Assignment 2: Dataflow Analysis

Gruppo 8
Componenti:

- Giovanni Orlandi
- Francesco Mancuso
- Luca Ballotti

Introduzione

Questo documento presenta il lavoro svolto dal gruppo 8 per l'Assignment 2 del corso di Compilatori, incentrato sull'implementazione di analisi di flusso di dati (**Dataflow Analysis**) all'interno di un compilatore.

L'obiettivo principale dell'assegnamento è progettare e implementare un'analisi utilizzando tecniche standard di forward e backward analysis. L'analisi è stata sviluppata seguendo i principi teorici introdotti a lezione, adattandoli al contesto pratico fornito.

---

1. **Very Busy Expression Analysis**

Un'espressione `e` è *very busy* in un punto `p` se su **ogni** cammino da `p` all'`exit block`:

- `e` viene valutata **prima** che uno dei suoi operandi venga ridefinito.

Serve ad ottimizzare via **code hoisting**, ovvero anticipare il calcolo di espressioni senza invalidarle.

	Very busy expression
Domain	$\text{Sets of expressions}$
Direction	Backword
Transfer function	$f_b(x) = Gen_b \cup \left(x - Kill_b \right)$
in[b]	$f_b(out[b]) $
out[b]	^ $in[succ(b)]$
Meet Operation (∧)	$\cap$
Boundary Condition	$in[\text{Exit}] = \emptyset$
Initial interior points	$in[b] = \text{Universal set}$

dove:

- Gen_b: espressioni generate da istruzioni nel basic block. Es: `a = x + y` genera `x + y`
- $Kill_b$: ogni espressione di assegnamento killa le espressioni in cui la variabile riassegnata è un operando

L'implementazione è stata pensata con un bit vector per ogni basic block. La lunghezza di ogni bit vector è pari al numero di espressioni presenti nel CFG.
In questo caso specifico il bit vector sarà lungo 2:

- **indice 0** → `b-a` (e1)
- **indice 1** → `a-b` (e2)

Esempio:

- `[1, 0]` vuol dire che `b-a` è busy, `a-b` no.

Esercizio

Tabella di Generazione/Kill

	Gen	Kill
BB1	∅	∅
BB2	∅	∅
BB3	e1	∅
BB4	e2	∅
BB5	e1	∅
BB6	∅	e1,e2
BB7	e2	∅
BB8	∅	∅

Tabella dei Bit Vector

	In[b]	Out[b]
BB1 (Entry)	/	1 0
BB2	1 0	1 0
BB3	1 1	0 1
BB4	0 1	0 0
BB5	1 0	0 0
BB6	0 0	0 1
BB7	0 1	0 0
BB8 (Exit)	0 0	/

---

2. **Dominator analysis**

In un **CFG** diciamo che un nodo *X* **domina** un altro nodo *Y* se il nodo *X* appare in ogni percorso del grafo che porta dal blocco *Entry* al blocco *Y*.

La funzione di trasferimento risulta molto semplice: ogni blocco deve aggiungere se stesso all'insieme dei dominator ricevuto in ingresso e propagare questa lista aggiornata in uscita.
Poiché un blocco *A* è dominatore di un blocco *B* se e solo se tutti i percorsi che portano a *B* passano per *A*, la **meet operation** da usare è l'intersezione degli output di tutti i predecessori del blocco *B*.

L'implementazione, pur essendo realizzabile nuovamente tramite bit-vector, è stata pensata come una lista contenente i **dominatori**, rappresentati dai nomi dei BB, per risultare più agevole da gestire su carta e più intuitiva alla lettura umana.

	Dominator Analysis
Domain	Sets of BB
Direction	Forward
Transfer function	$f_b(x) = b \cup x$
in[b]	^ $out[pred(b)]$
out[b]	$f_b(in[b]) $
Meet Operation (∧)	$\cap$
Boundary Condition	$out[\text{Entry}] = \text{Entry}$
Initial interior points	$out[b] = \text{Universal set}$

Esercizio

Lista dei dominatori

BB	**In[b]**	**Out[b]**
A (Entry)	∅	{A}
B	{A}	{A, B}
C	{A}	{A, C}
D	{A, C}	{A, C, D}
E	{A, C}	{A, C, E}
F	{A, C}	{A, C, F}
G (Exit)	{A}	{A, G}

---

3. **Constant propagation**

L'obiettivo della **constant propagation** è determinare in quali punti del programma le variabili assumono un valore costante.
In ogni BB, consideriamo non più una singola informazione, ma una coppia **<variabile, valore costante>**.

Infatti, se in un blocco *b* abbiamo la coppia *<x, c>*, significa che ogni volta che si raggiunge *b*, la variabile *x* ha valore costante pari a *c*.

Con la nostra implementazione, siamo in grado di determinare costanti anche quando il valore deriva da espressioni binarie su variabili che risultano costanti in ingresso al blocco.

Le funzioni *Gen* e *Kill* risultano qui più articolate:

- **Gen**: un'espressione di assegnamento, con left-value *k*, genera la coppia *<k, value>* nei seguenti casi:
    1. Se il right-value è una costante (es. `k = 2`).
    2. Se il right-value è un'espressione tra variabili che sono costanti in ingresso (es. `k = a * 3`, e tra le informazioni in ingresso a *b* c'è *<a, 4>*).
- **Kill**: ogni assegnamento invalida tutte le precedenti coppie associate al left-value, indipendentemente dal valore.

Dal punto di vista tecnico, si utilizza una lista per ospitare le coppie. Ancora una volta, si potrebbe usare un vettore non binario ma con l'eventuale valore costante salvato dentro, ma risulta poco pratico a mano.

Poiché una variabile è considerata costante in un blocco solo se è costante in **tutti** i suoi predecessori, la **meet operation** adottata è l'**intersezione**.
Questo spiega anche la scelta dell'**universal set** come initial interior points. Inizialmente assumiamo che ogni variabile possa assumere qualsiasi valore costante.

	Constant propagation
Domain	(var, valore)
Direction	Forward
Transfer function	$f_b(x) = Gen_b \cup \left(x - Kill_b \right)$
in[b]	^ $out[pred(b)]$
out[b]	$f_b(in[b]) $
Meet Operation (∧)	$\cap$
Boundary Condition	$out[\text{Entry}] = \emptyset$
Initial interior points	$out[b] = \text{Universal set}$

Tabella di Gen/Kill

> **Nota:** con `c` si intende un **valore costante generico**.

BB	**Gen**	**Kill**
BB1	∅	∅
BB2	<k,2>	<k,c>
BB3	∅	∅

BB	Gen	Kill
BB4	<a,k+2>	<a,c>
BB5	<x,5>	<x,c>
BB6	<a,k*2>	<a,c>
BB7	<x,8>	<x,c>
BB8	<k,a>	<k,c>
BB9	∅	∅
BB10	<b,2>	<b,c>
BB11	<x,a+k>	<x,c>
BB12	<y,a*b>	<y,c>
BB13	<k,k+1>	<k,c>
BB14	∅	∅
BB15	∅	∅

Iterazioni In/Out

BB	In (1st Iter.)	Out (1st Iter.)	In (2nd Iter.)	Out (2nd Iter.)
BB1	/	∅	/	∅
BB2	∅	{<k,2>}	∅	{<k,2>}
BB3	{<k,2>}	{<k,2>}	{<k,2>}	{<k,2>}
BB4	{<k,2>}	{<k,2>, <a,4>}	{<k,2>}	{<k,2>, <a,4>}
BB5	{<k,2>, <a,4>}	{<k,2>, <a,4>, <x,5>}	{<k,2>, <a,4>}	{<k,2>, <a,4>, <x,5>}
BB6	{<k,2>}	{<k,2>, <a,4>}	{<k,2>}	{<k,2>, <a,4>}
BB7	{<k,2>, <a,4>}	{<k,2>, <a,4>, <x,8>}	{<k,2>, <a,4>}	{<k,2>, <a,4>, <x,8>}
BB8	{<k,2>, <a,4>}	{<k,4>, <a,4>}	{<k,2>, <a,4>}	{<k,4>, <a,4>}
BB9	{<k,4>, <a,4>}	{<k,4>, <a,4>}	{<a,4>}	{<a,4>}
BB10	{<k,4>, <a,4>}	{<k,4>, <a,4>, <b,2>}	{<a,4>}	{<a,4>, <b,2>}
BB11	{<k,4>, <a,4>, <b,2>}	{<k,4>, <a,4>, <b,2>, <x,8>}	{<a,4>, <b,2>}	{<a,4>, <b,2>}
BB12	{<k,4>, <a,4>, <b,2>, <x,8>}	{<k,4>, <a,4>, <b,2>, <x,8>, <y,8>}	{<a,4>, <b,2>}	{<a,4>, <b,2>, <y,8>}

BB	In (1st Iter.)	Out (1st Iter.)	In (2nd Iter.)	Out (2nd Iter.)
BB13	{<k,4>, <a,4>, <b,2>, <x,8>, <y,8>}	{<k,5>, <a,4>, <b,2>, <x,8>, <y,8>}	{<a,4>, <b,2>, <y,8>}	{<a,4>, <b,2>, <y,8>}
BB14	{<k,4>, <a,4>}	{<k,4>, <a,4>}	{<a,4>}	{<a,4>}
BB15	{<k,4>, <a,4>}	{<k,4>, <a,4>}	{<a,4>}	{<a,4>}