Analisi dei sistemi informatici

Amedeo Zampieri

January 2019

0.1 Interpretazione astratta

Dominio astratto : Un dominio astratto è un insieme di *oggetti astratti*, più le operazioni astratte su tali oggetti (operazioni che approssimano quelle concrete).

Astrazione : Una funzione di astrazione α mappa ogni oggetto concreto o in una sua approssimazione rappresentata da un oggetto astratto $\alpha(o)$

Comparazione di astrazioni : Una maggiore perdita di informazioni significa che il grado di astrazione è maggiore. Non è sempre possibile paragonare le astrazioni, poiché possono perdere informazione in maniera diversa.

Concretizzazione : Una funzione di concretizzazione γ mappa oggetti astratti $\bar{o}$ in oggetti concreti $\gamma(\bar{o})$ che essi rappresentano.

Collecting semantics : È una semantica che colleziona tutti gli stati che possono presentarsi in una traccia. Non esiste una semantica universale poiché l'informazione da mantenere dipende dal problema specifico.

Least upper bound (lub) : Un *lub* in un poset $\mathbf{X}$ è un $\mathbf{x}$ tale che:

- $\mathbf{x}$ è un *upper bound* di $\mathbf{X}$;
- $\mathbf{x}$ è il minore degli *upper bound* di $\mathbf{X}$;

Quando esiste, un *lub* è unico. Non è detto che un *lub* debba appartenere al poset.

Greatest lower bound (glb) : È il duale di *lub*.

Join semireticolo : Un *join semireticolo* $\langle P, \leq, \sqcup \rangle$ è un poset $\langle P, \leq \rangle$ tale che dati due elementi $\mathbf{x}, \mathbf{y} \in P$ abbiano un **lub** $\mathbf{x} \sqcup \mathbf{y}$.

Meet semireticolo : è il duale del *join semireticolo*, quindi è definito come $\langle P, \leq, \sqcap \rangle$.

Reticolo : Un reticolo è sia un join che un meet semireticolo. Dispone quindi di **lub** e di **glb**. È definito quindi come $\langle P, \leq, \sqcup, \sqcap \rangle$.

Sottoreticolo : Sia $\langle L, \leq, \sqcup, \sqcap \rangle$ un reticolo. $S \subseteq L$ è un sottoreticolo di L **se e solo se**

$$\forall x, y \in S : x \sqcup y \in S \wedge x \sqcap y \in S$$

Reticolo completo : Un reticolo $\langle L, \leq, \sqcup, \sqcap \rangle$ si dice completo se ogni sottoinsieme $X \subseteq L$ dispone di *lub* e di *glb*. Un reticolo completo ha un elemento **bottom** $\perp = \sqcup\emptyset$ e un elemento **top** $\top = \sqcup P$.

Punti fissi : Sia $f \in L \to L$ un operatore sul poset $\langle L, \sqsubseteq \rangle$

- **fixpoints** : $\mathrm{fp}(f) \stackrel{\text{def}}{=} \{x \in L | f(x) = x\}$
- **pre-fixpoints** : $\mathrm{prefp}(f) \stackrel{\text{def}}{=} \{x \in L | f(x) \sqsubseteq x\}$
- **post-fixpoints** : $\mathrm{postfp}(f) \stackrel{\text{def}}{=} \{x \in L | f(x) \sqsupseteq x\}$

Il *least fixpoint* (**lfp**) di f è il minimo dei suoi fixpoint. Segue dualmente il *greatest fixpoint*(**gfp**).

L' *astrazione* è il processo con il quale si rimpiazza qualcosa di concreto con una descrizione di *alcune* delle sue proprietà. Queste proprietà sono descritte in maniera precisa, mentre quelle che vengono tralasciate generano l'imprecisione dell'astrazione.
Sia $\mathscr{P}(\Sigma)$ l'insieme delle proprietà di Σ. Sia $A \subseteq \mathscr{P}(\Sigma)$ un'astrazione: gli elementi in A sono descritti precisamente dall'astrazione (senza perdita di precisione, mentre gli elementi che non appartengono ad A devono essere rappresentati attraverso oggetti di A, introducendo così perdita di precisione.

Reticolo delle proprietà : Sia $\mathscr{P}(\Sigma)$ l'insieme delle proprietà degli oggetti in Σ un reticolo completo e distributivo $\langle \mathscr{P}(\Sigma), \subseteq, \emptyset, \cup, \cap, \neg \rangle$:

- Una proprietà P è un sottoinsieme di Σ;
- $\subseteq$ è l'implicazione logica fra proprietà;
- Σ è vero;
- $\emptyset$ è falso;
- $\cup$ è la disgiunzione, ovvero, dati $P, Q \subseteq \mathscr{P}(\Sigma)$, gli elementi di P o di Q appartengono a $P \cup Q$);
- $\cap$ è la congiunzione, ovvero, dati $P, Q \subseteq \mathscr{P}(\Sigma)$, gli elementi di P e di Q appartengono a $P \cap Q$);
- $\neg$ è la negazione, ovvero, dato $P \subseteq \mathscr{P}(\Sigma)$, gli elementi che stanno in $\Sigma \setminus P$;

Quando si approssima una proprietà concreta P con una astratta $\overline{P}$ ci sono essenzialmente due casi:

- Approssimazione per difetto: $\overline{P} \subseteq P$. La proprietà astratta è più vincolante di quella concreta.

$$x \in \overline{P} \rightarrow x \in P$$

- Approssimazione per eccesso: $\overline{P} \supseteq P$. La proprietà astratta è meno vincolante di quella concreta, aggiungendo così rumore. A volte analizzare un insieme più grande permette di avere una miglior rappresentazione degli elementi grazie al fatto che i vincoli sono rilassati.

$$x \notin \overline{P} \rightarrow x \notin P$$

Essendo casi duali, basta studiarne uno solo. Inoltre $\overline{P}$ deve essere **decidibile**.

Miglior astrazione : La miglior approssimazione di una proprietà concreta è il *glb* di tutti gli elementi astratti che godono di tale proprietà:

$$\overline{P} = \bigcap \{ \overline{P}' \in A | P \in \overline{P}' \}$$

Galois connections : Dati due poset $(A, \leq_A)$ e $(C, \leq_C)$, una *connessione di Galois* fra i due poset consiste di due funzioni *monotone*:

- Astrazione: $\alpha : C \to A$ su $(A, \leq_A)$
- Concretizzazione: $\gamma : A \to C$ su $(C, \leq_C)$

tali che $\forall a \in A.\alpha(c) \leq_A a \Leftrightarrow \forall c \in C.c \leq_C \gamma(a)$. Le connessioni di Galois servono per formalizzare la relazione fra concreto ed astratto, garantendo l'esistenza della migliore approssimazione. Una GC viene rappresentata come

$$(C, \leq_C) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \leq_A)$$

Data α o γ la corrispondente GC è determinata univocamente.

Galois insertions : Una inserzione di Galois di A su B è una GC nella quale l'operatore di chiusura è l'identità su A.

$$\forall a \in A.\alpha\gamma(a) \leq_A a \ \& \ \forall c \in C.c \leq_C \gamma\alpha(c)$$

Upper closure operators : Una funzione $\rho P \to P$ su un poset $\langle P, \leq_P\rangle$ è detta **uco** se:

- È estensiva (aggiunge rumore): $\forall x \in P.x \leq_A \rho(x)$
- È monotona
- È idempotente (il rumore viene aggiunto tutto in una volta e poi non ne viene più aggiunto): $\forall x \in P.\rho(x) = \rho\rho(x)$

Nelle *GC* e nelle *GI* $\gamma\alpha$ sono *uco*.

Moore families : Sia L un reticolo completo. $X \subseteq L$ è una *Moore family* di L se $X = \mathscr{M}(X) \overset{\text{def}}{=} \{\bigwedge S | S \subseteq X\}$, dove $\bigwedge \emptyset = \top \in \mathscr{M}(X)$. Essere una Moore family garantisce l'esistenza della miglior approssimazione.

0.2 Reticolo delle astrazioni

Consideriamo il reticolo completo $\langle C, \leq \wedge, \vee, \top, \bot\rangle$, $A_i \in uco(C)$:
Il reticolo delle astrazioni è il reticolo degli *uco* $A \equiv \rho(C)$

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

I domini astratti possono essere comparati in base al loro grado di precisione: un'astrazione A_1 è più precisa di un'astrazione A_2 se la concretizzazione di A_1 contiene quella di A_2 poiché ogni oggetto di A_1 è approssimato in qualcosa di "più piccolo" rispetto agli oggetti di A_2.
I domini astratti possono essere modificati dai *glb*, poiché il glb di domini astratti è il più piccolo dominio astratto contenente l'unione delle concretizzazioni.
I domini astratti possono essere combinati tramite *lub*, poiché il lub di domini astratti è l'intersezione delle concretizzazioni dei domini astratti.
L'astrazione più imprecisa (o astratta) porta tutto a $\top$, mentre la più precisa astrae tutto a sé stesso, diventando la funzione identità (il dominio astratto A più preciso è infatti quello che coincide con il rispettivo C).

0.2.1 Soundness/Correctness

Correctness Consideriamo $C \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} A$, una funzione concreta $f : C \to C$ e una funzione astratta $f^\sharp : A \to A$. Si dice che $f^\sharp$ sia un'approssimazione *sound/correct* di f in A se:

$$\forall c \in C : \alpha(f(c)) \leq_A f^\sharp(\alpha(c)) \equiv \forall a \in A : f(\gamma(a)) \leq_C \gamma(f^\sharp(a))$$

Dalla definizione segue che la *bca* (best correct approximation) di $f : C \to C$ sia $\alpha \circ f \circ \gamma : A \to A$, ma dato che questa approssimazione compie computazioni concrete è del tutto inutile ai fini della semantica.

0.2.2 Completeness

Per quanto riguarda la *completeness* bisogna considerare due casi:

- **Backward**: considera la completezza sull'astrazione dell'output delle operazioni
- **Forward**: considera la completezza sull'astrazione dell'input delle operazioni

Backward completeness

In questo caso intendiamo che non vi è perdita di precisione in caso di approssimazione dell'input. In pratica anche se l'input viene approssimato si è ancora in grado di calcolare precisamente la proprietà astratta (il rumore in input non disturba la semantica della computazione).
Un esempio di *incompletezza backward* è sul dominio dei segni: se si guardano solo i segni degli input non possiamo sapere il segno di somma fra $\mathbb{Z}^-$ e $\mathbb{Z}^+$, andando così a $\top$. Allo stesso tempo però questo dominio è *forward complete* poiché astraendo sull'output otteniamo sempre lo stesso risultato che non facendolo. In pratica non c'è modo di avere un output più preciso.

Forward completeness

In questo caso non vi è perdita di precisione approssimando l'output dell' astrazione, ovvero, se si approssima l'output, la computazione è ancora precisa.
Un esempio di *incompletezza forward* è sull'espressione non deterministica $e_1 \Box e_2$ e la propagazione delle costanti. Infatti non sapendo quale operazione verrà eseguita otterremo $\top$. Un modo per evitare il problema sarebbe mettere tutte le possibilità nel risultato. Questo è anche un esempio di completezza *backward* poiché non c'è modo di cambiare l'input per ottenere una computazione più precisa.

0.2.3 Accelerazione della convergenza

In certi casi la convergenza non è assicurata, o magari è raggiungibile in un numero infinito di passi. Vorremmo quindi evitare entrambi i casi, poiché vorremmo sempre convergere e possibilmente in fretta (altrimenti lo strumento è inutile per applicazioni pratiche). Si introduce quindi in **widening**, una tecnica che permette l'accelerazione della convergenza.

Widening Un widening $\nabla \in P \times P \to P$ su un poset $\langle P, \sqsubseteq \rangle$ soddisfa:

- $\forall x, y \in P : x \sqsubseteq (x \nabla y) \wedge y \sqsubseteq (x \nabla y)$
- per ogni catena crescente $x^0 \sqsubseteq x^1 \sqsubseteq \dots$ la catena crescente ottenuta come $y^0 \stackrel{\text{def}}{=} x^0, \dots, y^{n+1} \stackrel{\text{def}}{=} y^n \nabla x^{n+1}, \dots$ **non** è **strettamente crescente**.

Sullo stesso dominio possono essere definiti diversi *widening*. Con questo operatore raggiungiamo dei *post punti fissi*. Vorremmo quindi trovare un modo per ottenere dei risultati più raffinati, in quanto non sappiamo quanto impreciso sia il post punto fisso raggiunto (F^∇).

Narrowing Un narrowing $\triangle \in P \times P \to P$ sul poset $\langle P, \sqsubseteq \rangle$ è un'operazione tale che:

- $\forall x, y \in P : y \sqsubseteq x \implies y \sqsubseteq x \triangle y \sqsubseteq x$ ($x \triangle y$ in quel punto ci garantisce di non scendere al di sotto di un punto fisso)
- per ogni catena decrescente $x^0 \sqsupseteq x^1 \sqsupseteq \dots$ la catena decrescente ottenuta come $y^0 \stackrel{\text{def}}{=} x^0, \dots, y^{n+1} \stackrel{\text{def}}{=} y^n \nabla x^{n+1}, \dots$ **non** è **strettamente decrescente**.

0.2.4 Linguaggio

Di seguito il linguaggio su cui verrà basato il resto:

Variabili:	x
Espressioni aritmetiche:	e
Assegnamenti:	$x \leftarrow e$
Lettura memoria:	$x \leftarrow M[e]$
Scrittura memoria:	$M[e]_1 \leftarrow e_2$
Condizioni:	if (e) S_1 else SS_2
Salto incondizionato:	goto L

0.2.5 Control Flow Graphs (CFG)

I *CFG* sono un modo utile per rappresentare il flusso del codice sotto forma di grafi (*con radice*):

- i **vertici** corrispondono ai program points
- gli **archi** corrispondono ai passi di computazione etichettati con la corrispondente azione:

Test:	$NonZero(e)$ or $Zero(e)$
Assegnamenti:	$x \leftarrow e$
Lettura memoria:	$x \leftarrow M[e]$
Scrittura memoria:	$M[e_1] \leftarrow e_2$
Statement vuoto:	;

Ogni statement condizionale ha due archi corrispondenti: uno etichettato con *NonZero(e)* che corrisponde alla condizione vera, l'altro etichettato con *Zero(e)*. Le computazioni vengono eseguite quando il corrispondente arco viene attraversato, modificando così lo stato del programma. Una computazione può quindi essere definita come un percorso fra due nodi.

Basic block I *basic blocks*, che vanno a comporre i nodi del nostro *CFG*, sono sequenze massime di statements con un singolo punto di entrata e un singolo punto di uscita. Un esempio semplice di basic block può essere dividere tutte le singole istruzioni del codice. Un modo migliore invece per costruirli è l'identificazione dei *leader*. Per determinare i *leader* si procede come segue:

- Il primo statement in una sequenza è un leader
- Qualsiasi statement s che sia target di un branch o un jump è un leader
- Qualsiasi statement che segua un branch o un return è un leader.

Per ogni leader il suo corrispondente basic block comprende tutto il codice dal leader (compreso) al successivo leader (escluso). I basic block sono comodi per l'ottimizzazione locale del codice, l'eliminazione della ridondanza e l'allocazione dei registri. Si prestano inoltre bene alla rappresentazione delle astrazioni su un linguaggio.

Per ogni nodo n del CFG si ha:

- $Pred(n)$ insieme dei predecessori di n;
- $Succ(n)$ insieme dei successori di n;
- un *branch node* è un nodo con più di un successore;
- un *join node* è un nodo che ha più di un predecessore;

Extended Basic Block (EBB) Un *EBB* è un insieme massimale di nodi in un CFG che *non* contengono altri *join node* al di fuori dell'entry node. Sono utilizzati per motivi di ottimizzazione.

Natural Loop Un loop è una collezione di nodi in un CFG tale che:

- Tutti i nodi nella collezione sono fortemente connessi;
- La collezione di nodi ha un'unica entry che permette l'accesso al loop;

Sono utilizzati per le ottimizzazioni. Una proprietà di cui godono è la seguente: a meno che due loop non abbiano la stessa entry sono o disgiunti o interamente uno contenuto nell'altro.

Inner Loop Un *inner loop* è un loop che non contiene altri loop.

Relazione di Dominance Un nodo d in un CFG *domina* un nodo n se **ogni** percorso dall'entry node del CFG che passa da n passa anche per d. Ogni nodo n ha un unico dominatore immediato, che è l'ultimo suo dominatore in ogni percorso possibile dal nodo entry.

Chapter 1

Analisi Statica

L'analizi statica è uno strumento utilizzato per studiare le proprietà (semantiche) dei programmi. Dal *teoreme di Rice* sappiamo però che le uniche proprietà studiabili in maniera precisa sono quelle banali. Nello specifico si parla di due tipi di analisi:

- Control flow analysis;
- Data flow analysis (distributive/non-distributive);

1.1 Analisi sui CFG

L'analisi si basa sul CFG generato dal relativo codice. Si dirama in tre differenti tipi:

- **Locali (livello blocco)**: sono eseguite all'interno di *singoli* basic block;
- **Intra-procedurali**: considerano il flusso di informazioni nel singolo CFG (della procedura);
- **Inter-procedurali**: considerano il flusso di informazioni tra le procedure.

Le *data-flow analyses* si basano sulla caratterizzazione di *come* l'informazione venga trasformata *all'interno* di un blocco, ottenendo una soluzione di una equazione di punto fisso definita per ogni blocco. Questa equazione viene (spesso) ottenuta così:

- Definizione dell'informazione in *ingresso* al blocco come *unione* dell' informazione in uscita dei blocchi precedenti;
- Definizione dell'informazione in *uscita* dal blocco come informazione in ingresso *trasformata* dalle operazioni eseguite nel blocco;
- Combinazione delle precedenti definizioni in un'equazione di punto fisso;

1.1.1 Schema delle analisi

Forward: In questo tipo di analisi il risultato è calcolato a partire dagli output dei blocchi. Il termine *forward* infatti ci suggerisce che il flusso vada dall'inizio alla fine del CFG.

$$MegasistemaForward$$

Backward: In questo caso il flusso va dalla fine all'inizio, risalendo il CFG. Infatti l'informazione è ottenuta guardando gli input dei blocchi.

$$MegasistemaBackward$$

Possible: Un'analisi possible definisce che una certa proprietà *potrebbe* valere in un certo punto di programma. Di questo tipo di analisi bisogna prendere l'insieme complementare, poiché non rispetta la *soundness*.

$$\oplus \equiv \cup$$

Definite: Un'analisi definite ci garantisce (*soundness*) che una certa proprietà valga in un blocco.

$$\oplus \equiv \cap$$

1.2 Formal Framework

Bisogna definire e formalizzare l'informazione *astratta* necessaria per l'analisi. Bisogna inoltre definire un *abstract edge effect* (funzione di trasferimento) monotona su questo nuovo dominio astratto. Cerchiamo la soluzione **MOP** (*merge over all paths* che permette di ottenere una trasformazione *sound* del programma. Nel caso in cui l'abstract edge effect sia *distributivo* ciò coincide con la soluzione *minima* del sistema di disuguaglianze nel quale le incognite sono le informazioni astratte in ogni punto di programma.
La terminazione è assicurata dalla monotonicità dellafunzione di trasferimento e da un reticolo ACC. La soundness invece segue per costruzione per punto fisso. Inoltre, per quanto riguarda la precisione, con questo metodo gli abstract edge effects tengono in considerazione anche percorsi che nell' esecuzione sarebbero invece impercorribili. Ciò quindi implica che tutti i percorsi possibili sono valutati (sound) e che ne vengono valutati alcuni in più (aggiunta di rumore).

1.3 Tre tipi di soluzioni: MFP, MOP e Ideale

Nella *data-flow analysis* si possono avere tre tipi di soluzioni che sono appunto *MFP*, *MOP* e *Ideale*. In alcuni casi MOP e MFP concidono, ovvero quando le funzioni di trasferimento sono *distributive*.Un'ulteriore condizione sarebbe che *ogni* punto di programma deve essere raggiungibile dalla entry, ma nel caso sia violata tale condizione basta rimuovere il codice morto (possibile in quanto *non* cambia la semantica del programma).

MFP (maximum fixed point) : Nel caso forward è l'insieme delle soluzioni delle equazioni degli ingressi in tutti i blocchi. È un' approssimazione della *MOP* (coincidono solo se la funzione di trasferimento è distributiva). È comunque un metodo **safe** di calcolo in quanto tiene conto di tutte le possibili varianti del concreto, ed in più aggiunge rumore.

MOP (merge over all paths) : È una soluzione migliore della *MFP* che prende in considerazione tutti i percorsi percorribili **sul CFG** e calcola la soluzione prendendo in considerazione solo questi. Purtoppo però la quantità di percorsi possibili può crescere esponenzialmente, o addirittura essere infinita. Non è quindi sempre computabile.

Ideale : Vengono presi in considerazione **solo** i percorsi percorribili sul programma. Ciò non è ovviamente possibile in quanto per sapere quali sono tutti e i soli percorsi possibili dovremmo anche conoscere *tutti* gli input. In pratica è la totale assenza di astrazione e il calcolo dovrebbe essere fatto nel concreto.

$$\textbf{MFP} \leq \textbf{MOP} \leq \textbf{Ideale}$$

1.3.1 Distributivo vs Non Distributivo

I problemi distributivi sono quelli considerati *semplici*, in quanto trattano proprietà riguardanti **come** il programma esegue (*live variables, available expressions, reaching definitions, very busy expressions*). Invece, tipicamente, i problemi non distributivi trattano proprietà su **cosa** il programma calcola (*l'output è una costante, un valore positivo o appartiene ad un intervallo*).

1.4 Data-flow analysis (Problemi distributiv)

Data-flow analysis : La *data-flow analysis* è una tecnica per raccogliere informazioni su *come* i dati si muovono a run time nei vari punti di un programma.

1.4.1 Busy expressions

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

- k è un assegnamento $(x \leftarrow e)$;
- π_1 non contiene utilizzi di x (x non compare r-side);
- π_2 non contiene modifiche di $\{x\} \cup Var(e)$ (x non compare l-side, nessuna delle variabili r-side nell'assegnamento di x viene modificata);

Very busy expression : Un assegnamento si dice *very busy* in s se risulta *busy* su ogni path da v ad *exit*.

Tipologia analisi

Backward, definite

Equazioni

$$VBOut(b) = \begin{cases} \emptyset & \text{se } p = exit \\ \bigcap\limits_{p \in succ(b)} VBIn(p) & \text{altrimenti} \end{cases}$$

$$VBIn(b) = Gen(b) \cup (VBOut(b) \setminus Kill(b))$$

$$VBOut(b) = \bigcap_{p \in succ(b)} Gen(p) \cup (VBOut(p) \setminus Kill(p))$$

Un assegnamento $x \leftarrow e$ è ucciso in un blocco n se una
variabile $y \in e$ è modificata o se x viene utilizzata nel blocco.

$$Kill(b) = \{x \leftarrow e | \exists e' \in b. x \in Var(e') \vee y \leftarrow e' \in b. y \in Var(e) \vee x = y\}$$

Un assegnamento $x \leftarrow e$ è generato in un blocco n se $x \leftarrow e \in b$ e $x \notin e$

$$Gen(b) = \{x \leftarrow e | x \leftarrow e \in b, x \notin Var(e)\}$$

Semantica astratta

$B = 2^{Ass}$, dove Ass corrisponde agli assegnamenti presenti nel codice. Questo dominio forma un reticolo completo con relazione d'ordinamento $\supseteq$.

$$\begin{aligned}
[\![;]\!]^\sharp B &= B \\
[\![NonZero(e)]\!]^\sharp B &= [\![Zero(e)]\!]^\sharp B = B \setminus Ass(e) \\
[\![x \leftarrow e]\!]^\sharp B &= \begin{cases} B \setminus (Occ(x) \cup Ass(e)) \cup \{x \leftarrow e\} & \text{if } x \notin Var(e) \\ B \setminus (Occ(x) \cup Ass(e)) & \text{altrimenti} \end{cases} \\
[\![x \leftarrow M[e]]\!]^\sharp B &= B \setminus (Occ(x) \cup Ass(e)) \\
[\![M[e_1] \leftarrow e_2]\!]^\sharp B &= B \setminus (Ass(e_1) \cup Ass(e_2))
\end{aligned}$$

1.4.2 Available expressions

Available expression : Un' espressione e si dice *available* in v se è stata valutata prima di v, il suo valore è stato assegnato ad x e $x, Var(e)$ non vengono modificate tra la definizione e v.

Tipologia analisi

Forward, Definite

Equazioni

$$AvailIn(b) = \begin{cases} \emptyset & \text{se } b = entry \\ \bigcap\limits_{p \in succ(b)} AvailOut(p) & \text{altrimenti} \end{cases}$$

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

Un assegnamento $x \leftarrow e$ è ucciso in un blocco b se una variabile $y \in e$ è modificata o se $x \in e$.

$$Kill(b) = \{x \leftarrow e | \exists y \in Var(e).(y \leftarrow e' \in b \vee y = x)\}$$

Un assegnamento $x \leftarrow e$ è generato in un blocco b se $x \leftarrow e \in b$ e $x \notin Var(e)$

$$Gen(b) = \{x \leftarrow e | x \leftarrow e \in b \wedge x \notin Var(e)\}$$

Semantica astratta

$B = 2^{Ass}$, dove Ass corrisponde agli assegnamenti presenti nel codice. Questo dominio forma un reticolo completo con relazione d'ordinamento $\supseteq$.
$Occ(x) = \{y \leftarrow e | x = y \vee x \in Var(e)\}$

$$\begin{aligned} [\![;]\!]^\sharp B &= B \\ [\![NonZero(e)]\!]^\sharp B &= [\![Zero(e)]\!]^\sharp B = B \\ [\![x \leftarrow e]\!]^\sharp B &= \begin{cases} B \setminus Occ(x) \cup \{x \leftarrow e\} & \text{if } x \in Var(e) \\ B \setminus Occ(x) & \text{altrimenti} \end{cases} \\ [\![x \leftarrow M[e]]\!]^\sharp B &= B \setminus Occ(x) \\ [\![M[e_1] \leftarrow e_2]\!]^\sharp B &= B \end{aligned}$$