

Indice

1	Interpretazione astratta	5
1.1	Introduzione	5
1.2	Connessione di Galois	6
1.3	Famiglie di Moore	7
1.4	Upper closure operator	7
1.5	Reticolo delle interpretazioni astratte	7
1.6	Computazioni astratte e concrete	8
1.7	Correttezza	8
1.8	Completezza	9
1.9	Accelerazione della convergenza	9
1.9.1	Widening	9
1.9.2	Narrowing	10
1.10	Linguaggio e semantica	10
1.10.1	Collecting Semantics	10
1.11	Control-Flow-Graph (CFG)	11
1.11.1	Notazione dei CFG	12
2	Analisi Statica	15
2.1	Introduzione	15
2.2	Analisi sul CFG	15
2.3	Soluzioni MFP - MOP - IDEAL	17
2.4	Data Flow Analysis	18
2.4.1	Available Expressions	18
2.4.2	Very Busy Expressions	19
2.4.3	Liveness	20
2.4.4	Reaching Definition	21
2.4.5	Riepilogo	22
2.5	Problemi Non-Distributivi	23
2.5.1	Costanti	23
2.5.2	Segni	23
2.5.3	Intervalli	24
3	Analisi Dinamica	25
3.1	Testing	25
3.2	Debugging	25
3.3	Program Slicing	25

Capitolo 1

Interpretazione astratta

1.1 Introduzione

Lo scopo è quello di trovare un'approssimazione di una semantica $\langle P \rangle$ di $\llbracket P \rrbracket$ tale per cui valgano:

- *correttezza*: $\llbracket P \rrbracket \subseteq \langle P \rangle$;
- *decidibilità*: $\langle P \rangle \subseteq Q$ è decidibile (Q è un insieme di semantiche che soddisfa la proprietà di interesse).

Se entrambe le proprietà sono soddisfatte, allora vale che

$$(\langle P \rangle \subseteq Q) \Rightarrow (\llbracket P \rrbracket \subseteq Q)$$

La semantica è data da una coppia $\langle D, f \rangle$ dove D è una coppia $\langle D, \leq_D \rangle$ rappresentante un dominio semantico e $f : D \rightarrow D$ è una funzione di trasferimento con una soluzione a punto fisso.

Dato un oggetto concreto, definiamo:

- un **oggetto astratto** come una rappresentazione matematica sovra-approssimata del corrispondente concreto;
- un **dominio astratto** come un insieme di oggetti astratti con delle operazioni astratte, che approssimano quelle concrete;
- una funzione di **astrazione** α che mappa oggetti concreti in oggetti astratti;
- una funzione di **concretizzazione** γ che mappa oggetti astratti in oggetti concreti.

La caratteristica peculiare delle astrazioni è che solo alcune proprietà vengono osservate con esattezza, le altre vengono solo approssimate. In sostanza, dato un dominio astratto A , gli elementi di A sono osservati con esattezza, gli altri sono approssimati o l'informazione è persa del tutto.

Proprietà. L'insieme delle proprietà $\mathcal{P}(\Sigma)$ di oggetti in Σ è l'insieme di elementi che gode di quella proprietà. Questo insieme di proprietà costituisce un reticolo completo

$$\langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \cup, \cap, \neg \rangle$$

dove:

- \subseteq è l'implicazione logica;
- Σ è **true**;
- \cup è la disgiunzione (oggetti che godono di P o di Q appartengono a $P \cup Q$);
- \cap è la congiunzione (oggetti che godono di P e di Q appartengono a $P \cap Q$);
- \neg è la negazione (oggetti che non godono di P stanno in $\Sigma \setminus P$).

Direzione dell'astrazione. Quando si approssima una proprietà concreta $P \in \mathcal{P}(\Sigma)$ usando una proprietà astratta \bar{P} , deve essere stabilito un criterio per definire quando \bar{P} è un'approssimazione di P .

Si distinguono quindi i seguenti casi:

- approssimazione *da sopra*: $P \subseteq \bar{P}$;
- approssimazione *da sotto*: $P \supseteq \bar{P}$.

Dato un oggetto o , si vuole quindi sapere se $o \in P$:

$$P \supseteq \bar{P} : \begin{cases} \text{"Si"} & o \in \bar{P} \\ \text{"Non lo so"} & o \notin \bar{P} \end{cases} \quad P \subseteq \bar{P} : \begin{cases} \text{"No"} & o \notin \bar{P} \\ \text{"Non lo so"} & o \in \bar{P} \end{cases}$$

Migliore approssimazione. Definiamo come *migliore approssimazione* di una proprietà P in A il glb delle over-approximation di P in A , ossia:

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

1.2 Connessione di Galois

Imponiamo il vincolo che α e γ siano monotone, allora concludiamo che:

- $\gamma \circ \alpha : C \rightarrow C$ è **estensiva**: $\gamma(\alpha(c)) \geq c$;
- $\alpha \circ \gamma : A \rightarrow A$ è **riduttiva**: $\alpha(\gamma(a)) \leq a$.

Le definizioni qui sopra dicono rispettivamente che:

- α perde informazione, e γ non la può recuperare;
- γ non perde informazione.

Definizione 1.2.0.1 (Connessione di Galois). *Dati due poset $\langle A, \leq_A \rangle$ e $\langle C, \leq_C \rangle$, e due funzioni monotone $\alpha : C \rightarrow A$ e $\gamma : A \rightarrow C$, diciamo che $\langle C, \alpha, \gamma, A \rangle$ è una connessione di Galois se:*

- $\forall c \in C : c \leq_C \gamma(\alpha(c))$
- $\forall a \in A : \alpha(\gamma(a)) \leq_A a$

Se inoltre vale che $\forall a \in A : \alpha(\gamma(a)) = a$, allora $\langle C, \alpha, \gamma, A \rangle$ è un'inserzione di Galois.

Una connessione e un'inserzione di Galois sono rappresentate rispettivamente come

$$C \xleftarrow[\alpha]{\gamma} A \quad C \xleftarrow[\alpha]{\gamma} \gg A$$

La funzione α è detta *aggiunta sinistra*, mentre la funzione γ è detta *aggiunta destra*.

Teorema 1.2.0.1. *Data una connessione di Galois $C \xleftarrow[\alpha]{\gamma} A$, sono equivalenti:*

- $C \xleftarrow[\alpha]{\gamma} \gg A$;
- α è *suriettiva*;
- γ è *iniettiva*.

Inoltre, dati due domini astratti, non esistono due coppie (α, γ) che formino una connessione di Galois; quindi la connessione di Galois tra due domini è **unica**, e le funzioni sono identificabili attraverso:

$$\begin{aligned} \alpha(c) &= \bigwedge \{ a \in A \mid c \leq_C \gamma(a) \} \\ \gamma(a) &= \bigvee \{ c \in C \mid \alpha(c) \leq_A a \} \end{aligned}$$

1.3 Famiglie di Moore

Definizione 1.3.0.1 (Famiglia 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, ipotizzando che ogni proprietà concreta abbia una migliore astrazione $\bar{P} \in A$, implica che il dominio A è una famiglia di Moore.

1.4 Upper closure operator

Definizione 1.4.0.1 (Upper closure operator). *Una funzione $f : P \rightarrow P$ su un poset $\langle P, \leq_P \rangle$ è un upper closure operator (uco) se soddisfa le seguenti proprietà:*

- *estensività:* $\forall x \in P : x \leq_P \rho(x)$
- *monotonia:* $\forall x, y \in P : (x \leq_P y) \Rightarrow (\rho(x) \leq_P \rho(y))$
- *idempotenza:* $\forall x \in P : \rho(x) = \rho(\rho(x))$

I lower closure operator sono definiti in modo duale, specificando che ρ deve essere *riduttiva*, ovvero che $\forall x \in P : x \geq_P \rho(x)$.

Teorema 1.4.0.1. *Data una connessione di Galois $C \xleftrightarrow[\alpha]{\gamma} A$ si ha che $\gamma \circ \alpha$ è un uco e $\alpha \circ \gamma$ è un lco.*

Teorema 1.4.0.2. *$C \xleftrightarrow[\alpha]{\gamma} A$ se e solo se A è isomorfo¹ ad una Moore family di C .*

Teorema 1.4.0.3. *Sia $\rho \in \text{uco}(C)$. Allora $\forall A \simeq \rho(C)$ si ha che $\exists \alpha, \gamma : C \xleftrightarrow[\alpha]{\gamma} A$*

1.5 Reticolo delle interpretazioni astratte

I vari domini astratti possono essere comparati sulla base della loro precisione. In generale si può dire che un dominio astratto A_1 è più preciso di A_2 (indicato attraverso $A_1 \sqsubseteq A_2$) quando

$$\forall a_2 \in A_2, \exists a_1 \in A_1 \quad \text{tali che} \quad \gamma_1(a_1) = \gamma_2(a_2)$$

ovvero quando

$$\gamma(A_2) \subseteq \gamma(A_1)$$

Collegando agli uco, possiamo dire che

$$A_1 \sqsubseteq A_2 \Leftrightarrow \rho_1 \sqsubseteq \rho_2 \Leftrightarrow \rho_2(C) \subseteq \rho_1(C)$$

Definizione 1.5.0.1 (Reticolo delle int. astratte). *Se C è un reticolo completo o un cpo, allora*

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

è un reticolo completo dove $\forall \rho, \eta \in \text{uco}(C), \{\rho_i\}_{i \in I} \subseteq \text{uco}(C)$ e $x \in C$:

- $\rho \sqsubseteq \eta \Leftrightarrow \forall y \in C. \rho(y) \leq \eta(y) \Leftrightarrow \eta(C) \subseteq \rho(C)$
- $\left(\bigsqcap_{i \in I} \rho_i \right)(x) = \bigwedge_{i \in I} \rho_i(x)$
- $\left(\bigsqcup_{i \in I} \rho_i \right)(x) = x \Leftrightarrow \forall i \in I. \rho_i(x) = x$
- $\lambda x. \top, \lambda x. x$ sono rispettivamente *top* e *bottom*.

¹Con isomorfismo si intendono reticoli con la stessa struttura.

1.6 Computazioni astratte e concrete

Definizione 1.6.0.1 (Correttezza). *Data un'inserzione di Galois $C \xleftrightarrow[\alpha]{\gamma} A$, una funzione concreta $f : C \rightarrow C$ e una funzione astratta $f^\# : A \rightarrow A$ diciamo che $f^\#$ è un'approssimazione corretta di f se*

$$\forall c \in C : \alpha(f(c)) \leq_A f^\#(\alpha(c)) \quad \text{backward}$$

o equivalentemente

$$\forall a \in A : f(\gamma(a)) \leq_C \gamma(f^\#(a)) \quad \text{forward}$$

Rinforzando la definizione e imponendo uguaglianza si perde l'equivalenza delle due espressioni sopra.

Definizione 1.6.0.2 (Completezza). *Data un'inserzione di Galois $C \xleftrightarrow[\alpha]{\gamma} A$, una funzione concreta $f : C \rightarrow C$ e una funzione astratta $f^\# : A \rightarrow A$ diciamo che $f^\#$ è:*

- *backward-completa per f se $\forall c \in C : \alpha(f(c)) = f^\#(\alpha(c))$*
- *forward-completa per f se $\forall a \in A : f(\gamma(a)) = \gamma(f^\#(a))$*

La definizione rappresenta una situazione ideale in cui non si ha perdita di precisione durante il calcolo astratto. Inoltre la backward-completeness lavora sull'astrazione dell'input delle operazioni, la forward-completeness sull'output.

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

- $\rho \in \text{uco}(C)$ è backward-completo per f se $\rho \circ f = \rho \circ f \circ \rho$
- $\rho \in \text{uco}(C)$ è forward-completo per f se $f \circ \rho = \rho \circ f \circ \rho$

Inoltre quando ρ è sia backward che forward-completo allora vale che $\rho \circ f = f \circ \rho$.

Teorema 1.6.0.1. *Data $C \xleftrightarrow[\alpha]{\gamma} A$, una funzione concreta $f : C \rightarrow C$ e una funzione astratta $f^\# : A \rightarrow A$ allora*

$$\forall c \in C : \alpha(f(c)) \leq_A f^\#(\alpha(c)) \Leftrightarrow \alpha \circ f \circ \gamma \sqsubseteq f^\#$$

Definizione 1.6.0.3 (Best correct approximation). *Data $C \xleftrightarrow[\alpha]{\gamma} A$ e una funzione concreta $f : C \rightarrow C$ allora $\alpha \circ f \circ \gamma : A \rightarrow A$ è la best correct approximation di f in A .*

1.7 Correttezza

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

$$\forall c \in C : \alpha(f(c)) \leq_A f^\#(\alpha(c))$$

oppure, equivalentemente:

$$\forall a \in A : f(\gamma(a)) \leq_C \gamma(f^\#(a))$$

Nel processo di astrazione è ammessa una perdita di informazioni, ciò non è possibile nel processo di concretizzazione, dunque possiamo dire che se $c \in C$. Possiamo dire che $\alpha(c)$ è l'elemento astratto più preciso che rappresenta c .

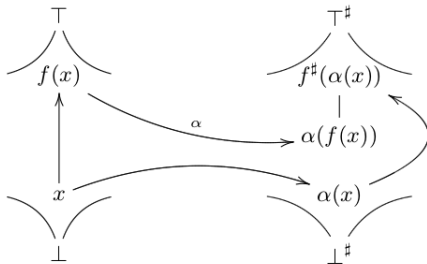


Figura 1.1: Condizione di correttezza: $\alpha(f(c)) \leq_A f^\#(\alpha(c))$

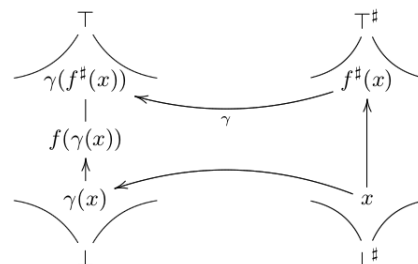


Figura 1.2: Condizione di correttezza: $f(\gamma(a)) \leq_C \gamma(f^\#(a))$

1.8 Completezza

Consideriamo $C \xleftrightarrow[\alpha]{\gamma} A$, una funzione concreta $f : C \rightarrow C$ e una funzione astratta $f^\# : A \rightarrow A$. Possiamo dire che:

- $f^\#$ è backward-complete per f se: $\forall c \in C : \alpha(f(c)) = f^\#(\alpha(c))$;
- $f^\#$ è forward-complete per f se: $\forall a \in A : f(\gamma(a)) = \gamma(f^\#(a))$.

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

- La **B**-completezza considera l'astrazione sull'output delle operazioni e non si accumula nessuna perdita di precisione astraendo in p gli argomenti di f ;
- La **F**-completezza considera l'astrazione sull'input delle operazioni e non si accumula nessuna perdita di precisione approssimando il risultato della funzione f calcolata in p .

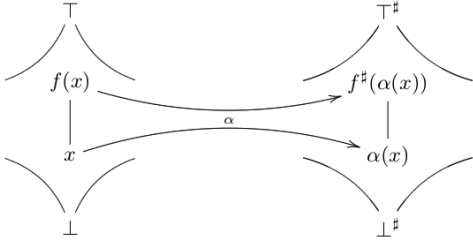


Figura 1.3: Condizione di **B**-completezza

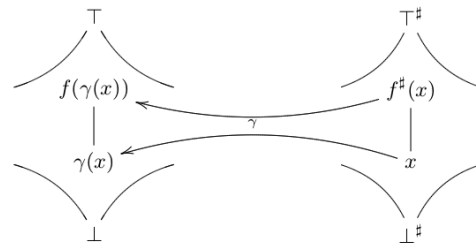


Figura 1.4: Condizione di **F**-completezza

1.9 Accelerazione della convergenza

1.9.1 Widening

Un widening

$$\nabla : P \times P \rightarrow P$$

su un poset $\langle P, \leq_P \rangle$ è una funzione che soddisfa:

- $\forall x, y \in P : x \sqsubseteq (x \nabla y) \wedge y \sqsubseteq (x \nabla y)$
- per ogni catena ascendente $x_0 \sqsubseteq x_1 \sqsubseteq \dots \sqsubseteq x_n$ la catena definita come $y_0 = x_0, \dots, y_{n+1} = y_n \nabla x_{n+1}$ non è strettamente crescente.

Dato che in interpretazione astratta è necessario garantire/accelerare la convergenza, viene usato il widening (che si sostituisce al least upper bound), dal momento che anche il calcolo astratto può divergere. Il risultato di un widening è un post-puntofisso di F^∇ , ovvero una sovrapprossimazione del punto fisso più piccolo di $\text{lfp} F$.

Ad esempio, il widening su intervalli funziona come segue:

$$[a, b] \nabla [c, d] = [e, f] \quad \text{tale che}$$

$$e = \begin{cases} -\infty & \text{se } c < a \\ a & \text{altrimenti} \end{cases} \quad \text{e } f = \begin{cases} +\infty & \text{se } b < d \\ b & \text{altrimenti} \end{cases}$$

1.9.2 Narrowing

Dato che il widening raggiunge un post-fixpoint, può capitare che si abbiano eccessive perdite di informazione, in questo caso viene usato il narrowing.

Definizione 1.9.2.1. *Il narrowing è una funzione $\triangle : P \times P \rightarrow P$ tale che:*

- $\forall x, y \in \mathcal{P} : y \leq x \implies y \leq x \triangle y \leq x$
- *Per ogni catena discendente $x_0 \geq x_1 \geq \dots$, la catena discendente $y_0 = x_0, \dots, y_{i+1} = y_i \triangle x_{i+1}$ non è strettamente decrescente.*

Per gli intervalli il narrowing funziona come segue:

$$[a, b] \triangle [c, d] = [e, f] \quad \text{tale che}$$

$$e = \begin{cases} c & \text{se } a = -\infty \\ a & \text{altrimenti} \end{cases} \quad \text{e } f = \begin{cases} d & \text{se } b = +\infty \\ b & \text{altrimenti} \end{cases}$$

1.10 Linguaggio e semantica

Introduciamo in questa sezione il linguaggio che verrà usato nel resto della dispensa e la sua semantica.

Statement	Codice
Variabili	x
Espressioni aritmetiche	e
Assegnamenti	$x \leftarrow e$
Lettura da memoria	$x \leftarrow M[e]$
Scrittura in memoria	$M[e]_1 \leftarrow e_2$
Condizionali	if (e) S₁ else S₂
Salto non condizionale	goto L

La memoria M è vista come un array arbitrariamente grande dove i valori possono essere inseriti e letti.

- x e $M[e]$ sono contenitori di valori;
- il contenuto di $M[e]$ non è visibile fino alla valutazione di e ;
- x è solamente il nome tramite cui accedere al contenitore associato.

1.10.1 Collecting Semantics

È l'insieme dei comportamenti osservabili nella semantica operativa. La *Collecting Semantics* è il punto di partenza per ogni tipo di analisi (non ne esiste una universale).

La **trace semantics** di un programma accumula informazioni temporali riguardo l'esecuzione: una traccia tiene conto dell'ordine in cui i *program states* sono raggiunti durante l'esecuzione. Le tracce analizzate possono essere dei seguenti tipi:

- L'insieme di tutti i discendenti dello stato iniziale.
- L'insieme di tutti i discendenti dello stato iniziale che può raggiungere uno stato finale.
- Lo stato di tutte le tracce finite dallo stato iniziale.
- L'insieme di tutte le tracce infinite e finite dallo stato iniziale ecc.

Però non sempre siamo interessati alle informazioni temporali ma solamente agli invarianti presenti ad ogni *program point*. Questi invarianti possono essere astratti dalle informazioni temporali attraverso la **collecting semantics**.

Più formalmente, un invariante del programma P al punto di programma l è una qualsiasi proprietà $I \in P$ (store) che è presente ogni talvolta che l viene raggiunto.

La *collecting semantics* di P è semplicemente l'associazione tra i vari *program point* e le corrispondenti invarianti ben precise.

Lo stato di input non è noto al momento della compilazione, quindi vengono collezionati tutti gli stati raggiungibili da tutti i possibili ingressi del programma.

Si tratta di una collezione di stati che possono apparire su alcune tracce nei diversi *program point*. Trattandosi di un'astrazione, non è più possibile risalire alle tracce di esecuzione del programma conoscendo solamente i vari *program states*.

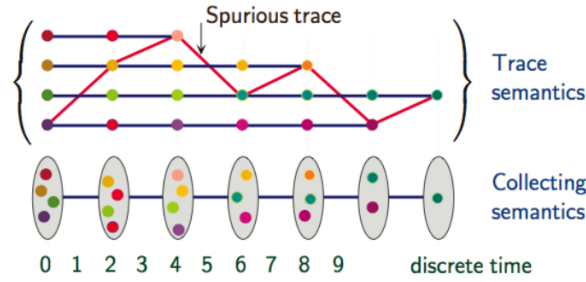


Figura 1.5: Esiste la traccia rossa? *Trace semantics*: NO; *collecting semantics*: NON LO SO.

1.11 Control-Flow-Graph (CFG)

E costituito da:

- **nodi**: corrispondono ai *program points*;
- **archi**: passi di computazione etichettati con la corrispondente azione; sono della forma $K = (u, lab, v)$, dove u è il nodo sorgente, v è il nodo di destinazione e lab è l'etichetta.

Test	$NonZero(e)$ or $Zero(e)$
Assegnamenti	$x \leftarrow e$
Lettura da memoria	$x \leftarrow M[e]$
Scrittura in memoria	$M[e]_1 \leftarrow e_2$
Statement vuoto	;

Ogni passo di computazione della semantica operativa trasforma gli stati del programma:

$$(\rho, \mu) \text{ dove } \rho : Var \rightarrow int \text{ e } \mu : \mathbb{N} \rightarrow int$$

- La funzione ρ mappa le variabili del programma al loro valore attuale;
- la funzione μ mappa ogni cella dell'array al suo contenuto nelle celle di memoria.

Una *computazione* è un percorso che va da un nodo di partenza u e termina in un nodo v . Il percorso è un insieme di archi del *CFG*. La trasformazione dello stato è data dalla composizione degli effetti degli archi.

$$\llbracket \pi \rrbracket = \llbracket k_n \rrbracket \circ \dots \circ \llbracket k_1 \rrbracket$$

Il **Control Flow Graph** è generato dalla sintassi del programma ed è utile per capire la struttura del codice.

Viene utilizzato per effettuare *debugging*, *testing* ed individuare *dead code*.

Basic Block. Sequenza massima di statements consecutivi con un singolo entry point, un singolo exit point e nessun branch interno.

I *basic block* si identificano facilmente poiché iniziano con un *leader* che può essere dei seguenti tipi:

- l'*entry point* del programma (il primo statement);
- ogni statement che è target di branch (condizionali o non condizionali) che contengono dei *GoTo*
- ogni statement che segue un branch (condizionale o non condizionale) o un *return*.

Dopo aver diviso il codice in *basic block* (individuati tramite i *leader* di ciascun blocco), essi verranno collegati dagli archi, in corrispondenza di:

- *GoTo* non condizionali;
- branch condizionali / archi multipli;
- flusso di programma (il controllo passa ad un altro blocco se non ci sono branch alla fine).

Se non c'è un unico *entry-node* n_0 ed un unico *exit-node* n_f , si aggiungono *dummy nodes* e gli archi necessari (nessun arco entrante in n_0 e nessun arco uscente da n_f).

1.11.1 Notazione dei CFG

Dato un $CFG = \langle N, E \rangle$:

- Se c'è un arco $n_i n_j \in E$:
 - n_i è predecessore di n_j ;
 - n_j è un successore di n_i .
- Per ogni nodo $n \in N$:
 - $\text{textitPred}(n)$: è l'insieme dei predecessori di n ;
 - $\text{textitSucc}(n)$: 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;

Depth First Traversal. Il CFG è un grafo diretto e con radice (*entry-node*). Deve essere attraversato partendo dalla radice ed esplorando in profondità il più possibile ciascun ramo prima di fare *backtracking*.

E' possibile costruire uno **spanning tree** per il grafo che contenga tutti i nodi, tale che:

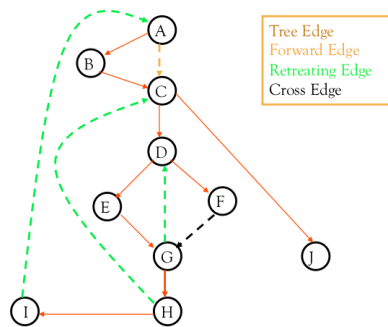
- ci sia un percorso dalla radice ad ogni nodo che sia raggiungibile nel grafo originale;
- non devono esserci cicli.

I nodi vengono numerati nell'ordine in cui verranno visitati.



Classificazione degli archi. Dato un arco $x \rightarrow y$ in un CFG, esso sarà:

- un **arco avanzante**: se x è predecessore di y nell'albero;
 - **tree edge**: se è parte dello spanning tree;
 - **forward edge**: se non è parte dello spanning tree e x è predecessore di y nell'albero.
- un **arco all'indietro**: se y è un predecessore di x nell'albero;
- un **cross edge**: se non è parte dello spanning tree e nessun nodo è predecessore dell'altro.



Extended Basic Block. Insieme massimo di nodi che non contiene nessun nodo di join (oltre all'entry node). Ha un solo ingresso e più uscite.

Natural Loop. Un Loop è un insieme di nodi strettamente connessi. Ha un unico ingresso (l'unico modo per visitarlo). Deve contenere un unico arco all'indietro per ripercorrere il loop.

Un loop che non contiene altri loops è un *inner loop*.

Per trovare un loop all'interno di un grafo è sufficiente cercare gli archi all'indietro ($n \rightarrow d$). Per costruire un loop si aggiunge d , si aggiunge n (se $n \neq d$), si considera ogni nodo $m \neq d$ all'interno del loop (inserendo tutti i predecessori di m).

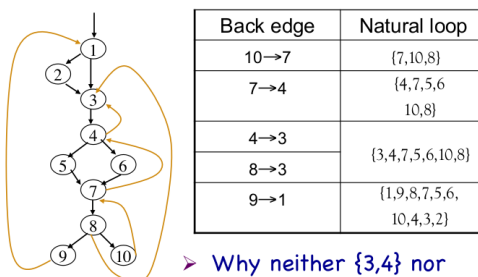


Figura 1.6: Natural loops example

Dominance. Un nodo d domina un nodo n se ogni percorso dall'entry node del grafo fino a n passa attraverso d ($d \text{ dom } n$).

- $Dom(n)$: l'insieme dei dominatori del nodo n ;
- ogni nodo domina se stesso: $n \in Dom(n)$;
- il nodo d domina strettamente n se $d \in Dom(n)$ e $d \neq n$;
- *Dominance-based loop recognition*: la entry di un loop domina tutti i nodi interni al loop.

Ogni nodo n ha un unico *dominatore immediato* m che è l'ultimo dominatore di n su ogni percorso dall'entry node a n ($m \text{ idom } n$), $m \neq n$.

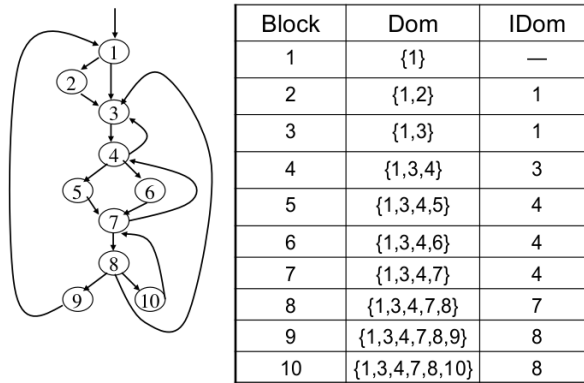


Figura 1.7: Dominator example

Capitolo 2

Analisi Statica

2.1 Introduzione

L'obiettivo dell'analisi statica è quello di dire, osservando le proprietà semantica di un programma, se una certa proprietà vale o meno. Esistono diverse tipologie di analisi statica:

- Control flow Analysis;
- Data flow Analysis (distributive e non-distributive);

2.2 Analisi sul CFG

Viene generato un CFG per ogni procedura. Le analisi che vengono eseguite sono localizzate a 3 livelli:

1. **Locali al blocco:** sono eseguite all'interno di uno stesso *basic block*;
2. **Intra-procedurali:** considerano il flusso di informazioni nel singolo CFG;
3. **Inter-procedurali:** considerano il flusso di informazioni tra le procedure (con archi che rappresentano le chiamate di funzione).

L'analisi di *data-flow* dice come l'informazione viene manipolata in un blocco. L'informazione è caratterizzata dalla soluzione dell'equazione di punto fisso definita per ogni blocco.

In alcuni casi questa equazione è ottenuta in 3 passaggi:

- definendo l'informazione entrante in un blocco, che è l'unione dell'informazione di uscita del blocco precedente;
- definendo l'informazione in uscita dal blocco che è l'informazione in ingresso, modificata dalle operazioni eseguite nel blocco;
- queste definizioni vengono poi combinate nell'equazione del punto fisso.

Le analisi di *data-flow* seguono il seguente schema:

* Forward

$$FAin(n) = \begin{cases} \bigoplus_{m \in Pred(n)} FAout(m) & n = n_0 \\ & \tau(FAin(m)) = \text{gen}(m) \cup (FAout(m) \setminus \text{kill}(m)) \end{cases}$$

* Backward

$$BAout(n) = \begin{cases} \bigoplus_{m \in Succ(n)} BAin(m) & n = n_f \\ & \tau(BAout(m)) = \text{gen}(m) \cup (BAin(m) \setminus \text{kill}(m)) \end{cases}$$

* Possible analyses $\longrightarrow \oplus = \cup$ * Definite analyses $\longrightarrow \oplus = \cap$

2.3 Soluzioni MFP - MOP - IDEAL

Per le equazioni di *data-flow analysis* esistono 3 tipi di soluzioni:

- **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; questa soluzione approssima la *MOP*.
- **MOP** (*merge over all paths*): è la soluzione più precisa rispetto alla *MFP* ($MOP \supseteq MFP$) poiché combina i valori dell'analisi di tutti i possibili percorsi del CFG dopo averli attraversati tutti. In generale, questa soluzione non è computabile perché ci posso essere un numero esponenziale (o infinito) di percorsi possibili:
 - loop con guardia sempre vera;
 - un programma che contiene N *if* statement avrà 2^N percorsi di esecuzione;
- **IDEAL**: è la soluzione migliore ma non è computabile. A differenza della *MOP*, prende in considerazione solamente i percorsi che verranno attraversati sicuramente da almeno qualche esecuzione. Calcola il valore alla fine di ogni possibile percorso di esecuzione e calcola poi il *meet* di questi valori.
 - ogni soluzione più grande di *IDEAL* è scorretta;
 - ogni soluzione più piccola di *IDEAL* è conservativa (*safe*);

Se la funzione di trasferimento di ogni arco è **distributiva** ($f(x \cup y) = f(x) \cup f(y)$) (e ogni program point è raggiungibile dall'entry point), allora la soluzione delle equazioni di *data-flow* è la stessa per *MOP* e *MFP* ($MOP = MFP$). Dunque per le funzioni di trasferimento distributive, è possibile calcolare la soluzione *MOP* attraverso l'algoritmo iterativo del punto fisso.

I **problemi distributivi** sono i cosiddetti problemi "*semplici*", come ad esempio: *live variables*, *available expressions*, *reaching definitions* e *very busy expressions* (tutte proprietà che ci dicono *COME* un programma viene eseguito).

I **problemi non-distributivi** sono quelli che ci dicono *COSA* calcola un programma (ad esempio che l'output è costante, valori positivi, intervalli etc.). Un esempio di problema non distributivo è la **constant propagation analysis**.

```
if(<some condition>) {
    A = 2;
    B = 3;
}
else {
    A = 3;
    B = 2;
}
C=A+B;
```

Se consideriamo la *constant propagation*, in questo programma il valore di C sarà sempre 5, indipendentemente dal valore della guardia dello statement *if*.

Con una soluzione *MFP*, C non verrà mai considerata una costante, al contrario, con una soluzione *MOP* otterremo come informazione che la variabile C è una costante.

2.4 Data Flow Analysis

Insieme di tecniche che raccolgono informazione su come i dati fluiscono durante l'esecuzione.

2.4.1 Available Expressions

L'espressione e è *available* se è valutata e assegnata ad una variabile prima di v (uso della variabile). Tra la valutazione e v non vengono ridefinite le variabili dell'espressione e x ($x:=e$).

Proprietà: Forward & Definite

Punto fisso:

$$AvailIn(n) = \begin{cases} \emptyset & \text{se } n = n_0 \\ \bigcap_{m \in pred(n)} AvailOut(m) & \text{altrimenti} \end{cases}$$

$$AvailOut(n) = Gen(n) \cup (AvailIn(n) \setminus Kill(n))$$

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

Semantica:

Dominio astratto = Ass = {assegnamenti $x \leftarrow e \mid x \notin Var(e)$ }

$A \subseteq Ass$

$$\llbracket ; \rrbracket^\# A = A$$

$$\llbracket NonZero(e) \rrbracket^\# A = \llbracket Zero(e) \rrbracket^\# A = A$$

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

$$\llbracket x \leftarrow M[e] \rrbracket^\# A = A \setminus Occ(x)$$

$$\llbracket M[e_1] \leftarrow e_2 \rrbracket^\# A = A$$

$Occ(x) = \{\text{Assegnamenti che coinvolgono } x \text{ a destra o a sinistra}\}$

$Gen(n) = \{\text{espressioni valutate nel blocco } n \text{ e nessun operando di } e \text{ è definito nuovamente tra l'ultima valutazione di } e \text{ in } n \text{ e la fine di } n\}$

$Kill(n) = \{\text{espressioni uccise da una nuova definizione di } n\}$

2.4.2 Very Busy Expressions

Un assegnamento è *busy* su un cammino π se $\pi = \pi_1 \ k \ \pi_2$ con:

- k è un assegnamento $x \leftarrow e$;
- π_1 non contiene usi di x ;
- π_2 non contiene modifiche di $\{x\} \cup Var(e)$.

Un assegnamento è *very busy* se è *busy* su ogni percorso da v a *exit*.

Dice come e quali espressioni anticipare.

Un assegnamento è ucciso in un blocco n se una delle sue variabili è modificata o se e viene usata.

Un assegnamento è generato in un blocco n se si trova nel blocco e l'espressione non contiene la variabile che si sta assegnando.

Proprietà: Backward & Definite

Punto fisso:

$$VB_{exit}(p) = \begin{cases} \emptyset & \text{se } p = v_{exit} \\ \bigcap_{q \in succ(p)} VB_{entry}(q) & \text{altrimenti} \end{cases}$$

$$VB_{entry}(p) = Gen(p) \cup (VB_{exit}(p) \setminus Kill(p))$$

$$VB_{exit}(p) = \bigcap_{q \in succ(p)} Gen(q) \cup (VB_{exit}(q) \setminus Kill(q))$$

Semantica:

$$B = 2^{Ass} = \mathcal{P}(Ass)$$

$$\llbracket ; \rrbracket^\# B = B$$

$$\llbracket NonZero(e) \rrbracket^\# B = \llbracket Zero(e) \rrbracket^\# B = B \setminus Ass(e)$$

$$\llbracket x \leftarrow e \rrbracket^\# B = \begin{cases} B \setminus (Occ(x) \cup Ass(e)) \cup \{x \leftarrow e\} & \text{se } x \notin Var(e) \\ B \setminus (Occ(x) \cup Ass(e)) & \text{altrimenti} \end{cases}$$

$$\llbracket x \leftarrow M[e] \rrbracket^\# B = B \setminus (Occ(x) \cup Ass(e))$$

$$\llbracket M[e_1] \leftarrow e_2 \rrbracket^\# B = B \setminus (Ass(e_1) \cup Ass(e_2))$$

$$Use(n) = \{\text{occorrenza di una variabile sul lato destro di uno statement}\}$$

2.4.3 Liveness

x è *live* all'uscita del blocco b se verrà usata successivamente. x non è *live* o (*dead*) se viene ridefinita prima di un successivo uso.

x è *live* in un cammino π ($v \rightarrow exit$) se:

- π non contiene $Def(x)$ e,
- esiste almeno un uso di x in π che segue la $Def(x)$;

x è *live* se si trova tra una definizione ed un uso.

Dice se a e b possono essere memorizzate nella stessa locazione, cioè se a e b non sono mai *live* insieme, allora posso sostituire a con b .

- $x \in Use(n) \Rightarrow x \text{ LiveIn in } n$
- $x \text{ è LiveOut in } n \text{ e } x \notin VarKill(n) \Rightarrow x \text{ LiveIn in } n$;
- $x \text{ è LiveIn in almeno un } Succ(n) \Rightarrow x \text{ LiveOut}(n)$;

Falsi positivi:

- x è accessibile attraverso altri nomi \Rightarrow Liveness fallisce;
- analizzi anche cammini non possibili;
- inizializzazione in altre procedure (perché questa analisi è intra-procedurale);

Proprietà: Backward & Possible

Punto fisso:

$$\begin{aligned} LiveOut(n) &= \begin{cases} \emptyset & \text{se } n = exit \\ \bigcup_{m \in Succ(n)} LiveIn(m) & \text{altrimenti} \end{cases} \\ LiveIn(n) &= Use(n) \cup (LiveOut(n) \setminus VarKill(n)) \\ LiveOut(n) &= \bigcup_{m \in Succ(n)} Use(m) \cup (LiveOut(m) \setminus VarKill(m)) \end{aligned}$$

Semantica:

Dominio astratto = $\mathcal{P}(Var)$

$L \subseteq Var$

$$\begin{aligned} \llbracket ; \rrbracket^\sharp L &= L \\ \llbracket NonZero(e) \rrbracket^\sharp L &= \llbracket Zero(e) \rrbracket^\sharp L = L \cup Var(e) \\ \llbracket x \leftarrow e \rrbracket^\sharp L &= Var(e) \cup (L \setminus \{x\}) \\ \llbracket x \leftarrow M[e] \rrbracket^\sharp L &= Var(e) \cup (L \setminus \{x\}) \\ \llbracket M[e_1] \leftarrow e_2 \rrbracket^\sharp L &= L \cup Var(e_1) \cup Var(e_2) \end{aligned}$$

$LiveIn(n) = \{\text{sono le variabili } live \text{ in } n \text{ che sono } live \text{ su almeno un arco entrante}\}$

$LiveOut(n) = \{\text{sono le variabili } live \text{ in } n \text{ che sono } live \text{ su almeno un arco uscente}\}$

$VarKill(n) = Def(n)$, cioè le definizioni presenti in n

True Liveness: un *true use* è un uso in un assegnamento ad una variabile *live*. Se assegno x ad una variabile *non-live*, allora anche x non è *live*.

Copy Propagation

L'analisi ad ogni program point tiene traccia delle copie di x .

Se ho un assegnamento $T \leftarrow x + 1$ e poi $y \leftarrow T$, allora quest'ultimo è inutile.

Proprietà: Forward & Definite

Punto fisso:

$$\begin{aligned} Copie_{entry}(n) &= \bigcap_{m \in Pred(n)} Copie_{exit}(m) \\ Copie_{exit}(n) &= \bigcap_{m \in Pred(n)} Gen(m) \cup (Copie_{exit}(m) \setminus Kill(m)) \end{aligned}$$

Semantica:

Dominio astratto $= \mathcal{V}_x = \{V \subseteq Var \mid x \in V\}$ perché x è copia di se stesso.

$V \subseteq Var$

Entry $V_0 = \{x\}$ perché x è copia di se stesso e cerco le altre sue copie.

$$\begin{aligned} \llbracket ; \rrbracket^\# V &= V \\ \llbracket NonZero(e) \rrbracket^\# V &= \llbracket Zero(e) \rrbracket^\# V = V \\ \llbracket x \leftarrow e \rrbracket^\# V &= \llbracket x \leftarrow M[e] \rrbracket^\# V = \{x\} \\ \llbracket z \leftarrow y \rrbracket^\# V &= \begin{cases} V \cup \{z\} & \text{se } y \in V \text{ (y è copia di x)} \\ V \setminus \{z\} & \text{altrimenti} \end{cases} \\ \llbracket y \leftarrow e \rrbracket^\# V &= V \setminus \{y\} \\ \llbracket M[e_1] \leftarrow e_2 \rrbracket^\# V &= V \end{aligned}$$

$Gen(n) = \{(x == y) \mid n \text{ contiene } x \leftarrow y\}$

$Kill(n) = \{(x == y) \mid x \text{ è ridefinita in } n\}$

2.4.4 Reaching Definition

Dato un program point p vogliamo identificare le definizioni di variabili che raggiungono p .

Viene usata in *code motion*: se uso un assegnamento in tutto il ciclo senza modificarlo, allora lo sposto all'entrata del ciclo.

Proprietà: Forward & Possible

Punto fisso (non c'è la semantica):

$$\begin{aligned} RD_{entry}(n) &= \begin{cases} i = \{(x, ?) \mid x \in Var\} & \text{se } n = entry \\ \bigcup_{m \in Pred(n)} RD_{exit}(m) & \text{altrimenti} \end{cases} \\ RD_{exit}(n) &= Gen(n) \cup (RD_{entry}(n) \setminus Kill(n)) \end{aligned}$$

$\{(x, p) \mid x \in Vars, p \text{ punto di programma}\}$

Inizializzazione: $i = \{(x, ?) \mid x \in Vars, \text{variabile non inizializzata}\}$

$Gen(n) = \{\text{definizioni } (x, l) \text{ dentro } n \text{ e disponibili alla fine di } n\}$

$Kill(n) = \{(x, p) \mid x \text{ è ridefinita in } n\}$

2.4.5 Riepilogo

	Possible (\cup)	Definite (\cap)
Forward	Reaching Definition	Available Expr, Copy Propagation
Backward	Liveness	Very Busy Expr

Available Expressions:

$$AvailIn(n) = \begin{cases} \emptyset & \text{se } n = n_0 \\ \bigcap_{m \in pred(n)} AvailOut(m) & \text{altrimenti} \end{cases}$$

$$AvailOut(n) = Gen(n) \cup (AvailIn(n) \setminus Kill(n))$$

Very Busy:

$$VB_{exit}(p) = \begin{cases} \emptyset & \text{se } p = v_{exit} \\ \bigcap_{q \in succ(p)} VB_{entry}(q) & \text{altrimenti} \end{cases}$$

$$VB_{entry}(p) = Gen(p) \cup (VB_{exit}(p) \setminus Kill(p))$$

Liveness:

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

$$LiveIn(n) = Use(n) \cup (LiveOut(n) \setminus VarKill(n))$$

Reaching Definition:

$$RD_{entry}(n) = \begin{cases} i = \{(x, ?) \mid x \in Var\} & \text{se } n = entry \\ \bigcup_{m \in Pred(n)} RD_{exit}(m) & \text{altrimenti} \end{cases}$$

$$RD_{exit}(n) = Gen(n) \cup (RD_{entry}(n) \setminus Kill(n))$$

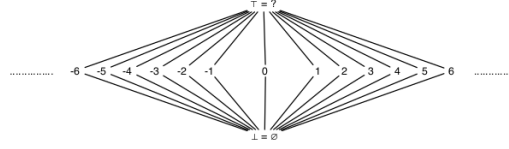
2.5 Problemi Non-Distributivi

2.5.1 Costanti

Ogni singoletto non è confrontabile con gli altri. Se una costante assume due valori va in \top . È un reticolo completo poiché contiene \emptyset ed è ACC perché è finito in altezza.

$$\alpha(\{0,1\}) = \top$$

$$\alpha(\{5\}) = 5$$



Abstract states: $\text{Var} \rightarrow \text{Const}$

Dominio concreto: $\mathbb{V} \rightarrow \mathbb{Z}$

Dominio astratto: $\mathbb{V} \rightarrow \mathcal{P}(\mathbb{Z})$

Semantica astratta delle espressioni:

$op = \text{operatore}$

$$a \text{ op } b = \begin{cases} a \text{ op } b & \text{se } a \text{ e } b \text{ sono costanti} \\ \top & \text{se } a = \top \vee b = \top \end{cases}$$

$$\llbracket c \rrbracket^\# D = c$$

$$\llbracket op \ e \rrbracket^\# D = op^\# \llbracket e \rrbracket^\# D$$

$$\llbracket e_1 \text{ op } e_2 \rrbracket^\# D = \llbracket e_1 \rrbracket^\# D \text{ op }^\# \llbracket e_2 \rrbracket^\# D$$

$$\llbracket x \rrbracket^\# D = D(x)$$

Semantica astratta dei comandi:

$D = \text{memoria}$

$$\llbracket ; \rrbracket^\# D = D$$

$$\llbracket NonZero(e) \rrbracket^\# D = \begin{cases} \perp & \text{se } \llbracket e \rrbracket^\# D = 0 \\ D & \text{se } \llbracket e \rrbracket^\# D \neq 0 \end{cases}$$

$$\llbracket Zero(e) \rrbracket^\# D = \begin{cases} D & \text{se } 0 \subseteq \llbracket e \rrbracket^\# D \\ \perp & \text{se } 0 \not\subseteq \llbracket e \rrbracket^\# D \end{cases}$$

$$\llbracket x \leftarrow e \rrbracket^\# D = D[x \mapsto \llbracket e \rrbracket^\# D]$$

$$\llbracket x \leftarrow M[e] \rrbracket^\# D = D[x \mapsto \top] \quad \text{non so cos'è } M[e] \text{ perché lo valuterò dopo}$$

$$\llbracket M[e_1] \leftarrow e_2 \rrbracket^\# D = D$$

2.5.2 Segni

Dominio rappresentato da un semipiano (un insieme di punti), quindi non va subito a \top .

2.5.3 Intervalli

Il dominio degli Intervalli non è *ACC*, dunque non garantisce la terminazione: per questo viene introdotto il *widening*.

$$\mathbb{I} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{+\infty\}, l \leq u\}$$

[a, b] dove $a \leq x \leq b$ (convessi)

Semantica astratta delle espressioni:

- $[l_1, u_1] +^\# [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$
- $-^\# [l, u] = [-u, -l]$
- $[l_1, u_1] *^\# [l_2, u_2] = [a, b]$ dove:
 - $a = \min(l_1 * l_2, l_1 * u_2, l_2 * u_1, l_2 * u_2)$
 - $b = \max(l_1 * l_2, l_1 * u_2, l_2 * u_1, l_2 * u_2)$
- $[l_1, u_1] =^\# [l_2, u_2] = \begin{cases} [1, 1] & \text{se } l_1 = l_2 = u_1 = u_2 (\text{costanti}) \\ [0, 0] & \text{se } u_1 < l_2 \vee u_2 < l_1 \\ [0, 1] & \text{altrimenti (intervalli uguali che approssimano valori diversi)} \end{cases}$
- $[l_1, u_1] <^\# [l_2, u_2] = \begin{cases} [1, 1] & \text{se } u_1 < l_2 \\ [0, 0] & \text{se } u_2 \leq l_1 \\ [0, 1] & \text{altrimenti} \end{cases}$

Semantica astratta dei comandi:

$D : \mathbb{V} \rightarrow \mathbb{I}$

$$\begin{aligned} \llbracket ; \rrbracket^\# D &= D \\ \llbracket \text{NonZero}(e) \rrbracket^\# D &= \begin{cases} \perp & \text{se } [0, 0] = \llbracket e \rrbracket^\# D \\ D & \text{se } [0, 0] \neq \llbracket e \rrbracket^\# D \text{ (contiene anche altri valori)} \end{cases} \\ \llbracket \text{Zero}(e) \rrbracket^\# D &= \begin{cases} D & \text{se } [0, 0] \subseteq \llbracket e \rrbracket^\# D \text{ (lo 0 è uno dei possibili valori)} \\ \perp & \text{se } [0, 0] \not\subseteq \llbracket e \rrbracket^\# D \text{ (l'intervallo } e \text{ non contiene 0)} \end{cases} \\ \llbracket x \leftarrow e \rrbracket^\# D &= D[x \mapsto \llbracket e \rrbracket^\# D] \\ \llbracket x \leftarrow M[e] \rrbracket^\# D &= D[x \mapsto [-\infty, +\infty]] \text{ (elemento } \top \text{ degli intervalli)} \\ \llbracket M[e_1] \leftarrow e_2 \rrbracket^\# D &= D \end{aligned}$$

Capitolo 3

Analisi Dinamica

L'analisi dinamica di un programma si basa sulla sua esecuzione e viene utilizzata in vari ambiti: *testing, debugging, emulation/virtualization, profiling/tracing, monitoring, dynamic slicing*.

Nelle sezioni seguenti ne analizziamo alcuni nel dettaglio.

3.1 Testing

Si tratta principalmente dell'esecuzione di un programma basata su un campione di dati (molto piccolo) passato come input.

L'**obiettivo** è la ricerca di bug/errori/difetti del software, senza correggerli. Questa operazione viene svolta nella fase di testing da professionisti con un'esperienza nella ricerca e identificazione dei bug.

Durante la fase di testing si devono ricercare:

- **mistake**: un'azione umana che ha prodotto un risultato scorretto;
- **fault**: un passaggio scorretto (una definizione di variabile...) all'interno del programma;
- **failure**: la mancata abilità da parte del sistema di svolgere le funzioni richieste;
- **errori**: la differenza tra il valore atteso e il valore effettivamente calcolato/osservato;
- **specifiche**: un documento che specifica, in modo completo e preciso, le richieste e le caratteristiche del sistema e/o dei componenti e spesso delle procedure per verificare quali delle disposizioni sono state soddisfatte.

3.2 Debugging

L'**obiettivo** 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.

3.3 Program Slicing

Si tratta di una tecnica di decomposizione che trasforma un 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* che descrive i parametri di interesse: V (insieme delle variabili di interesse) e n (punti di interesse del programma).

Ci sono diversi motivi per i quali effettuare il *program slicing*: *program debugging, testing* (lo slicing riduce i costi del *regression testing* dopo una trasformazione del codice), *parallelizzazione, compressione di un programma* (effettuare lo slicing aiuta a comprendere come viene eseguito un

programma e quali variabili verranno modificate nei vari percorsi) e *mantenimento del software* (per modificare il codice senza *side effects* indesiderati in giro per il programma).

Esistono **diversi tipi di program slicing**:

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

Esistono, inoltre, **diverse forme di program slicing**:

- **Korel & Laski (KL)**: è una forma di slicing molto forte in cui il programma e lo slice devono seguire *paths* identici. Il programma e lo slice hanno la stessa semantica operativa. Il *path* seguito dallo slice deve essere un *subpath* dell'esecuzione originale.
- **Iteration Count (IC)**: richiede che lo slice e il programma si pareggino solo ad una certa iterazione k di un program point n (cioè quando lo statement al program point n viene eseguito per la k -esima volta), e non per tutte le iterazioni dello stesso program point.
- **KL-IC** (combinazione dei precedenti): richiede che il programma e lo slice seguano *paths* identici e siano uguali solamente ad una particolare iterazione di un certo program point.