Gruppo: 18 Nome: Enrico Luppi Matricola: 152749

Dataflow Analysis Assignment

1) Very Busy Expressions

Nell'analisi delle Very Busy Expressions ci interessa sapere, in un punto p, quali espressioni saranno valutate in futuro, permettendo l'eliminazione di valutazioni ridondanti. Vi è quindi necessità di propagare informazioni dalla fine all'inizio del CFG (backward).
Un blocco genera un'espressione (Gen_b) quando la valuta, mentre la uccide ($Kill_b$) quando ridefinisce uno degli operandi.
Per valutare i soli nodi del CFG dato, andrà posto vuoto l'insieme in ingresso al blocco Exit. L'espressione dovrà essere valutata in tutti i percorsi presi da p, quindi il meet operator sarà ∩ e di conseguenza le condizioni iniziali saranno date dall'insieme universo $\mathcal{U}$.

	Very Busy Expressions
Domain	Sets of Expressions
Direction	Backward: $in[b] = f_b(out[b])$ $out[b] = \wedge\, in[succ(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet operation (Λ)	∩
Boundary Condition	In[exit] = Ø
Initial interior points	In[b] = $\mathcal{U}$

	Iterazione 1		Iterazione 2*
	IN [B]	OUT[B]	
BB1 (entry)		b-a	
BB2	b-a	b-a	
BB3	a-b, b-a	a-b	
BB4	a-b	Ø	
BB5	b-a	Ø	
BB6	Ø	a-b	
BB7	a-b	Ø	
BB8 (exit)	Ø		

Con Bitvector < (a-b) (b-a) >

	Iterazione 1		Iterazione 2*
	IN [B]	OUT[B]	
BB1 (entry)		<01>	
BB2	<01>	<01>	
BB3	<11>	<10>	
BB4	<10>	<00>	
BB5	<01>	<00>	
BB6	<00>	<10>	
BB7	<10>	<00>	
BB8 (exit)	<00>		

*La seconda iterazione è uguale (non essendoci backedges) e si avrà convergenza.

2) Dominator Analysis

Nella Dominator Analysis ci interessa costruire l'insieme DOM[b] per ogni nodo. L'idea è quella di aggiungere il nodo in esame all'insieme DOM in ingresso; per cui si ha una propagazione in avanti (forward). Per valutare i soli nodi del CFG dato, poniamo vuoto l'insieme DOM in uscita dal blocco Entry.
Dato che più nodi predecessori, che confluiscono nello stesso nodo, non lo dominano, il meet operator sarà ∩ e di conseguenza le condizioni iniziali saranno date dall'insieme universo $\mathcal{U}$.

	Dominator Analysis
Domain	Sets of Basic Blocks
Direction	Forward: out[b] = f_b(in[b]) in[b] = ∧ out [pred(b)]
Transfer function	f_b(x) = x U {b}
Meet operation (∧)	∩
Boundary Condition	out[entry] = Ø
Initial interior points	out[b] = $\mathcal{U}$

	Iterazione 1		Iterazione 2*
	IN [B]	OUT[B]	
BB1 (entry)		Ø	
BB2	Ø	A	
BB3	A	A, B	
BB4	A	A, C	
BB5	A, C	A, C, D	
BB6	A, C	A, C, E	
BB7	A, C	A, C, F	
BB8	A	A, G	
BB9 (exit)	A, G		

Con Bitvector < A B C D E F G >

	Iterazione 1		Iterazione2*
	IN [B]	OUT[B]	
BB1 (entry)		<0000000>	
BB2	<0000000>	<1000000>	
BB3	<1000000>	<1100000>	
BB4	<1000000>	<1010000>	
BB5	<1010000>	<1011000>	
BB6	<1010000>	<1010100>	
BB7	<1010000>	<1010010>	
BB8	<1000000>	<1000001>	
BB9 (exit)	<1000001>		

*La seconda iterazione è uguale (non essendoci backedges) e si avrà convergenza.

3) **Constant Propagation**

Nell'analisi Constant Propagation ci interessa sapere in quali punti del programma le variabili possono essere sostituite da costanti. L'idea è, quindi, quella di procedere dall'inizio del CFG (forward) calcolando coppie <variabile, valore>.
Una coppia è generata da un blocco (Gen_b) quando definita attraverso costanti e/o altre variabili già associate ad una costante in una coppia in ingresso; mentre viene uccisa dal blocco ($Kill_b$) quando la variabile viene ridefinita.
Per valutare i soli nodi del CFG dato, poniamo vuoto l'insieme in uscita da Entry. Essendo possibile avere, su percorsi diversi, coppie diverse con la stessa variabile, il meet operator sarà ∩ e di conseguenza le condizioni iniziali saranno date dall'insieme universo $\mathcal{U}$.

	Constant Propagation
Domain	Sets of Pairs < variable, value >
Direction	Forward: out[b] = f_b(in[b]) in[b] = Λ out [pred(b)]
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$
Meet operation (Λ)	∩
Boundary Condition	out[entry] = Ø
Initial interior points	out[b] = $\mathcal{U}$

	Iterazione 1		Iterazione 2		Interazione 3*
	IN [B]	OUT[B]	IN [B]	OUT[B]	
BB1 (entry)		Ø		Ø	
BB2	Ø	(k,2)	Ø	(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)	(a,4) (k,4)	(k,2) (a,4)	(a,4) (k,4)	
BB9	(a,4) (k,4)	(a,4) (k,4)	(a,4)	(a,4)	
BB10	(a,4) (k,4)	(a,4) (k,4) (b,2)	(a,4)	(a,4) (b,2)	
BB11	(a,4) (k,4) (b,2)	(a,4) (x,8) (k,4) (b,2)	(a,4) (b,2)	(a,4) (b,2)	
BB12	(a,4) (x,8) (k,4) (b,2)	(a,4) (x,8) (k,4) (b,2) (y,8)	(a,4) (b,2)	(a,4) (b,2) (y,8)	
BB13	(a,4) (x,8) (k,4) (b,2) (y,8)	(a,4) (x,8) (b,2) (y,8) (k,5)	(a,4) (b,2) (y,8)	(a,4) (b,2) (y,8)	
BB14	(a,4) (k,4)	(a,4) (k,4)	(a,4)	(a,4)	
BB15 (exit)	(a,4) (k,4)		(a,4)		

*La terza iterazione è uguale alla seconda e si avrà convergenza.

Con Bitvector (in ordine di calcolo) < (k,2) (a,4) (x,5) (x,8) (k,4) (b,2) (y,8) (k,5) >

	Interazione 1		Interazione 2		Interazione 3*
	IN [B]	OUT[B]	IN [B]	OUT[B]	
BB1 (entry)		<00000000>		<00000000>	
BB2	<00000000>	<10000000>	<00000000>	<10000000>	
BB3	<10000000>	<10000000>	<10000000>	<10000000>	
BB4	<10000000>	<11000000>	<10000000>	<11000000>	
BB5	<11000000>	<11100000>	<11000000>	<11100000>	
BB6	<10000000>	<11000000>	<10000000>	<11000000>	
BB7	<11000000>	<11010000>	<11000000>	<11010000>	
BB8	<11000000>	<01001000>	<11000000>	<01001000>	
BB9	<01001000>	<01001000>	<01000000>	<01000000>	
BB10	<01001000>	<01001100>	<01000000>	<01000100>	
BB11	<01001100>	<01011100>	<01000100>	<01000100>	
BB12	<01011100>	<01011110>	<01000100>	<01000110>	
BB13	<01011110>	<01010111>	<01000110>	<01000110>	
BB14	<01001000>	<01001000>	<01000000>	<01000000>	
BB15 (exit)	<01001000>		<01000000>		

*La terza iterazione è uguale alla seconda e si avrà convergenza.