Very Busy Expressions

Il problema di ottimizzazione delle Very Busy Expressions è di tipo backwards, in quanto le espressioni vengono propagate o meno a punti precedenti del codice, se almeno uno degli operandi viene ridefinito in un'istruzione precedente o meno: ciò permette di propagare istruzioni che rimangono invariate fino a tale evento. Questo approccio permette di valutare se è possibile fare code hoisting con specifiche istruzioni ed è particolarmente utile nel caso siano presenti loop.
Le funzioni di Gen_B e $Kill_B$ sono definite come segue:

$$Gen_B = \{X \text{ op } Y \mid \text{Espressioni del tipo X op Y utilizzata nel Basic Block B}\}$$

$$Kill_B = \{X \text{ op } Y \mid \text{Espressioni del tipo X op Y dove X o Y vengono definite nel Basic Block B}\}$$

Inizialmente l'input di tutti i Basic Block è posto all'insieme vuoto, in quanto nessuno di questi ha espressioni Very Busy all'inizio della Dataflow Analysis.

Very Busy Expressions	
Domain	Set di espressioni del tipo X op Y
Direction	Backwards: in[B] = f_B(out[B]) out[B] = ^in[successor(B)]
Transfer Function	$f_B(x) = Gen_B \cup (x - Kill_B)$
Meet Operation (^)	∩
Boundary Condition	in[EXIT] = ∅
Initial Interior Points	in[B] = ∅

Fig. 1: Specchietto fornito compilato per il problema delle Very Busy Expressions.

La seguente tabella mostra le iterazioni relative al CFG presente nelle slide. Le iterazioni sono state fatte partire da BB8 e a risalire seguendo i predecessori:

Very Busy Expressions	1		2	
	IN	OUT	IN	OUT
BB1	{B-A,A!=B}	{B-A, A!=B}	{B-A,A!=B}	{B-A, A!=B}
BB2	{B-A,A!=B}	{B-A}	{B-A,A!=B}	{B-A}
BB3	{A-B,B-A}	{A-B}	{A-B,B-A}	{A-B}
BB4	{A-B}	∅	{A-B}	∅
BB5	{B-A}	∅	{B-A}	∅
BB6	∅	{A-B}	∅	{A-B}
BB7	{A-B}	∅	{A-B}	∅
BB8	∅	∅	∅	∅

Fig. 2: Tabella delle iterazioni dell'algoritmo relativo alle Very Busy Expressions applicato al CFG fornito.

Si arriva a convergenza piuttosto rapidamente, in quanto non sono presenti loop di alcuna natura. Il codice, in base ai risultati ottenuti, potrebbe essere ottimizzato hoistando l'espressione B-A prima del BB2 come una variabile nuova, come ad esempio T=B-A, che andrà a sostituire l'espressione in tutti i punti successivi a BB2 in cui compare.

Dominator Analysis

Un dominatore viene definito come un Basic Block che, dall'entry point ad un punto p del programma, è necessario attraversare per raggiungere p.
Il problema della Dominator Analysis è di tipo forward, in quanto i dominatori di un Basic Block vengono propagati ai successori di quest'ultimo. La funzione di Gen_B è definita come:

$$Gen_B = B$$

in quanto Bi ∈ DOM[Bi].
Inizialmente out[B] è posto a Insieme Universo, altrimenti, ogni volta che viene calcolato in[B] in situazioni come, ad esempio, un self loop o più predecessori con almeno uno di questi non ancora valutato, si va a svuotare l'insieme dei dominatori propagati.

Dominator Analysis	
Domain	Basic Blocks
Direction	Forward: out[B] = f_B(in[B]) in[B] = ^out[predecessor(B)]
Transfer Function	$f_B(x) = Gen_B \cup x$
Meet Operation (^)	∩
Boundary Condition	out[ENTRY] = ∅
Initial Interior Points	out[B] = Insieme Universo

Fig. 3: Specchietto fornito compilato per il problema della Dominator Analysis.

Dominator Analysis	1		2	
	IN	OUT	IN	OUT
A	∅	{A}	∅	{A}
B	{A}	{A,B}	{A}	{A,B}
C	{A}	{A,C}	{A}	{A,C}
D	{A,C}	{A,C,D}	{A,C}	{A,C,D}
E	{A,C}	{A,C,E}	{A,C}	{A,C,E}
F	{A,C}	{A,C,F}	{A,C}	{A,C,F}
G	{A}	{A,G}	{A}	{A,G}

Fig. 4: Tabella delle iterazioni dell'algoritmo relativo alla Dominator Analysis applicato al CFG fornito.

Constant Propagation

Il problema della Constant Propagation si occupa di verificare se raggiunto un punto p del codice una o più variabili sono costanti.
Per rappresentare lo stato delle costanti si è optato per un insieme di tuple <variabile, valore>, vengono invece omesse le variabili non costanti o non dichiarate contrassegnate con ∅. Si nota che è possibile ottenere un insieme di tutte le variabili e il loro relativo stato inserendo come valore possibile lo stato di Non Costanza e quello di Non Dichiarazione e adattando lievemente la Transfer Function e la Meet Operation.
Sia e(variabile) la funzione che mappa il nome della variabile al suo valore, le funzioni di Gen_B e $Kill_B$ sono definite come segue:

$$Gen_B = \{<x,c> \mid x \text{ è ottenuto come } x = y \text{ o } x = y \text{ op } z \text{ dove } e(y)\neq\emptyset \text{ e } e(z)\neq\emptyset\}$$

$$Kill_B = \{<x,c> \mid \text{nel caso in cui } x = y \text{ o } x = y \text{ op } z\ \}$$

Pertanto, vengono generate le tuple variabile – valore per variabili a cui sono stati assegnati valori costanti e vengono rimosse dall'insieme le relativa alla variabile a cui si sta assegnando un valore: se a tale variabile verrà assegnato un valore costante questa sarà inserita nell'insieme dalla funzione generatrice.

Constant Propagation	
Domain	Insieme di coppie <variabile,valore>
Direction	Forward: out[B] = f_B(in[B]) in[B] = ^out[predecessor(B)]
Transfer Function	$f_B(x) = Gen_B \cup (x - Kill_B)$
Meet Operation (^)	Vedi fig. 6
Boundary Condition	out[ENTRY] = ∅
Initial Interior Points	out[B] = ∅

Fig. 5: Specchietto fornito compilato per il problema della Dominator Analysis.

Per la Meet Operation si è optato per un operatore che propaghi la tupla della variabile x se date due tuple, <x,c> e <x,d>, si ha c=d mentre in tutti gli altri casi non viene propagata. Rispetto all'operatore di intersezione, questa Meet Operation differisce nel caso in cui si ha <x,c> e <x, ∅>, in quanto <x,c> viene propagato anche se <x, ∅> non è presente nell'insieme. Si tratta, in sintesi, di un operatore di intersezione che propaga gli elementi unici a un insieme e non all'altro.

^	**n**	**m**	∅
n	n	∅	n
m	∅	m	m
∅	n	m	∅

Fig. 6: Definizione del comportamento della meet operation.

Constant Propagation	**1**	
	IN	OUT
k=2	∅	<k,2>
if	<k,2>	<k,2>
a=k+2	<k,2>	<k,2>, <a,4>
x=5	<k,2>,<a,4>	<k,2>,<a,4>,<x,5>
a=k*2	<k,2>	<k,2>, <a,4>
x=8	<k,2>,<a,4>	<k,2>,<a,4>,<x,8>
k=a	<k,2>,<a,4>	<k,4>,<a,4>
while	<k,4>,<a,4>	<k,4>,<a,4>
b=2	<k,4>,<a,4>	<k,4>,<a,4>,<b,2>
x=a+k	<k,4>,<a,4>,<b,2>	<k,4>,<a,4>,<b,2>,<x,8>
y=a*b	<k,4>,<a,4>,<b,2>,<x,8>	<k,4>,<a,4>,<b,2>,<x,8>,<y,8>
k++ | k=k+1	<k,4>,<a,4>,<b,2>,<x,8>,<y,8>	<k,5>,<a,4>,<b,2>,<x,8>,<y,8>
print(a + x)	<k,4>,<a,4>	<k,4>,<a,4>
Constant Propagation	**2**	
	IN	OUT
k=2	∅	<k,2>
if	<k,2>	<k,2>
a=k+2	<k,2>	<k,2>, <a,4>
x=5	<k,2>,<a,4>	<k,2>,<a,4>,<x,5>
a=k*2	<k,2>	<k,2>, <a,4>
x=8	<k,2>,<a,4>	<k,2>,<a,4>,<x,8>
k=a	<k,2>,<a,4>	<k,4>,<a,4>
while	<a,4>,<b,2>,<x,8>,<y,8>	<a,4>,<b,2>,<x,8>,<y,8>
b=2	<a,4>,<b,2>,<x,8>,<y,8>	<a,4>,<b,2>,<x,8>,<y,8>
x=a+k	<a,4>,<b,2>,<x,8>,<y,8>	<a,4>,<b,2>,<y,8>
y=a*b	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
k++ | k=k+1	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
print(a + x)	<a,4>,<b,2>,<x,8>,<y,8>	<a,4>,<b,2>,<x,8>,<y,8>
Constant Propagation	**3**	
	IN	OUT
k=2	∅	<k,2>
if	<k,2>	<k,2>
a=k+2	<k,2>	<k,2>, <a,4>
x=5	<k,2>,<a,4>	<k,2>,<a,4>,<x,5>
a=k*2	<k,2>	<k,2>, <a,4>
x=8	<k,2>,<a,4>	<k,2>,<a,4>,<x,8>
k=a	<k,2>,<a,4>	<k,4>,<a,4>
while	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
b=2	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
x=a+k	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
y=a*b	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
k++ | k=k+1	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
print(a + x)	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
Constant Propagation	**4**	
	IN	OUT
k=2	∅	<k,2>
if	<k,2>	<k,2>

a=k+2	<k,2>	<k,2>, <a,4>
x=5	<k,2>,<a,4>	<k,2>,<a,4>,<x,5>
a=k*2	<k,2>	<k,2>, <a,4>
x=8	<k,2>,<a,4>	<k,2>,<a,4>,<x,8>
k=a	<k,2>,<a,4>	<k,4>,<a,4>
while	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
b=2	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
x=a+k	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
y=a*b	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
k++ | k=k+1	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>
print(a + x)	<a,4>,<b,2>,<y,8>	<a,4>,<b,2>,<y,8>

Fig. 7: Tabella delle iterazioni dell'algoritmo relativo alla Constant Propagation applicato al CFG fornito.