Marras, Colli, Lazzarini

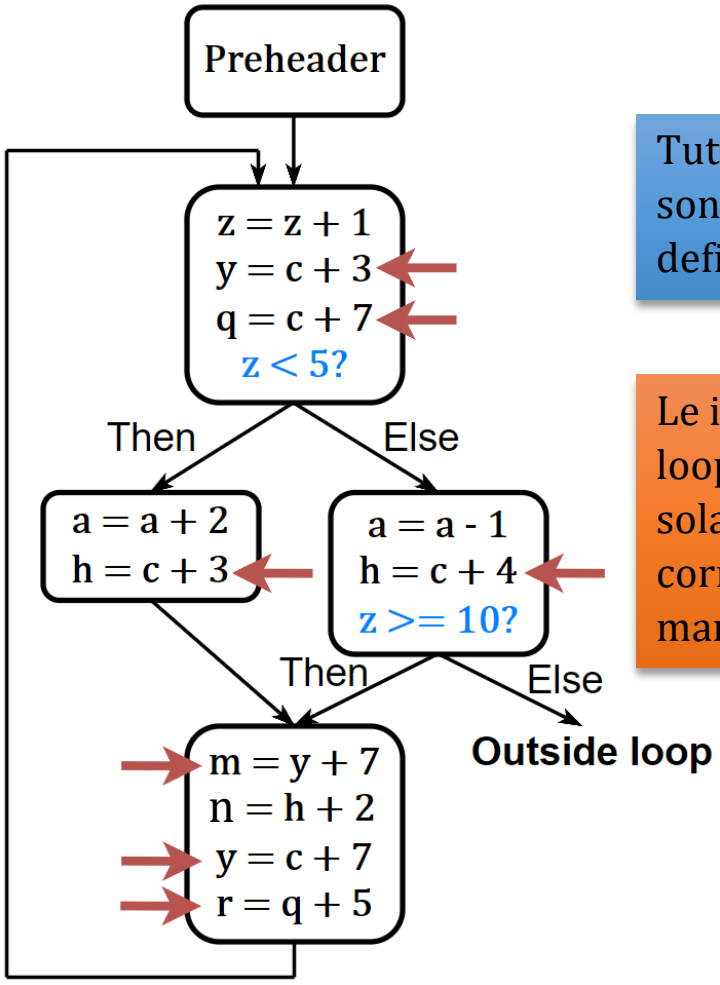
LINGUAGGI E COMPILATORI - ASSIGNMENT 3

Loop-Invariant Code Motion senza la forma SSA

Istruzioni Loop-invariant:

Le seguenti istruzioni indicate dalle frecce rosse sono **loop-invariant** perché possiedono una struttura del tipo $A = B \pm C$ dove:

1) Tutte le definizioni di **B** e **C** che raggiungono l'istruzione sono fuori dal loop.
2) Alternativamente c'è una e una sola **reaching definition** per ogni operando , e si tratta di un'istruzione già marcata come **loop-invariant**.

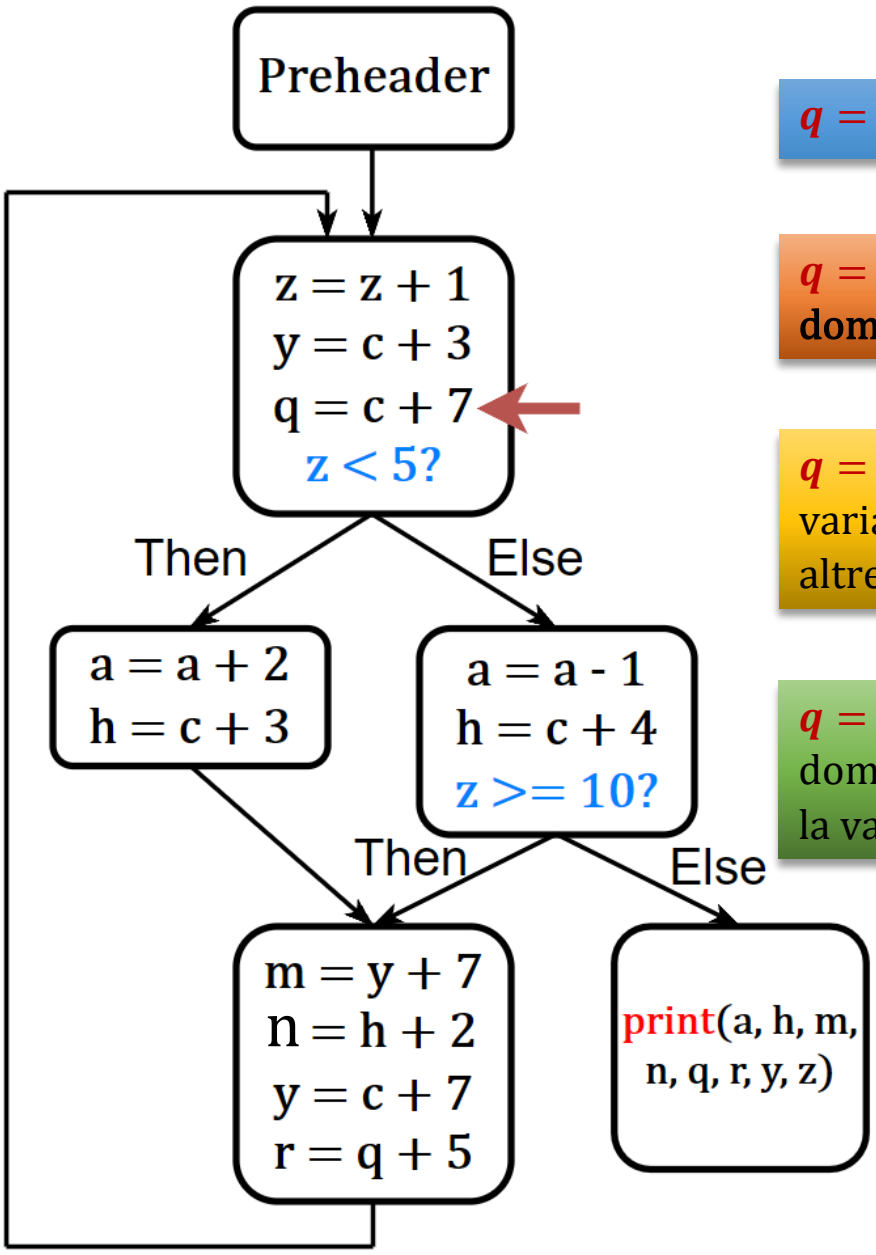


Tutte le istruzioni dalla forma $c + costante$ sono loop-invariant perché c è una variabile definita fuori dal loop.

Le istruzioni $m = y + 7$ e $r = q + 5$ sono loop-invariant perché possiedono una e una sola reaching definition di y e q, le quali corrispondono a istruzioni che sono già state marcate come loop-invariant.

Istruzioni candidate alla Code Motion:

La seguente istruzione indicata dalla freccia rossa è **candidata alla Code Motion** perché:



$q = c + 7$ è un'istruzione **loop-invariant.**

$q = c + 7$ si trova in un blocco che **domina tutte le uscite del loop.**

$q = c + 7$ assegna un valore alla variabile q, la quale non è assegnata in altre parti del loop.

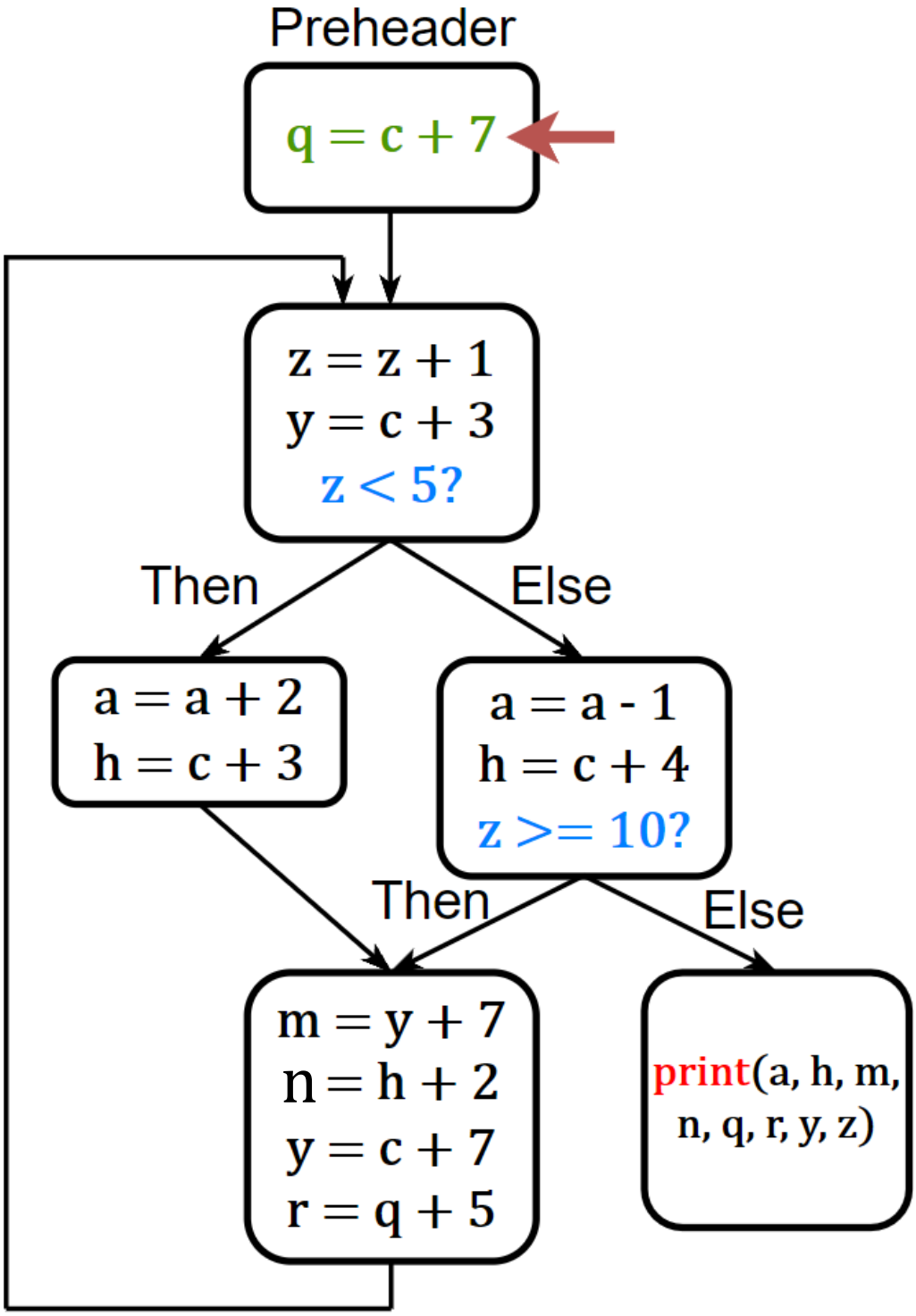
$q = c + 7$ si trova in un blocco che domina tutti i blocchi nel loop che usano la variabile q → $r = q + 5$.

Istruzioni che permettono la Code Motion:

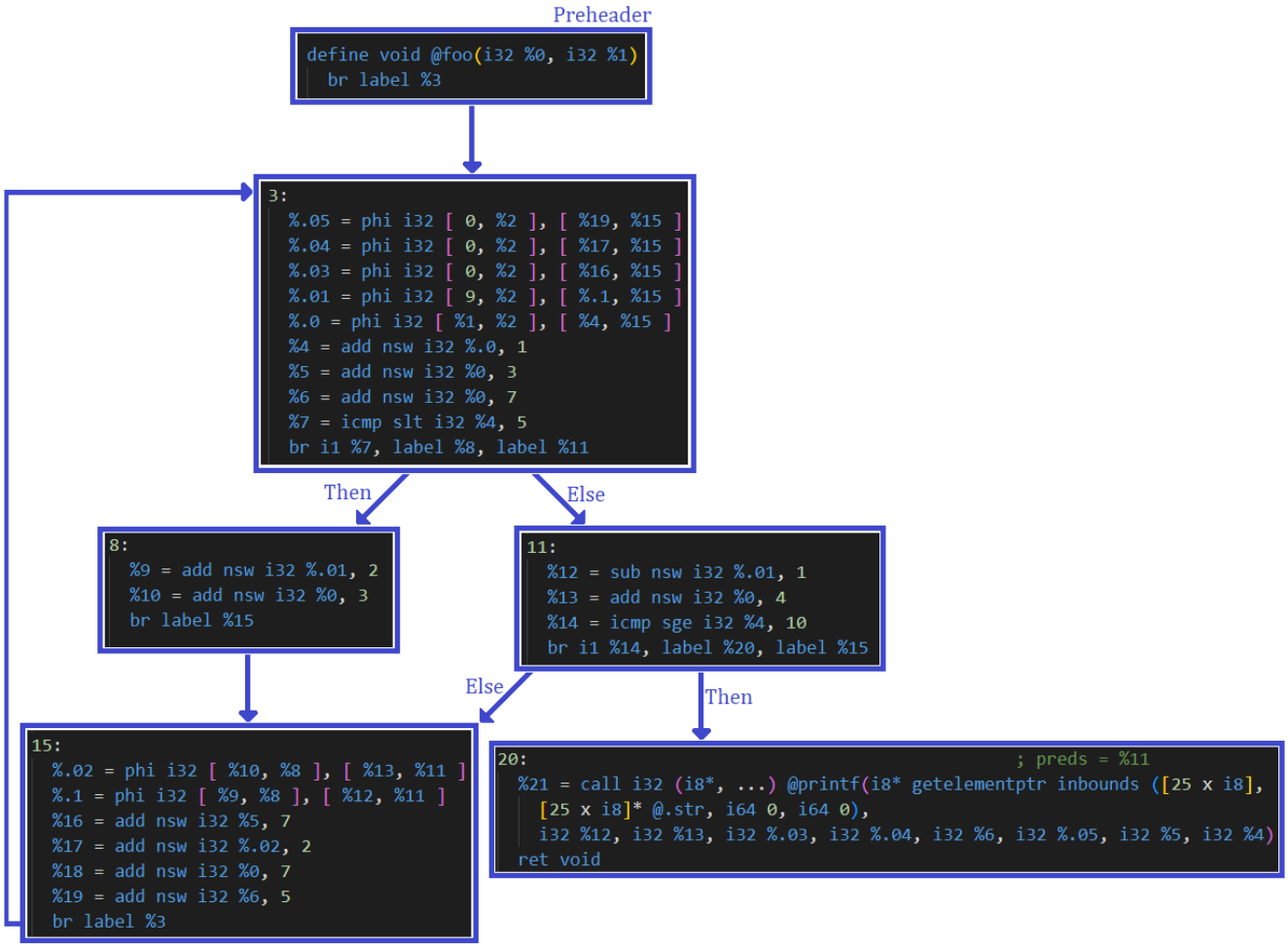
$q = c + 7$ permette la Code Motion, perché l'istruzione c da cui dipende non viene definita internamente al loop.

Risultato della Code Motion:

L'istruzione $q = c + 7$ è stata spostata nel Preheader.



Forma SSA:



Loop-Invariant Code Motion in forma SSA:

Istruzioni Loop-invariant:

Il riconoscimento delle istruzioni **Loop-invariant** si effettua nello stesso modo della forma **non SSA**, e pertanto si trovano lo stesso numero di istruzioni **Loop-invariant**:

```
Ricerca delle istruzioni Loop-invariant:
L'istruzione   %5 = add nsw i32 %0, 3 è loop invariant perché il primo operando è stato definito fuori dal loop
L'istruzione   %6 = add nsw i32 %0, 7 è loop invariant perché il primo operando è stato definito fuori dal loop
L'istruzione   %13 = add nsw i32 %0, 4 è loop invariant perché il primo operando è stato definito fuori dal loop
L'istruzione   %10 = add nsw i32 %0, 3 è loop invariant perché il primo operando è stato definito fuori dal loop
L'istruzione   %16 = add nsw i32 %5, 7 è loop invariant perché il primo operando è a sua volta loop invariant
L'istruzione   %18 = add nsw i32 %0, 7 è loop invariant perché il primo operando è stato definito fuori dal loop
L'istruzione   %19 = add nsw i32 %6, 5 è loop invariant perché il primo operando è a sua volta loop invariant
```

Istruzioni candidate alla Code Motion:

A differenza della forma **non SSA**, il risultato ottenuto risulta diverso. In particolare, le istruzioni **loop-invariant**:

- $\%5 = add\ nsw\ i32\ \%0, 3 \rightarrow y = c + 3$
- $\%18 = add\ nsw\ i32\ \%0, 7 \rightarrow y = c + 7$

che precedentemente non erano valide candidate alla **Code Motion**, ora lo sono dal momento che:

- $\%5$ si trova in un **Basic Block** che **domina** tutte le uscite del loop, assegna un valore a variabili non assegnate altrove nel loop (sempre rispettato dalla forma **SSA**) e si trova in un blocco che domina tutti i blocchi che usano $\%5$.
- $\%18$ si trova in un **Basic Block** che **NON domina** tutte le uscite del loop, ma è **dead** all'uscita, assegna un valore a variabili non assegnate altrove nel loop (sempre rispettato dalla forma **SSA**) e non ci sono usi di $\%18$ all'interno del Loop

```
Ricerca delle istruzioni candidate alla Code Motion:
  %5 = add nsw i32 %0, 3 NON è dead all'uscita del loop
  %6 = add nsw i32 %0, 7 NON è dead all'uscita del loop
  %10 = add nsw i32 %0, 3 NON domina un'uscita
  %10 = add nsw i32 %0, 3 NON domina un suo uso
  %13 = add nsw i32 %0, 4 NON è dead all'uscita del loop
  %13 = add nsw i32 %0, 4 NON domina un suo uso
  %16 = add nsw i32 %5, 7 NON domina un'uscita
  %16 = add nsw i32 %5, 7 NON domina un suo uso
  %18 = add nsw i32 %0, 7 NON domina un'uscita
  %19 = add nsw i32 %6, 5 NON domina un'uscita
  %19 = add nsw i32 %6, 5 NON domina un suo uso
-------------------------------
Istruzioni candidate alla Code Motion:
  %5 = add nsw i32 %0, 3
  %6 = add nsw i32 %0, 7
  %18 = add nsw i32 %0, 7
```

Istruzioni che permettono la Code Motion:

Le istruzioni che permettono la Code Motion sono le stesse che si ottengono dal passo precedente:

- $\%5 = add\ nsw\ i32\ \%0, 3 \rightarrow y = c + 3$
- $\%6 = add\ nsw\ i32\ \%0, 7 \rightarrow q = c + 7$
- $\%18 = add\ nsw\ i32\ \%0, 7 \rightarrow y = c + 7$

```
Istruzioni movable:
  %5 = add nsw i32 %0, 3
  %6 = add nsw i32 %0, 7
  %18 = add nsw i32 %0, 7
```

in quanto, l'istruzione **%0** da cui dipendono non viene definita internamente al loop.

Risultato della Code Motion:

Istruzioni su cui è stata effettuata la Code Motion

```
6   define void @foo(i32 %0, i32 %1) {
7     %3 = add nsw i32 %0, 3
8     %4 = add nsw i32 %0, 7
9     %5 = add nsw i32 %0, 7
10    br label %6
11
12  6:                                                ; preds = %16, %2
13    %.05 = phi i32 [ 0, %2 ], [ %19, %16 ]
14    %.04 = phi i32 [ 0, %2 ], [ %18, %16 ]
15    %.03 = phi i32 [ 0, %2 ], [ %17, %16 ]
16    %.01 = phi i32 [ 9, %2 ], [ %.1, %16 ]
17    %.0 = phi i32 [ %1, %2 ], [ %7, %16 ]
18    %7 = add nsw i32 %.0, 1
19    %8 = icmp slt i32 %7, 5
20    br i1 %8, label %9, label %12
21
22  9:                                                ; preds = %6
23    %10 = add nsw i32 %.01, 2
24    %11 = add nsw i32 %0, 3
25    br label %16
26
27  12:                                               ; preds = %6
28    %13 = sub nsw i32 %.01, 1
29    %14 = add nsw i32 %0, 4
30    %15 = icmp sge i32 %7, 10
31    br i1 %15, label %20, label %16
32
33  16:                                               ; preds = %12, %9
34    %.02 = phi i32 [ %11, %9 ], [ %14, %12 ]
35    %.1 = phi i32 [ %10, %9 ], [ %13, %12 ]
36    %17 = add nsw i32 %3, 7
37    %18 = add nsw i32 %.02, 2
38    %19 = add nsw i32 %4, 5
39    br label %6
40
41  20:                                               ; preds = %12
42    %21 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([25 x i8],
43      [25 x i8]* @.str, i64 0, i64 0),
44      i32 %13, i32 %14, i32 %.03, i32 %.04, i32 %4, i32 %.05, i32 %3, i32 %7)
45    ret void
46  }
```

Implementazione:

1) Verifica dell'applicabilità della Loop Invariant Code motion: si controlla che il loop sia in forma normalizzata, che abbia un preheader e uno o più exit block.

```
virtual bool isLoopSuitableToLICM(Loop *L)
{
    if ((*L).isLoopSimplifyForm() && (*L).getLoopPreheader() && (!(*L).hasNoExitBlocks()))
        return true;
    else
        return false;
}
```

2) Ricerca delle istruzioni loop invariant:

- Tutte le istruzioni del programma vengono inserite in una mappa che permette di taggarle nel caso siano loop invariant.

```
std::map<Instruction*, bool> loopInvariantMap;
```

- Vengono controllate solamente le istruzioni binarie, dal momento che sono le uniche sulle quali si potrebbe idealmente applicare la Loop invariant code motion. Tra le istruzioni non considerate ideali ci sono anche le **PHI**, in quanto non sarebbe possibile spostarle nel preheader senza cambiare la struttura del loop.

```
if (InstIter.isBinaryOp())
```

- Si verificano tutti i possibili casi nei quali un'istruzione viene considerata loop invariant con particolare attenzione al caso nel quale un'istruzione non abbia una vera rappresentazione in formato SSA.
  Ad esempio, il caso in cui il loop sia definito in una funzione e un operando di un'istruzione sia passato come parametro alla funzione stessa.

```
if(!(firstOperandInstruction = dyn_cast<Instruction>(InstIter.getOperand(0)))
    && !(secondOperandInstruction = dyn_cast<Instruction>(InstIter.getOperand(0))))
{
    loopInvariantMap[&InstIter] = true;
}
```

Se il cast a Instruction di entrambi gli operandi non ha successo, significa che non ci sono definizioni delle istruzioni corrispondenti a entrambi gli operandi, quindi l'istruzione è loop invariant.

3) Ricerca tra le istruzioni loop invariant delle istruzioni candidate alla code motion:

- Le istruzioni loop invariant candidate vengono aggiunte al vettore `std::vector<Instruction*> candidateMovableInstructions;`
- Affinché venga aggiunta, un'istruzione deve:
  - **Dominare tutte le uscite** o alternativamente **dev'essere dead all'uscita del loop**.
  - **Dominare tutti i suoi usi**.

```
if ((dominatesAllExits || deadAfterExit) && dominatesAllUses)
    candidateMovableInstructions.push_back(iter->first);
```

4) Ricerca tra le istruzioni candidate alla code motion delle istruzioni movable:

- Le istruzioni movable vengono aggiunte al vettore:
  `std::vector<Instruction*> movableInstructions;`
- Viene eseguita una ricerca **depth-first** in **reverse post order** sui basic block:

```
LoopBlocksDFS DFS(L);
DFS.perform(LI);
```

```
for (auto BB = DFS.beginRPO(); BB != DFS.endRPO(); ++BB)
```

  Se una candidata alla code motion dipende solamente da istruzioni che sono state spostate nel preheader, allora sarà un'istruzione movable.
- Si pone particolare attenzione sempre al caso menzionato prima, nel quale un operando di una candidata non abbia un'istruzione associata. In questo caso l'operando non ostacola la code motion:

```
if(!(operandInstruction = dyn_cast<Instruction>(instBB.getOperand(numOperand))))
    operandAllowsToCodeMotion[numOperand] = true;
```

5) Code motion:

- Si itera sulle istruzioni movable e le si sposta nel preheader.

```
virtual void codeMotion(Loop *L)
{
    BasicBlock * preHeader = (*L).getLoopPreheader();
    Instruction * preHeaderTerminator = preHeader->getTerminator();
    for (auto inst : movableInstructions)
    {
        inst->moveBefore(preHeaderTerminator);
    }
```