 2: $x := x + 1$ 3: od; 4:	$\begin{cases} X_1 = [1, 1] \\ X_2 = (X_1 \cup X_3) \cap [-\infty, 9999] \\ X_3 = X_2 \oplus [1, 1] \\ X_4 = (X_1 \cup X_3) \cap [10000, +\infty] \end{cases}$
---	--	---	--

(a) Convergenza

(b) Widening e narrowing

5.12 Altre analisi su domini (non) relazionali



5.13 Risoluzione degli esercizi

Per risolvere un esercizio su una particolare analisi non distributiva è necessario:

1. Fornire una *descrizione* intuitiva dell'analisi richiesta (**costanti**, **intervalli**).
2. Fornire la *semantica astratta* (**costanti**, **intervalli**).
3. Definire il *widening*, se presente (**intervalli**).
4. Calcolare in forma tabellare l'analisi su un programma definendo le *equazioni* sui punti di programma (**costanti**, **intervalli**).
5. Calcolare in forma tabellare l'analisi su un programma mediante *widening* (**intervalli**).

Per questioni di spazio, sono stati riportati tra parentesi i collegamenti ai paragrafi di questo documento in cui cercare le risposte ai relativi passi.

Capitolo 6

Analisi dinamica

6.1 Introduzione

L'*analisi dinamica* di un programma si basa sulla sua esecuzione. Essa viene utilizzata in vari ambiti, quali: *testing*, *monitoring*, debugging, emulazione e virtualizzazione, profiling e tracing, *slicing*, disassembly, decompilation.

6.2 Testing

Il *testing* consiste, principalmente, nell'esecuzione di un programma su un campione di dati (molto piccolo) passato come input. Si tratta quindi di una tecnica di analisi dinamica, in cui ovviamente occorre eseguire il programma. Inoltre, vale il concetto di *inaccuratezza ottimistica*, ovvero si suppone che il programma sia corretto se supera una serie di test scelti precedentemente. Più formalmente, il testing può essere definito come un processo di ricerca di bug/erri/difetti del software.

6.2.1 Obiettivi del testing

L'obiettivo ultimo del testing è quello di cercare e *trovare un errore*. Infatti, questo è l'unico caso in cui il test è completamente decidibile in quanto possiamo dire con assoluta certezza che il programma non funziona correttamente. Inoltre, il testing permette di raggiungere una certa *confidenza* ed eventualmente di andare a validare un certo numero di specifiche del software.

6.2.2 Tipologie di errori

Durante la fase di testing si possono trovare varie tipologie di errori, tra cui:

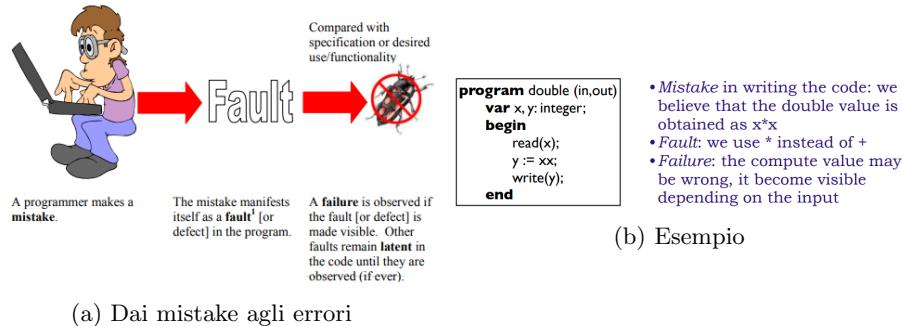
- *Mistake* (errore umano), ovvero un'azione umana che genera un risultato non corretto.
- *Fault* (difetto), ovvero un passo/processo/dato non corretto presente nel codice.
- *Failure* (fallimento), ovvero la mancata abilità da parte del sistema di eseguire la funzionalità attesa.

- *Error* (errore), ovvero la misura della differenza tra il comportamento atteso e quello osservato.

Il comportamento atteso del programma risiede nelle *specifiche*, ovvero in documenti che descrivono le richieste e le caratteristiche del sistema.

6.2.3 Progressione di un fallimento

Un *mistake* del programmatore causa un difetto nel codice (*fault*) che potrebbe causare un *fallimento*. Il testing vuole quindi cercare i difetti nel codice con l'obiettivo di metterli in evidenza per fare in modo che possano essere corretti dagli sviluppatori. L'obiettivo ultimo del testing è comunque quello di migliorare la qualità generale del codice. Infatti, tramite l'*errore*, il testing misura quanto l'esecuzione del codice si allontana dal suo comportamento atteso.



6.3 Aspetti teorici

In generale possiamo dire che un *programma* P è una funzione che va da un insieme di dati in input D ad un insieme di dati in output R .

- Se $P(d)$ rispetta le specifiche (nessun fallimento) allora è sound ($ok(P, d)$), altrimenti non lo è.
- Un programma è sound ($ok(P)$) se per ogni input non va in errore, ovvero: $\forall d \in D. ok(P, d)$.

Diciamo poi che un programma è sound sul *test* $T \subseteq D$ ($ok(P, T)$) se supera la correttezza per ogni dato di test t : $\forall t \in T. ok(P, t)$. L'obiettivo del testing è quindi quello di trovare un T tale per cui $\neg ok(P, T)$.

- Un test T è di successo ($success(T, P)$) se trova uno o più fallimenti su P ($\neg ok(P, T)$).
- Un test T è detto ideale (*ideal*) se il suo fallimento implica la correttezza del programma P : $ok(P, T) \Rightarrow ok(P)$.

Infine, il testing deve poter selezionare i predicati su cui procedere con il test stesso. Quindi, un *criterio di selezione* di test C per P sarà un insieme di predicati su D .

- Un test T è selezionato da C ($selected(C, T)$) se $\forall t \in T. \exists c \in C$. tale per cui $c(t)$ è vero e $\forall c \in C. \exists t \in T$. tale per cui $c(t)$ è vero.
- Un criterio di selezione C è affidabile per P ($reliable(P, C)$) se tutti i test selezionati da C producono o tutti successi o tutti fallimenti, ovvero: $success(T_1, P) \Leftrightarrow success(T_2, P)$.
- Un criterio di selezione C è valido per P ($valid(C, P)$) se trova almeno un test T in cui il programma P fallisce, ovvero: se $\exists d \in d. \neg ok(P, d)$ allora $\exists T. selected(C, T)$ tale per cui $\neg ok(P, T)$.

```
program double (in,out)
var x, y: integer;
begin
    read(x);
    y := xx;
    write(y);
end
```

- If $selected(C, T)$ is such that $T \subseteq \{0,2\}$, then $reliable(double, C)$ but $\neg valid(C, double)$
- If $selected(C, T)$ is such that $T \subseteq \{0,1,2,3,4\}$ then $\neg reliable(double, C)$ but $valid(C, double)$
- If $selected(C, T)$ is such that T contains only values greater than 3, then $reliable(double, C)$ and $valid(C, double)$

6.3.1 Teoremi

- Teorema di *Goodenough & Gerhart*: se si intersecano i test affidabili con quelli validi, si ottengono quelli di fallimento per P ; se poi si interseca il tutto con i test selezionati per quei casi in cui P non fallisce, allora si ottiene che il programma P è sempre corretto. In altre parole, ogni test selezionato da un criterio affidabile e valido è ideale.

$$reliable(C, P) \wedge valid(C, P) \wedge selected(C, T) \wedge \neg success(T, P) \Rightarrow ok(P)$$

- Teorema di *Howden*: non esiste un algoritmo che dato P genera un test finito ideale, ovvero un test selezionato da C , affidabile e valido. Se P è corretto solo su $d \in D$, un criterio C affidabile e ideale deve generare esattamente d . Questo teorema non dice quindi che C non esiste, bensì che C può esistere ma non esiste un modo per calcolarlo.
- Tesi di *Dijkstra*: il testing può provare la presenza di difetti nel programma ma non la loro assenza. In altre parole, il testing può dimostrare che un programma non è corretto, ma non può dimostrare che lo è.
- Teorema di *Weyuker*: dato un programma P , i problemi seguenti sono indecidibili.
 - Esiste un input che fa eseguire un certo comando?
 - Esiste un input che fa eseguire un certo branch?
 - Esiste un input che fa eseguire un certo cammino?

6.4 Processo di testing

L'*esecuzione* dei casi di test è solo una piccola parte del processo di testing. La parte difficile è la *generazione* dei casi di test, i quali devono essere pensati a seconda di quello che fa il programma. Occorre quindi *pianificare* il processo di testing per trovare forme di *misura* che aiutino a capire quando fermare il test.

6.4.1 Generazione dei casi di test

Per generare dati di test esistono diversi *approcci*, tra cui:

- *Random*.
- Analisi *funzionale* (black box), basata sulle specifiche.
- Analisi *strutturale* (white box), basata sul codice.
- *Grey box*, basato su informazioni parziali.
- *Fault-based*, basato sulle classi dei difetti.

6.4.2 Approccio code-based

Nell'analisi statica abbiamo guardato alla semantica analizzando la sintassi. Nel testing possiamo decidere il grado di esaustività del test guardando al codice. Per generare i casi di test basandosi sul codice si può ricorrere alla *coverage* del codice su CFG, ovvero ad una metrica che misura la completezza del testing. Per misurare la coverage, si parte sempre dalla seguente domanda: quali parti devono essere ancora testate nel codice? Infatti, se una parte del CFG non viene mai testata, non si scopriranno mai gli errori. Con *parte del CFG* si intendono: comandi (blocchi), branch (archi), condizioni e cammini.

6.5 Coverage testing

Il *coverage testing* viene effettuato sulle varie componenti del CFG. Si avranno quindi i seguenti criteri di coverage:

- Statement coverage.
- Branch coverage.
- Condition coverage.
- Path coverage.

Nei prossimi paragrafi li analizzeremo uno alla volta usando il seguente esempio:

```

1  int foo (int a, int b, int c, int d, float e) {
2      float e;
3      if (a == 0) {
4          return 0;
5      }
6      int x = 0;
7      if ((a==0) OR ((c == d) AND bug(a))) {
8          x=1;
9      }
10     e = 1/x;
11 }
12 }
```

6.5.1 Statement coverage

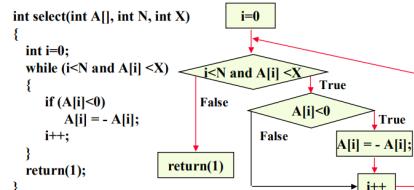
È un criterio di adeguatezza minimale, in quanto prevede che ogni *comando* debba essere eseguito almeno una volta. La coverage viene quindi misurata come: $\# \text{ comandi eseguiti} / \# \text{ comandi totali}$.

- * Statement coverage is a measure of the percentage of statements that have been executed by test cases.
- * Your objective should be to achieve 100% statement coverage through your testing.
- * In the example, consider `foo(0,0,0,0)`, as result, we had 42% ($5/12$) statement coverage
- * With `foo(1,1,1,1)` we achieve 100% because we execute now also lines 6-12

```

2 int foo (int a, int b, int c, int d, float e) {
3     float x;
4     if (a == 0)
5         return 0;
6     if (x == 0)
7         if ((c == 0) OR ((c == d) AND bug(a)))
8             x = 1/x;
9     else
10        e = 1/x;
11    return e;
12 }
```

(a) Statement testing



One test datum ($N=1, A[0]=7, X=9$) is enough to guarantee statement coverage of function select
Faults in handling positive values of $A[i]$ would not be revealed

(b) Esempio

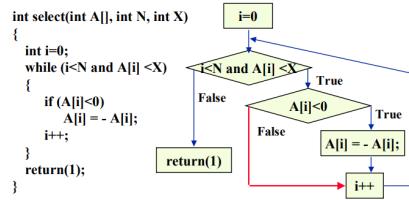
6.5.2 Branch coverage

È un criterio di adeguatezza che prevede di considerare tutte le *diramazioni*, ovvero di testare ogni guardia come vera e come falsa. La coverage viene quindi misurata come: $\# \text{ branch eseguiti} / \# \text{ branch totali}$.

Line #	Predicate	True	False
3	(a == 0)	Test Case 1 <code>foo(0, 0, 0, 0)</code> return 0	Test Case 2 <code>foo(1, 1, 1, 1)</code> return 1
7	((a==b) OR ((c==d) AND bug(a)))	Test Case 1 <code>foo(1, 1, 1, 1)</code> return 1	

- * With the previous used test cases we reach 75% of coverage
- * We have to consider `foo(1,2,1,2,1)` for reaching 100%
- * In many cases, the objective is to achieve 100% branch coverage in your testing, though in large systems only 75%-85% is practical. Only 50% branch coverage is practical in very large systems of 10 million source lines of code or more (Beizer, 1990)

(a) Branch testing



We must add a test datum ($N=1, A[0]=7, X=9$) to cover branch False of the if statement. Faults in handling positive values of $A[i]$ would be revealed. Faults in exiting the loop with condition $A[i] < X$ would not be revealed

(b) Esempio

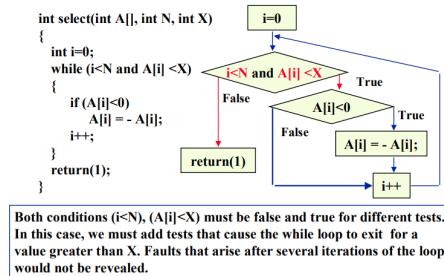
6.5.3 Condition coverage

È un criterio di adeguatezza che prevede di considerare tutte le *condizioni* che compongono una guardia, ovvero di testare tutte queste condizioni come vere e come false. La coverage viene quindi misurata come: $\# \text{ valori di verità delle condizioni eseguite} / 2 * \# \text{ condizioni totali}$.

- * We write Test Case 4 to address test (c==d) as true: foo(1, 2, 1, 1, 1), expected return value 1
- * To finalize our condition coverage, we must force bug(a) to be false. So we create Test Case 5, foo(3, 2, 1, 1, 1), expected return value "division by zero".

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, 1, 1, 1) return value 0	Test Case 3 foo(1, 2, 1, 1, 1) division by zero!
(c==d)	Test Case 4 foo(1, 2, 1, 1, 1) return value 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)	Test Case 4 foo(1, 2, 1, 1, 1) return value 1	Test Case 5 foo(3, 2, 1, 1, 1) division by zero!

(a) Condition testing



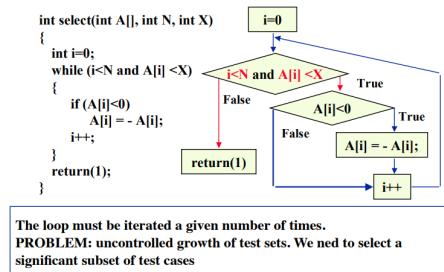
(b) Esempio

6.5.4 Path coverage

È possibile anche valutare tutti i percorsi di un CFG; tuttavia, nella pratica, essi sono molti di più dei branch. La coverage sarebbe quindi misurata come: $\# \text{cammini eseguiti} / \# \text{cammini totali}$.

- * The number of paths in a program with loops is unbounded
 - * the simple criterion is usually impossible to satisfy
- * For a feasible criterion: Partition infinite set of paths into a finite number of classes
- * Useful criteria can be obtained by limiting
 - * the number of traversals of loops
 - * the length of the paths to be traversed
 - * the dependencies among selected paths

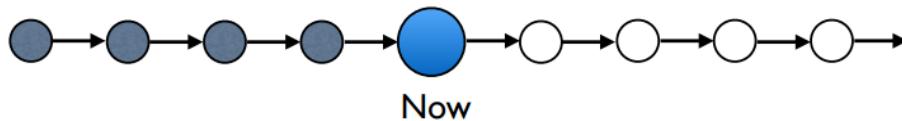
(a) Path testing nella pratica



(b) Esempio

6.6 Monitoring

Il *monitoring* tiene conto delle tracce di esecuzione presenti (sotto analisi) e passate (conosciute). Con traccia si intende l'esecuzione di un sistema modellata come una sequenza di stati.



6.6.1 Execution monitor

Sia Ψ l'insieme di tutte le possibili tracce sugli stati di un sistema e $\Sigma \subseteq \Psi$ l'insieme delle tracce di esecuzione di un fissato sistema S . Definiamo una politica di sicurezza p come un predicato su Ψ (ossia sull'insieme delle esecuzioni possibili) tale che:

- Un sistema S soddisfa p se $p(\Sigma)$ è vero.

- Se p vale sull'insieme delle esecuzioni di S , allora S soddisfa la politica descritta da p .
- p può essere verificata da un *execution monitor* se esiste un predicato \bar{p} sulle singole esecuzioni tale che $p(\Sigma) \Leftrightarrow \forall \sigma \in \Sigma. \bar{p}(\sigma)$, ossia p vale su Σ (per S) se e solo se ogni esecuzione di S soddisfa \bar{p} .

Inoltre, $\Gamma \subseteq \Psi$ è detta proprietà se la condizione $\sigma \in \Gamma$ dipende esclusivamente da σ , ossia per decidere se σ gode della proprietà Γ ($\sigma \in \Gamma$) posso osservare esclusivamente σ . Infine, può esistere un monitor per p solo se p è riscrivibile come proprietà sulle singole esecuzioni del sistema e se la verifica della proprietà si basa esclusivamente sul passato della traccia (in quanto un monitor non può vedere il futuro della traccia).

6.6.2 Proprietà di safety

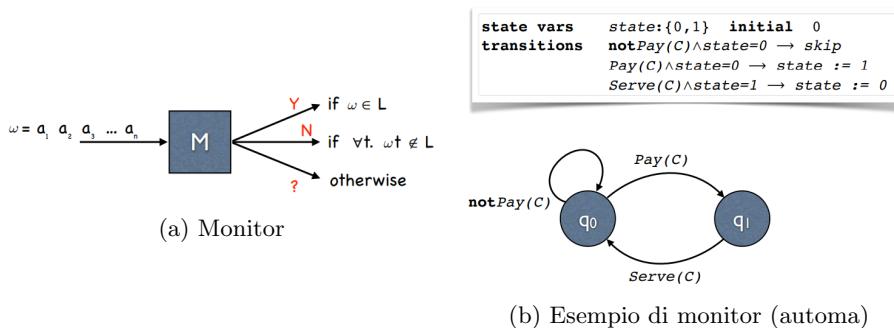
Una proprietà Γ è di *safety* (niente di cattivo avverrà) se per ogni esecuzione σ vale che: $\sigma \notin \Gamma \Rightarrow \exists \delta \leq \sigma. \forall \tau \in \Psi. \delta\tau \notin \Gamma$. Se una esecuzione non è safe vuol dire che in un certo punto della traccia succede qualcosa che fa violare in modo irreversibile Γ . Se una politica non è safety essa non ammette un execution monitor. Possiamo quindi distinguere tre classi di politiche:

- Politiche di *controllo degli accessi*, le quali sono proprietà di safety.
- Politiche di *flusso di informazione*, le quali non sono proprietà.
- Politiche di *disponibilità*, le quali sono proprietà ma non di safety.

6.6.3 Monitorare un linguaggio formale

Un *monitor* M per un linguaggio $L \subseteq A^*$ è un meccanismo che prende in input una sequenza di simboli $a \in A$, uno alla volta, ed emette un valore per il prossimo simbolo a_n . Il *problema del monitoring* può essere descritto come segue:

1. Dato un linguaggio logico L per descrivere la proprietà.
2. Data una formula $f \in Formule(L)$ per formalizzare la proprietà in L .
3. Costruire un monitor $M_{Ling(f)}$ per il linguaggio di f .

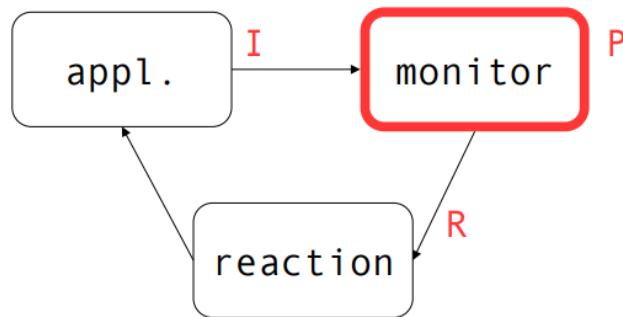


6.6.4 Sfide del monitoring

Le sfide del monitoring possono essere legate all'*strumentazione del codice* (aspetto importante nell'implementazione) o alla *specifica del monitor* stesso. Tra queste ultime troviamo:

- Linguaggio scelto.
- Facilità di creazione dalle specifiche.
- Efficienza.
- Integrazione dell'analisi statica con quella dinamica.
- Come agire in caso di violazione.

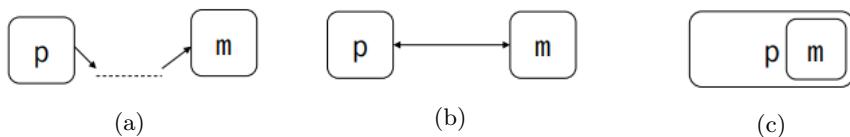
- * (I) Instrumentation language
- * (P) Property specification language
- * (R) Reaction language



6.6.5 Integrazione del monitor nell'applicazione

Per integrare il monitor nell'applicazione ci sono vari approcci:

- (a) *Offline*, sempre possibile.
- (b) *Online e outline*, in cui il monitor è eseguito in parallelo all'applicazione (la comunicazione tra di loro può essere sincrona o asincrona).
- (c) *Online e inline*, in cui il monitor è inglobato nel programma.



6.6.6 Violazione e validazione

L'*obiettivo* di un monitor è quello di confutare (rilevare violazioni) o di validare. In particolare, *violare* significa verificare che il sistema soddisfi la proprietà (cercando eventuali violazioni), mentre *validare* è un approccio che si usa quando è più facile descrivere la negazione (ciò che non vogliamo che avvenga).

6.6.7 Outline enforcement

Un monitor è un trasformatore di tracce, e quindi fa enforcement (forzare) perché agisce sull'esecuzione. Le possibili *operazioni* di un outline monitor sono:

- *ACCEPT*, ovvero l'accettazione dell'azione eseguita.
- *HALT*, ovvero lo stop dell'esecuzione.
- *SUPPRESS*, ovvero la soppressione dell'azione da eseguire.
- *INSERT*, ovvero l'inserimento di operazioni.

Inoltre, l'enforcement ha le seguenti *proprietà*:

- *Correttezza*, ossia tutto ciò che è osservabile soddisfa la politica.

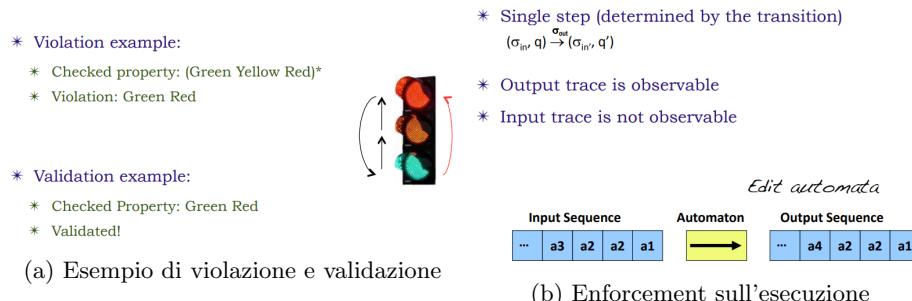
$$\forall \sigma_{in}. \exists q'. \exists \sigma_{out}. (\sigma_{in}, q_0) \Rightarrow (\text{empty}, q') \wedge P(\sigma_{out})$$

- *Trasparenza*, ossia la semantica delle esecuzioni che soddisfano la proprietà deve essere preservata.

$$P(\sigma_{in}) \Rightarrow \sigma_{in} \cong \sigma_{out}$$

Infine, esistono i seguenti *tipi* di enforcement monitor:

- *Conservativo*, se soddisfa solo la correttezza.
- *Effettivo*, se soddisfa entrambe le proprietà con qualche restrizione sulle sequenze valide.
- *Preciso*, se soddisfa entrambe le proprietà senza restrizioni.

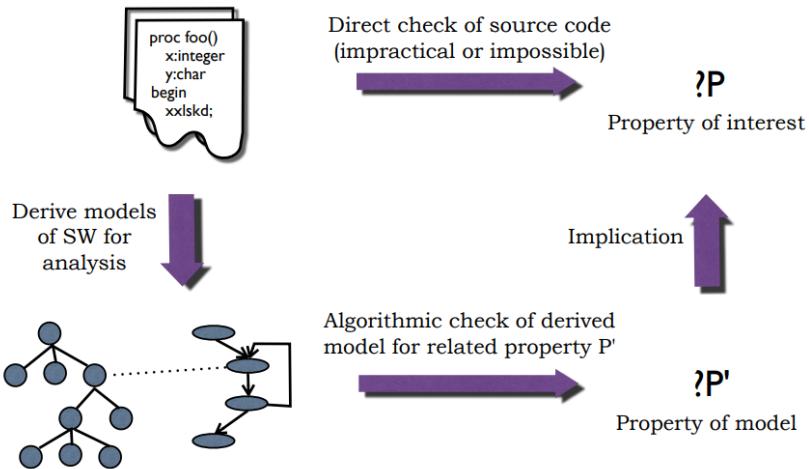


Capitolo 7

Model checking

7.1 Introduzione

Il *model checking* è un metodo per verificare che un sistema soddisfi una certa proprietà. È una tecnica automatica, è corretto e preciso, ma non può essere applicato a tutti i programmi bensì solo ai sistemi finiti sul modello del sistema.



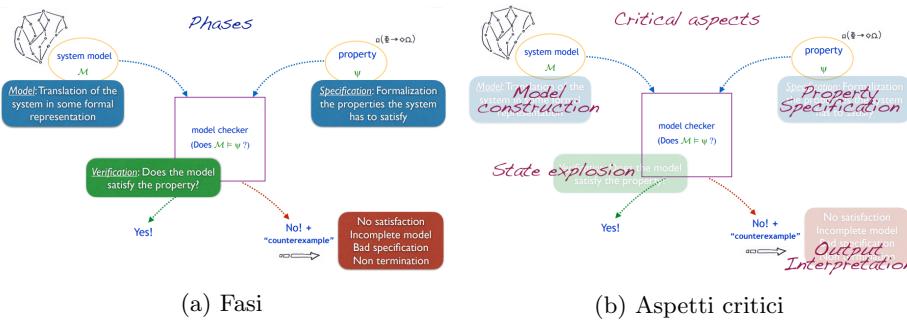
7.1.1 Fasi e aspetti critici

Le *fasi* del model checking sono:

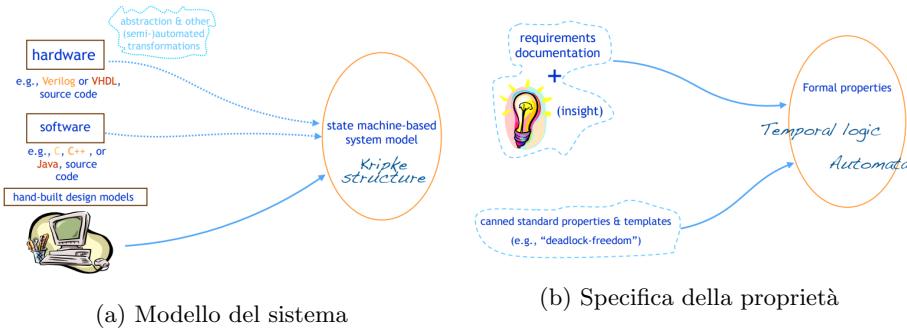
1. Rappresentazione del sistema.
2. Formalizzazione delle proprietà (attraverso un linguaggio formale).
3. Algoritmo di verifica.
4. Risposta (nessuna soddisfazione, modello incompleto, errore di specifica della proprietà, non terminazione).

Gli *aspetti critici* del model checking, invece, sono:

1. Costruzione del modello.
2. Specifica della proprietà.
3. Esplosione degli stati.
4. Interpretazione della risposta.



Riguardo alla *costruzione del modello* si ha che il sistema da modellare può essere hardware, software oppure un sistema informatico. L'obiettivo è quindi quello di tradurre tale sistema in una *struttura di Kripke*, ovvero in una macchina a stati finiti. Riguardo alla *specifica della proprietà*, infine, si usano i requisiti del sistema e altre informazioni contestuali per formalizzare la proprietà usando *logiche temporali* o automi.



7.2 Scelta del modello del sistema

Dobbiamo poter rappresentare uno *stato* del sistema che evolve nel tempo e dobbiamo rappresentare la manipolazione dei dati attraverso le *variabili*. Inoltre, vogliamo il *non-determinismo* (per rappresentare informazioni non note) e la *concorrenza* (per l'interazione tra processi). In definitiva, vogliamo che il modello permetta la progettazione di algoritmi efficienti.

7.2.1 Strutture di Kripke

Per avere tutto ciò che abbiamo appena descritto, dobbiamo costruire il modello attraverso le strutture di Kripke. Se AP è un insieme di predicati atomici, allora una *struttura di Kripke* viene definita come $M = \langle S, S_0, R, L \rangle$ dove:

- S è un insieme finito di stati.
- $S_0 \subseteq S$ è l'insieme degli stati iniziali.
- $R \subseteq S \times S$ è la relazione (totale) di transizione.

$$\forall s \in S. \exists s' \in S. R(s, s')$$

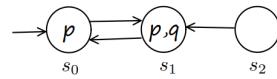
- $L : S \rightarrow \mathcal{P}(AP)$ è la funzione di etichettatura che associa ad ogni stato i predicati atomici che lo stato soddisfa.

Un cammino nella struttura M che parte da uno stato s è una sequenza infinita di stati $\pi = s_0 s_1 s_2 \dots$ tale che $s_0 = s$ e $\forall i > 0. R(s_i, s_{i+1})$.

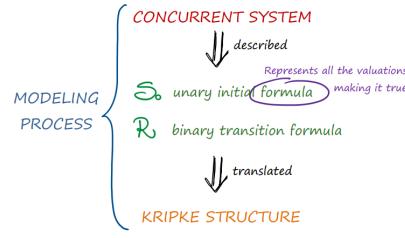
$$S = \{s_0, s_1, s_2\} \quad S_0 = \{s_0\} \quad AP = \{p, q\}$$

$$R = \{(s_0, s_1), (s_1, s_0), (s_2, s_1)\}$$

$$L = \{s_0 \mapsto \{p\}, s_1 \mapsto \{p, q\}, s_2 \mapsto \emptyset\}$$



(a) Esempio di struttura di Kripke



(b) Processo di modellazione

7.2.2 Esempio completo

Sia V un insieme di variabili (con V' copia di V) e D un insieme finito di valori per V . Gli stati $s \in S$ saranno quindi funzioni $s : V \rightarrow D$ e i predicati atomici AP saranno del tipo $v = d$ (dove $v \in V$ e $d \in D$) con $v = d$ vero in s se $s(v) = d$. Si consideri ora il sistema $x := (x + y) \bmod 2$ con $V = \{x, y\}$ e $D = \{0, 1\}$ che parte con x e y a 1. La corrispondente struttura di Kripke viene costruita seguendo i passaggi sottostanti.

1. Definizione di \mathcal{S}_0 : $\mathcal{S}_0(x, y) \equiv x = 1 \wedge y = 1$.
2. Definizione di \mathcal{R} : $\mathcal{R} \equiv x' = (x + y) \bmod 2 \wedge y' = y$.
3. Costruzione di $M = \langle S, S_0, R, L \rangle$ a partire dalle formule \mathcal{S}_0 e \mathcal{R} :

• S set of all valuations for V	$S = D \times D$	$\mathcal{S}_0 = \{(1, 1)\}$
• $\mathcal{S}_0 \subseteq S$ set of all the valuations satisfying \textcircled{S}	$\mathcal{R} = \{((1, 1), (0, 1)), ((0, 1), (1, 1)), ((1, 0), (1, 0)), ((0, 0), (0, 0))\}$	
• $\forall s, s' \in S. R(s, s') \text{ if } \forall v \in V. \textcircled{R}(s(v), s'(v))$	$L((1, 1)) = \{x=1, y=1\}$	$L((0, 1)) = \{x=0, y=1\}$
• $L(s) = \{v = d \in AP \mid s(v) = d\}$	$L((1, 0)) = \{x=1, y=0\}$	$L((0, 0)) = \{x=0, y=0\}$

The only possible path is $\gamma = (1, 1)(0, 1)(1, 1)(0, 1) \dots$

7.2.3 Granularità

Le *transizioni* devono essere atomiche, perché nessuno stato osservabile del sistema può risultare dall'esecuzione di una parte della transizione. Inoltre, transizioni troppo grossolane perdono stati osservabili, mentre transizioni troppo raffinate contengono stati che non sono raggiungibili.

$$\begin{aligned}
 & \text{Suppose now to define the system with: } V = \{x, y, R_1, R_2\} \\
 & \text{Consider the system with: } V = \{x, y\}, D = \text{Nat} \text{ and} \\
 & \quad S_0(x, y) \equiv x = 1 \text{ and } y = 2 \\
 & \quad R_1(x, y, R_1, R_2) \equiv \underbrace{R_1' = x}_{\alpha_0} \quad R_1(x, y, R_1, R_2) \equiv \underbrace{R_1' = R_1 + y}_{\alpha_1} \\
 & \quad R_2(x, y, R_1, R_2) \equiv \underbrace{R_2' = y}_{\beta_0} \quad R_2(x, y, R_1, R_2) \equiv \underbrace{R_2' = R_2 + x}_{\beta_1} \\
 & \quad R_3(x, y, R_1, R_2) \equiv \underbrace{x' = R_1}_{\alpha_2} \quad R_3(x, y, R_1, R_2) \equiv \underbrace{y' = R_2}_{\beta_2} \\
 & \quad \alpha \beta \Rightarrow x = 3 \text{ and } y = 5 \\
 & \quad \beta \alpha \Rightarrow x = 4 \text{ and } y = 3 \\
 & \quad \alpha_0 \beta_0 \alpha_1 \beta_1 \alpha_2 \beta_2 \Rightarrow x = 3 \text{ and } y = 3
 \end{aligned}$$

7.2.4 Dai programmi alle strutture di Kripke

Per passare da un programma P alla formula \mathcal{R} , etichettiamo il programma con dei valori che identificano il punto di programma, ossia: entry point m , exit point m' e la *trasformazione* dell'etichetta $P \rightarrow P^{\mathcal{L}}$. Quest'ultimo valore viene definito per induzione sulla struttura del nostro linguaggio imperativo:

- Se P è un comando terminale, $P^{\mathcal{L}} = P$.
- Se $P = P_1; P_2$, $P^{\mathcal{L}} = P_1^{\mathcal{L}}; l'' : P_2^{\mathcal{L}}$.
- Se $P = \text{if } b \text{ then } P_1 \text{ else } P_2$, $P^{\mathcal{L}} = \text{if } b \text{ then } l_1 : P_1^{\mathcal{L}} \text{ else } l_2 : P_2^{\mathcal{L}}$.
- Se $P = \text{while } b \text{ do } P_1$, $P^{\mathcal{L}} = \text{while } b \text{ do } l_1 : P_1^{\mathcal{L}}$.

Sui valori l varia la variabile pc . Introducendo la notazione

$$same(Y) = \bigwedge_{y \in Y} (y' = y)$$

abbiamo che $S_0(V, pc) \equiv pre(V) \wedge pc = m$ (dove pre indica le condizioni iniziali del programma). Definiamo quindi il predicato che descrive l'*esecuzione* $\mathcal{C}(l, P, l')$ ricorsivamente sulla grammatica del linguaggio:

- Assegnamento $\mathcal{C}(l, v := e, l') \equiv pc = l \wedge pc' = l' \wedge v' = e \wedge same(V \setminus \{v\})$.
- Comando vuoto $\mathcal{C}(l, \text{skip}, l') \equiv pc = l \wedge pc' = l' \wedge same(V)$.
- Composizione sequenziale $\mathcal{C}(l, P_1; l'': P_2, l') \equiv \mathcal{C}(l, P_1, l'') \vee \mathcal{C}(l'', P_2, l')$.
- Condizionale $\mathcal{C}(l, \text{if } b \text{ then } l_1 : P_1 \text{ else } l_2 : P_2, l') \equiv (pc = l \wedge pc' = l_1 \wedge b \wedge same(V)) \vee (pc = l \wedge pc' = l_2 \wedge \neg b \wedge same(V)) \vee \mathcal{C}(l_1, P_1, l') \vee \mathcal{C}(l_2, P_2, l')$.
- While $\mathcal{C}(l, \text{while } b \text{ do } l_1 : P_1, l') \equiv (pc = l \wedge pc' = l_1 \wedge b \wedge same(V)) \vee (pc = l \wedge pc' = l' \wedge \neg b \wedge same(V)) \vee \mathcal{C}(l_1, P_1, l)$.

Consider the program $P = x:=e; \text{while } x < 10 \text{ do } x := x + 2 \text{ endw};$

$$\begin{aligned}
 P^L &= m: x := 3; 1: \text{while } x < 10 \text{ do } 2: x := x + 2 \text{ endw}; m' \\
 S_0(p_c, x) &\equiv p_c = m \text{ and } x = 0 \\
 R &\rightarrow C(m, P^L, m') = C(m, x := 3, 1) \text{ or } C(1, \text{while } x < 10 \text{ do } 2: x := x + 2, m') \\
 &= (p_c = m, p_c' = 1, x' = 3) \text{ or } (p_c = 1, p_c' = 2, x' < 10) \text{ or} \\
 &\quad (p_c = 1, p_c' = m', x' < 10) \text{ or } C(2, x := x + 2, 1), \\
 &= (p_c = m, p_c' = 1, x' = 3) \text{ or } (p_c = 1, p_c' = 2, x' < 10) \text{ or} \\
 &\quad (p_c = 1, p_c' = m', \neg x < 10) \text{ or } (p_c = 2, p_c' = 1, x' = x + 2)
 \end{aligned}$$

7.3 Logiche temporali

Una *logica temporale* è un formalismo usato per descrivere le sequenze di transizioni tra stati in un sistema. Esistono diverse logiche temporali che differiscono tra loro nella semantica degli operatori che forniscono.

7.3.1 CTL*

La logica temporale su cui ci concentriamo è la cosiddetta CTL*. Le *formule* CTL* descrivono le proprietà degli alberi di esecuzione, ovvero alberi in cui lo stato iniziale è uno stato della struttura Kripke. Tali formule sono composte da:

- *Path quantifiers* (quantificatori), che descrivono la struttura di ramificazione nell'albero di esecuzione.
- *Operatori temporali*, che descrivono le proprietà di un cammino che attraversa l'albero.

Path quantifiers

$\mathbf{A}_f \rightarrow f$ holds for *all* computation paths

$\mathbf{E}_f \rightarrow f$ holds for *some* computation path

Temporal operators

$\mathbf{X}_f \rightarrow f$ holds in the *next* state of the path

$\mathbf{F}_f \rightarrow f$ holds in a *future* state of the path

$\mathbf{G}_f \rightarrow f$ holds in *each* state of the path

$f \mathbf{U}_g \rightarrow f$ holds in *each* state *before* g holds

$f \mathbf{R}_g \rightarrow$ Logical dual of until

7.3.2 Sintassi e semantica

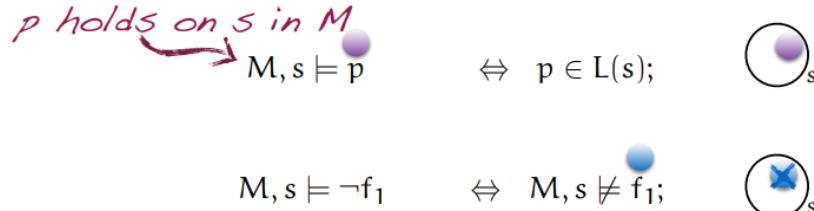
In CTL* esistono due tipi di formule: *formule di stato* (che valgono in uno specifico stato) e *formule di cammino* (che valgono lungo un cammino).

- | | |
|--|---|
| $\text{AP} = \text{set of atomic proposition}$
$\left\{ \begin{array}{l} \cdot p \in \text{AP}, \text{ then } p \text{ is a state formula} \\ \cdot \text{If } f \text{ and } g \text{ are state formulas then } \textcolor{red}{\Box}f, \textcolor{blue}{\Diamond}g, \textcolor{red}{\Box}\Diamond g, \textcolor{blue}{\Diamond}\Box g \text{ are state formulas} \\ \cdot \text{If } f \text{ is a path formula, then } \textcolor{red}{E}f \text{ and } \textcolor{blue}{A}f \text{ are state formulas} \end{array} \right.$ | $\text{AP} = \text{set of atomic proposition}$
$\left\{ \begin{array}{l} \cdot p \text{ is a state formula then } p \text{ is a path formula} \\ \cdot \text{If } f \text{ and } g \text{ are path formulas then } \textcolor{red}{\Box}f, \textcolor{blue}{\Diamond}g, \textcolor{red}{\Box}\Diamond g, \textcolor{blue}{\Diamond}\Box g \text{ and } \textcolor{red}{F}f, \textcolor{blue}{G}g, \textcolor{red}{F}\Diamond g \text{ and } \textcolor{blue}{G}\Box g \text{ are path formulas} \end{array} \right.$ |
|--|---|

(a) Formule di stato

(b) Formule di cammino

Per definire la *semantica* di CTL* facciamo riferimento ad una struttura di Kripke del tipo $M = \langle S, S_0, R, L \rangle$, in cui un cammino è una sequenza infinita di stati tale per cui ogni coppia di stati consecutivi è nella relazione R . Usando la seguente notazione



definiamo quindi la semantica degli operatori di CTL* come:

Semantics	Semantics
$M, s \models f_1 \vee f_2 \Leftrightarrow M, s \models f_1 \text{ or } M, s \models f_2;$ 	$M, \pi \models \Diamond f_1 \Leftrightarrow s \text{ is the first state of } \pi \text{ and } M, s \models f_1;$
$M, s \models f_1 \wedge f_2 \Leftrightarrow M, s \models f_1 \text{ and } M, s \models f_2;$ 	$M, \pi \models \neg g_1 \Leftrightarrow M, \pi \not\models g_1;$
$M, s \models E g_1 \Leftrightarrow \text{Exists a path } \pi \text{ from } s \text{ s.t. } M, \pi \models g_1;$ 	$M, \pi \models g_1 \vee g_2 \Leftrightarrow M, \pi \models g_1 \text{ or } M, \pi \models g_2;$
$M, s \models A g_1 \Leftrightarrow \text{Each path } \pi \text{ from } s \text{ s.t. } M, \pi \models g_1;$ 	$M, \pi \models g_1 \wedge g_2 \Leftrightarrow M, \pi \models g_1 \text{ and } M, \pi \models g_2;$
Semantics	Semantics
$M, \pi \models X g_1 \Leftrightarrow M, \pi^1 \models g_1;$ 	$M, \pi \models g_1 U g_2 \Leftrightarrow \exists k \geq 0 \text{ s.t. } M, \pi^k \models g_2 \text{ for each } 0 \leq j < k, M, \pi^j \models g_1;$
$M, \pi \models F g_1 \Leftrightarrow \text{Exists } k \geq 0 \text{ s.t. } M, \pi^k \models g_1;$ 	$M, \pi \models g_1 R g_2 \Leftrightarrow \text{For each } j \geq 0, \text{ if for each } i < j \text{ } M, \pi^i \neq g_1 \text{ then } M, \pi^j \models g_2;$
$M, \pi \models G g_1 \Leftrightarrow \text{For each } i \geq 0 M, \pi^i \models g_1;$ 	

7.3.3 Altre logiche temporali

Da CTL* derivano altre due logiche temporali (con diverso potere espressivo):

- CTL (*branching-time*), in cui gli operatori quantificano sui cammini che partono da uno stato, ovvero non si applicano operatori temporali a formule di cammino.
- LTL (*linear-time*), in cui gli operatori descrivono gli eventi sui singoli cammini che partono da uno stato.

* CTL is obtained restricting the syntax of path formulas as follows

- If f and g are state formulas then $\neg f$, $\text{F}f$, $\text{G}f$, $\text{F}\Box g$ and $\text{F}\Diamond g$ are path formulas

* LTL is obtained by restricting path formulas as follows

- $p \in AP$, then p is a path formula
- If f and g are path formulas then $\neg f$, $\text{F}f$, $\text{G}f$, $\text{F}\Box g$ and $\text{F}\Diamond g$ are path formulas

(a) Sintassi delle formule

* LTL (no CTL)



* CTL*



$\exists \cdot$



* CTL (no LTL)



$\exists \cdot$



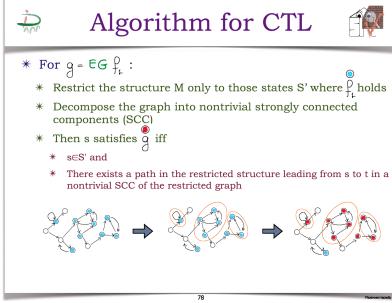
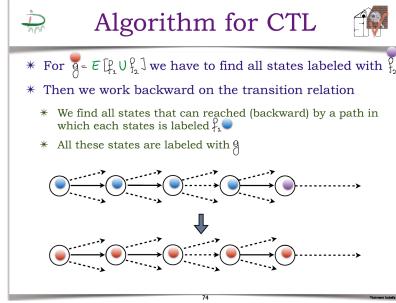
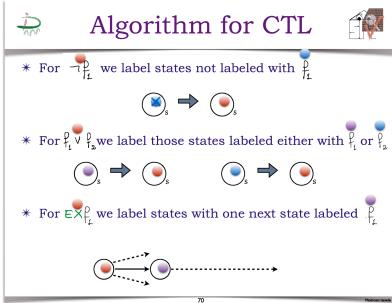
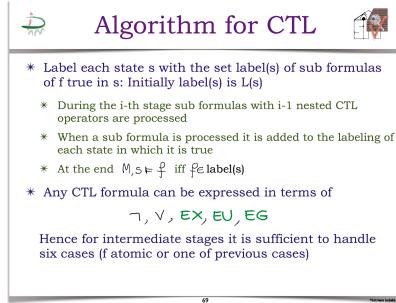
$\exists \cdot$



(b) Esempi

7.3.4 Algoritmi per CTL

Data una struttura di Kripke $M = \langle S, S_0, R, L \rangle$, il model checking ha l'obiettivo di trovare l'insieme di tutti gli stati in S che soddisfano una certa formula temporale f : $\{s \in S \mid M, s \models f\}$.



Capitolo 8

Slicing

8.1 Intuizione

Si tratta di una tecnica di decomposizione che trasforma il programma originale cancellandone alcune istruzioni che non hanno alcun effetto sulle *variabili di interesse nei punti di interesse*. Lo *slice* è il programma trasformato secondo il criterio di *slicing*, il quale descrive i seguenti parametri di interesse:

- V , ossia l'insieme delle variabili di interesse.
- n , ossia i punti di interesse del programma.

```
1  read(n);
2  i := 1;
3  s := 0;
4  p := 1;
5  while (i <= n) do
6      s := s + i;
7      p := p * i;
8      i := i + 1;
9  od
10 write(s)
11 write(p);
```

8.2 Motivazioni

Ci sono diversi motivi per i quali effettuare il program slicing:

- *Program debugging.*
- *Testing*, in quanto lo slicing riduce i costi del regression testing dopo una trasformazione del codice.
- *Parallelizzazione.*
- *Comprensione di un programma*, in quanto effettuare lo slicing aiuta a comprendere come viene eseguito un programma e quali variabili verranno modificate nei vari percorsi.
- *Mantenimento del software*, ossia modificare il codice senza avere side effects.

8.3 Tipi di program slicing

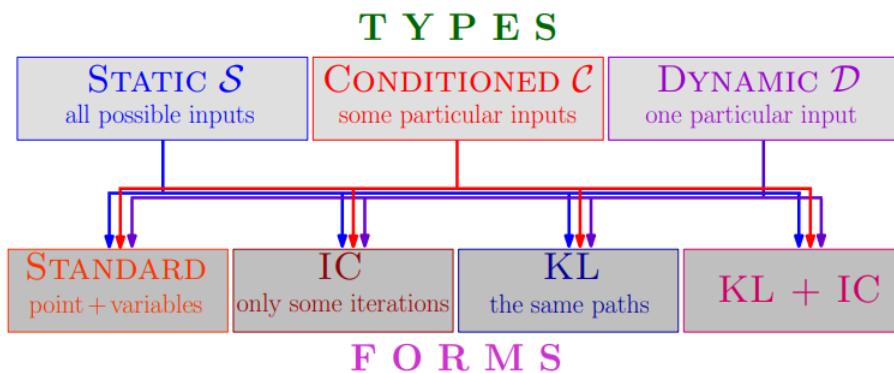
Esistono diversi tipi di program slicing:

- *Static slicing*, in cui l'equivalenza tra programma originale e slice deve (implicitamente) essere valida per ogni possibile input.
- *Conditioned slicing*, il quale preserva il significato del programma originale per un insieme di input che soddisfa una particolare condizione ϕ .
- *Dynamic slicing*, il quale considera una particolare computazione (e dunque un particolare input) in modo da preservare il significato del programma unicamente per quell'input.

8.4 Forme di program slicing

Esistono diverse forme di program slicing:

- *Standard*, che considera un singolo punto di programma rispetto ad un set di variabili.
- *Korel & Laski* (KL), ovvero una forma di slicing molto forte in cui il programma e lo slice (avendo la stessa semantica operazionale) devono seguire cammini identici (il cammino seguito dallo slice deve essere un sotto-cammino dell'esecuzione originale).
- *Iteration Count* (IC), la quale richiede che lo slice e il programma concordino solo ad una certa iterazione k di un punto di programma n (cioè quando il comando al punto di programma n viene eseguito per la k -esima volta) e non per tutte le iterazioni dello stesso punto di programma.
- *KL-IC* (combinazione delle precedenti), la quale richiede che il programma e lo slice seguano cammini identici e concordino solo ad una certa iterazione di un punto di programma.

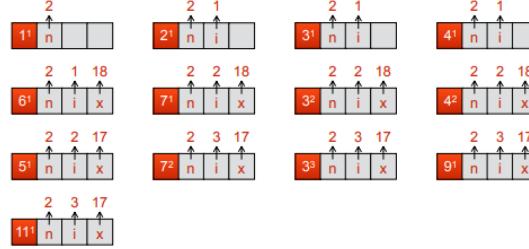


8.5 Esempi

Consideriamo il seguente programma:

```
1 read(n);
```

```
2 i := 1;
3 while (i <= n) do {
4   if (i mod 2 = 0) {
5     x := 17; }
6   else { x := 18; }
7   i := i + 1;
8 }
9 if (i = 1) {
10  x = 17; }
11 write(x);
```



8.5.1 Standard slicing

Vediamo due esempi di *standard slicing* per x al punto 11:

```
1 read(n);
2 i := 1;
3 while (i <= n) do {
4   if (i mod 2 = 0) {
5     x := 17; }
6   else { x := 18; } 6
7   i := i + 1;
8 }
9 if (i = 1) {
10  x = 17; }
11 write(x);
```



(a) Eliminazione della riga 6

```
1 read(n);
2 i := 1;
3 while (i <= n) do {
4   if (i mod 2 = 0) {
5     x := 17; }
6   else { x := 18; } 6
7   i := i + 1;
8 }
9 if (i = 1) {
10  x = 17; }
11 write(x);
```



(b) Eliminazione delle righe da 3 a 8

8.5.2 KL slicing

Vediamo due esempi di *KL slicing* per x al punto 11:

```
1 read(n);
2 i := 1;
3 while (i <= n) do {
4   if (i mod 2 = 0) {
5     x := 17; }
6   else { x := 18; } 6
7   i := i + 1;
8 }
9 if (i = 1) {
10  x = 17; }
11 write(x);
```



(a) Eliminazione della riga 6

```
1 read(n);
2 i := 1;
3 while (i <= n) do {
4   if (i mod 2 = 0) {
5     x := 17; }
6   else { x := 18; } 6
7   i := i + 1;
8 }
9 if (i = 1) {
10  x = 17; }
11 write(x);
```



The only difference is in point 6 not present in the slice
The difference is in point 10 which is present in the slice

(b) Eliminazione delle righe da 3 a 8

8.5.3 IC slicing

Vediamo un esempio di *IC slicing* per x alla seconda iterazione del punto 4:

```

1 read(n);
2 i := 1;
3 while (i <= n) do {
4   if (i mod 2 = 0) {
5     x := 17; }           2 2 17
6   else { x := 18; }      2 2 17
7   i := i + 1;           2 2 17
8 }
9 if (i = 1) {           2 2 17
10  x = 17; }             2 2 17
11 write(x);
```

(a) Programma originale

```

1 read(n);
2 i := 1;
3 while (i <= n) do {
4   if (i mod 2 = 0) {
5     x := 17; }           2 2 17
6   else { x := 18; }      2 2 17
7   i := i + 1;           2 2 17
8 }
9 if (i = 1) {           2 2 17
10  x = 17; }             2 2 17
11 write(x);
```

The variable x has a different value

(b) Eliminazione della riga 6

Appendice A

Domande di riepilogo

A.1 Semantica e approssimazioni (capitoli 1-4)

Quando la semantica astratta sovra-approssima quella concreta? E quando la sotto-approssima?

- Sovra-approssimazione: se la semantica astratta gode della proprietà, allora anche la semantica concreta la soddisfa.
- Sotto-approssimazione: se la semantica astratta non gode della proprietà allora, anche la semantica concreta non la soddisfa.

Come astrae la semantica delle tracce la collecting semantics?

La collecting semantics astrae la semantica delle tracce approssimando insiemi di tracce in tracce di insiemi.

Cosa osserva la collecting semantics?

Colleziona gli stati raggiunti in ogni punto di programma.

In che cosa consiste il processo di astrazione?

Il processo di astrazione consiste nel:

- Caratterizzare le proprietà che vogliamo osservare.
- Caratterizzare come computare la semantica sulle proprietà osservate.

Come può essere formalizzato un dominio astratto per garantire correttezza?

Per garantire correttezza un dominio astratto può essere formalizzato come:

- Una coppia di funzioni (astrazione e concretizzazione) che formano una connessione di Galois.
- Una coppia di funzioni (astrazione e concretizzazione) che formano una inserzione di Galois.
- Una funzione sul domino concreto che sia monotona, estensiva e idempotente.
- Una famiglia di Moore del dominio concreto.

Quando un dominio astratto A è più preciso di un altro dominio astratto B?

Un dominio astratto A è più preciso di un altro dominio astratto B se A contiene B insiemisticamente.

Dire quali affermazioni riguardanti il dominio dei domini astratti sono false.

- Il greatest lower bound di domini astratti è l'unione insiemistica dei domini: **F**
- Il least upper bound di domini astratti è l'intersezione insiemistica dei domini: **V**
- Il dominio più astratto è il dominio contenente solo \top , ovvero l'elemento massimo del dominio concreto: **V**
- Il dominio astratto più concreto è il dominio contenente solo \perp , ovvero l'elemento minimo del dominio concreto: **F**

Dire quali affermazioni sulla precisione sono false.

- Le due definizioni di correttezza (forward e backward) non sono equivalenti: **F**
- Le due definizioni di completezza (forward e backward) non sono equivalenti: **V**
- Se un dominio è forward completo per una funzione allora è anche backward completo: **F**
- Esistono domini che non sono forward completi e non sono neanche backward completi: **V**
- Se un dominio è backward corretto allora è anche forward corretto: **V**

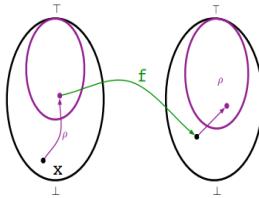
Quali sono le fasi del processo di astrazione del calcolo semantico su domini infiniti e non ACC?

Le fasi del processo di astrazione del calcolo semantico su domini infiniti e non ACC sono:

1. Calcolo iterativamente la semantica astratta per punto fisso mediante l'uso di un acceleratore di calcolo del punto fisso.
2. Raggiungo un punto post-fisso.
3. Applico un'operazione di raffinamento per raggiungere un punto fisso (che sarà un'approssimazione del minimo punto fisso).

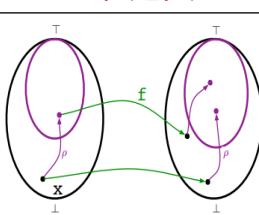
Associa l'immagine con la sua definizione.

Correttezza forward



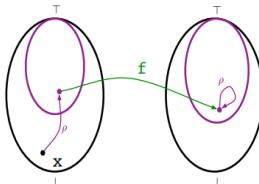
Mostra che astraendo l'output si può solo essere meno precisi

Correttezza backward



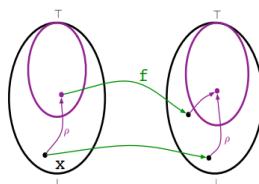
Mostra che astraendo l'input si può solo essere meno precisi

Completezza forward



L'approssimazione dell'output non genera imprecisione

Completezza backward



L'approssimazione dell'input non genera imprecisione

A.2 Analisi di data-flow (capitolo 5)

Dire quali affermazioni sull'analisi statica sono false.

- Estrae proprietà semantiche dei programmi in modo approssimato: **V**
- Estrae precisamente proprietà semantiche dei programmi: **F**
- Estrae precisamente proprietà sintattiche dei programmi: **V**

Quali sono le fasi dell'analisi di un programma mediante CFG?

Le fasi dell'analisi di un programma mediante CFG sono:

1. Costruire il CFG.
2. Raccogliere l'informazione sui blocchi (*Gen* e *Kill*).
3. Risolvere le equazioni di punto fisso per ogni punto di programma.

Quali sono i passi di un'analisi forward?

I passi di un'analisi forward sono:

1. Fisso l'informazione per il blocco di ingresso del CFG.
2. Fisso l'informazione di base per il calcolo del punto fisso (possible = \emptyset , definite = \top).
3. Calcolo l'informazione cercata in input ad ogni blocco come la combinazione (possible = \cup , definite = \cap) dell'informazione in output dei blocchi predecessori.
4. Calcolo l'informazione in output ad ogni blocco come l'informazione generata dal blocco (*Gen*) unito l'informazione propagata dal blocco, ovvero quella in input da cui tolgo quella distrutta (*Kill*) dal blocco.
5. Ripeto i passi precedenti fino a quando raggiungo il punto fisso delle informazioni in input ai blocchi.

Quali sono i passi di un'analisi backward?

I passi di un'analisi backward sono:

1. Fisso l'informazione per il blocco di uscita del CFG.
2. Fisso l'informazione di base per il calcolo del punto fisso (possible = \emptyset , definite = \top).
3. Calcolo l'informazione cercata in output ad ogni blocco come la combinazione (possible = \cup , definite = \cap) dell'informazione in input dei blocchi successori.
4. Calcolo l'informazione in input ad ogni blocco come l'informazione generata dal blocco (*Gen*) unito l'informazione propagata dal blocco, ovvero quella in output da cui tolgo quella distrutta (*Kill*) dal blocco.
5. Ripeto i passi precedenti fino a quando raggiungo il punto fisso delle informazioni in output ai blocchi.

Da cosa è garantita la terminazione nel framework monotono?

La terminazione nel framework monotono è garantita da:

- Funzione di trasferimento monotona su dominio ACC.
- Funzione di trasferimento monotona su dominio finito.

Quali sono i passi di un'analisi nel framework monotono?

I passi di un'analisi nel framework monotono sono:

1. Definire l'informazione astratta che vogliamo osservare, ovvero le proprietà da analizzare che formano il dominio dell'analisi.
2. Definire la semantica astratta, ovvero la funzione semantica di trasferimento (trasformazione) definita sulle proprietà astratte da analizzare.
3. Costruire un sistema di disequazioni (una per ogni punto di programma) la cui soluzione migliore descrive la trasformazione dell'informazione astratta in ogni punto di programma.
4. Trovare per punto fisso la soluzione migliore del sistema di disequazioni, che coincide con la soluzione del corrispondente sistema di equazioni.
5. Fornire la proprietà invariante di ogni punto di programma come la soluzione (MFP) del sistema di equazioni costruito.

Dov'è che l'analisi può perdere precisione?

L'analisi può perdere precisione:

- Nel dover considerare tutti cammini del CFG.
- Nel calcolare la semantica in modo locale al singolo blocco.
- Nel guardare a proprietà semantiche guardando solo la sintassi.

Dire quali affermazioni sulla soluzione dell'analisi sono corrette.

- La soluzione più precisa è la IDEAL, cioè quella che considera solo la semantica dei cammini eseguibili (ma che non è calcolabile): **V**
- La soluzione più precisa è la MOP, cioè quella che considera la semantica di tutti i cammini del CFG (e che è potenzialmente calcolabile): **F**
- La soluzione calcolabile più precisa è la MOP nel caso in cui la funzione di trasferimento è monotona e distributiva: **V**
- La soluzione meno precisa è la MFP, cioè quella che calcola per punto fisso la proprietà per ogni punto di programma combinando le semantiche locali al blocco (e che è sempre calcolabile): **V**
- La soluzione MOP è sempre più precisa della soluzione MFP: **F**

A.3 Analisi statica non distributiva (capitolo 5)

Cosa riguarda l'analisi non distributiva?

L'analisi non distributiva riguarda cosa calcola il programma.

Cosa calcola l'analisi delle costanti per ogni punto di programma?

L'analisi delle costanti calcola, per ogni punto di programma, l'insieme delle variabili che hanno precisamente sempre lo stesso valore (ugualmente calcolato) costante ogni volta che la computazione raggiunge il punto di programma.

Dire quali affermazioni sul dominio per l'analisi delle costanti sono corrette.

- È un dominio finito: **F**
- È un dominio ad altezza finita (ACC): **V**
- Ha come elementi tutti e soli (\top e \perp esclusi) gli insiemi di cardinalità 1: **V**
- Permette solo di determinare se le variabili sono costanti o meno: **F**

Cosa calcola l'analisi degli intervalli in ogni punto di programma?

L'analisi degli intervalli calcola, in ogni punto di programma, il range dei valori che una variabile può avere nel punto di programma.

Dire quali affermazioni sul dominio per l'analisi degli intervalli sono corrette.

- È un dominio finito: **F**
- È un dominio ad altezza finita (ACC): **F**
- Può avere catene ascendenti infinite: **V**
- Contiene il dominio delle costanti: **V**
- Ha come least upper bound l'unione insiemistica: **F**

Dire quali affermazioni sul widening sono corrette.

- È un operatore commutativo che sovra-approssima il least upper bound del reticolo completo: **F**
- È un acceleratore per raggiungere un'approssimazione del minimo punto fisso del calcolo: **V**
- È un acceleratore che permette di raggiungere più velocemente il minimo punto fisso del calcolo: **F**
- È un operatore non commutativo che sovra-approssima il least upper bound del reticolo: **V**
- È un operatore la cui definizione dipende fortemente dagli elementi del reticolo: **V**

Dire quali affermazioni sul narrowing sono corrette.

- È un operatore che permette di recuperare parte dell'informazione persa calcolando mediante l'uso del widening restando tra i punti fissi del calcolo: **V**
- È un operatore che permette di raffinare il risultato del widening facendo raggiungere il minimo punto fisso: **F**
- È un operatore non commutativo: **V**
- È un operatore che può non restituire un punto fisso: **F**

Appendice B

Esempi di domande d'esame

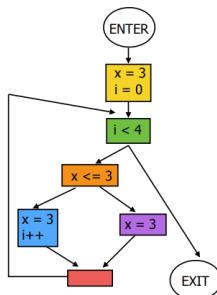
B.1 Domande sull'analisi statica

Definire cosa è un Control Flow Graph (CFG) e descriverne la procedura di costruzione. Dare inoltre un esempio.

Il *Control Flow Graph* rappresenta il flusso di controllo di un programma. Il CFG viene generato dalla sintassi del programma e permette di capire la struttura del codice. Viene utilizzato per effettuare testing, debugging e per individuare codice morto. Un CFG è un grafo diretto $G = \langle N, E \rangle$ dove n indica i nodi che corrispondono ai punti di programma (basic blocks) ed e indica gli archi (passi di computazione). Per ogni arco $e = (n_i, n_j)$: n_i è detto predecessore di n_j ; n_j è detto successore di n_i . Per ogni nodo n : $Pred(n)$ è l'insieme dei predecessori di n ; $Succ(n)$ è l'insieme dei successori di n ; un branch node è un nodo che ha più di un successore; un join node è un nodo che ha più di un predecessore.

I *basic blocks* sono una sequenza massimale di comandi aventi un singolo entry point, un singolo exit point e nessun branch interno. Essi si costruiscono individuando i blocchi leader, ossia: il primo statement del programma; ogni statement che è target di un punto di branch; ogni statement che segue immediatamente un punto di branch. Quindi, un basic block è la sequenza di comandi che si trova tra un leader (incluso) e un altro leader (escluso).

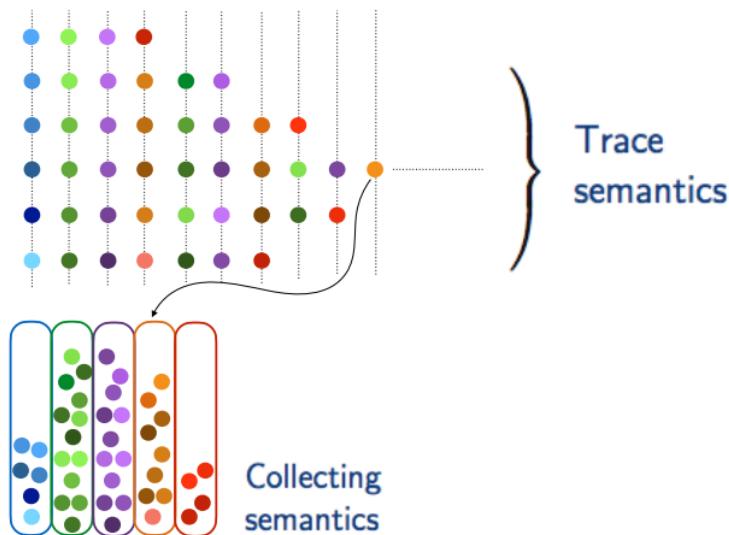
La *procedura di costruzione* di un CFG prevede di dividere il codice in basic blocks e poi di collegare questi ultimi tra loro in corrispondenza di: salti non condizionali (goto); branch condizionali; passaggi di controllo tra blocchi.



Definire il concetto di collecting semantics e dire in che modo essa astrae la semantica operazionale delle tracce.

La semantica delle tracce è l'insieme di tutte le possibili esecuzioni di un programma. Siccome tali esecuzioni sono potenzialmente infinite, è necessaria una semantica ancora più astratta. La *collecting semantics* colleziona gli stati raggiunti in ogni punto di programma e quindi astrae la semantica delle tracce approssimando insiemi di tracce in tracce di insiemi (collezioni di tutte le possibili esecuzioni). L'idea alla base della collecting semantics è quindi quella di rimuovere la sequenza temporale e mantenere solo le informazioni date dal punto di programma considerato.

Trattandosi di un'*astrazione*, non è più possibile risalire alle tracce di esecuzione del programma. Inoltre, la collecting semantics differisce dalla semantica delle tracce se nel codice sono presenti dei cicli in quanto, per iterazioni successive di tali cicli, la collecting semantics porrà i relativi stati sempre nello stesso insieme (perché si perde l'ordine temporale degli stati). In sostanza si collezionano gli stati in base a proprietà simili. Se paradossalmente tutti gli stati differissero gli uni dagli altri, allora raggiungerei un'equivalenza con la semantica delle tracce.



Definire le soluzioni MOP, MFP e IDEAL descrivendo formalmente le differenze.

La *MFP* (Maximum Fixed Point) è la soluzione che combina i valori dell'analisi quando il CFG ha dei nodi in cui convergono due o più percorsi. La soluzione MFP approssima la soluzione MOP. Infatti, la *MOP* (Merge Over all Paths) è una soluzione più precisa rispetto alla MFP ($MOP \sqsupseteq MFP$) poiché combina i valori dell'analisi di tutti i percorsi del CFG (dopo averli attraversati tutti). La soluzione MOP è quella che vorremmo ma, in generale, essa non è computabile in quanto i percorsi possono essere infiniti. Le soluzioni MOP e MFP coincidono ($MOP = MFP$) quando tutte le funzioni di trasferimento (abstract edge effect) sono distributive e dunque è possibile calcolare la soluzione MOP attraverso l'algoritmo iterativo del punto fisso.

La *IDEAL* è la soluzione migliore (più precisa), ma non è computabile. Infatti a differenza della MOP, la IDEAL prende in considerazione solamente i percorsi possibili, ossia quelli che verranno sicuramente attraversati da qualche esecuzione. La soluzione IDEAL calcola quindi i valori alla fine di ogni possibile percorso di esecuzione e ne fa la combinazione. Il problema della non computabilità sta nel fatto che non è possibile conoscere in maniera statica i cammini eseguibili. Qualsiasi soluzione convergente più piccola della IDEAL sarà quindi meno precisa, in quanto avrà considerato un numero di vincoli maggiore di quello necessario.

Riassumendo, la soluzione MOP prende in considerazione tutti i cammini che dall'entry vanno in un determinato blocco. La perdita di precisione sta nel fatto che possono essere considerati anche cammini che non verranno mai eseguiti. Nella MOP avrà quindi i cammini della IDEAL più alcuni cammini spuri ($MOP \sqsubseteq IDEAL$). Combinando queste informazioni con quanto già osservato nel confronto tra MOP e MFP si ottiene la seguente relazione:

$$MFP \sqsubseteq MOP \sqsubseteq IDEAL$$

Definire il concetto di widening e per cosa viene utilizzato.

L'idea alla base del *widening* è quella di accelerare la convergenza del calcolo del punto fisso, anche a costo di una possibile perdita di precisione, spostandosi nel calcolo dei punti post-fissi. Il *widening* è un operatore non commutativo, la cui definizione dipende fortemente dagli elementi del reticolo, che sovra-approxima il minimo punto fisso (least upper bound del reticolo).

Si può cercare di migliorare l'approssimazione del punto fisso calcolata con l'uso del *widening*, raffinando i punti post-fissi mediante il *narrowing*. Il *narrowing*, infatti, è un operatore che permette di recuperare parte di quell'informazione che è stata persa con il *widening*, restando tra i punti fissi del calcolo.

B.2 Domande sull'analisi dinamica

Cos'è lo slicing? Cosa distingue uno slicing dinamico da uno statico o condizionale?

Lo *slicing* è una tecnica di decomposizione che trasforma il programma originale cancellandone alcune istruzioni che non hanno alcun effetto sulle variabili di interesse nei punti di interesse. Tali parametri di interesse definiscono il criterio di slicing secondo cui il programma viene poi trasformato. Ci sono diversi motivi per i quali effettuare il program slicing tra cui: debugging, testing, parallelizzazione, comprensione di un programma, mantenimento del software.

Esistono diversi *tipi* di program slicing:

- Slicing statico, in cui l'equivalenza tra programma originale e slice deve essere valida per ogni possibile input.
- Slicing condizionale, il quale preserva il significato del programma originale per un insieme di input che soddisfa una particolare condizione.
- Slicing dinamico, il quale considera una particolare computazione (e dunque un particolare input) in modo da preservare il significato del programma unicamente per quell'input.

Esistono infine diverse *forme* di program slicing:

- Standard, che considera un singolo punto di programma rispetto ad un set di variabili.
- Korel & Laski (KL), in cui il cammino seguito dallo slice deve essere un sotto-cammino dell'esecuzione del programma originale.
- Iteration Count (IC), la quale richiede che lo slice e il programma originale concordino solo ad una certa iterazione di un certo punto di programma.
- KL-IC, combinazione delle precedenti.

Definire il concetto di monitoring. In cosa consiste? Che caratteristiche formali deve avere una proprietà per essere monitorabile?

Il monitoring tiene conto delle tracce di esecuzione presenti (sotto analisi) e passate (conosciute). Con traccia si intende l'esecuzione di un sistema modellata come una sequenza di stati. Un monitor M per un linguaggio $L \subseteq A^*$ è quindi un meccanismo che prende in input una sequenza di simboli $a \in A$, uno alla volta, ed emette un valore per il prossimo simbolo a_n . Il *monitor* è un trasformatore di tracce che ha come obiettivo quello di confutare (rilevare violazioni) o di validare. In particolare, violare significa verificare che il sistema soddisfi la proprietà (cercando eventuali violazioni), mentre validare è un approccio che si usa quando è più facile descrivere la negazione (cioè che non vogliamo che avvenga).

Per integrare il monitor nell'applicazione ci sono vari approcci: offline (sempre possibile); online e outline (il monitor è eseguito in parallelo all'applicazione); online e inline (il monitor è inglobato nel programma). Le possibili *operazioni* (trasformazioni) di un outline monitor sono: ACCEPT (accettazione dell'azione eseguita); HALT (stop dell'esecuzione); SUPPRESS (soppressione dell'azione da eseguire); INSERT (inserimento di operazioni).

Per essere monitorabile, una proprietà deve essere di *safety* (niente di cattivo avverrà). Se una esecuzione σ non è safe vuol dire che in un certo punto della traccia Ψ succede qualcosa che fa violare in modo irreversibile la proprietà Γ :

$$\sigma \notin \Gamma \Rightarrow \exists \delta \leq \sigma. \forall \tau \in \Psi. \delta\tau \notin \Gamma$$

Definire testing e debugging, descrivendone le principali differenze.

Il *testing* consiste, principalmente, nell'esecuzione di un programma su un campione di dati (molto piccolo) passato come input. Si tratta quindi di una tecnica di analisi dinamica, in cui ovviamente occorre eseguire il programma. Inoltre, vale il concetto di inaccuratezza ottimistica, ovvero si suppone che il programma sia corretto se supera una serie di test scelti precedentemente. Più formalmente, il testing può essere definito come un processo di ricerca di bug/erri/difetti del software. Infatti, quando cerchiamo e troviamo un errore possiamo dire con assoluta certezza che il programma non funziona correttamente. Tra le varie tipologie di errori troviamo: mistake (errore umano); fault (difetto o dato non corretto presente nel codice); failure (fallimento nell'eseguire una funzionalità attesa); error (misura della differenza tra il comportamento atteso e quello osservato).

L'obiettivo del *debugging*, invece, è l'identificazione, l'isolamento e la risoluzione dei problemi/bug. Questa operazione si può svolgere durante la fase di sviluppo del software oppure in una fase apposita in cui vengono sistemati i bug riportati dopo i test.

Credits

Basato sulle slide fornite dalla *prof.ssa Isabella Mastroeni*

Vedere anche qui: <https://github.com/davbianchi/Dispense-Info-Univr/tree/master/magistrale/analisi-di-sistemi>

Repository github: <https://github.com/zampierida98/UniVR-informatica>

Indirizzo e-mail: zampieri.davide@outlook.com