

A.A. 2021/22

FONDAMENTI DI
ANALISI E VERIFICA DEL SOFTWARE

PROF. ISABELLA MASTROENI

FABS :)

NOTA

Questi appunti/sbominatura/versione “discorsiva” delle slides sono per mia utilità personale,
quindi pur avendole revisionate potrebbero essere ancora presenti typos, commenti/aggiunte personali (che
anzi, lascio di proposito) e nel caso peggiore qualche inesattezza!

Comunque spero siano utili! 

Appunti e puttane sono scritte *principalmente* in questo stile.

**Questo file fa parte della mia collezione di sbobinature,
che è disponibile (e modificabile!) insieme ad altre in questa repo:
<https://github.com/fabfabretti/sbominamento-seriale-uniVR>**

INDICE

NOTA	2
Indice.....	3
I concetti base dell'analisi dei programmi	5
Modellazione dei programmi	11
Control flow graph	14
Approssimazione	16
Significato dell'approssimazione.....	21
analisi statiche	24
Data flow analysis. Available expression.....	26
Data flow: analisi statica e approccio semantico	31
Funzione distributiva vs non distributiva.....	36
Data flow: sintesi sull'approccio	38
Data flow: Analisi di liveness.....	41
Data flow: true liveness. Falsi positivi dell'analisi di liveness non dovuti all'approssimazione.....	47
Data flow: Copy propagation.....	50
Data flow: Reaching definitions.....	54
Data flow: sintesi finale.....	58

I CONCETTI BASE DELL'ANALISI DEI PROGRAMMI

L0.1 - Introduzione - I concetti base

Analisi e verifica

Cerchiamo di capire **cosa significa analizzare e verificare un sistema informatico**. Innanzitutto, si parte dall'idea di capire se quello che abbiamo sviluppato fa esattamente quello che vogliamo.

Il nostro sistema fa esattamente quello per cui è stato sviluppato?

La risposta arriva dall'analisi della progettazione e dello sviluppo, e dalla conoscenza che possiamo acquisire attraverso l'analisi e la verifica del nostro prodotto.

L'obiettivo dell'analisi è quindi quello di garantire in qualche modo che un SW faccia esattamente quello che ci si aspetta.

Per esempio:

- Se il software **interagisce con mondo esterno** devo **garantire che non sarà indotto a violare la sicurezza del computer**, o se stiamo parlando di un software di controllo delle macchine voglio garantire che il software non porti la macchina a causare un incidente.
- Se il software **gestisce immagini mediche**, vogliamo **garantire che non introduca pixel** (perché a volte anche piccole macchie sono indici di cose).

L'esecuzione dipende dal significato di ciò che è scritto nel codice: quindi vogliamo estrarre e analizzare il significato, ovvero la semantica.

Def: Semantica

La semantica di un programma è la **descrizione** (generalmente **formale**) **dell'esecuzione** di un programma.

Def: Proprietà semantica

Una proprietà semantica è una **proprietà del comportamento a tempo di esecuzione**.

Verificare che un SW faccia esattamente quello che ci si aspetta, quindi, corrisponde a **verificare proprietà del comportamento a tempo di esecuzione**, e quindi **verificare proprietà semantiche**.

Quando si fa analisi

L'analisi permette agli sviluppatori di fare la quality assurance. Questo è lo scopo principale, ma è usata anche per mantenere il codice.

- **Sviluppo:** quality assurance
- **Mantenimento del SW:** consiste nella correzione di errori e integrazioni di nuove funzionalità. Una modifica potrebbe generare nuove situazioni inattese, quindi è importante usare strumenti di analisi per garantire che il programma sia ancora funzionante, oppure per acquisire informazioni su quello che viene detto legacy code come ad esempio cobol.
- **Sicurezza:** serve a garantire determinate proprietà di sicurezza.
- **Software security/malware**

Insomma, è importantissimo comprendere la semantica del codice per tantissimi scopi diversi

Definizioni

Programmi

In base all'obiettivo:	In base al metodo:
<ul style="list-style-type: none">• Domani analysis 🔎: possiamo costruire delle analisi specifiche per un programma o per una famiglia di programmi. Sarà molto efficiente, ma poco sfruttabile.• Analisi generiche 🌎: hanno obiettivo generico, e sono usate da compilatori, interpreti o ambienti che non trattano applicazioni specifiche. Per esempio, l'analisi per capire cosa fa un programma sono di questo tipo.	<ul style="list-style-type: none">• Analisi a livello di programma: sono eseguite su codice sorgente del programma, e tipicamente coinvolgono un procedimento simile a un interprete/compilatore: si ricostruisce l'albero di parsing per capire quello che il programma può eseguire a livello approssimato.• Analisi a livello di modello: costruisco un modello sul programma, e poi eseguo l'analisi sul modello. Un esempio è il control flow graph.

Proprietà

- **Proprietà di safety**

Esclude specifici comportamenti (in tempo finito). Sono proprietà che posso verificare durante l'esecuzione, e stabilire su quanto dell'esecuzione è già avvenuto: per esempio, "non sono in deadlock", controllo degli accessi...

- **Proprietà di liveness**

Garantisce il verificarsi di comportamenti attesi

- **Proprietà di information flow**

Assomigliano a quelle di safety nel senso che posso stabilirle in un tempo (es. "non è avvenuto questo flusso di informazione"), ma sono diverse perché non posso verificarle su una singola esecuzione: solo confrontando numerose esecuzioni posso dire se è avvenuto un flusso di informazioni.

Tipo di analisi

La distinzione è fra statica e dinamica.

- **Statica:** eseguiamo l'analisi senza eseguire il programma. È eseguita *prima* dell'esecuzione, quindi cerco di estrarre informazioni prima. È one shot, e può essere meno efficiente per garantire precisione.

Questo è possibile perché:

- Tutto ciò che verrà eseguito è nel codice sorgente: quindi qualunque analisi io estraggo, sicuramente contiene l'informazione eseguita a tempo di esecuzione.
- Sappiamo che dire cose sul codice è decidibile, a differenza delle cose sulla semantica. Quindi spostare l'osservazione sul codice consente di avere una risposta.

- **Dinamica:** eseguiamo l'analisi sull'esecuzione del codice. È più facile da progettare, ma deve garantire un'alta performance!

Analizzatore ideale

Idealmente, vorremmo calcolare in modo automatico il risultato in tempo finito.

Def. Analizzatore ideale

Dato un linguaggio di programmazione P e una proprietà di interesse π . $\forall P, \forall \pi$ l'analizzatore **ideale** deve calcolare:

1. in modo completamente **automatico**,
2. il risultato **preciso**,
3. in **tempo finito**.

È ideale perché è impossibile. I linguaggi con cui abbiamo a che fare sono turing-completi, quindi abbiamo i limiti:

- **La terminazione non è decidibile:** non esiste un programma $halt$ tale che $\forall P, halt(P) = true \Leftrightarrow P \text{ terminates}$. Quindi, per esistere, l'analizzatore ideale violerebbe questa proprietà.
- **Teorema di Rice:** non possiamo determinare in modo automatico proprietà semantiche non banali di programmi

Soluzione

Quindi è impossibile analizzare? No, **basta indebolire uno o più di questi limiti** per poter bypassare l'impossibilità. Possiamo fare due cose.

Rinunciare all'automazione

Significa richiedere una certa quantità di input da parte dell'utente; introduce errori.

Accettare risposte approssimate

Rimaniamo su un'analisi automatica, ma rinunciamo a una risposta precisa; accettiamo risultati non accurati, che non significa sbagliati. Per esempio, accettiamo che risponda "non so".

Analisi precisa per un programma $\pi \Rightarrow$

$$\forall P, analisi(p) = true \Leftrightarrow p \text{ soddisfa } \pi.$$

Questo è ideale però!

Possiamo indebolire questo se e solo se in due modi:



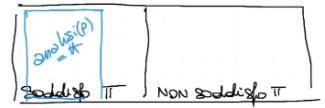
- **Soundness (sovraffermo)**

Pretendo che è vera se la proprietà è soddisfatta.

- $\forall p, analisi(p) = \text{true} \Rightarrow p \text{ soddisfa } \pi$

...Ovvero, se risponde true allora p soddisfa π , mentre se $analisi(p) = \text{false}$ allora non so se p soddisfa o meno π .

È fondamentale quando l'cosa importante è sapere se π è soddisfatta



- **Completeness (sottostimo)**

$\forall p, analisi(p) = \text{true} \Leftarrow p \text{ soddisfa } \pi$,

o anche $\forall p, analisi(p) = \text{false} \Rightarrow p \text{ non soddisfa } \pi$

...cioè se soddisfa la proprietà abbiamo sempre risposta vera; se p non la soddisfa allora potrebbe dare sia vero che falso.

È fondamentale quando l'cosa importante è sapere se π NON è soddisfatta (es. violazione della sicurezza)

- **Restringere la classe di programmi**

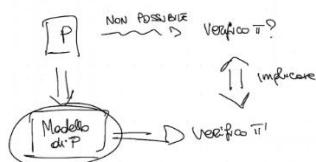
Rinuncio all'automazione

Significa richiedere una certa quantità di input da parte dell'utente.

Famiglie di analisi

Possiamo dividere le analisi in base a cosa indeboliamo.

- **Verifica/model checking:** garantisce il verificarsi di proprietà. Vi rientra il model checking.



- **Analisi statica** (conservativa): estrazione di proprietà senza esecuzione, generando una sovrastima delle proprietà. Tipicamente è automatica, sound ma incompleta.
- **Bug finding:** strumenti che cercano di trovare cause di incorrettezza. È un procedimento automatico; spesso non è corretto/preciso, perché si basa su euristiche.
- **Testing:** è una forma di analisi dinamica che cerca comportamenti anomali del sistema su esecuzioni finite.

Per completare questa parte introduttiva, andiamo queste classi sulle caratteristiche dell'analisi già descritte.

	Dinamica (blackbox) o statica (whitebox)?	Automatica?	Per ogni programma?	Soundness?	Completeness?
Model checking : verifica sul modello di un programma se vale una proprietà di interesse in modo esaustivo	Statico	Sì	No!!!! Lavora su modelli, e non su programmi, e i modelli devono essere finiti.	Sì ma sul modello! Se il modello rappresenta correttamente la realtà allora sì, ma il modello non garantisce che non ci sia un errore di modellazione.	
Analisi statica conservativa	Statica	Sì	Sì	Sì	NO
Bug finding	Statica	Sì	Sì	No	No
Testing	Dinamica	Sì	Sì	No	Sì

Le caratteristiche su cui davvero si approssima quando si cerca di avere una tecnica di analisi automatica e deidibile sono fondamentalmente:

- Correttezza
- Completezza
- Proprietà semplificate

(ultima lezione ultimi minuti TODO)

L0.2 - Introduzione - Esempi e ingredienti

Esempi

Vediamo qualche ingrediente essenziale :)

Buffer overflow

È un attacco abbastanza datato, quindi ormai si presuppone che non sia più possibile; è importante per capire il tipo di attacco possibile. Si basa sul concetto di allocazione delle variabili in un certo modo della memoria.

L'idea di fondo è **che si sfrutta l'allocazione del record di attivazione**: esso è riempito in un certo ordine, e quando poi sovrascriviamo il buffer andiamo in direzione opposta e raggiungiamo/sovrascriviamo il frame pointer e il return pointer, alterando il processo di ritorno.

Quindi, bisogna **capire come è allocata la memoria** e cosa **scrivere per sovrascrivere in modo giusto il buffer**; per far sì che il codice sovrascriva.

Nel caso sopra, se sopra iname mettiamo una dimensione superiore dell'allocazione osvrascriviamo parti importanti di memoria.

Ruolo dell'analisi

È importante, quindi l'analisi: **un'analisi studiata ad hoc permetterebbe di individuare questi flussi di informazioni**. Ma che analisi uso?

Innanzitutto pensiamo ad analisi di **dataflow** (=come i dati viaggiano nelle analisi), ma questo non è esente da problemi:

💡 We have to represent the data flow in C:

```
a = 2;
*p = 3;
...
← is the value of a still 2?
```

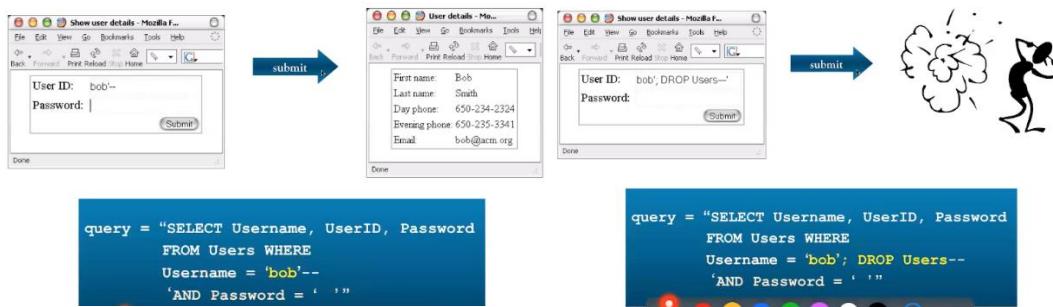
💡 Yes, if we can prove that p cannot point to a

Nel caso sopra, potrebbe succedere che altre variabili (p) sono diventate alias di a e ne hanno modificato il valore.

Webapp security: sql injection

```
String query = "SELECT Username, UserID, Password
                FROM Users WHERE
                username = " + user + " AND
                password = " + password;
```

Questa query è estremamente ingenua: potrei mettere degli input particolari in user e causare iniezione di codice.



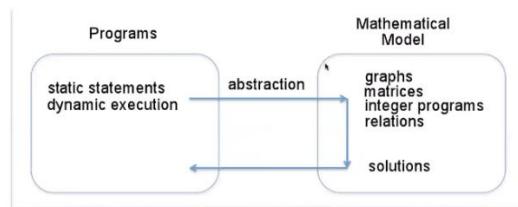
Ruolo dell'analisi

Questo è un **problema di input**, quindi posso analizzarlo staticamente: con la pointer analysis, posso verificare a quali oggetti una particolare variabile punta. È molto pesante quindi richiede di essere approssimata.

La cosa si fa più complicata in linguaggi come Java, dove i riferimenti sono nascosti. In questo caso, bisogna capire staticamente come l'informazione viaggia fra parametri, strutture dati etc. con riferimenti che possono anche essere di tipo alias.

Ingredienti matematici: l'astrazione

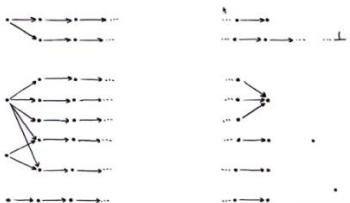
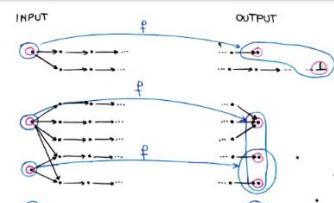
Abbiamo bisogni di strumenti matematici che ci permettano di lavorare sui programmi.



Generality, Power, Simplicity and Efficiency

La semantica è la descrizione formale dell'esecuzione.

I puntini sono gli stati. I puntini sono foto dello stato, la cui granularità dipende da cosa ci interessa analizzare.

Semantica operazionale	Semantica denotazionale
<p>È per quando ci interessa ogni singolo passo: la semantica operazione descrive l'evoluzione della mia macchina al variare dell'input.</p>  <p>Operational semantics</p>	<p>Guarda solo la funzione input/output; è il caso dell'analisi dinamica.</p>  <p>Denotational semantics</p>

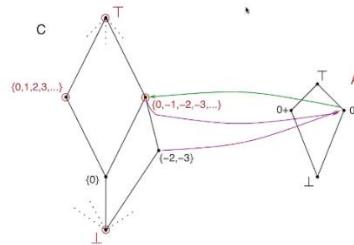
Il concetto di astrazione **non è solo un buttare via dettagli**, ma farlo secondo delle **regole precise**; se seguo queste regole **posso garantire di poter "tornare indietro"** e **estrarre informazioni sull'esecuzione concreta**.

Astrazione matematica

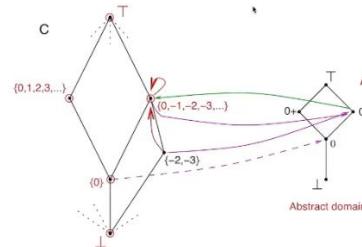
Per poter garantir ciò, quindi, il procedimento **dove garantire determinati vincoli**, fissati dalla teoria dell'interpretazione astratta. Questa teoria mi dà le caratteristiche che deve avere l'osservazione astratta perché io possa studiare nel modello astratto e dedurre una corrispondente proprietà.

Per capire il concetto di astrazione ad alto livello, supponiamo di avere le proprietà dei numeri interi.

Consider $C = \wp(\mathbb{Z})$: [Cousot & Cousot'77]



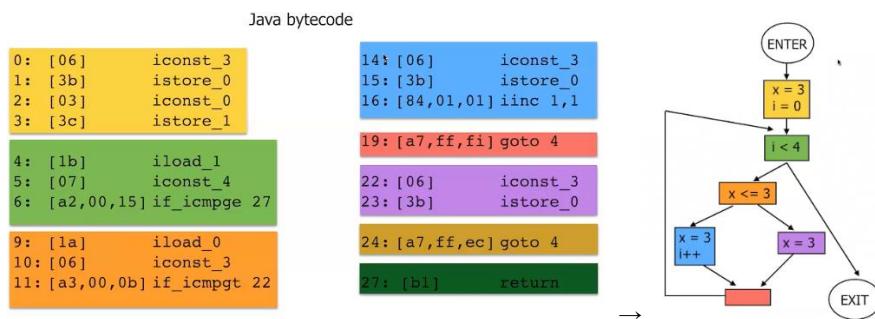
Accetto di approssimare tutto tranne il segno. Il problema qui è che non ho nessun modo di rappresentare lo 0 astrattamente, quindi questo dominio non rispetta i vincoli di un dominio astratto.



Ora va bene.

Control flow graph

È una rappresentazione finita del flusso di controllo dei nostri programmi. Consiste nel dividere il codice o bytecode in gruppi, in modo tale che ciascun gruppo – detto basic block – contenga solo istruzioni sequenziali.



Il CFG è un modello **del tutto equivalente**, quindi **non ho perdita di informazione**; non è un modello astratto, ma un **modello diverso**.

Quello che cambia è che non ho if e while, quindi nel CFG generano semplici archi. Questo semplifica molto rispetto all'operazionale, perché la smallstep fa troppi passi e la bigstep riassume in un solo passo il while – che è un po' too much. Iò CFG permette di approssimare la semantica per però avere un calcolo decidibile.

MODELLAZIONE DEI PROGRAMMI

L1 - Modellare - Linguaggio e semantiche

Parliamo di **come rappresentiamo l'esecuzione dei programmi**. È importante, perché su questi modelli costruiamo la nostra analisi.

Abbiamo già detto che abbiamo bisogno di modelli su cui ragionare; il punto di partenza poi sarà il linguaggio che utilizzeremo.

Partiamo dagli elementi di base. Innanzitutto, per poter eseguire un'analisi sui programmi, dobbiamo:

- **Fissare il linguaggio di programmazione.** Il linguaggio nel caso reale sarà reale; per noi sarà un linguaggino semplice imperativo con elementi base e una sua semantica.
- **Descrivere il modello per ragionare sul programma;** decidiamo di usare:
 - CFG: permette di separare modifica dello stato da controllo
 - Semantica delle tracce: semantica più dettagliata

Il linguaggio IMP

Lo usiamo in tutto il corso.

Sintassi

$m \in \mathbb{V}$ valori $x \in \mathbb{X}$ variabili $\odot ::= + | - | * | ... | \odot ::= < | >$
 $E ::= m | x | E \odot E$ espressioni $B ::= x \odot m$ expr booleane
 $C ::= \text{skip} | C; C | x := E | \text{input}(x) | \text{if}(B) \{ C \} \text{else} \{ C \} | \text{while}(C)$
comandi
 $P ::= C$ programma

Semantica

Caratterizza mediante una funzione matematica gli effetti dell'esecuzione del programma sugli stati della macchina.

Semantica : insieme di stati di input \rightarrow Insieme di stati di output

Di fatto, quindi, **lavora come un interprete**: esegue il programma sugli stati di input. **La differenza è che mentre la semantica è una funzione da insieme ad insieme , l'interprete tipicamente lavora su un singolo insieme input.**

Tipicamente, **le semantiche sono compostionali**. Questo significa che la semantica di un costrutto più complesso è ottenuta come composizione delle semantiche dei costrutti che lo compongono. Quindi, per esempio, la semantica del ; è la composizione dei comandi che compongono le due espressioni.

Rappresentiamo lo stato della macchina come una **fotografia della memoria**. La semantica in generale, quindi, diventa una **trasformazione di memorie**.

Memoria

La memoria è un'associazione variabili-valori, ovvero una funzione

$$M = X \rightarrow V$$

Per semplicità, abbiamo un nostro insieme di variabili V di uno stesso tipo fissato.

Semantiche

Semantica delle espressioni

Categoria sintattica che rappresenta valori

$$\llbracket E \rrbracket : M \rightarrow V$$

Espressioni → categoria sintattica che rappresenta

$$\llbracket E \rrbracket : M \rightarrow V$$

$m \in M$

$$\llbracket n \rrbracket(m) = n$$

$$\llbracket x \rrbracket(m) = m(x)$$

$$\llbracket E_1 \circ E_2 \rrbracket(m) = P_{\circ}(\llbracket E_1 \rrbracket(m), \llbracket E_2 \rrbracket(m))$$

$$\llbracket B \rrbracket : M \rightarrow B = \{t, f\}$$

$$\llbracket x \odot m \rrbracket(m) = P_{\odot}(m(x), m)$$

Semantica dei comandi

La semantica di un comando prende un insieme di memorie e restituisce un insieme. È una trasformazione di insiemi di memorie.

$$\llbracket C \rrbracket : I(M) \rightarrow I(M)$$

$$\llbracket \text{skip} \rrbracket(M) = M$$

$$\llbracket C_0; C_1 \rrbracket(M) = \llbracket C_0 \rrbracket(\llbracket C_1 \rrbracket(M))$$

$$\llbracket x := E \rrbracket(M) = \{m[x \mapsto \llbracket E \rrbracket(m)] \mid m \in M\}$$

$$\llbracket \text{input}(x) \rrbracket(M) = \{m[x \mapsto m] \mid m \in M, m \in V\}$$

$$\llbracket \text{if } B \text{ then } C \text{ else } C' \rrbracket(M) = \llbracket C \rrbracket(\llbracket B \rrbracket(M)) \cup \llbracket C' \rrbracket(\llbracket B \rrbracket(M))$$

$$\llbracket \text{while } B \text{ do } C \rrbracket(M) = \{m \in M \mid \llbracket B \rrbracket(m) = t\}$$

$$\llbracket \text{while } B \text{ do } C \rrbracket(M) =$$

E il while?

Per il while dobbiamo costruire man mano l'esecuzione dopo i vari passi. Di fatto eseguo il corpo e retesto la guardia; continuiamo a eseguire il comando su un insieme di memoria risultanti dall'esecuzione precedente.

Definiamo M_i come le memorie risultanti dopo aver eseguito il while i volte

$$M_i = \mathfrak{F}_{\neg B}(\llbracket C \rrbracket \circ \mathfrak{F}_B)^i(M)$$

E ad esempio

$$M_1 = \mathfrak{F}_{\neg B}(\llbracket C \rrbracket(\mathfrak{F}_B(M)))$$

Solitamente, lo scopo di usare questa i non è quello di capire cosa risulta dopo i volte, ma piuttosto capire il numero massimo di esecuzioni; cosa succede dopo che finisce il while? Filtro il risultato con le memorie che rendono falsa la guardia.

Definiamo quindi M_i come l'insieme di stati che avessi come risultato se applicassi il while i volte.

Prendiamo l'unione di queste memorie: esisterà sempre, per i cicli terminanti, un i massimo dopo il quale M_i sono costanti; ovvero non ho più memorie che rendono vera la guardia, e quindi il calcolo resta sempre uguale.

Al contrario, se il ciclo non termina continuo per sempre a calcolare l'iterazione.

Quindi la semantica del while è l'unione di queste due condizioni:

$$\bigcup_i M_i = \bigcup_i \mathfrak{F}_{\neg B}((\llbracket C \rrbracket \circ \mathfrak{F}_B)^i(M))$$

essendo la funzione \mathfrak{F}_B additiva, posso portare l'unione dentro.

$$\bigcup_i M_i = \mathfrak{F}_{\neg B} \bigcup_i ((\llbracket C \rrbracket \circ \mathfrak{F}_B)^i(M))$$

Quindi, non è altro che l'unione di tutte le esecuzioni del while di tutte le volte necessarie a farlo terminare (o divergente se non è terminante). Nel caso in cui termina, prendo di tutte le memorie risultanti quelle che rendono falsa la guardia – e che quindi fanno uscire dal while

Questo può anche essere scritto come

$$\bigcup_i M_i = \bigcup_i \ellfp_M F \text{ con } F : M' \rightarrow M \cup \llbracket C \rrbracket \circ \mathfrak{F}_B(M')$$

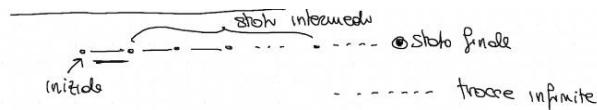
Con ℓfp , ovvero il least fixpoint. Spiega questa roba nella parte che non chiede quindi apposht.

La semantica denotazionale, quindi, vede il while come un'unica istruzione.

Semantica operazionale/transizionale

Va a guardare **come l'esecuzione del programma cambia step-by-step lo stato della macchina**. La principale è proprio il while!

Richiede di descrivere il programma come **l'insieme di tutte le sequenze di transizioni di stati, a partire da quelli iniziali**. Queste sono le esecuzioni del programma, e lo strumento matematico che permette di rappresentarlo è un sistema di transizione.



Anche questa viene descritta tramite un punto fisso: esso è l'iteratore di stati, ch elocalmente trasforma il singolo stato.

È una **semantica di fixpoint**, ovvero **costruita per punto fisso**. Questo è molto importante, perché significa che **posso costruirla in modo iterativo e costruttivo**.

Semantica di punto fisso

Una semantica di punto fisso è una semantica dove ho:

- Un dominio semantico D_{\leq} di ordine parziale
- Un trasformatore semantico $F : D \rightarrow D$ monotono, totale e iterabile

Sistema di transizione

Un sistema di transizione è una coppia $\langle \Sigma, \tau \rangle$ dove

- Σ è un insieme di stati non vuoto
- τ è una relazione di transizione tra stati, quindi $\tau \subseteq \Sigma \times \Sigma$

Stati appartenenti a τ

Una coppia di stati appartiene a τ se posso transizionarvi, ovvero

$$(\sigma, \sigma') \in \tau \text{ se } \sigma \xrightarrow{\tau} \sigma'$$

Stato terminale

Uno stato σ è terminale se per ogni altro stato sigma non può andare mediante τ in σ'

$$\forall \sigma', \sigma \not\rightarrow_{\tau} \sigma'$$

Calcolo di punto fisso delle tracce finite

Prendo gli stati terminali e guardo tutte le tracce che in un passo terminano in quegli stati.

Si possono anche costruire quelle infinite, ma poi il calcolo non è decidibile.

Def. Semantica operazionale formale sulla sintassi

$$[\cdot] : M \rightarrow M$$

$$[\text{skip}] (m) = m \quad [C_1; C_2] (m) = [C_1] ([C_2] (m))$$

$$[x := E] m = m[x \mapsto [E] m]$$

$$[\text{input}(x)] m = m[x \mapsto m]$$

$$[\text{if } B \text{ then } C_1 \text{ else } C_2] m = \begin{cases} [C_1] (m) & [\text{if } B] (m) = \text{tt} \\ [C_2] (m) & [\text{if } B] (m) = \text{ff} \end{cases}$$

$$[\text{while } B \text{ do } C] m = \begin{cases} [C; \text{while } B \text{ do } C] (m) & [\text{if } B] (m) = \text{tt} \\ m & [\text{if } B] (m) = \text{ff} \end{cases}$$

$$\begin{aligned} X_0 &= \emptyset \\ X_1 &= \{ \circ \} \quad \text{stati terminali} \\ X_2 &= \{ \circ \} \cup \{ \xrightarrow{\cdot} \circ \} = \{ \circ, \xrightarrow{\cdot} \circ \} \\ X_3 &= \{ \circ \} \cup \{ \xrightarrow{\cdot} \circ, \xrightarrow{\cdot \rightarrow \cdot} \circ \} \\ &\vdots \\ X^m &= \{ \circ, \xrightarrow{\cdot} \circ, \xrightarrow{\cdot \rightarrow \cdot} \circ, \dots, \xrightarrow{\cdot \rightarrow \dots \rightarrow \cdot} \circ \} \end{aligned} \quad F^t$$

$$\sum^+ = \bigcup_{n=0}^{\infty} F^n$$

CONTROL FLOW GRAPH

L2 – Modellare

Definizione

È un **grafo diretto**, generato a partire dalla sintassi del programma. Rivediamo il modello del programma, visualizzandolo ed evidenziandone la struttura di controllo.

Questo è utile per analisi statica, testing, debugging... È quindi molto usato in letteratura.

$$G = (N, E)$$

$n \in N$ è un **basic block**; un basic block è una sequenza massimale di istruzioni con singola entrata e singola uscita. Ho un punto di ingresso e un punto di uscita. Per semplicità ipotizziamo un n_o unico punto di ingresso del programma, e n_f unico punto di uscita; se non ci sono basta fare un sovrannodo.

$e \in E \subseteq N \times N$, con $e = (n_i, n_j)$ descrive un **possibile trasferimento di controllo** dal blocco n_i al blocco n_j .



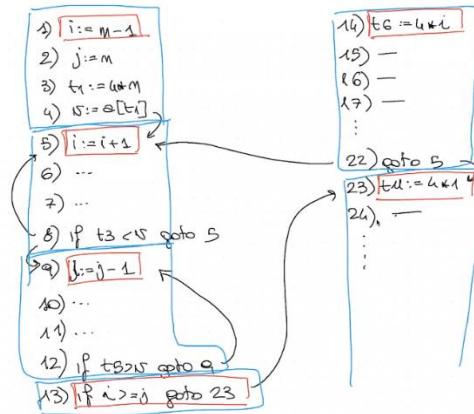
Costruire i basic block partendo da un programma

È un algoritmo.

Input: sequenza di istruzioni

1. **Determiniamo i leaders.** I leaders sono delle istruzioni che formeranno la prima istruzione di ogni basic block. Possono essere:
 - a. **Entry point** di una sequenza di istruzioni
 - b. Ogni istruzione **raggiunta da più archi entranti**
 - c. Qualunque istruzione che **segue un branch. O un return.**
2. **Costruiamo il basic block** come costituzione di leader + sequenza di istruzioni fino al leader successivo, escluso.

Esempio



Nozioni sui CFG

Dato un $CFG = (N, E)$

<p><i>Predecessore e successore</i> Se $\exists e = (n_i, n_j) \in E$, n_i è detto predecessore di n_j e n_j è il successore di n_i. Posso usare questo per creare insiemi: $Pred(n) =$ insieme di tutti i predecessori di n</p>	<p><i>Successori</i> $Succ(n) =$ insieme di tutti i successori di n</p>
<p><i>Join node</i> <i>Join node</i> nodo con più di un predecessore.</p>	<p><i>Branch node</i> È un nodo con più di un successore</p>

Linguaggio IMP-CFG

Spostandoci nel grafo, **trasformiamo le informazioni di controllo negli archi**, e sulle etichette rimane solamente la **modifica alla memoria**. È quindi importante capire il linguaggio del CFG, ovvero il linguaggio IMP-CFG – che è il linguaggio delle etichette di CFG generato da programmi in IMP.

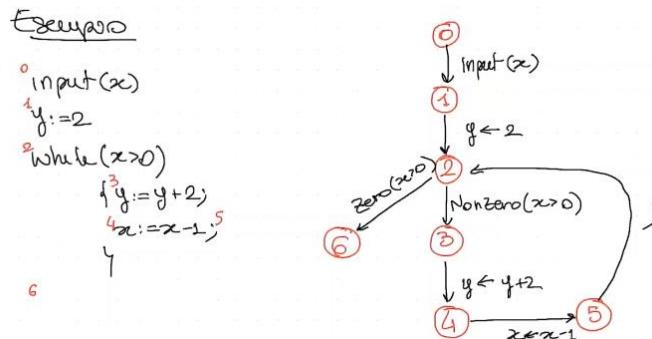
Ogni nodo corrisponde a un punto di programma. Decidiamo per comodità di far corrispondere ogni singolo nodo a un punto di programma.

Ogni arco è etichettato con la modifica dello stato corrispondente all'istruzione eseguita; le etichette, di fatto, sono comandi. Possiamo avere:

- **Test:** archi uscenti da un branch node: Nonzero(e) o zero(e), ovvero rispettivamente “è vero” o “è falso”
- **Assegnamento:** giusto per distinguere da IMP, la indichiamo con freccia \leftarrow
- **Comando vuoto:** ; (di là è skip)
- **Input:** input(x)

Non abbiamo altre istruzioni, perché **if e while sono integrati negli archi** quando costruiamo il CFG.

Vedremo che possiamo costruirli sia **algoritmicamente** con analisi statica e basic block, sia **accademicamente a mano** come faremo noi: ogni punto di comanda è un nodo e ogni comando diventa l'etichetta dell'arco uscente. Esempio:



Def. Arco

L'arco quindi è una **tripla** $k = (u, lab, v)$ che sono rispettivamente il **predecessore**, l'**etichetta** e il **successore**.

Def. Semantica

Describe l'effetto dell'arco sullo stato. È chiamato anche **abstract edge effect**, perché descrive l'effetto dello statement corrispondente all'etichetta sullo stato.

Una computazione, quindi, è un **cammino sul control flow graph**; aka una **sequenza di archi**:

$$\prod_{\text{con } k_i = (u_i, \text{lab}_i, u_{i+1})} = k_1 \dots k_n$$

u nodo iniziale e $u_1 = u$
v nodo finale e $u_n = v$.

m memoria

$$[\![;]\!](m) = m$$

$$[\![\text{zero}(e)]\!](m) = m \quad \text{se } [\![e]\!](m) = \text{falso}$$

$$[\![\text{Nonzero}(e)]\!](m) = m \quad \text{se } [\![e]\!](m) = \text{vero}$$

$$[\![x \leftarrow e]\!](m) = m [x \mapsto [\![e]\!](m)]$$

$$[\![\text{input}(x)]\!](m) = m [x \mapsto m(x)]$$

contiene valori iniziali

Dato che la semantica si può fare per composizione, possiamo dire che (**attenzione all'ordine inverso!**):

$$[\![\prod]\!] = [\![k_n]\!] \circ \dots \circ [\![k_2]\!] \circ [\![k_1]\!]$$

APPROSSIMAZIONE

L3 – Approssimare – Approssimare i dati

Cerchiamo di capire *come* approssimare i dati, e scegliere l'informazione che ci interessa sul nostro programma.

Partiamo da una proprietà sull'esecuzione dei programmi.

Proprietà di un programma

Una proprietà è **l'insieme di tutte le esecuzioni che godono di quella proprietà** che rappresentiamo – o anche, la proprietà rappresentata da un insieme non è altro che **l'invariante che accomuna tutte e sole le proprietà dell'insieme stesso**.

$\llbracket P \rrbracket \subseteq Q$ = tutte le esecuzioni di P devono essere in Q, ovvero godere della proprietà rappresentata da Q.

Problema...	... e soluzione
Esiste il teorema di Rice. In generale, testare che una semantica ogda di una certa proprietà... non è decidibile!	Approssimazione! Costruiamo una semantica approssimata che rappresenta con un certo grado di errore la semantica concreta. In pratica, la semantica approssimata contiene tutte le esecuzioni che abbiamo nel concreto.

Ovviamente ci serve un vincolo per poter guadagnare qualcosa: vogliamo che il test per la nostra specifica proprietà Q sia decidibile; **questo è possibile perché costruiamo la semantica specificatamente in funzione della proprietà Q da dimostrare.**

$$\llbracket P \rrbracket^{\#} \supseteq \llbracket P \rrbracket$$

$$\llbracket P \rrbracket^{\#} \subseteq Q \text{ deve essere decidibile}$$

Approssimare

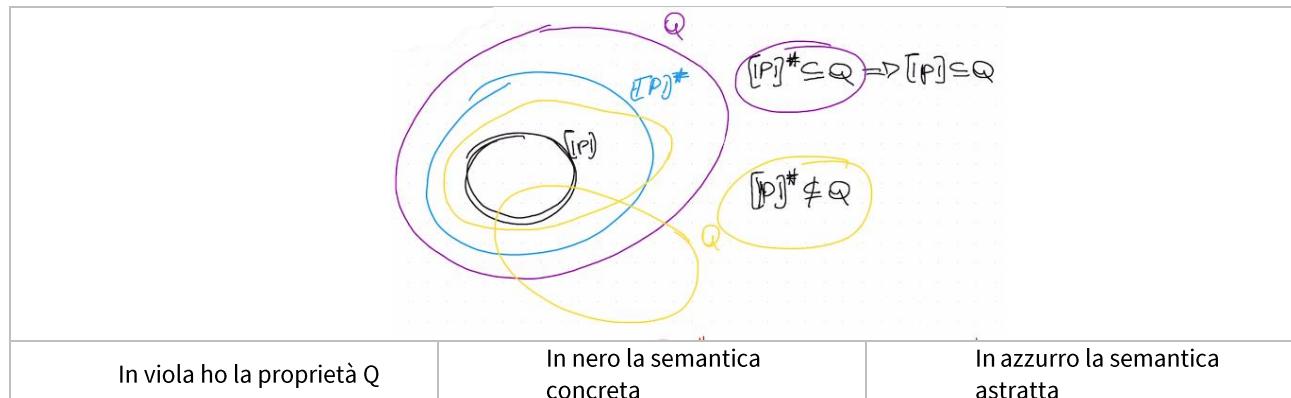
Approssimare vuol dire fare in modo che questa semantica astratta contenga Q, e che il test “gode della proprietà Q sia decidibile”.

Questo funziona perché una volta che garantiamo che la semantica approssimata contenga la semantica concreta,, allora per la transitività del contenimento abbiamo che anche la semantica concreta sicuramente soddisfa Q.

$$\llbracket P \rrbracket^{\#} \subseteq Q \Rightarrow \llbracket P \rrbracket \subseteq \llbracket P \rrbracket^{\#} \subseteq Q$$

Avendo che $\llbracket P \rrbracket^{\#} \subseteq Q$ è decidibile, diventa anche automatico che P soddisfa la proprietà. Viceversa, quando la proprietà P non soddisfa Q allora non possiamo dire nulla.

Graficamente:



Se la semantica astratta gode della proprietà Q – ovvero è contenuta nella proprietà Q – allora automaticamente, per transitività, anche la semantica P è contenuta nella proprietà Q.

In questo senso abbiamo una risposta approssimata: in un senso, abbiamo una risposta certa; nell'altra non lo abbiamo. Quindi ora vogliamo capire **come costruire questa semantica approssimata $\llbracket P \rrbracket^*$ a partire dalla semantica concreta $\llbracket P \rrbracket$** .

Astrarre la semantica

Abbiamo già definito la semantica come coppia $\langle \text{funzione}, \text{dominio} \rangle$. Con D ordinato e la funzione $f: D \rightarrow D$ con fixpoint.

Dobbiamo fare due cose:

- **Approssimare il dominio di dati D come un dominio che contiene oggetti astratti.** Questi oggetti astratti sono le osservazioni astratte e la loro relazione con gli oggetti concreti.
- **Approssimare la funzione f , che è la computazione, sugli oggetti astratti.** Ovvero specificare come la semantica manipola queste osservazioni astratte.

Procedimento

Dobbiamo **individuare la proprietà astratta che vogliamo osservare** e che vogliamo manipolare al posto degli oggetti concreti. Fissati gli oggetti concreti, fissiamo quale sia l'osservazione di questi oggetti concreti che vogliamo manipolare in astratto.

Scegliere l'osservazione = individuare cosa vogliamo osservare con precisione.

Tutto ciò che non mettiamo nell'osservazione astratta è ciò su cui aggiungiamo errore.

Avendo quindi:

- Σ insieme di oggetti concreti, su cui lavora normalmente la semantica
- $\wp(\Sigma)$ insieme delle proprietà degli elementi di Σ
- **Dominio astratto:** è un sottoinsieme di parti di Σ . Ovvero è l'**insieme di tutte quelle proprietà che vogliamo osservare con precisione**.

$A \subseteq \wp(\Sigma)$, dove A è l'**insieme delle proprietà (non tutte) sugli elementi di Σ osservate con precisione**

Per esempio, scegliendo come proprietà $A = \text{segni}$, prendiamo l'insieme

$\{n \mid n < 0\} \in A$ = è osservato precisamente; se calcoliamo esattamente tutti i numeri negativi, questo insieme lo troviamo con precisione.

Viceversa, se il nostro programma calcolasse solo $\{-1, -2\} \notin A$. Questo insieme non è in A , quindi non possiamo osservarlo con precisione; noteremo la proprietà invariante dei suoi elementi, ma dato che stiamo buttando via i valori aggiungiamo rumore (=abbiamo aggiunto TUTTI gli elementi che hanno la proprietà "numero negativo", non solo -1 e -2).

Esempi di oggetti che possiamo approssimare:

- **Valori:** booleani, interi... v
- **Memorie:** \mathbb{V} insieme di variabili $\Rightarrow M: \mathbb{V} \rightarrow v$
- **Stack, heap** e tutti gli oggetti concreti che possono far parte di una computazione
- ...

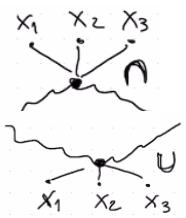
Esempi di proprietà sui valori:

- **Numeri dispari:** $odd = \{n \mid n \bmod 2 = 1\}$
- **Numeri pari:** $even = \{n \mid n \bmod 2 = 0\}$
- **Range di valori:** $\{n \mid \min \leq n \leq \max\}$
- ...

In generale, quando abbiamo un oggetto concreto, un qualunque insieme $I \in \wp(\Sigma)$ (=alle parti di sigma, aka qualunque proprietà di Σ), rappresenta una proprietà invariante tra tutti gli elementi di I .

Proprietà degli insiemi di proprietà di un insieme di oggetti concreti

- Σ Insieme di **oggetti concreti**
- $\wp(\Sigma)$ Insieme delle **proprietà** su Σ



Questa collezione è un **reticolo completo** (distributivo ma lo vediamo dopo)

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

- \subseteq = contenimento = implicazione logica
- \emptyset = elemento più piccolo di $\wp(\Sigma)$ = falso (non è un oggetto)

- \cup = least upper bound (lub) = or logico: unione di tutti gli elementi che contengono

- | | |
|---|--|
| <ul style="list-style-type: none"> • $\Sigma = \text{elemento più grande di } \wp(\Sigma) = \text{vero}$
(qualunque oggetto) | <ul style="list-style-type: none"> • $\cap = \text{greatest lower bound (glp)} = \text{and logico: il più grande di quelli che stanno sotto}$ • $\neg = \text{complemento} = \text{negazione}$ |
|---|--|

Esempi:

$$\begin{aligned} & \{m \mid m \bmod 2 = 0\} \cup \{n \mid n \bmod 2 = 1\} = \Sigma \\ & \neg \{m \mid m \bmod 2 = 0\} = \{m \mid \neg (m \bmod 2 = 0)\} \\ & \quad = \{m \mid n \bmod 2 = 1\} \end{aligned}$$

16:47 TODO

How to send a screenshot via email, according to my 60 yo coworker:
 - Go to the page you wanna screenshot
 - Hit the PrntScr button and paste the image on Word
 - Print the Word page
 - Scan the printed page on the company printer
 - Send the scanned image via email

1:50 PM · May 2, 2023 · 99 Views

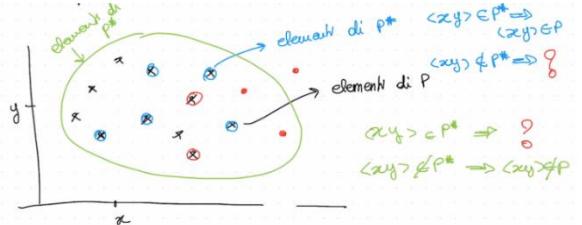
Direzione approssimazione

Fissato che vogliamo approssimare, non c'è una sola direzione di approssimazione; ne esistono fondamentalmente 2.

- $P \subseteq P^{\#}$ **Approssimazione per eccesso:** è quella che abbiamo appena descritto; la semantica concreta è contenuta nella semantica approssimata
- $P^{\#} \subseteq P$ **Approssimazione per difetto:** prendiamo e approssimiamo la semantica togliendo elementi, aka riducendo quello che osserviamo.

Osserviamo che:

- $\langle x, y \rangle \in P^{\#} \Rightarrow \langle x, y \rangle \in P$
(perché non abbiamo preso nulla in più!)
- $\langle x, y \rangle \notin P^{\#} \Rightarrow ???$ (non posso dir nulla)



Al contrario: prendo un insieme di elementi che è più grande di P .

$$\langle x, y \rangle \in P^{\#}$$

TIPICAMENTE ci focalizziamo su quelle **per eccesso** perché quelle per difetto sono più difficili da rappresentare; comunque, per farle basta manipolare l'altra versione.

Abbiamo una proprietà concreta di oggetti, tipicamente:

- **Complessa da rappresentare:** non sempre rappresentabile su una macchina
- **Potenzialmente infinita**

La proprietà **astratta** è un **insieme più grande**, e questo la rende **più facile da rappresentare** perché **individuiamo l'invariante** che vogliamo osservare tra gli oggetti concreti.

Esempio

Prendiamo l'insieme $\{2, 3\}$. Anziché rappresentare gli elementi elencandoli, possiamo rappresentare direttamente l'invariante che abbiamo scelto di osservare – in questo caso il **segno negativo**.

→ Estendiamo l'insieme a tutti gli elementi che hanno esattamente quell'invariante: tutti gli elementi negativi!

Esempio2

Un esempio più interessante è ad esempio “**interi negativi pari**”; per rappresentarli nel concreto dovrei elencarli tutti uno alla volta, ma non è possibile dato che sono infiniti; quindi quello che faccio è scegliere di osservare solo l'invariante.

Quindi, aumentando l'insieme semplifichiamo le cose. **Controintuitivo ma actually very nice.**

Minima approssimazione possibile

Un'altra osservazione importante è che non **vogliamo estendere arbitrariamente**, e quindi **non ci va bene un qualunque elemento** di A che approssimi per eccesso la proprietà concreta. **Vogliamo che la nostra approssimazione sia la più piccola possibile.**

Se ogni $\bar{P}^{\#}$ è una approssimazione di P , allora

$$P^{\#} = \cap \{\bar{P}^{\#} \in A \mid \bar{P}^{\#} \supseteq P\} \in A$$

(con A dominio astratto)

Ovvero, se vale allora A contiene la migliore approssimazione possibile per P.

L'interpretazione astratta che vediamo ci garantisce che questa unica, migliore scelta esista. (credo)

Esempio completo: dominio dei segni

Oggetti concreti $\Sigma = \mathbb{Z}$

- **Programmi:** moltiplicazione e addizione
- **Proprietà:** segno dei valori
- **Semantica concreta:** semantica delle operazioni su \mathbb{Z}
- **Semantica astratta:** regola dei segni su \mathbb{Z}

1. Definiamo l'approssimazione dei dati sulla semantica.

Questo consiste nello scegliere quali sono gli elementi che decido di osservare con precisione.

- $\{n \mid n \geq 0\} \equiv 0 +$
- $\{n \mid n \leq 0\} \equiv 0 -$
- $\{0\} \equiv 0$

Di conseguenza nel nostro esempio avrò che

- **Dominio concreto:** $\wp(\mathbb{Z})$
- **Dominio astratto:** $\{\mathbb{Z}, 0+, 0-, \emptyset\} \subseteq \wp(\mathbb{Z})$

(\mathbb{Z} rappresenta l'elemento astratto di cui non ho info sul segno; \emptyset rappresenta il falso e va messo perché di solito nelle analisi serve anche un elemento per il caso di errore. È il punto di partenza.)

Tutti gli elementi di $\wp(\mathbb{Z})$, che è astratto, che non sono nel mio dominio astratto... vengono approssimati.

Quindi le trasformazioni saranno

- $\{n \mid n \geq 0\} \rightarrow 0 +$
- $\{-5, -4\} \rightarrow 0 -$ (abbiamo aggiunto un errore: tutti i numeri negativi!)
- $\{-5, 4\} \rightarrow \mathbb{Z}$

2. Approssimare le operazioni.

In questo caso, praticamente, le operazioni sarebbero la regola dei segni.

$+$ *	$\mathbb{Z}, 0+, 0-, \emptyset$	$*$ *	$\mathbb{Z}, 0+, 0-, \emptyset$
\mathbb{Z}	$\mathbb{Z} \ \mathbb{Z} \ \mathbb{Z} \ \mathbb{Z} \ \mathbb{Z}$	\mathbb{Z}	$\mathbb{Z} \ \mathbb{Z} \ \mathbb{Z} \ 0 \ \mathbb{Z}$
$0+$	$\mathbb{Z} \ 0+ \ \mathbb{Z} \ 0+ \ 0+$	$0+$	$0+ \ 0- \ 0 \ 0+$
$0-$	$\mathbb{Z} \ \mathbb{Z} \ 0- \ 0- \ 0-$	$0-$	$0- \ 0+ \ 0 \ 0-$
0	$\mathbb{Z} \ 0+ \ 0- \ 0 \ 0$	0	$0 \ 0 \ 0 \ 0 \ 0$
\emptyset	$\mathbb{Z} \ 0+ \ 0- \ 0 \ \emptyset$	\emptyset	$\mathbb{Z} \ 0+ \ 0- \ 0 \ \emptyset$

Consideriamo vuoto come elemento neutro, aka come se non stessi facendo niente.

Tipi di approssimazione

Supponiamo di avere il dominio a due dimensioni coi punti calcolati dal programma.

	<p>Segni Sono tutti valori positivi.</p>
	<p>Intervalli Gli intervalli prendono il più piccolo e il più grande e solo l'area in cui si trovano tutti i valori.</p>
	<p>Ottagoni Guardano i punti secondo 8 possibili rette che passano tra i punti. In questo caso, prendo o solo le rette perpendicolari alle assi, o con倾inazione di 45 gradi. È un ottagono che delimita in modo un poco più preciso l'area del dominio.</p>
	<p>Poliedri Non hanno vincoli sull'inclinazione delle rette, per limitare dove sono i nostri punti.</p>
<p>Congruente $3\mathbb{Z} + 5$ $7\mathbb{Z} + 3$</p>	<p>Tutti questi sono domini complessi che riempiono le aree; poi esistono domini non complessi, come le congruenze e che semplicemente elencano tutti i punti distanti un certo tot da un punto di partenza.</p>

SIGNIFICATO DELL'APPROXIMAZIONE

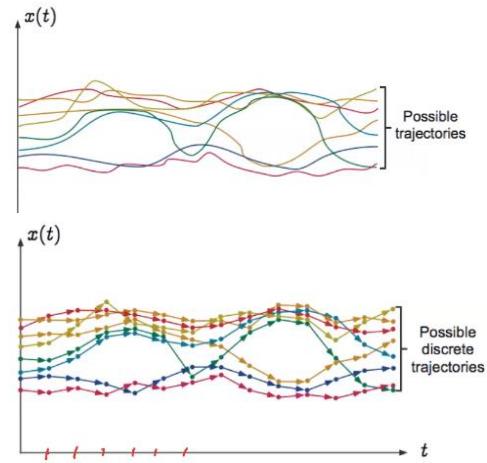
6 - Approssimare - Approssimare le computazioni

Abbiamo detto che l'approssimazione della computazione è **l'approssimare la semantica sul dominio dell'osservazione**.

Traccia

Abbiamo già visto che una **computazione** non è che una **traccia nel tempo del valore dello stato del sistema**. Quindi, a partire da ogni stato iniziale, ho una serie di possibili traiettorie: esse possono essere infinite se la computazione diverse, e il numero di tracce da dover osservare potrebbe essere infinito.

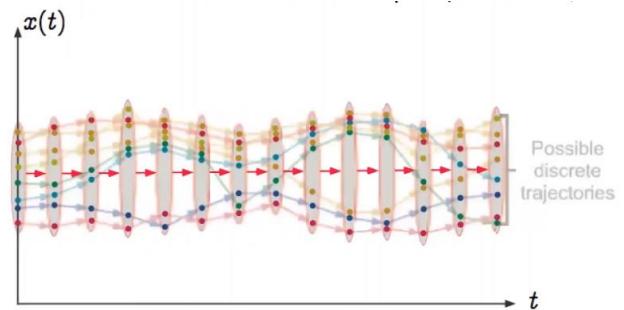
In realtà le traiettorie non sono continue, ma avviene una **discretizzazione delle tracce**: ovvero fissiamo degli step di tempo che tipicamente corrispondono alle singole istruzioni e ai singoli cambiamenti di stato, e spezziamo la traccia discretamente in queste evoluzioni dello stato della macchina.



Calcolo sugli insiemi: collecting semantics

Il primo passo che avviene nella ricerca della decidibilità dell'analisi – aka nel cercare di raggiungere la decidibilità – è sicuramente quello di **non guardare più i signoli stati ma guardarli come insiemi**. È un passo necessario per arrivare a parlare di proprietà, e vogliamo parlare di proprietà per ottenere un'analisi decidibile.

Il primo passo quindi è quello di **trasformare l'insieme di computazioni in un'unica computazione che avviene fra insiemi**. I punti sono comunque **concreti**: dal pov di ciò che possiamo calcolare, aka degli stati che possiamo raggiungere, **non abbiamo perso nessun tipo di informazione**. Questo calcolo non lo vogliamo fare direttamente “oneshot”, perché l'infinità delle traiettorie non viene alterata (abbiamo ancora una traccia di insiemi potenzialmente infiniti)

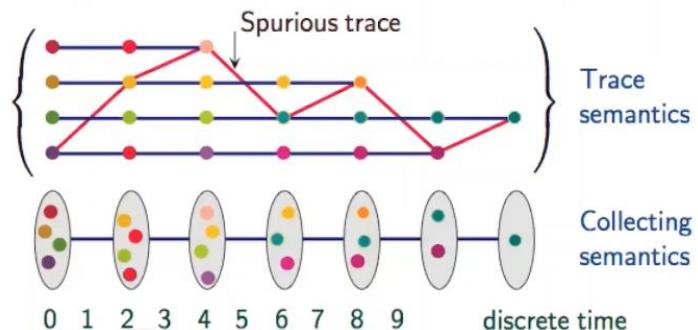


Il calcolo anche sull'insieme **avviene per punto fisso**: partiamo dall'insieme di stati iniziali e andiamo a collezionare via via tutti gli stati che raggiungiamo durante l'esecuzione. Questa è detta anche reachability semantics, o collecting semantics. Abbiamo già descritto questa semantica come la semantica calcolata induttivamente sulla struttura del linguaggio, che a partire da un insieme di stati **collezione gli insiemi di stati che posso raggiungere**.

Ho perdita d'informazione?

1. Dal punto di vista della **raggiungibilità e degli stati**, l'informazione è **precisa**: l'insieme di stati che ho in un punto della computazione è esattamente l'insieme di stati che verrebbe raggiunto nelal semantica concreta.
2. Di fatto però, ho **perdita di informazione sull'insieme delle tracce che sto rappresentando**. Sto mettendo assieme tutte le tracce, quindi non so quale pallino porta a quale! Ho perso informazione rispetto alle tracce che rappresentiamo.

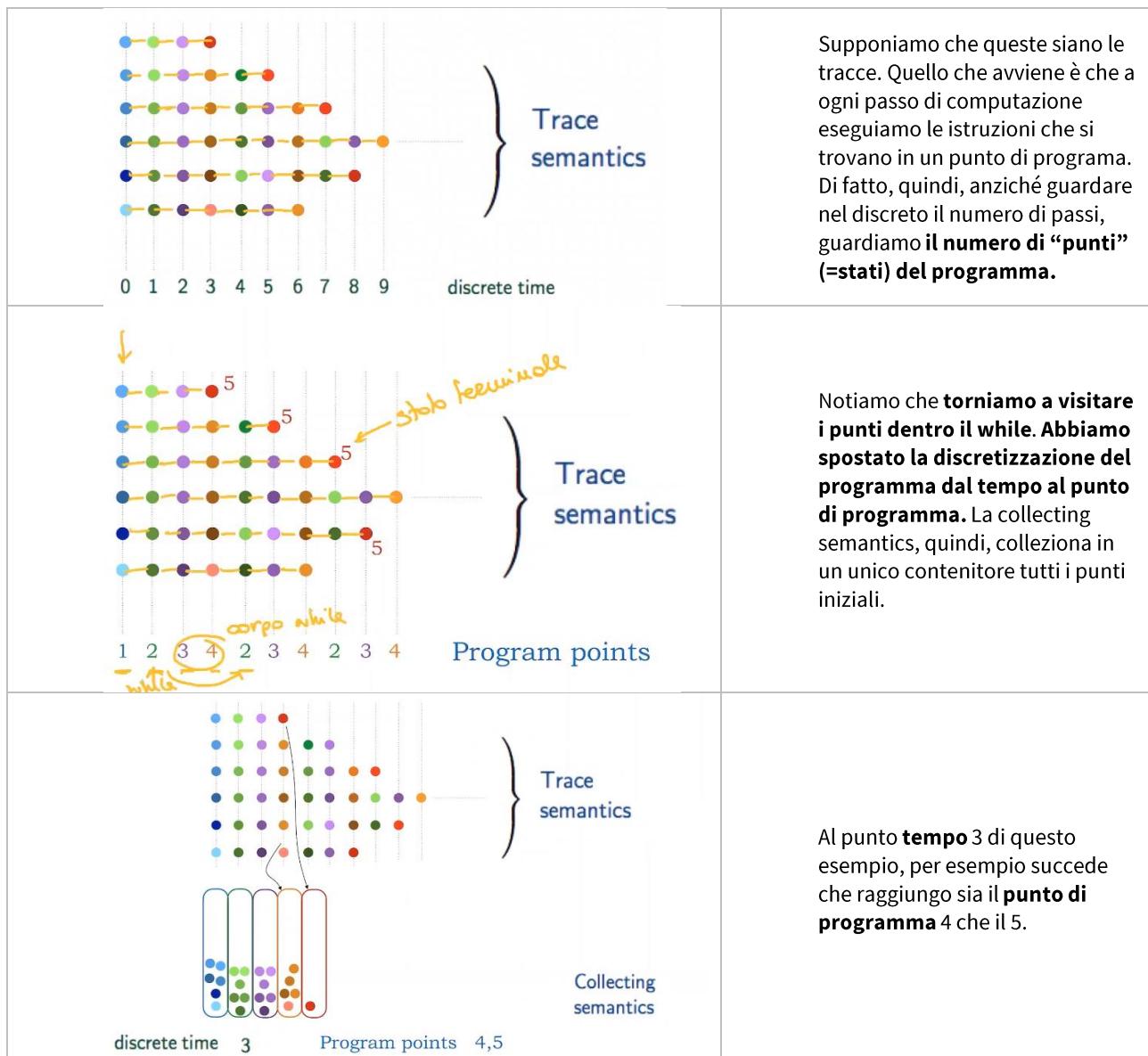
– This is an **abstraction**. Does the red trace exists?
Trace semantics: **no**, collecting semantics: **I don't know**.



Stiamo rappresentando l'insieme massimale di tutte le tracce concrete che hanno questa come traccia astratta.

Collecting semantics e decidibilità

Abbiamo risolto la decidibilità? Non ancora.



Quello che avviene è che **per ogni punto di programma l'insieme degli stati è sempre incrementale rispetto al punto di programma: la semantica è calcolata del tutto solo quando gli stati smettono di crescere**.

Questo **non permette ancora di risolvere la decidibilità** per due ragioni:

1. Stiamo ancora guardando gli insiemi in modo concreto/preciso, e l'insieme raggiungibile è sempre concreto.
2. La **computazione stessa rimane potenzialmente non terminante se ho un ciclo while che non termina**.

Quindi non siamo ancora nelle condizioni di garantire la terminazione. Ma è un passo: ci possono essere dei cicli che non terminano ma non calcolo nulla di nuovo, e quindi gli oggetti "si stabilizzano", come nell'esempio a destra.

In questo senso, la semantica collecting mi fa avvicinare alla decidibilità... ma non sempre, nella maggior parte delle volte le variabili nel while sono modificate e quindi gli insiemi continuano a crescere.

Abbiamo quindi bisogno di qualcosa altro.

$x := 0$
while $x > 0$
 $x := x - 1$ for

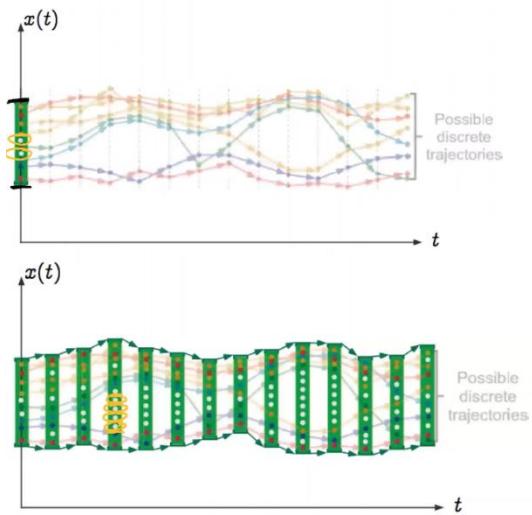
Calcolare sulle proprietà: property semantics

Quello che doviamo fare è fare (per davvero) l'approssimazione: non vogliamo calcolare la computazione tra insiemi di stati raggiungibili, ma tra **proprietà degli stati raggiungibili**. Ci spostiamo sulla proprietà: come per i dati, anziché elencare tutti gli elementi raggiunti **estraiemo la proprietà invariante** che vogliamo osservare aggiungendo ulteriore rumore.

Supponiamo, quindi, che anziché guardare gli elementi singoli guardiamo l'intervallo. La issue è che così aggiungiamo anche quei punti che in verità non ci sarebbero:

Cominciando a calcolare da questo, in realtà il calcolo dell'elemento successivo abbiamo anche degli stati che vengono aggiunti. Quindi non abbiamo solo le computazioni spurie dovute al fatto che stiamo guardando fra insiemi, ma anche computazioni spurie che partono da stati che non vengono mai raggiunti nel concreto.

Il secondo passo è computare il punto fisso su proprietà di stati raggiungibili.



Collecting semantics	Collecting on properties
Esecuzioni spurie ma solo fra stati raggiungibili	Stati spuri (data l'approssimazione), e quindi abbiamo computazioni spurie che includono stati spuri.
<p>Set of traces: α₁ ↓</p> <p>Trace of sets: α₃ ↓</p> <p>Sicure dell'observazione Reachable states</p>	<p>Set of traces: α₁ ↓ <i>passaggio necessario per prendere di proprietà</i></p> <p>Trace of sets: α₂ ↓ <i>passaggio alle proprietà</i></p> <p>Trace of intervals</p>

Riassumendo:

- Vogliamo **considerare proprietà** per approssimare l'informazione per poterla rendere calcolabile ed evitare il problema della non decidibilità
- Ci spostiamo sulle **computazioni di insiemi** anziché su insiemi di computazioni, senza però approssimare I/O, solo la semantica delle tracce (small step) perché ad ogni passo anziché guardare la relazione di transizione fra uno stato e il successivo abbiamo collezionato gli insiemi di stati, perdendo la singola relazione. La calcoliamo per punto fisso, per catturare la convergenza (es. ciclo non terminante che continua a ricalcolare gli stessi stati).
 - Non abbiamo ancora risolto il problema perché la maggioranza delle computazioni continuano a generare nuovi elementi, e anche la computazione degli elementi diverge.
- Ulteriore passo di approssimazione per approssimare la computazione, **guardando le proprietà e la semantica collective sulle proprietà**.
 - Queste proprietà possono già agevolare la terminazione:
In questo caso l'insieme continua a crescere (0,2,4,...), ma la proprietà resta immutata (even) ed è fixpoint.
Ancora non è sufficiente: la computazione può essere non terminante. Non possiamo andare oltre, semplicemente dipenderà dal dominio astratto (nell'esempio sopra, se come dominio astratto scegliessi gli intervalli questo non sarebbe più fixpoint perché farebbe [0,0],[0,2],[0,4]...)

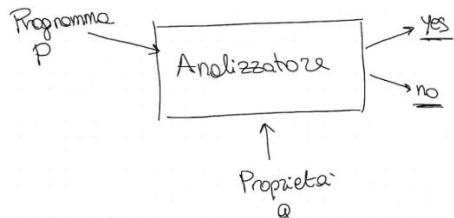
$x = 0$
 while $x > 0$
 $x := x + 2$
 s instead of concrete
 Even still without pp

ANALISI STATICHE

7 - Analisi statica: data flow - Impostazione del problema

Iniziamo a capire cosa si può fare con l'analisi statica. Per distinguerla da quella basata sull'interpretazione astratta la chiamiamo analisi di data flow basata su CFG.

Innanzitutto recuperiamo il significato di analizzatore:



Vogliamo una **risposta sì o no** data su una **approssimazione del significato di P**. Su P riusciamo (in modo decidibile) a dare in generale solo risposte approssimate.

Esempio

<pre> n := n0; i := n; while (i <> 0) do j := 0; while (j <> i) do j := j + 1 od; i := i - 1 od </pre>	<p>Prendiamo un programma che manipola un certo valore j.</p> <p>Cerchiamo proprietà del programma. Per esempio, sapendo che si parte da $n > 0$, possiamo fare alcune valutazioni</p>
<pre> {n0>=0} ← n := n0; {n0=n, n0>=0} i := n; {n0=i, n0=n, n0>=0} while (i <> 0) do {n0=n, i>=1, n0>=i} j := 0; {n0=n, j=0, i>=1, n0>=i} while (j <> i) do {n0=n, j>=0, i>=j+1, n0>=i} j := j + 1 ← j < n0 so no upper overflow {n0=n, j>=1, i>=j, n0>=i} od; {n0=n, i=j, i>=1, n0>=i} ← i > 0 so no lower overflow i := i - 1 {i+1=j, n0=n, i>=0, n0>=i+1} od {n0=n, i=0, n0>=0} </pre>	<p>Le asserzioni vengono ottenute a partire dallo stato terminante; le propaghiamo all'indietro.</p> <p>Su questo tipo di informazione, l'analizzatore riesce a dare risposte.</p>

I nostri ingredienti sono:

- **[[·]] Semantica** con $[[\cdot]] : \mathcal{L} \rightarrow \wp(D)$, dove \mathcal{L} è il linguaggio di programmazione e $\wp(D)$ è l'insieme di denotazioni che descrivono il significato del programma (per noi sarà l'insieme delle tracce di computazioni)
- **Proprietà Q** $\subseteq \wp(D)$, aka è un sottoinsieme di tutte le semantiche ed è quel sottoinsieme che soddisfa l'invariante
- $\wp(D)$ è l'**insieme di tutte le semantiche dei programmi**, quindi X è la semantica di un programma e quindi $X \subseteq D$ e $X \in \wp(D)$
- $Q \subseteq \wp(D)$ dove Q è il **dominio di tutti i programmi che soddisfano la proprietà** rappresentata.

Analisi statiche: overview

Control flow analysis

Proprietà analizzabili sulla **sintassi**.

Data flow analysis

Proprietà che guardano **come l'informazione fluisce dentro il programma**.

Abbiamo già un livello di difficoltà in più: dobbiamo guardare ai dati, e quindi alla **semantica**. È comunque un'analisi che **riesco ad approssimare abbastanza bene dalla sintassi**, perché **non entro nel merito della memoria** e mi basta guardare la relazione sintattica tra gli elementi del programma; di fatto, ragionare esclusivamente sulla sintassi permette lo stesso di raggiungere un livello accettabile di precisione.

Ne vediamo alcune:

- **Available expression:** mi dice se posso evitare di ricalcolare una certa espressione perché già calcolata in un certo punto del programma
- **Copy propagation:** mi permette di vedere se due variabili sono una la copia dell'altra
- **Liveness analysis:** più delicata, è un'analisi che mi dice se una certa definizione è viva perché viene data prima di un utilizzo; quindi se il valore di una variabile definita in un certo punto è effettivamente utilizzabile in un altro punto del programma
- **Reaching definition:** dice in ogni punto del programma quali sono le definizioni di variabili che raggiungono quel punto.

Analisi non distributive

Hanno bisogno di **più strumenti** e le vedremo sui dati: segni, intervalli...

DATA FLO. AVAILABLE EXPRESSION

Esiste una teoria sottostante, ma tipicamente questa analisi consiste in un **algoritmo ricorsivo di calcolo della proprietà** desiderata sul CFG.

Come funziona?

Supponiamo di avere un CFG per ogni procedura.

L'analisi può essere:

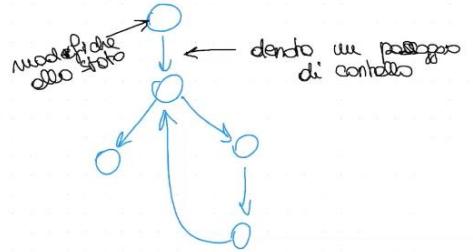
Locale	Intra-procedurale	Inter-procedurale
dentro il basick block (istruzioni senza branching interni ececc). È banale quindi non ci interessa	Analizza i singoli CFG in modo indipendente	analizza l'intero programma. Devo anche tener conto della relazione chiamata-ritorno fra procedure; avrò una foresta di CFG e dovrò tener conto degli archi che portano da un CFG all'altro.

Noi guardiamo le **intra-procedurali**.

Per l'analisi di data flow, quindi, guardiamo **come l'informazione di interesse viene trasformata dentro un blocco**.

Dato un CFG, quello che ci interessa, quindi, è capire **come** l'informazione di interesse viene trasformata nei blocchi.

1. Caratterizzare l'informazione di interesse che **entra** nel blocco. Questa sarà una **combinazione delle informazioni che arrivano dai blocchi predecessori**: dovremo unirla o intersecarla.
2. Caratterizzare l'informazione che **esce** dal blocco come **manipolazione delle informazione in entrata**.



Il calcolo avviene per punto fisso, cioè continuiamo a visitare il grafo finché l'informazione caratterizzata in entrata e in uscita dal blocco non viene più modificata, aka abbiamo trovato un'invariante.

Available expression

Supponiamo di avere questo codice:

```

 $z \leftarrow 1$ 
 $y \leftarrow M[B]$ 
A:  $x_1 \leftarrow y + z$ 
    :
B:  $x_2 \leftarrow y + z$ 
  
```

In B, stiamo rivalutando qualcosa di già calcolato in A?

Possiamo quindi, anziché ricalcolare il valore, prendere qualcosa di già calcolato?

Quindi non è così banale! Ci poniamo questo problema perché siamo nel mondo **dell'ottimizzazione**, e ci importa poter diminuire il numero di accessi. Per rispondere, dobbiamo capire se:

1. Arrivo a B sempre dopo aver visitato A?
2. Le variabili coinvolte, aka y, z, x_1 , vengono mai modificate tra A e B?

Esempio: bubble sort

Codice ad alto livello	Bytecode	Bytecode ottimizzato
<pre>for (i = n-2; i >= 0; i--) { for (j = 0; j <= i; j++) { if (A[j] > A[j+1]) { temp = A[j]; A[j] = A[j+1]; A[j+1] = temp; } } }</pre>	<pre>i := n-2 s5: if i<0 goto s1 j := 0 s4: if j>i goto s2 t1 = 4*j t2 = &A t3 = t2+t1 t4 = *t3 :A[j] t5 = j+1 t6 = 4*t5 t7 = &A t8 = t7+t6 t9 = *t8 :A[j+1] if t4 < t9 goto s3 t10 = 4*j t11 = &A t12 = t11+t10 temp = *t12 :temp=A[j] t13 = j+1 t14 = 4*t13 t15 = &A t16 = t15+t14 t17 = *t16 :A[j+1] t18 = 4*t17 t19 = &A t20 = t19+t18 :A[j] *t20 = t17 :A[j]=A[j+1] t21 = j+1 t22 = 4*t21 t23 = &A t24 = t23+t22 s3: j = j+1 s4: i = i-1 s2: goto s5 s1: s1: (*t4=*t3 means read memory at address in t3 and write to t4: *t20=t17: store value of t17 into memory at address in t20)</pre>	<pre>i = n-2 t27 = 4*i t28 = &A t29 = t27+t28 t30 = t28+4 5: if t29 < t28 goto s1 t25 = t28 t26 = t30 4: if t25 > t29 goto s2 t4 = *t25 :A[j] t9 = *t26 :A[j+1] if t4 <= t9 goto s3 temp = *t25 :temp=A[j] t17 = *t26 :A[j+1] *t25 = t17 :A[j]=A[j+1] *t26 = temp 3: t25 = t25+4 t26 = t26+4 goto s4 2: t29 = t29-4 goto s5</pre>

Osserviamo che ci sono **molte espressioni che sono ricalcate più volte**, e questo è particolarmente spesso in codice intermedio generato da calcolatore. Applicando un'ottimizzazione di questo tipo, il codice si riduce notevolmente: Quindi, generalizzando le due domande:

- È vero che una espressione è valutata sempre prima di un'altra?
- Una variabile in un punto di programma ha sempre lo stesso valore che aveva prima di un altro punto di programma?

Per poter ottenere una risposta abbiamo bisogno di:

- Una semantica operazionale
- Un metodo per dare risposta a queste due domande.

In particolare dobbiamo capire quale risposta accettiamo approssimata.

- Nel caso del nostro programma, se verifichiamo che l'espressione calcolata al posto A è **sempre prima** di quella al posto B **possiamo sostituire la roba in A al posto di quella in B**. Se dico che la computazione è ridondante, allora **devo esserne certa**. Non dobbiamo classificare come ridondante un calcolo che non lo è! Non posso ammettere **falsi positivi**.
- Viceversa, è ammesso **non catturare tutte le espressioni ridondanti**, quindi sono ammessi **falsi negativi**

Def. Espressione e disponibile in una variabile x al punto p

Data una variabile x e un punto di programma p , un'espressione si dice disponibile se:

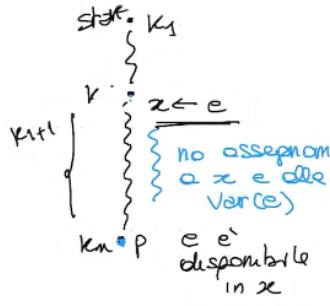
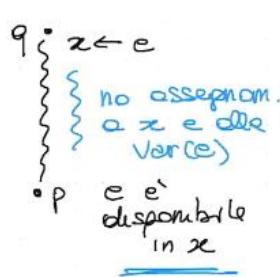
1. Condizione necessaria:
l'espressione e deve essere stata valutata in un punto precedente a p e salvata in x.

2. x e tutte le variabili in e non devono essere modificate tra la valutazione di e in q e p.

→ formalmente

Sia $\pi = k_1 \dots k_n$ un cammino dal punto start a p. allora diciamo che l'espressione e è disponibile in x al punto p se:

- a. π contiene k_i etichettato $x \leftarrow e$, ovvero se c'è un arco intermedio che ha eseguito l'assegnamento
- b. Nessuno tra gli archi $k_{i+1} \dots k_n$ è etichettato con un assegnamento ad una variabile in $\{x\} \cup Var(e)$, ovvero nessuna delle variabili deve essere modificata.



Esempio grafico

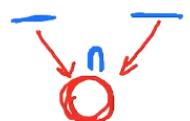
1. Caratterizziamo l'informazione in ingresso al blocco

- Al blocco iniziale n_0 non arriva **alcun'informazione**.
 - Ai blocchi successivi arriva una **combinazione dell'informazione che esce** dai blocchi precedenti.
- Ma come la combiniamo? Cup o cap? Dipende da come stiamo ragionando.

$$\text{AvailIn}(n) = \left\{ \bigcap_{m \in \text{Pred}(n)} \text{AvailOut}(m) \right\}$$

blocco start

$$\boxed{\begin{array}{l} z = 5 \\ a = w + 5 \end{array}}_n$$



Decidiamo che ogni cammino che arriva a p deve avere quella espressione come disponibile su quel contenitore. Quindi il modo di combinare è con l'intersezione; **quello che è disponibile in entrata a quel blocco è solo ciò che è disponibile in uscita a TUTTI i blocchi predecessori**.

Questo perché basta ci sia un cammino precedente che non lo ha e, nel caso in cui tanto bene arrivo da quel cammino, mi trovo a non avere quell'espressione calcolata :

Quindi ciò che è disponibile in entrata è:

- **Vuoto** se siamo nel blocco iniziale
- **Intersezione** (Ciò che è disponibile in entrata a TUTTI i blocchi predecessori) per gli altri.

2. Caratterizziamo ciò che è disponibile in uscita calcolando come il blocco manipola l'informazione disponibile in ingresso.

Qui entra in campo la **semantica**.

Innanzitutto **vediamo quali espressioni sono definite nel blocco** (e quindi essere disponibili da lì in poi).

Nell'esempio, $w+5$ è disponibile in output.

Quindi, in output mettiamo quello che viene generato nel blocco , unite a quelle che arrivano (togliendo quelle che vengono modificate)

2) Combinazione ciò che disponibile in uscita calcolando come il blocco manipola l'info disponibile in ingresso

$$\boxed{\begin{array}{l} z = 5 \\ a = w + 5 \end{array}}_n$$

$$\text{AvailOut}(m) = \text{Gen}(m) \cup (\text{AvailIn}(m) \setminus \text{Kill}(m))$$

AvailOut

Nell'esempio:

$$\text{AvailIn}(n) = \left\{ \bigcap_{m \in \text{Pred}(n)} \text{AvailOut}(m) \right\}$$

$n_0 = n$

AvailOut(m) = Gen(m) ∪ (AvailIn(m) ∖ Kill(m))

Associamo quindi ad ogni nodo n delle informazioni:

Associamo ad ogni modo $m \rightarrow \text{AvailIn}(m)$ contiene le espressioni disponibili all'ingresso di m

Initializzazione: $\text{AvailIn}(n) = \begin{cases} \emptyset & m = n_0 \\ \{x \leftarrow e \mid x := e \text{ è nel CFG}\} & \text{prendo tutti gli assegnamenti presenti nel CFG} \end{cases}$

$$\begin{aligned} \text{AvailIn}(m) &= \bigcap_{m \in \text{pred}(m)} \text{AvailOut}(m) \\ \text{AvailOut}(m) &= \text{Gen}(m) \cup (\text{AvailIn}(m) \setminus \text{Kill}(m)) \\ \text{AvailIn}(m) &= \bigcap_{m \in \text{pred}(m)} \text{Gen}(m) \cup (\text{AvailIn}(m) \setminus \text{Kill}(m)) \end{aligned}$$

Vediamo formalmente Gen e Kill: dato un blocco b , definisco

$$\text{Gen}(m) = \{x \in e \mid x := e \in b, x \notin \text{var}(e)\}$$

$$\text{Kill}(m) = \{x \in e \mid \exists y = e' \in m, y \in \text{Var}(e) \vee x = y\}$$

Ad esempio

$$(x) := (x) + y \quad \begin{matrix} x=5 & y=2 \\ \leftarrow & \end{matrix} \quad 2 \leftarrow x+y = 7$$

$$x+y = 9 \quad \leftarrow$$

Come calcoliamo AvailIn?

1. Costruiamo il CFG
2. Raccogliamo le informazioni iniziali sul programma: Gen e Kill non dipendono dal calcolo, quindi li facciamo tutti all'inizio
3. Risolviamo l'equazione per AvailIn per ogni blocco

<pre>// assume block b has k operations of form x <- y op z for each block b Init(b) Init(b) Gen DEExpr(b) <- ∅ Kill 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)</pre>	<pre>// 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</pre>
---	---

Esempio di esecuzione dell'analisi

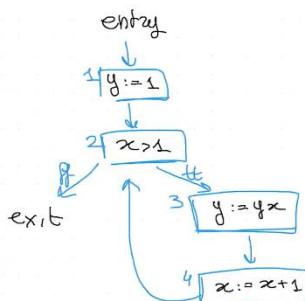
$y := 1;$
 $\text{while } (x > 4) \{$
 $y := y * x;$
 $x := x - 1;$

Calcolo tutti gli assegnamenti papabili:

~~$y \leftarrow 1$~~
 ~~$y \leftarrow y * x$~~
 ~~$x \leftarrow x - 1$~~

Qui notiamo che sia il secondo che il terzo assegnamento hanno la variabile modifica tra le variabili dell'espressione, e quindi non saranno mai disponibili dopo averle eseguite.

1. Costruiamo il control flow graph.

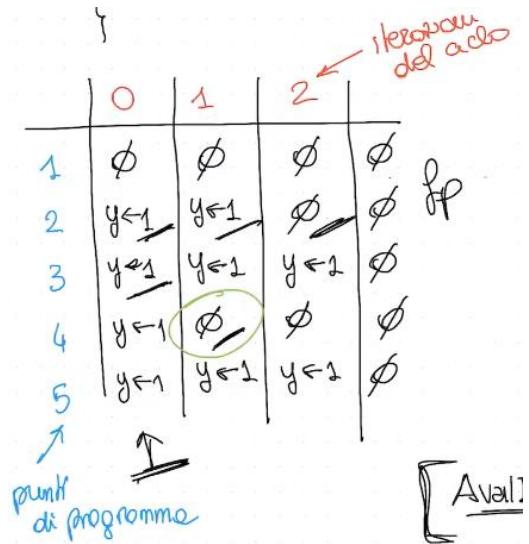


2. Assegnamo le informazioni iniziali.

	<p>Uccidiamo tutti gli assegnamenti che coinvolgono la variabile da qualche parte. Buttiamo via questa espressione = questa è l'approssimazione: non ci accorgiamo che assegnamo esattamente lo stesso valore, e che quindi in teoria sarebbe ancora disponibile.</p>
	<p>è un test quindi non generiamo né uccidiamo nulla.</p>
	<ul style="list-style-type: none"> Dato che la variabile modificata è dentro l'espressione, non la generiamo Uccidiamo tutti gli assegnamenti che coinvolgono y perché y viene modificata.
	<ul style="list-style-type: none"> Anche qui non generiamo nulla Kill(4) ucciderebbe tutti gli assegnamenti che coinvolgono x, ma nel nostro dominio non abbiamo assegnamenti che coinvolgono x e quindi possiamo scrivere vuoto.

Calcoliamo fino ad arrivare al punto fisso.

1:23:40 L5 TODO



DATA FLOW: ANALISI STATICÀ E APPROCCIO SEMANTICO

L8 - Analisi Statica Dafaflow 2 - Formal framework

Questo framework formale vuole riscrivere esattamente lo stesso approccio ma in funzione di una semantica. In questo modo, se vogliamo cambiare la cosa di osservare, basta cambiare la semantica e poi l'algoritmo si modifica "ereditando". La struttura di analisi è più modulare.

In più, questo framework formale ci permette di spostarci da quest'analisi di data flow molto vicine alla sintassi per poi spostarci/estenderci verso semantiche che guardano anche dentro la memoria, cambiando il livello di approssimazione dell'analisi.

Gli ingredienti

- **Formalizziamo l'informazione astratta che vogliamo analizzare.** Nell'esempio di prima, per l'available expression, vogliamo osservare per ogni punto di programma quali espressioni sono disponibili di una certa variabile.
- **Definiamo l'abstract edge effect**, ovvero l'effetto dell'arco dal punto di vista dell'informazione che stiamo osservando. È una funzione di trasferimento, che ci dice come l'esecuzione di una certa istruzione ha effetto sull'informazione che stiamo analizzando.
- **Costruiamo un sistema di disequazioni** (una disequazione per ogni punto di programma), cercando la migliore soluzione possibile. Vedremo che sono possibili due strade: l'iterazione di punto fisso naif senza particolari accortezze e l'iterazione round robin che cerca di accelerare la convergenza verso il punto fisso.
- **Risolvendo il sistema** otteniamo la soluzione del sistema che in generale approssima la soluzione MOP.

L'obiettivo è avere il **MOP**. Quello che invece possiamo calcolare è la soluzione **MFP**, ovvero la soluzione del sistema di disequazioni.

Quando le due coincidono abbiamo ridotto al minimo la perdita di informazione, quando non coincidono significa che nel processo di calcolo abbiamo aggiunto ulteriore perdita di informazione.

Applicazione a availability

Quindi dobbiamo:

1. Caratterizzare l'informazione astratta.

Vogliamo caratterizzare l'insieme di tutte le espressioni disponibili in un certo punto di programma dentro una certa variabile. Per rappresentare questa informazione rappresentiamo gli assegnamenti del tipo $x \leftarrow e$, dove $x \notin var(e)$, e chiamiamo questo insieme Ass 🎵.

Quindi, Dominio astratto : $\wp(Ass)$, $A \subseteq Ass$ (l'insieme delle parti dei sottoassegnamenti).

2. Caratterizzare la funzione di trasferimento.

La definiamo come semantica astratta, definita come manipolazione dell'informazione astratta, e la definiamo in modo induttivo sul linguaggio del CFG.

Definiamo l'arco come $k = (u \ lab \ v)$

$$u \xrightarrow{k} v \quad \text{lab} \xrightarrow{k} \llbracket \text{lab} \rrbracket A$$

$$\begin{aligned} \llbracket j \rrbracket^* A &= A & \llbracket \text{Nonzero}(e) \rrbracket^* A &= \llbracket \text{Zero}(e) \rrbracket^* A = A \\ \llbracket x \leftarrow e \rrbracket^* A &= \begin{cases} A \setminus \text{Occ}(x) & x \in \text{Var}(e) \\ (A \setminus \text{Occ}(x)) \cup \{x \leftarrow e\} & \end{cases} & \text{Occ}(x) = \{y \leftarrow e \mid \begin{array}{l} y=x \vee \\ x \text{ compare in } y \end{array}\} \\ \llbracket \text{Input}(x) \rrbracket^* A &= A \setminus \text{Occ}(x) & \text{qualsiasi assegnamento} \\ \llbracket \prod \rrbracket^* A &= \llbracket k_m \rrbracket^* \circ \dots \circ \llbracket k_1 \rrbracket \circ \llbracket k_0 \rrbracket A & \text{in cui compare } x \\ \prod &= k_0 k_1 \dots k_m & \text{o } x \leftarrow e \text{ o } x \leftarrow \text{symbol} \end{aligned}$$

*qualsiasi assegnamento
in cui compare x
o $x \leftarrow e$ o $x \leftarrow \text{symbol}$
del simbolo di assegnamento*

Def. Definitivamente disponibile

Un'espressione è definitivamente disponibile in v se è disponibile in ogni cammino che va dall'entrata del CFG a v. Quindi combino in intersezione.

$$\mathcal{A}^*[v] = \cap \{[\pi]^*\emptyset \mid \pi: start \rightarrow v\}$$

all'entry
sono
expr disponibili

La terminazione di questo calcolo di fixpoint è garantita da:

- **La funzione di trasformamento** $[\pi]^*$, è una funzione **monotona** che quindi preserva l'ordine,
- **Il dominio astratto** $\wp(\text{Ass})$ e Acc, ovvero **non ha catene ascendenti infinite**.

Dove perdiamo precisione?

Consideriamo i cammini non eseguibili = uso del cfg = prendiamo tutti i cammini del CFG che contengono i cammini eseguibili; chiaramente se ho branch veri/alsi, in realtà è possibile che uno dei due branch non venga mai eseguito!

Computazione della soluzione

Esempio simbolico

Gli available per ogni modo saranno:

$\underline{\mathcal{A}}[\text{start}] \subseteq \emptyset$	$\left\{ \begin{array}{l} \mathcal{A}[v] \subseteq ([\text{lab}]^* \mathcal{A}[u]) \\ \forall v = (\text{u lab } N) \end{array} \right.$
Nodo iniziale	Nodo generico

Un esempio di calcolo è il seguente; vediamo come si calcola per punto fisso.

$$\begin{aligned} D &= \wp(a, b, c) \\ \begin{cases} x_1 \supseteq a \cup x_3 \\ x_2 \supseteq x_3 \cap ab \\ x_3 \supseteq x_1 \cup bc \end{cases} \end{aligned}$$

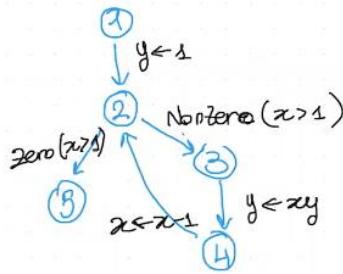
Si costruisce una tabella dove mettiamo a sinistra le incognite (quindi available in di ciascun modo) e sopra le iterazioni. Qui stiamo combinando per unione anziché per intersezione (e lo vediamo dal fatto che le equazioni hanno il "è contenuto"), quindi l'elemento neutro da cui partire è il vuoto.

	0	1	2	3	4
x_1	\emptyset	a	a, c	a, c	\wp
x_2	\emptyset	\emptyset	\emptyset	a	
x_3	\emptyset	bc	a, c	a, c	

NON HO CAPITO COME STA INCROCIANDO LE COSE. TODO 38:27 EP 8

$$\begin{aligned} &\text{Available expr} \\ &D = \wp(\text{Ass}) \\ \begin{cases} \mathcal{A}[\text{start}] \subseteq \emptyset \\ \mathcal{A}[v] \subseteq \cap \{([\text{lab}]^* \mathcal{A}[u]) \mid \\ &\quad u = \text{u lab } N \wedge \\ &\quad N \neq \text{start} \} \end{cases} \end{aligned}$$

Esempio istanziato #1



Vediamo gli available A per ogni punto di programma; se ho più archi entranti ho da fare l'intersezione. Per ogni arco devo fare le cose disponibili dall'arco applicate ad A(cerchio precedente)

$$\begin{aligned}
 A[1] &\subseteq \emptyset \\
 A[2] &\subseteq [y \leq s]^* A[1] \quad | \quad A[2] \subseteq \{y \leq s\} \\
 A[2] &\subseteq [x \leq x - 1]^* A[4] \quad \cap \quad [x \leq x - 1]^* A[4] \\
 A[3] &\subseteq [\text{Nonzero}(x > 1)]^* A[2] \\
 A[4] &\subseteq [y \leq xy]^* A[3] \\
 A[5] &\subseteq [\text{Zero}(x > 1)]^* A[2]
 \end{aligned}$$

Poi ne calcoliamo il sistema! Le nostre incognite sono A[1], A[2]...

$$A[1] = A[2] \dots = A[5] = \emptyset \rightarrow \text{soltuzione del corrispondente sistema di equazioni}$$

Usiamo l'eguaglianza anziché la diseguaglianza perché sappiamo che tipicamente tutte le soluzioni che raggiungono un nodo sono combinate per intersezione o per unione a seconda di quello che dobbiamo calcolare.

Costruiamo da qui la soluzione,

Esempio istanziato #2

Scriviamo le disequazioni istanziate dell'esempio di prima:

$$\begin{aligned}
 A[1] &\subseteq \emptyset \\
 A[2] &\subseteq [y \leq s]^* A[1] \quad | \quad A[2] \subseteq \{y \leq s\} \\
 A[2] &\subseteq [x \leq x - 1]^* A[4] \quad \cap \quad [x \leq x - 1]^* A[4] \\
 A[3] &\subseteq [\text{Nonzero}(x > 1)]^* A[2] \\
 A[4] &\subseteq [y \leq xy]^* A[3] \\
 A[5] &\subseteq [\text{Zero}(x > 1)]^* A[2]
 \end{aligned}$$

Diventa

$$\begin{cases}
 A[1] = \emptyset \\
 A[2] = \{y \leq s\} \cap (\text{A}[4] \setminus \text{occ}(x)) \\
 A[3] = \text{A}[2] \\
 A[4] = \text{A}[3] \setminus \text{occ}(xy) \\
 A[5] = \text{A}[2]
 \end{cases}$$

E a questo punto posso svolgere la tabellina.

- La prima colonna è tutto \mathbb{D} tranne il punto 1 perché so che è \emptyset .
 - Se arrivo a vuoto, non posso restringere oltre.
- La seconda colonna la ottengo applicando il sistema sulle $A[x]$ della colonna precedente!
- Continuo fino a fixpoint.

	0	1	2	3
1	\emptyset	\emptyset	\emptyset	\emptyset
2	$y \leftarrow 1$	$y \leftarrow 1$	$y \leftarrow 1$	\emptyset
3	$y \leftarrow 1$	$y \leftarrow 1$	$y \leftarrow 1$	\emptyset
4	$y \leftarrow 1$	\emptyset	\emptyset	\emptyset
5	$y \leftarrow 1$	$y \leftarrow 1$	$y \leftarrow 1$	\emptyset

↑
tutto

→ Nessuna espressione è disponibile in questi punti di programma.

Miglioria: Algoritmo di roundrobin

Se rifaccio lo stesso calcolo ma in roundrobin, allora ho che dall'iterazione 2 posso usare i valori già calcolati DELLA STESSA COLONNA, anziché andare in quella precedente.

Round Robin

	0	1	2
1	\emptyset	\emptyset	\emptyset
2	$y \leftarrow 1$	$y \leftarrow 1$	\emptyset
3	$y \leftarrow 1$	$y \leftarrow 1$	\emptyset
4	$y \leftarrow 1$	\emptyset	\emptyset
5	$y \leftarrow 1$	$y \leftarrow 1$	\emptyset

```

for (i=1; i<=m; i++) D[xi] = null
do {
    finished = true;
    for (i=1; i<=m; i++) {
        new = f_i(D[x1] ... D[xn])
        if (not(D[xi] >= new)) {
            finished = false;
            D[xi] = D[xi] ∨ new
        }
    }
} while (not finished)
  
```

Annotations:

- for ($i=1; i \leq m; i++$) $D[x_i] = \text{null}$: *using delle info calcolate* che stiamo calcolando
- do { *modo*
- finished = true;
- for ($i=1; i \leq m; i++$) { *valori della colonna precedente* eventualmente approssimati nella attuale iterazione
- $new = f_i(D[x_1] \dots D[x_n])$
- if (not($D[x_i] \geq new$)) { *vera se abbiamo modificato*
- $finished = \text{false}$;
- $D[x_i] = D[x_i] \vee new$

Soluzione MOP vs MFP

Soluzione MOP: soluzione desiderata

Come abbiamo visto, la soluzione che cerchiamo è la MOP (merge over all path). Ora la generalizziamo.

Gli ingredienti sono:

- Reticolo completo
- Valore/informazione iniziale
- Funzione che calcola la semantica su questa informazione.

Se I è l'informazione calcolata, e in particolare $I[v]$ è l'informazione al punto di programma v , allora in generale calcoliamo la soluzione mop come



In pratica, stiamo combinando gli effetti di ogni possibile cammino del CFG che mi portano a v .

Questa è la soluzione più precisa che possiamo calcolare, ma non è quella che abbiamo calcolato: la MOP non è decidibile, perché **ha la combinazione di un insieme di cammini che sono potenzialmente infiniti in quantità (se ho cicli)**.

Soluzione MFP: soluzione calcolata

$I[v]$ è il valore astratto calcolato al punto v .

$$\begin{cases} I[entry] \supseteq d_0 \\ I[v] \supseteq [k]^* I[u] \end{cases}$$

*direzione indica come
opporsi minimo*

$\forall k = \langle n, lab, u \rangle \quad [k]^* = [lab]^*$

Di questa cerchiamo la minima soluzione.

Teorema di Kam&Kildall

In generale, la soluzione MOP è più precisa della soluzione che calcoliamo mediante la risoluzione del sistema di transizione.

$$I[v] \sqsupseteq I^*[v]$$

La precisione è data in funzione dell'ordinamento, e in questo caso stiamo scrivendo tutto in modo che l'approssimazione è sovrastimare; quindi, ovviamente la soluzione è più precisa quando è più piccola. Questo significa anche che

$$I[v] \sqsupseteq [\pi]^# d_0$$

Ovvero che la soluzione calcolata in ogni punto di programma contiene ogni semantica dei cammini che portano dallo start fino a v partendo da d_0 .

Insomma, questa formula ci dice che la soluzione MOP è sempre più precisa (o potenzialmente uguale) alla soluzione del sistema, che chiamiamo MFP.

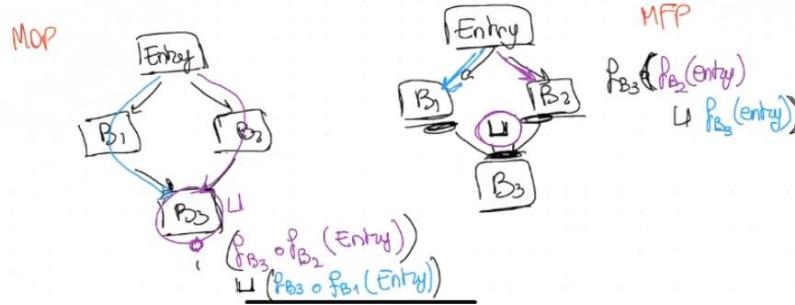
Differenza fra MOP e MFP

Per capire se e quando possono coincidere dobbiamo capire la differenza.

- **L'algoritmo non visita necessariamente i nodi nell'ordine di esecuzione;** siamo noi a decidere in che ordine mettere la valutazione nella tabella che abbiamo costruito.

- Nei punti di join, la computazione della soluzione algoritmica MFP calcola il merge ; potenzialmente, quindi, include valori non calcolati da esecuzioni.

Vediamolo graficamente.



(MOP unisce tutti i cammini separatamente, MFP unisce i nodi/archi. MOP è più preciso perché tiene conto di tutte le operazioni che avvengono NELL'ORDINE DI ESECUZIONE, mentre MFP collezionano tutto assieme nei punti di join, e applico il punto successivo sull'insieme più grande perché ho perso l'ordine di esecuzione.)

Questa differenza si esplicita nella distributività

FUNZIONE DISTRIBUTIVA VS NON DISTRIBUTIVA

Def. Funzione distributiva

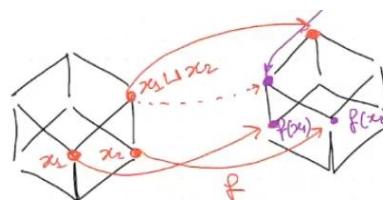
È una funzione che distribuisce sull'operatore di merge, ovvero applicare la f e poi applicare i risultati oppure applicare i risultati e poi la f è la stessa identica cosa.

$$f(x_1 \cup x_2) = f(x_1) \cup f(x_2)$$

Questo avviene se f è distributiva rispetto all'operatore di merge che andiamo a considerare nel problema, e se tutti i nodi del CFG del programma sono raggiungibili; in tal caso allora MOP e MFP coincidono, cioè

$$\begin{aligned} MOP[N] &= MFP[N] \quad \forall N \\ \mathcal{Y}[N] &= \mathcal{Y}^*[N] \end{aligned}$$

Graficamente, vogliamo che $f(x_1) \cup f(x_2)$ è nello stesso ordine di grandezza di $f(x_1 \cup x_2)$

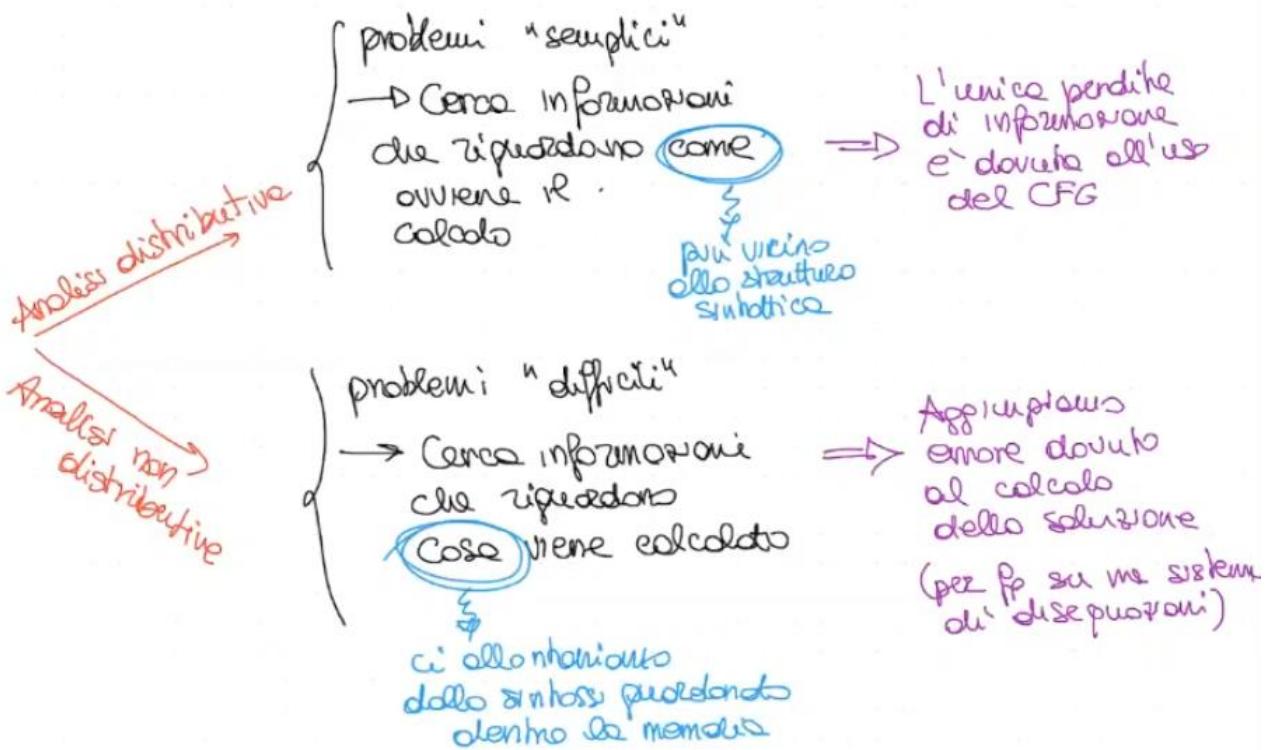


Se vogliamo avere un esempio di calcolo, supponiamo di avere un dominio con una coppia di valori ordinate punto a punto. Se prendiamo come operazione la somma, vogliamo che sommare prima o dopo aver fatto il least upper bound sia la stessa cosa.

$$\begin{aligned} (x_1 \otimes x_2) &\sqsubseteq (y_1, y_2) \\ \Leftrightarrow x_1 &\leq y_1 \quad x_2 \leq y_2 \end{aligned} \quad \begin{aligned} add((1, u) \vee (u, 1)) - \\ = add(4, u) = 8 \quad \cancel{*} \\ add(1, u) \vee add(u, 1) = 5 \end{aligned}$$

Analisi reale

A livello di corrispondenza con l'analisi reale, abbiamo che:



Tutte le data flow analysis, come le available expression, si trovano nel primo gruppo.

DATA FLOW: SINTESI SULL'APPROCCIO

Forward vs backwards

Forward	Backwards
<p>Costruiamo l'informazione in input di un nodo come dando l'informazione iniziale. Se $n \in n_0$ è una certa operazione di combinazione dell'informazione disponibile in output del predecessore.</p> <div style="text-align: center;"> $FA_{in}(n) = \begin{cases} \text{informazione iniziale} \\ \oplus \\ FA_{out}(m) \end{cases} \quad m \in pred(n)$ </div>	<p>Partiamo dall'uscita. Quindi, se partiamo dall'uscita, descriviamo ciò che c'è in uscita da un nodo come combinazione di quello che c'è in ingresso ai nodi successori.</p> <div style="text-align: center;"> $BA_{out}(n) = \begin{cases} \text{informazione in uscita} \\ \ominus \\ BA_{in}(m) \end{cases} \quad m \in succ(n)$ </div>
<p>L'out del nodo è una certa trasformazione delle informazioni in input:</p> $FA_{out}(n) = Gen(n) \cup (FA_{in}(n) \setminus kill(n))$	<p>L'in del nodo è una certa trasformazione delle operazioni di output:</p> $BA_{in}(n) = Gen(n) \cup (BA_{out}(n) \setminus kill(n))$
<p>Posso poi unire queste due cose in un'equazione da risolvere rispetto all'input:</p> $FA_{in}(n) = \bigoplus_{m \in pred(n)} Gen(m) \cup (FA_{in}(m) \setminus kill(m))$	<p>Posso poi unire queste due cose in un'equazione da risolvere rispetto all'output:</p> $BA_{out}(n) = \bigoplus_{m \in succ(n)} Gen(m) \cup (BA_{out}(m) \setminus kill(m))$

Il + cerchiato può essere unione per analisi possibile, e intersezione per definite.

$$\bigoplus \rightarrow \begin{cases} \cup & \text{possible} \\ \cap & \text{definite} \end{cases}$$

Funzione di trasferimento

In entrambi i casi è una funzione che descrive come avviene il calcolo, ma tra algoritmo e semantica viene data in modo diverso rispetto alla struttura che stiamo osservando:

Algoritmico	Semantico
<p>abbiamo una famiglia di funzioni di trasferimento tra valori astratti, dove ne abbiamo una per ogni blocco del CFG. La funzione mi spiega come viene manipolata l'informazione dentro il blocco.</p> <div style="text-align: center;"> $\text{algoritmo: } f : V \rightarrow V$ <p style="color: blue;">famiglia di funzioni</p> $f = f_B \mid B \text{ blocco del CFG}$ $f_B(x) = Gen_B \cup (x \setminus kill_B)$ <p style="color: green;">informazione in ingresso del blocco B</p> </div>	<p>Ho una funzione monotona definita induttivamente sulla struttura sintattica del linguaggio del CFG.</p> <div style="text-align: center;"> $\text{U merge: } \forall x, y \quad x \leq y \Rightarrow f(x) \subseteq f(y)$ $\equiv f(x) \cup f(y) \subseteq f(x \cup y)$ $\text{distributivo: } f(x) \cup f(y) = f(x \cup y)$ $\Rightarrow Mop = Mfp$ </div>

Computazione della soluzione

Algoritmico	Semantico
<p>INPUT:</p> <ul style="list-style-type: none"> • CFG (con simbolo entry e simbolo exit) • V valori essenziali • Un operatore di merge \sqcup • Un insieme di F di funzioni f_B (B blocco del CFG) • Valore iniziale N_{entry} ($\circ N_{exit}$) $\in V$ <p>OUTPUT: Insieme di valori in $V \vdash IN[B]$ e $OUT[B]$</p> <p style="text-align: center;"> </p> <p><u>Forward:</u></p> <pre> IN[Entry] = Nentry; for (each block B != Entry) IN[B] = false while (some IN changes) for (each block B != Entry) { IN[B] = ⊔ OUT[P]; P ∈ pred(B) OUT[B] = f_B(IN[B]); } </pre> <p style="text-align: center;"> $\text{merge} = \sqcup \Rightarrow \text{elemento neutro}$ $\text{è} \perp$ $\perp \sqcup \text{false} = \perp$ </p> <p><u>Backward:</u></p> <pre> OUT[Exit] = Nexit for (each block B != exit) OUT[B] = false while (some OUT changes) for (each block B != exit) { OUT[B] = ⊔ IN[S]; S ∈ succ(B) IN[B] = f_B(OUT[B]); } </pre> <p style="text-align: center;"> $\text{esclusivo del merge}$ neutro </p> <p style="text-align: center;"> \Rightarrow se corrisponde allora abbiamo le risultati dell'analisi </p> <p>Costruiamo le equazioni su $I[v]$ e risolviamo usando la semantica astratta come funzione di trasferimento (effetto dell'esecuzione degli archi)</p>	

Soluzione dell'analisi

In realtà né MOP né MFP sono soluzioni ideali. La soluzione ideale (che vediamo in versione forward) andrebbe calcolata su tutti i cammini possibili (**eseguibili**) dalla entry al nodo di interesse. Cioè, vorrei considerare SOLO i cammini che vengono eseguiti durante l'esecuzione del programma!!

Dati dei blocchi B , un cammino $p = Entry \rightarrow B_1 \rightarrow \dots \rightarrow B_k$ e una funzione di trasferimento totale sul cammino $f_p = f_{B_k} \circ f_{B_{k-1}} \circ \dots \circ f_{B_1}(entry)$, allora

$$IDEAL[B] = \bigcup_{\substack{p \text{ cammino} \\ \text{possibile} \\ \text{dalla entry} \\ \text{a } B}} f_p(N_{entry})$$

eseguibile

Ogni **soluzione più grande** di questa, aka che contiene IDEAL, è **corretta** perché contiene tutte le informazioni che sono effettivamente calcolate nell'insieme.

Viceversa, ogni **soluzione più piccola non è corretta** perché perde alcuni dei cammini eseguibili.

Rispetto a ideal, quindi:

MOP	MFP
<p>MOP approssima considerando OGNI cammino del CFG; consideriamo più cammini come possibili. La soluzione MOP è la merge su tutti i cammini del CFG</p> $MOP[B] = \bigsqcup_{P \text{ cammino del CFG}} f_P(N_{\text{entry}})$ <p>Quindi, se ho che l'insieme dei cammini possibili ne sto considerando di più; ideale è più preciso di MOP.</p> <p>$MOP[B] \supseteq IDEAL[B]$</p>	<p>MFP è la soluzione del calcolo solo considerando gli input (o gli output) di ogni blocco. È meno precisa della MOP e vi coincide se la funzione è distributiva.</p> $N[B] \supseteq MOP[B] \Rightarrow MFP \supseteq MOP$ <p>($MFP = MOP$ se funz. distributiva)</p>

Quindi per transitività

$$MFP \supseteq MOP \supseteq IDEAL$$

DATA FLOW: ANALISI DI LIVENESS

L10 - Analisi Statica DataFlow 4 - Liveness (approccio algoritmico)

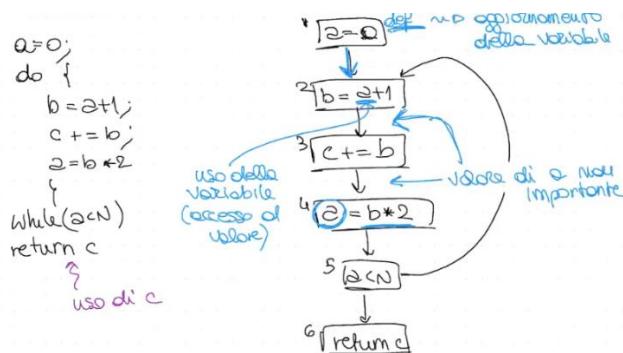
Approccio algoritmico

Cerca di individuare le variabili "vive" in una porzione di codice analizzata. L'idea di fondo si basa sul fatto che due variabili che sono vive contemporaneamente devono essere in punti di memoria diverse.

Quest'informazione è importante in molti contesti, in particolare per l'**ottimizzazione** (se trovo che due variabili non sono mai vive nello stesso momento allora possono occupare la stessa area di memoria), così come nel software **watermarking**.

Esempio

Una variabile è **viva** se il valore in quel momento è importante perché nel futuro verrà utilizzato. (es. se viene sovrascritto...). La variabile è viva in tutti i cammini che arrivano a un punto dove essa è viva, perché lì userò il valore e poiché userò il valore allora tutti i cammini che arrivano a quel punto usano il valore (che quindi non può essere modificato).



Al contrario dell'available expression, la liveness guarda da un punto di programma al passato: ciò che determina il fatto che una variabile è viva è il fatto di essere utilizzata. Rimane viva fino alla definizione successiva, perché lì la andrà a ridefinire e sovrascrivere.

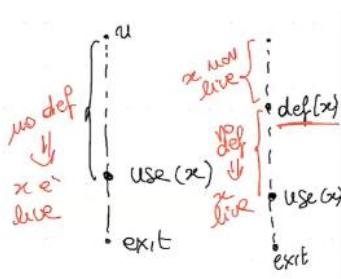
Andiamo a definire cosa sono le variabili usate e definite. È un concetto puramente sintattico.

- **Uso:** accesso al valore
- **Definizione:** aggiornamento del valore

Quindi possiamo definirlo in modo sintattico sulle etichette del linguaggio:

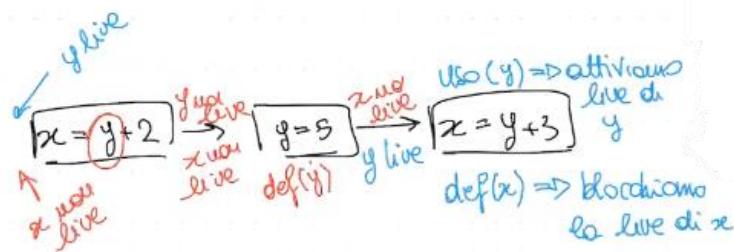
lab	use[lab]	def[lab]
j	∅	∅
Zero()	Var(x)	∅
Nonzero()	Var(x)	∅
$x \leftarrow e$	Var(x)	{x}
Input(x)	∅	{x}

x è dead se viene riassegnata prima del suo utilizzo.



In generale questi aspetti non sono decidibili solo sulla sintassi.

È chiaro che la costruzione dipende da quello che succede prima di un certo punto di programma; andremo all'indietro cercando dove quella variabile ha un valore che verrà utilizzato.



Esempio 20:42

Variabili usate nel nodo 4
 $2 \rightarrow 3$
 $3 \rightarrow 4, b$ live
 b def al nodo 2
 $\Rightarrow 1 \rightarrow 2 \rightarrow b$ non live
 $5 \rightarrow 3$
 $3 \rightarrow 6$

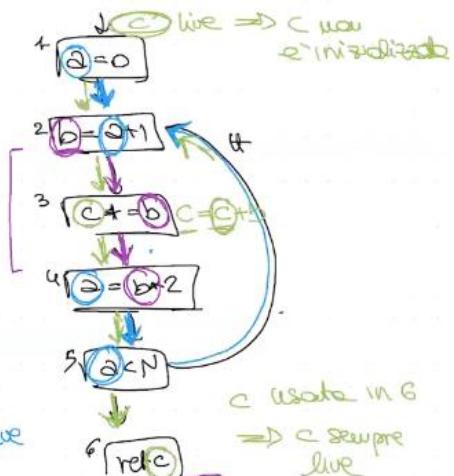
Variabili non usate al 6
 $\Rightarrow 5 \rightarrow 6$ non live

$4 \rightarrow 5$ z live
 z definita in 4 $\Rightarrow 3 \rightarrow 4$ z non live
 $2 \rightarrow 3$

z usata in 2 \Rightarrow z'stive liveness

$1 \rightarrow 2$
 $5 \rightarrow 2$ z live

z no live



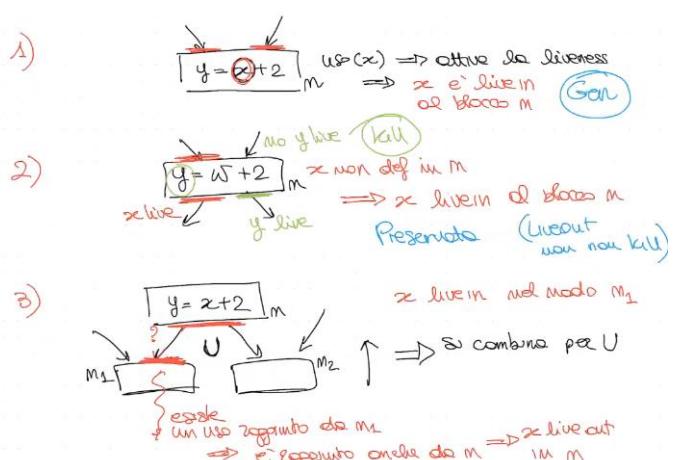
L12 - Analisi statica Dataflow 6 - True liveness

Falsi positivi dell'analisi di liveness non dovuti all'approssimazione

Si osserva che rimangono dei falsi positivi, non legati a cammini non eseguibili o semantica non osservabile, ma legati ad aspetti sintattici e che quindi possiamo catturare.

Partiamo da un esempio →

(basta un solo cammino dove è live perché essa sia live.
 Stiamo considerando anche cammini non eseguibili, quindi ho una perdita di precisione).



Metodo

Definisco le equazioni

$\text{LiveIn}[m]$ sono variabili live in ingresso al modo
 $\text{LiveOut}[m]$ sono variabili live in uscita dal modo

$$\left\{ \begin{array}{l} \text{LiveOut}[m] = \begin{cases} \emptyset & m = \text{exit} \\ \bigcup_{p \in \text{succ}(m)} \text{LiveIn}[p] & \end{cases} \\ \text{LiveIn}[m] = \underline{\text{Gen}[m]} \cup (\text{LiveOut}[m] \setminus \underline{\text{Kill}[m]}) \end{array} \right. \quad \begin{array}{l} \text{caso 3} \\ \text{caso 1 e 2} \end{array}$$

$$\begin{array}{l} \text{Gen}[m] = \{x \mid \exists e \in m . x \in \text{ver}(e)\} \\ \text{Kill}[m] = \{x \mid \exists z \in e \in m\} \end{array} \quad \begin{array}{l} \text{Raccolta} \\ \text{informazioni} \\ \text{iniziali sui CFG} \end{array}$$

L'equazione di punto fisso che risolviamo col nostro algoritmo, quindi, è

$$\boxed{\text{LiveOut}[m] = \bigcup_{p \in \text{succ}(m)} \text{Gen}[p] \cup (\text{LiveIn}[p] \setminus \text{Kill}[p])} \quad \begin{array}{l} \text{Raccolta} \\ \text{informazioni} \\ \text{iniziali sui CFG} \end{array}$$

L'algoritmo costruisce il CFG, raccoglie le info iniziali, e poi risolve l'equazione.

L'elemento neutro è il vuoto, perché stiamo unendo per sovrastima/unione.

Precisione

I problemi sono:

- Consideriamo cammini non eseguibili, quindi questo aggiunge potenzialmente falsi positivi; potremmo determinare come live variabili che non lo sono.
- Il contenuto di x potrebbe essere accessibile attraverso altri nomi; questo fa sì che potremmo perdere una definizione! Non ce ne accorgiamo perché ragioniamo solo sulla sintassi.



se live
 → no use
 → no eseguibili
 $use(x)$

→ def(y)
 → $use(y)$
 x non dovrebbe essere live
 perché il suo valore viene combiato
 $use(y)$

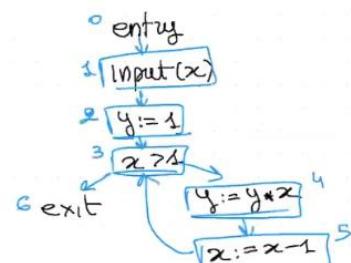
Esempio di analisi di liveness completo algoritmico

1. Definizione del CFG

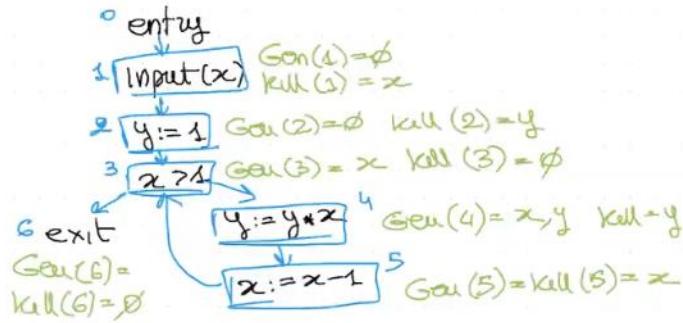
```

Input(x)
y := 1;
while (x > 1) {
    y := x * y;
    x := x - 1;
}
  
```

→



2. Raccolta delle informazioni iniziali, cioè Gen e Kill



3. Costruiamo le tabelle di LiveIn e LiveOut. Le fa separate per non fare confusione.

Facciamo l'analisi backward quindi li scrive all'indietro. Regola di LiveIn: $\text{LiveIn}[b] = \text{Gen} \cup (\text{LiveOut} \setminus \text{Kill})$

LiveOut	0	1
6	\emptyset	
5	\emptyset	
4	\emptyset	
3	\emptyset	
2	\emptyset	
1	\emptyset	

LiveIn	0	1
6	\emptyset	
5	x	
4	xy	
3	x	
2	\emptyset	
1	\emptyset	

Il passo 1 del liveout fa semplicemente l'unione del LiveIn dei nodi successori. Il passo 1 del livein applica la formula.

LiveOut	0	1
6	\emptyset	\emptyset
5	\emptyset	x
4	\emptyset	x
3	\emptyset	xy
2	\emptyset	x
1	\emptyset	\emptyset

LiveIn	0	1
6	\emptyset	\emptyset
5	x	$x \leftarrow x \cup x \setminus x$
4	xy	$xy \leftarrow xy \cup x \setminus y$
3	x	$xy \leftarrow xy \cup x \setminus y$
2	\emptyset	$x \leftarrow x \cup xy \setminus \emptyset$
1	\emptyset	$x \setminus y$

Vado avanti fino a fixpoint.

LiveOut	0	1	2
6	\emptyset	\emptyset	\emptyset
5	\emptyset	x	xy
4	\emptyset	x	x
3	\emptyset	xy	xy
2	\emptyset	x	xy
1	\emptyset	\emptyset	x

LiveIn	0	1	2
6	\emptyset	\emptyset	\emptyset
5	x	$x \leftarrow x \cup xy \setminus x$	xy
4	xy	xy	xy
3	x	$xy \leftarrow x \cup xy \setminus x$	xy
2	\emptyset	x	$\emptyset \cup xy \setminus y$
1	\emptyset	\emptyset	\emptyset

LiveOut	0	1	2	3
6	\emptyset	\emptyset	\emptyset	\emptyset
5	\emptyset	x	xy	xy
4	\emptyset	x	x	xy
3	\emptyset	xy	xy	xy
2	\emptyset	x	xy	xy
1	\emptyset	\emptyset	x	x

LiveIn	0	1	2
6	\emptyset	\emptyset	\emptyset
5	x	x	xy
4	xy	xy	xy
3	x	xy	xy
2	\emptyset	x	x
1	\emptyset	\emptyset	\emptyset

Approccio semantico

L11 - Analisi statica DataFlow 5 - Liveness

Al contrario della versione algoritmica (CFG, risoluzione dell'equazione di punto fisso) qui facciamo un'individuazione del dominio delle osservazioni astratte (=cosa osserviamo del programma) e poi andiamo a definire la semantica astratta induttivamente sulla sintassi del linguaggio del CFG, e su questa semantica definiamo come l'informazione viene manipolata.

- Dominio: $\wp(Var)$, ovvero la **collezione di tutti gli insiemi di variabili** (dominio finito)
- L'analisi è **backward**:
 - individua gli usi e l'uso attiva l'aliveness su tutti i cammini che arrivano a quell'uso. Anche in questo caso, significa che **diamo l'informazione iniziale sul nodo di uscita**.
 - Inoltre ci dice in che direzione **applichiamo la semantica**, aka **al contrario** (di solito era applicata in reverse, ora è in giusto): $\pi = k_1 \dots k_n \Rightarrow [\pi]^\# = [k_1] \circ \dots \circ [k_n]$
 - La soluzione MOP sarà $\mathcal{L}^*[v] = \bigcup\{\pi^\# \times | \pi : v \rightarrow exit\}$. Usiamo l'unione perché è possibile.
 - Influisce anche sul sistema di computazione ma ci arriviamo dopo.

Semantica

Diamo la semantica induttivamente sul linguaggio del CFG.

$$L \subseteq Ver \quad [\cdot]^\# L = L \quad [\text{NonZero}(e)]^\# L = [\text{Zero}(e)]^\# L = L \cup \text{Ver}(e)$$

assunzione:
variabili live
al momento
dell'esecuzione

$$[\text{input}(x)]^\# L = (L \setminus \{x\}) \cup \text{Ver}(e)$$
$$[\text{zero}(x)]^\# L = L \setminus \{x\}$$

Computazione MFP (miglior punto disso del sistema di disequazione)

Fissate x informazione iniziale al nodo exit

Quando abbiamo un arco vogliamo visitarlo in backward, quindi vogliamo visitarlo in direzione opposta; scriviamo l'informazione che vale in u in funzione di quello che valeva in v .

$$\left\{ \begin{array}{l} L[\text{exit}] \ni x \\ L[u] \supseteq [\text{lab}]^\# L[v] \end{array} \right. \quad u = (u \text{ lab } v)$$

possible

$[\cdot]^\#$ monotonous

Dominio finito } \Rightarrow soluzione decidibile
esiste MPP

Esempio

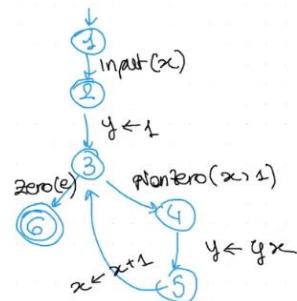
1. Costruiamo il CFG spostando il contenuto del blocco n sul blocco che va da n al successore.

```

input(x)
y := 1
while(x > 1) {
    y := y * x
    x := x - 1
}

```

→



2. Scriviamo il sistema di disequazioni TODO 22:12

$$L(6) \supseteq \emptyset$$

$$L(5) \supseteq [x \leq x+1]^* L(3) = L(3) \setminus \{x\} \cup \{x\}$$

$$= L(3) \cup \{x\}$$

$$L(4) \supseteq [y \leftarrow xy]^* L(5) = L(5) \setminus \{y\} \cup \{xy\}$$

$$= L(5) \cup \{xy\}$$

$$L(3) \supseteq L(6) \cup L(4) \cup \{x\}$$

$$L(2) \supseteq [y \leftarrow 1]^* L(3) = L(3) \setminus \{y\}$$

$$L(1) \supseteq [\text{input}(x)]^* L(2) = L(2) \setminus \{x\}$$

3. Calcolo la soluzione

	0	1	2	
6	∅	∅	∅	fp
5	∅	x =	xy	
4	∅	xy	xy	≡ LiveIn
3	∅	xy	xy	
2	∅	x	x	
1	∅	∅	∅	

NOTA BENE :)

Se cambiamo l'ordine l'unica cosa che viene calcolata in un passo in più è 5, perché dipende da 3. Modifichiamo così l'ordine per calcolare 3 prima di 5.

Quindi, cambiando l'ordine inficiamo solo la velocità di terminazione, null'altro. →

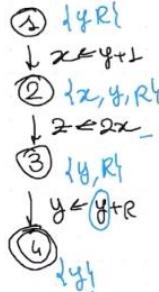
	0	1
6	∅	∅
4	∅	xy
3	∅	xy
5	∅	xy
2	∅	x
1	∅	∅

DATA FLOW: TRUE LIVENESS.

FALSI POSITIVI DELL'ANALISI DI LIVENESS NON DOVUTI ALL'APPROXIMAZIONE

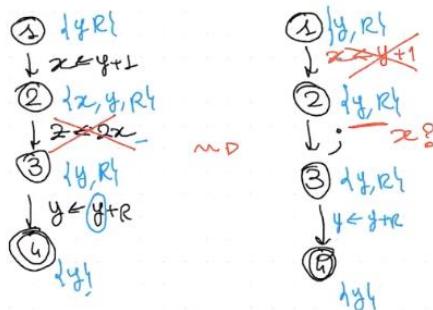
12- Analisis Statica DataFlow 6 - True liveness

Supponiamo di avere un programmino molto semplice con la seguente liveness



Notiamo che z non è mai live durante il programma, quindi l'istruzione che assegna z può essere eliminata.

Ora proviamo a ricalcolare la liveness su questa nuova versione:



Mi rendo così conto che anche x non è live, e può essere eliminata.

Potrei arrivare a sapere questa cosa applicando iterativamente l'analisi di liveness, ma sappiamo che l'applicazione iterativa non è efficiente. Dunque, ci chiediamo se possiamo sfruttare in modo ricorsivo quanto l'analisi sta calcolando per poter decidere anche chi rimane nella liveness.

In questo esempio, quando mettiamo x nelle variabili live lo facciamo in funzione di un utilizzo che avviene dentro un assegnamento a una variabile che già sappiamo non essere live – e quindi che non sarà usata in futuro. Questo assegnamento, quindi, è evidentemente inutile già dal fatto che x non è dentro le variabili live! Quindi, l'idea è di raffinare l'analisi andando a decidere se inserire una variabile usata tra le live in funzione di quanto abbiamo calcolato fra le variabili live fino a quel punto di programma. Definiamo in questo senso la **true liveness**, che dipende da un concetto di true use. Questa definizione di true use è ricorsiva, poiché è definita in funzione di sé stessa.

Def. Truly liveness

Una variabile x è truly live su $\pi : v \rightarrow^* \text{exit}$ se π :

- π non contiene definizioni di x
- π contiene un **true use** della variabile.

Cosa significa truly used in un arco u ? Lo guardo con una tabellina sulla semantica :)

	<u>y truly used in u</u>
j	false
$\text{zero}(e)$	$y \in \text{Var}(e)$
$\text{Nonzero}(e)$	$y \notin \text{Var}(e)$
$x \leftarrow e$	$y \in \text{Var}(e) \wedge x \text{ truly used in } u$
$\text{input}(x)$	false

Usando questa definizione posso osservare dall'esempio che:

Parto bottom up.

- 3-4: R è nell'espressione e Y è fra le usate nel target, quindi tutte le variabili dell'espressione sono aggiunte
- 2-3: Z è nelle usate? No, quindi x (che è nell'espressione) non è aggiunta
- 1-2: X è nelle usate? No, quindi Y (che è nell'espressione) non va aggiunta, ma è già presente quindi vbb.

In questo modo ci accorgiamo subito che z e x non sono usate. Quindi, posso riscrivere i due approcci di analisi di liveness (true liveness) usando il concetto di uso al posto dell'uso visto fino ad ora.

Quindi, in parallelo, vediamo l'approccio algoritmico e l'approccio sintattico:

Approccio algoritmico

Nella formula di base non cambia nulla, se non che sostituiamo LiveIn e LiveOut con $T\text{liveIn}$ e $T\text{liveOut}$.

Cambia invece Gen, e diventa TGen: deve esistere nel blocco un'espressione (che non sia un assegnamento) che appartiene alle variabili di e (ovvero, le variabili), oppure, siamo in una variabile dentro l'assegnamento e quindi verifico l'uso

$$\begin{aligned} T\text{live}_{\text{out}}(m) &= \left\{ \begin{array}{ll} X & m = \text{exit} \\ \bigcup_{p \in \text{succ}(m)} T\text{live}_{\text{in}}(p) & \end{array} \right. \\ T\text{live}_{\text{in}}(m) &= T\text{Gen}(m) \cup (T\text{live}_{\text{out}}(m) \setminus T\text{kill}(m)) \\ T\text{kill}(m) &= \{x \mid x \leftarrow e \text{ in } m\} \\ T\text{Gen}(m) &= \{x \mid \begin{array}{l} \text{Be in (no assgn)} \\ \text{or } x \text{ is var} \\ \text{or } (\exists y \in e. y \text{ is var}) \\ \wedge y \text{ is true live in } m \end{array}\} \end{aligned}$$

Approccio semantico

Dobbiamo ridefinire la semantica.

$$\begin{aligned} [;]^\# L &= L \\ [[\text{zero}(e)]]^\# L &= [[\text{Nonzero}(e)]]^\# L = \\ &\quad L \cup \text{var}(e) \\ [[\text{input}(x)]]^\# L &= L \setminus \{x\} \\ [[x \leftarrow e]]^\# L &= \begin{cases} (L \setminus \{x\}) \cup \text{var}(e) & x \notin L \\ L \setminus \{x\} & \text{altr.} \end{cases} \end{aligned}$$

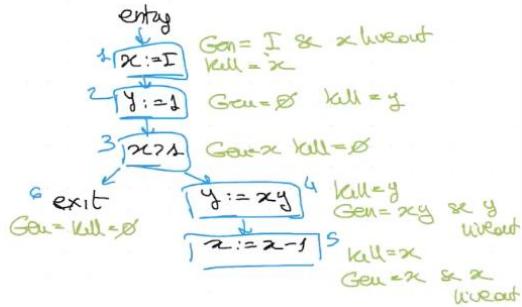
Si può dimostrare che è distributiva, ovvero non vambiano le proprietà che abbiamo già dimostrato.

Esempio

```

x := I
y := 1;
while (x > 1) {
    y := y * x;
    x := x - 1;
}

```



15:05 L12

Time out	0	1	2	3	
6	Ø	Ø	Ø	Ø	p p
5	Ø	x	x	x	p p
4	Ø	Ø	x	x	
3	Ø	Ø	Ø	x	
2	Ø	x	x	x	
1	Ø	Ø	x	x	I
0					

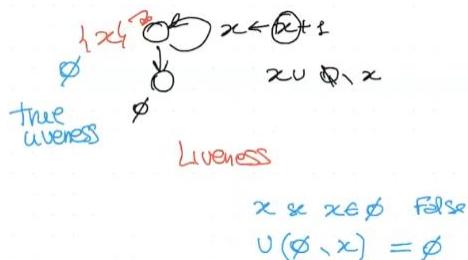
Time in	0	1	2	
6	Ø	Ø	Ø	p p
5	Ø	x	x	
4	Ø	Ø	x	
3	x	x	x	
2	Ø	x	x	
1	Ø	I	I	
0				

Semantica;

$$\begin{aligned}L(G) &= \emptyset \\L(S) &\supseteq \{x \in x - s y \mid L(S) = L(S) \setminus \{x\} \cup \{x \mid x \in L(S)\}\} \\L(U) &\supseteq \{y \in y \neq x \mid L(U) = L(U) \setminus \{xy\} \cup \{y \mid y \in L(U)\}\} \\L(B) &\supseteq L(C) \cup L(U) \cup \{x\} \\L(C) &\supseteq L(B) \cup \{y\} \\L(A) &\supseteq L(C) \cup A \cup \{x \mid x \in L(C)\}\end{aligned}$$

	O	1	2	
6	Ø	Ø	Ø	fp
5	Ø	Ø	x	
4	Ø	Ø	x	
3	Ø	x	x	
2	Ø	x	x	
1	Ø	Ø	I	

La true liveness cattura anche esempi che l'applicazione ripetuta della liveness normale non sarebbe in grado di catturare.



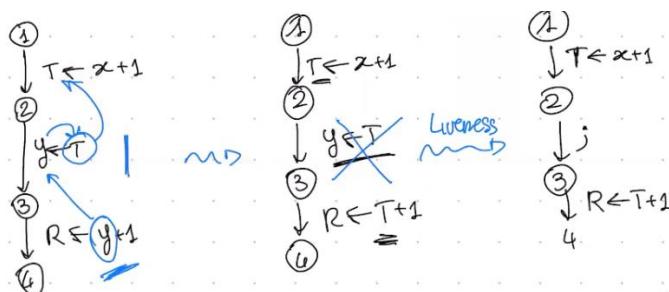
DATA FLOW: COPY PROPAGATION

L13 - Analisi statica dataflow 7 - Copy Propagation

La propagazione delle copie riguarda l'idea di **analizzare quali variabili durante l'esecuzione sono la copia dell'altra, ovvero contengono lo stesso valore.**

In questo modo, posso ottimizzare andando a riutilizzare una delle due come variabile. Per capire l'utilità di questo tipo di analisi dobbiamo pensare al contesto: spesso le analisi di dataflow nascono per l'ottimizzazione del codice intermedio, generato automaticamente in un compilatore e che quindi spesso crea delle copie di variabili senza ulteriori computazioni. In questo contesto, può succedere di avere un assegnamenti del tipo $y \leftarrow T$, dove la variabile y quindi ha una copia esatta di T .

Se vediamo questo all'interno di un'esecuzione, è evidente che effettivamente la variabile y è identica alla variabile T ; quindi eseguire l'assegnamento e accedere a y può essere sostituito dall'idea di andare a sostituire la variabile T all'interno dell'uso di y . In questo modo, applicando la liveness possiamo realizzare che y non è utile e otteniamo un codice ottimizzato; quindi la C.P. può preparare il terreno per un'altra ottimizzazione.



Come costruiamo l'informazione che ci serve per propagare queste copie?

Caso semplice: propagazione della copia di una variabile specifica x

Fissata la variabile, vogliamo che l'analisi **mantenga ad ogni punto di programma l'insieme di tutte le variabili che sicuramente contengono una copia del valore di x .**

Ottenuto questo, allora ogni occorrenza di una copia di x può essere sostituita con x . Dal punto di vista dell'analisi, è poco probabile sapere a priori quale sia la variabile per cui è interessante sapere le copie: quindi, in generale, **si estende questa idea e non più propagare le copie di una specifica variabile, ma propagare tutte le variabili.**

Informazione osservata: $\text{Var} \times \text{Var}$
 $(x, y) \Rightarrow x \text{ è copia di } y$

Nella lettura c'è un ordine: devo capire **quale variabile è la copia di quale variabile**, quindi vale **l'ordine di assegnamento**; se (y, t) sono una copia, allora y può essere sostituito da t .

Quindi ne creiamo un insieme per ogni punto di programma:

$\text{Copy}_{\text{out}}, \text{Copy}_{\text{in}} \subseteq \text{Var} \times \text{Var}$
 copie disponibili all'uscita del blocco copie disponibili in ingresso ad ogni blocco

Guardiamo il programma in direzione di esecuzione, e **collezioniamo variabili che SICURAMENTE contengono una copia**, quindi vogliamo che QUALUNQUE cammino che porta al punto di sostituzione ha effettivamente la copia, quindi l'analisi è **definita**.

Analisi:
 forward → guarda all'utilizzo di un assegnamento nel futuro
 definite → tutti i comandi che portano al punto doveva avere l'info di essere copia

L'unico elemento che manca è la definizione di come fistruggiamo e generiamo l'informazione. Definiamo quindi l'insieme gen e kill:

$$\text{Gen}(b) = \{(xy) \mid \exists z : z \in b\}$$

$$\text{Kill}(b) = \{(xy) \mid \exists z : z \in b \wedge \exists y' : y' \neq y \wedge z \in y'\}$$

Dobbiamo evitare di togliere $x=y$ (aka, non è in kill) perché una variabile è sempre copia di se stessa.

Informazione iniziale sulla entry è esattamente il fatto che ogni variabile è una copia di sé stessa:

$$X = \{(xx) \mid x \in \text{Var}\}$$

Approccio algoritmico

$$\text{Copy}_{in}(m) = \begin{cases} X & m = \text{entry} \\ \cap_{p \in \text{pred}(m)} \text{Copy}_{out}(p) & \text{otherwise} \end{cases}$$

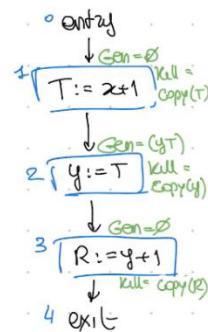
forward
definite

$$\text{Copy}_{out}(m) = \text{Gen}(m) \cup (\text{Copy}_{in}(m) \setminus \text{Kill}(m))$$

Esempio

Esempio

$$\begin{aligned} T := x+1; \quad (2z) &\stackrel{\text{def}}{=} \{(2z) \mid z \in \text{Var}\} \\ y := T; \quad (zw) &\stackrel{\text{def}}{=} \{(z,w) \mid z, w \in \text{Var}\} \\ R := y+1 \end{aligned}$$



(praticamente, usiamo z e w per indicare "qualunque altra variabile"!)

Tradotto, quindi, (z,w) corrisponderà a tutte le combinazioni (commutativamente):

$$(zw) = (xy)(xT)(xR) \\ (yt)(yR) \\ (RT)$$

Approccio algoritmico

Giro 1

Copy_{in}	0
0	(zz)
1	(zw)
2	(zw)
3	(zw)
4	(zw)

Copy_{out}	0
0	(zz)
1	$(xy)(xR)$
2	$(yt)(xT)$
3	$(xy)(xT)$

Entrando nel nodo 0, per esempio, sappiamo che non c'è nulla; questo però significa che è vera quella informazione che non è mai falsa, aka (z,z) (ogni variabile con sé stessa). Per tutti gli altri

Per gli output invece guardiamo il Gen unito quello che vale in input meno qualcosa, ma so che da (z,z) non tolgo mai nulla.

blocchi, invece, siamo in analisi definita quindi prendiamo tutte le possibili coppie, aka (z, w)

Giro 2

CopyIn	0	1	2	3	4	CopyOut	0	1	2	3
0	(zz)	(zz)				0	(zz)	(zz)		
1	(zw)	(zz)				1	(xy)(xr)	(zz)		
2	(zw)	(xy)(xr)	cyR			2	(yt)(xt)	(yt)(xr)	(xy)(xr)	
3	(zw)	(yt)(xt)	(xr)(rt)			3	(xy)(xt)	(yt)(xt)	(yt)(xr)	
4	(zw)	(xy)(xt)	(yt)							

semplicemente l'input di ogni blocco è l'output del blocco precedente

Per CopyOut, invece, si rifanno le stesse cose di prima...

Giro 3

CopyIn	0	1	2	CopyOut	0	1	2
0	(zz)	(zz)	(zz)	0	(zz)	(zz)	(zz)
1	(zw)	(zz)	(zz)	1	(xy)(xr)	(zz)	(zz)
2	(zw)	(xy)(xr)	(zz)	2	(yt)(xt)	(yt)(xr)	(yt)
3	(zw)	(yt)(xt)	(yt)(xr)	3	(xy)(xt)	(yt)(xt)	(yt)
4	(zw)	(xy)(xt)	(yt)				

in input si copia tutto...

Per copy out riapplichiamo la stessa cosa...

Quarto giro...

CopyIn	0	1	2	3	CopyOut	0	1	2	3
0	(zz)	(zz)	(zz)	(zz)	0	(zz)	(zz)	(zz)	(zz)
1	(zw)	(zz)	(zz)	(zz)	1	(xy)(xr)	(zz)	(zz)	(zz)
2	(zw)	(xy)(xr)	(zz)	(zz)	2	(yt)(xt)	(yt)(xr)	(yt)	(yt)
3	(zw)	(yt)(xt)	(yt)(xr)	(yt)	3	(xy)(xt)	(yt)(xt)	(yt)	(yt)
4	(zw)	(xy)(xt)	(yt)						

Abbiamo ottenuto, quindi, che y e T sono una la copia dell'altra, e in particolare T può essere copiata in y .

Approccio semantico

Informazione : $\text{Var} \times \text{Var}$

Informazione minima : $X = \{(xx) \mid x \in \text{Var}\}$

$C \subseteq \text{Var} \times \text{Var}$

$$[\cdot]^* C = C = [\text{NonZero}(e)]^* C = [\text{Zero}(e)]^* C$$

$$[x \in e]^* C = \begin{cases} C \setminus \text{copy}(x) \cup \{(xy)\} & e=y \\ C \setminus \text{copy}(x) & \text{otherwise} \end{cases}$$

$$[\text{input}(x)]^* C = C \setminus \text{copy}(x)$$

$$\text{copy}(x) = \{(xy) \mid y \in \text{Var}, x \neq y\} \cup \{(yx) \mid y \in \text{Var}, x \neq y\}$$

$$e^*[\nu] = \bigcup_{\pi} [\pi]^* X \mid \pi: \text{entry} \rightarrow \nu \quad [\pi]^* = [K_m] \circ \dots \circ [K_1]^* X$$

$$\text{map } e^*[\nu] = \bigcup_{\pi} [\pi]^* X \mid \pi: \text{entry} \rightarrow \nu \quad [\pi]^* = [K_m] \circ \dots \circ [K_1]^* X$$

Questa semantica è monotona e distributiva; $C^*[v]$ può essere calcolata come soluzione del sistema di disequazioni.

$$\left\{ \begin{array}{l} C[\text{entry}] \subseteq X \\ C[N] \subseteq [(lab)]^* C[u] \end{array} \right. \xrightarrow{\text{lab}} \underline{n} \quad \underline{\underline{n}}$$

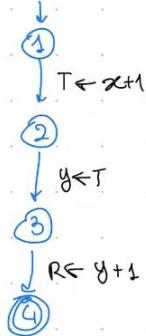
Grazie al fatto che questa semantica è distributiva, questa soluzione calcolata come MFP è esattamente uguale alla MOP.

Esempio

Per comodità riporto il CFG.

Costruiamo il sistema di informazioni.

$$\begin{aligned} C(1) &\in (zz) \\ C(2) &\subseteq [T \leftarrow z+1]^* C(1) \\ &= C(1) \setminus \text{copy}(T) \\ C(3) &\subseteq [y \leftarrow T]^* C(2) \\ &= C(2) \setminus \text{copy}(y) \cup (yT) \\ C(4) &\subseteq [R \leftarrow y+1]^* C(3) \\ &= C(3) \setminus \text{copy}(R) \end{aligned}$$



		Primo giro:	
1	O	1	<ul style="list-style-type: none"> 1 è contenuto in z,z quindi non può che essere z,z è $c(1)$ da cui tolgo qualcosa, ma $c(1)$ è già al minimo quindi è per forza z,z è $C(2)$ da cui tolgo qualcosa, ma di nuovo essendo al minimo dell'informazione è yT $C(4)$ è $C(3)$, quindi yT, da cui tolgo le copie di R cioè però non ci sono e quindi non tolgo nulla..
1	(zz)	(zz)	Punto fisso in un solo passaggio >:)
2	(zW)	(zW)	
3	(zW)	(yT)	
4	(zW)	(yT)	

L'analisi perde precisione perché

- 1 Sto considerando tutti i cammini possibili nel CFG, quindi anche i cammini potenzialmente non eseguibili, e quindi aumentiamo gli oggetti con cui facciamo l'intersezione; perdiamo delle copie, perché magari lo sono su un cammino che in realtà non viene mai eseguito
- 2 Guardiamo l'analisi semantica sulla sintassi, quindi possiamo perdere copie perché eseguiamo un assegnamento che in realtà non cambia il valore della variabile target, e he però noi invece consideriamo. Alcuni esempi sono:

$$\begin{aligned} x &:= x \\ x &:= 2x - x \end{aligned}$$

DATA FLOW: REACHING DEFINITIONS

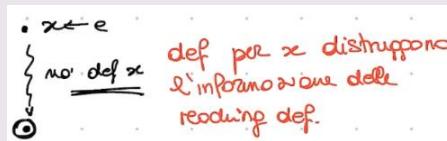
L14 - Analisi statica Dataflow 8 - Reaching definitions

Questa è l'ultima analisi statica che andiamo a guardare via approccio algoritmico/semantico.

Questa analisi ha come obiettivo quello di **individuare, dato un punto di programma, quali sono gli assegnamenti** – che qui chiamiamo definizioni – **che raggiungono quel punto di programma**; aka che non vengono sovrascritti prima di arrivare a quel punto.

Def. Reaching definitions

Individua tutte le definizioni (assegnamenti) che raggiungono un punto di programma senza riscrittura intermedia.



Una definizione viene uccisa quando c'è una nuova definizione in un punto intermedio.

Un esempio di utilizzo è il **loop code-motion**: vogliamo individuare assegnamenti invarianti interni a un ciclo, di modo che se lo sono possiamo spostarli fuori dal ciclo. Per esempio:



L'analisi è forward e possibile, ovvero che unisce per unione.

Informazione osservata: Var x PP
 (x, p) significa che x è definito al punto p
 e la coppia identifica questo assegnamento.

Informazione iniziale: \emptyset

(=non ci sono assegnamenti iniziali che raggiungono)

Come al solito abbiamo i due approcci.

Approccio algoritmico

Dobbiamo descrivere Gen e Kill.

$$\text{Gen}(m) = \{(x, m) \mid \exists x := e \in m \vee \text{input}(x) \in m\}$$

$$\text{Kill}(m) = \{(x, p) \mid \exists x := e \in m \vee \text{input}(x) \in m, p \neq p\}$$

Da qui possiamo definire RD_{in} e RD_{out} :

$$RD_{in}(m) = \begin{cases} \emptyset & m = entry \\ \bigcup_{p \in pred(m)} RD_{out}(p) & \end{cases}$$

$$RD_{out}(m) = Gen(m) \cup (RD_{in}(m) \cup Kill(m))$$

Approccio semantico

Abstract edge effect $R \subseteq Var \times PP$

$$[[\cdot]]^* R = R = [[NonZero(e)]]^* R = [[Zero]]^* R$$

$$\text{def}(x) = \{(xp) \mid p \in PP\}$$

$$[[x \leftarrow e]]^* R = R \setminus \text{def}(x) \cup \{ \langle x, v \rangle \mid \langle u, x \leftarrow e, v \rangle \text{ eseguito} \}$$

$$[[\text{input}(x)]]^* R = R \setminus \text{def}(x) \cup \{ \langle x, v \rangle \mid \langle u, \text{input}(x), v \rangle \text{ eseguito} \}$$

Questa semantica è distributiva, quindi MOP = MFP del sistema di disequazioni.

MOP: $R^*[v] = \bigcup_{\pi} [\pi]^* \emptyset \mid \pi: entry \rightarrow v \}$ $[\pi]^* \emptyset = [[K_1]]^* \circ [[K_2]]^* \emptyset$

Il sistema delle disequazioni sarà:

$$\begin{cases} R[entry] \supseteq \emptyset \\ R[v] \supseteq [[lab]]^* R[u] \end{cases} \quad v = (u, lab, n)$$

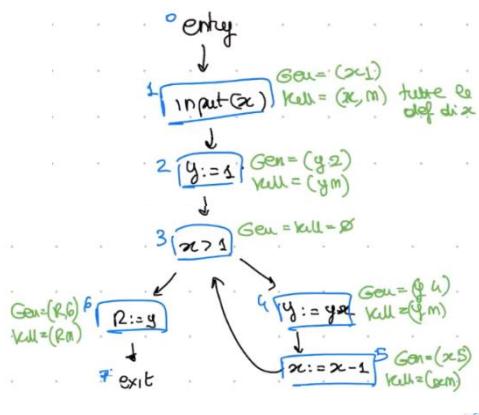
Esempi

```

input(x);
y := 1;
while (x > 1) {
    y := y * x;
    x := x - 1;
}
R := y;

```

Algoritmico



Primo giro:

RD_{in}	0		RD_{out}	0
0	\emptyset		0	\emptyset
1	\emptyset		1	(x_1)
2	\emptyset		2	(y_2)
3	\emptyset		3	\emptyset
4	\emptyset		4	(y_4)
5	\emptyset		5	(x_5)
6	\emptyset		6	(R_6)
7	\emptyset			

Tutto vuoto perché siamo in possible

RD_{out}	0
0	\emptyset
1	(x_1)
2	(y_2)
3	\emptyset
4	(y_4)
5	(x_5)
6	(R_6)

È solo il Gen

Secondo giro:

RD_{in}	0	1	RD_{out}	0	1
0	\emptyset	\emptyset	0	\emptyset	\emptyset
1	\emptyset	\emptyset	1	(x_1)	(x_1)
2	\emptyset	(x_1)	2	(y_2)	$(y_2)(x_1)$
3	\emptyset	$(y_2)(x_5)$	3	$(y_2)(x_5)(x_1)$	$(y_2)(x_5)(x_1)(y_4)$
4	\emptyset	\emptyset	4	$(y_2)(x_5)$	$(y_2)(x_5)$
5	\emptyset	(y_4)	5	(y_4)	(y_4)
6	\emptyset	\emptyset	6	(x_5)	$(x_5)(y_4)$
7	\emptyset	(R_6)	7	(R_6)	(R_6)

È l'unione dei predecessori in RDout

RD_{out}	0	1
0	\emptyset	\emptyset
1	(x_1)	(x_1)
2	(y_2)	$(y_2)(x_1)$
3	\emptyset	$(y_2)(x_5)$
4	(y_4)	(y_4)
5	(x_5)	$(x_5)(y_4)$
6	(R_6)	(R_6)

È l'Rdout del turno precedente unito al Rdin - kill

Terzo giro:

RD_{in}	0	1	2		RD_{out}	0	1	2
0	\emptyset	\emptyset	\emptyset		0	\emptyset	\emptyset	\emptyset
1	\emptyset	\emptyset	\emptyset		1	(x_1)	(x_1)	(x_1)
2	\emptyset	(x_1)	(x_1)		2	(y_2)	$(y_2)(x_1)$	$(y_2)(x_1)$
3	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)(x_1)$	$(y_2)(x_5)(x_1)(y_4)$	3	$(y_2)(x_5)$	$(y_2)(x_5)(x_1)$	$(y_2)(x_5)(x_1)(y_4)$
4	\emptyset	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)$	4	(y_4)	(y_4)	(y_4)
5	\emptyset	(y_4)	(y_4)	(y_4)	5	(x_5)	$(x_5)(y_4)$	$(x_5)(y_4)$
6	\emptyset	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)$	6	(R_6)	(R_6)	(R_6)
7	\emptyset	(R_6)	(R_6)	(R_6)				

Quarto giro:

RD_{in}	0	1	2	3	RD_{out}	0	1	2	3
0	\emptyset	\emptyset	\emptyset	\emptyset	0	\emptyset	\emptyset	\emptyset	\emptyset
1	\emptyset	\emptyset	\emptyset	\emptyset	1	(x_1)	(x_1)	(x_1)	(x_1)
2	\emptyset	(x_1)	(x_1)	\emptyset	2	(y_2)	$(y_2)(x_1)$	$(y_2)(x_1)$	$(y_2)(x_1)$
3	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)(x_1)$	$(y_2)(x_5)(x_1)(y_4)$	3	$(y_2)(x_5)$	$(y_2)(x_5)(x_1)$	$(y_2)(x_5)(x_1)(y_4)$	$(y_2)(x_5)(x_1)(y_4)$
4	\emptyset	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)$	4	(y_4)	(y_4)	(y_4)	(y_4)
5	\emptyset	(y_4)	(y_4)	(y_4)	5	(x_5)	$(x_5)(y_4)$	$(x_5)(y_4)$	$(x_5)(y_4)$
6	\emptyset	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)$	6	(R_6)	(R_6)	(R_6)	(R_6)
7	\emptyset	(R_6)	(R_6)	(R_6)					

Quinto giro:

RD_{in}	0	1	2	3	4	
0	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	
1	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	
2	\emptyset	(x_1)	(x_1)	\emptyset	\emptyset	
3	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)(x_1)$	$(y_2)(x_5)(x_1)(y_4)$	\emptyset	
4	\emptyset	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)$	\emptyset	
5	\emptyset	(y_4)	(y_4)	(y_4)	\emptyset	
6	\emptyset	\emptyset	$(y_2)(x_5)$	$(y_2)(x_5)$	\emptyset	
7	\emptyset	(R_6)	(R_6)	(R_6)	\emptyset	

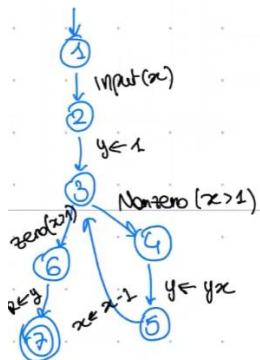
fp

Quest'analisi ci dice che all'entrata del while arrivano tutte le definizioni, così come al primo assegnamento;

Il primo assegnamento poi nasconde la definizione di y, infatti lì non arriva la definizione di y2; mentre al nodo 6 che è l'uscita possono arrivare tutte le definizioni (dato che posso non eseguire il while) e poi all'ultimo nodo arrivano tutte. In output invece ci accorgiamo che viene bloccata la definizione di x.

Approccio semantico

Innanzitutto costruiamo la forma giusta del CFG:



Costruiamo il sistema di disequazioni:

$$\begin{aligned}
 R(1) &\supseteq \emptyset \\
 R(2) &\supseteq [\text{input}(x)]^* \emptyset = (x_1) \\
 R(3) &\supseteq [(y \leftarrow 1)]^* R(2) = R(2) \setminus \text{def}(y) \cup (y_2) \\
 &\quad \cup (R(5) \setminus \text{def}(x) \cup (x_5)) \\
 R(4) &\supseteq R(3) \\
 R(5) &\supseteq [(y \leftarrow yx)]^* R(4) = R(4) \setminus \text{def}(y) \cup (y_4) \\
 R(6) &\supseteq R(3) \\
 R(7) &\supseteq [(x \leftarrow x - 1)]^* R(6) = R(6) \setminus \text{def}(x) \cup (R_6)
 \end{aligned}$$

Calcolo la soluzione:

	0	1	2	
1	\emptyset	\emptyset	\emptyset	\emptyset
2	\emptyset	(x_1)	(x_2)	
3	\emptyset	$(x_1)(y_2)(x_5)$	$(x_1)(y_2)(y_4)(x_5)$	
4	\emptyset	$(x_1)(y_2)(x_5)$	$(x_1)(y_2)(y_4)(x_5)$	
5	\emptyset	$(x_1)(x_5)(y_4)$	$(x_1)(x_5)(y_4)$	
6	\emptyset	$(x_1)(y_2)(x_5)$	$(x_1)(y_2)(y_4)(x_5)$	
7	\emptyset	$(x_1)(y_2)(x_5)$	$(x_1)(y_2)(y_4)(x_5)(R_6)$	

DATA FLOW: SINTESI FINALE

L15 - Analisi Statica DataFlow 9 - Sintesi Finale

Vediamo una tabellina di incrocio usando tutte le possibilità.

	Possible	Definite
Forward	Reading definition	<ul style="list-style-type: none"> Available expr Copy propagation
Backward	<ul style="list-style-type: none"> Liveness True liveness 	Very busy expr

Very Busy Expression: Definite + backwards (accenno)

Parte dall'idea che un'espressione è busy se verrà prima o poi calcolata nel futuro. Partendo quindi dal punto in cui ci interessa analizzare, vogliamo essere sicuro che nel futuro prima o poi questa espressione viene raggiunta

Very busyness

Un'espressione è every busy se tutti i cammini che portano da quel cammino all'uscita vedono quell'espressione come busy.

Stiamo parlando di TUTTI i cammini, quindi definite, e guardiamo al PASSATO, quindi è backwards.

È un'analisi che nasce nell'ambito dell'ottimizzazione, e riguarda la possibilità di ottimizzare per code motion: se due cammini hanno quella espressione calcolata a un certo punto, allora quel calcolo può essere spostato a monte di entrambi i cammini.

Non ci dà il dettaglio ma tanto non chiede gli esercizi so

Approccio algoritmico

$$X = \emptyset$$

$$\text{Gen}(m) = \{x \in e \mid x \in e_m \wedge x \notin \text{Var}(e)\}$$

$$\text{Kill}(m) = \{x \in e \mid \begin{cases} (y \in e) \wedge (y \in \text{Var}(e) \vee y \in m) \\ \exists e' \in m. x \in \text{Var}(e') \end{cases}\}$$

$$\text{VB}_{\text{out}}(m) = \begin{cases} \emptyset & m = \text{exit} \\ \bigcup_{p \in \text{succ}(m)} \text{VB}_{\text{in}}(p) & \text{otherwise} \end{cases}$$

$$\text{VB}_{\text{in}}(m) = \text{Gen}(m) \cup (\text{VB}_{\text{out}}(m) \setminus \text{Kill}(m))$$

Approccio semantico

$$\overline{[\Gamma]}^* X$$

$$B^*[\alpha] = \bigcap_{\pi} [\pi]^* \otimes [\pi : \alpha \rightarrow \text{exit}]$$

$$\begin{cases} B[\text{exit}] \subseteq \emptyset \\ B[\alpha] \subseteq [\text{lab}]^* B[\alpha] \end{cases}$$

$$K = (\alpha, \text{lab } \beta)$$

Reaching definition: possible + forward

Algorithmic

Gen e Kill

$$RD_{in}(m) = \begin{cases} X & m = \text{entry} \\ \bigcup_{p \in \text{pred}(m)} RD_{out}(p) \end{cases}$$

$$RD_{out}(m) = \text{Gen}(m) \cup (RD_{in}(m) \setminus \text{Kill}(m))$$

Semantic

$$\overline{[\cdot]}^*$$

$$R^*[\alpha] = \bigcup \{ \overline{[\pi]}^* X \mid \pi : \text{entry} \rightarrow \alpha \}$$

$$\begin{cases} R[\text{entry}] \supseteq X \\ R[\alpha] \supseteq [\text{lob}]^* R[\alpha] \end{cases}$$

$$(\alpha, \text{lob}, \nu)$$

Available expression: definite + forward

Algorithmic

Gen + Kill

$$Avail_{in}(f(m)) = \begin{cases} X & m = \text{entry} \\ \bigcap_{p \in \text{pred}(m)} Avail_{out}(p) \end{cases}$$

$$Avail_{out}(m) = \text{Gen}(m) \cup (Avail_{in}(m) \setminus \text{Kill}(m))$$

Semantic

$$\overline{[\cdot]}^*$$

$$A[\alpha] = \bigcap \{ \overline{[\pi]}^* X \mid \pi : \text{entry} \rightarrow \alpha \}$$

$$A[\text{entry}] \subseteq X$$

$$A[\nu] \subseteq [\text{lob}]^* A[\alpha]$$

$$\kappa = (\alpha, \text{lob}, \nu)$$

Copy propagation è analogo, cambia solo la definizione di Gen e Kill

Liveness: possible + backwards

Algorithmic

Kill + Gen

$$Live_{out}(m) = \begin{cases} X & m = \text{exit} \\ \bigcup_{p \in \text{succ}(m)} Live_{in}(p) \end{cases}$$

$$Live_{in}(m) = \text{Gen}(m) \cup (Live_{out}(m) \setminus \text{Kill}(m))$$

Semantic

$$\overline{[\cdot]}^*$$

$$L^*[\alpha] = \bigcup \{ \overline{[\pi]}^* X \mid \pi : \alpha \rightarrow \text{exit} \}$$

$$L[\text{exit}] \supseteq X$$

$$L[\alpha] \supseteq [\text{lob}]^* L[\alpha]$$

$$\kappa = (\alpha, \text{lob}, \nu)$$