Relazione assignment #3 - Linguaggi & Compilatori pt.2
Matteo Lugli, Carlo Uguzzoni

1 Introduzione

L'esperienza consiste di realizzare un passo di ottimizzazione (in particolare, una Loop-Invariant Code Motion) su di un loop preso come caso di studio, usando l'API di LLVM.

L'algoritmo, già visto a lezione, è il seguente:

Dato un insieme di nodi in un loop

- Calcolare le *reaching definitions*
- Trovare le istruzioni *loop-invariant*
- Calcolare i dominatori (*dominance tree*)
- Trovare le uscite del loop (i successori fuori dal loop)
- Le istruzioni candidate alla *code motion*:
  - Sono *loop invariant*
  - Si trovano in blocchi che dominano tutte le uscite del loop
  - Assegnano un valore a variabili non assegnate altrove nel loop
  - Si trovano in blocchi che dominano tutti i blocchi nel loop che usano la variabile a cui si sta assegnando un valore
- Eseguire una ricerca *depth-first* dei blocchi
  - Spostare l'istruzione candidata nel *preheader* se tutte le istruzioni invarianti da cui questa dipende sono state spostate

Di seguito sono riportate le varie fasi, con relativi commenti.

2 Svolgimento

Calcolo delle Reaching Definitions Le reaching definitions non sono state ottenute preventivamente, nella definizione dell'algoritmo. Essendo la IR di LLVM in forma SSA, è bastato effettuare (ove possibile) sugli operandi delle espressioni un downcasting per ottenere la reaching definition.

```
    for (auto operand = I.op_begin(); operand != I.op_end(); ++operand) {
        Value *Operand = *operand;
...
        Instruction *Inst = dyn_cast<Instruction>(Operand);
```

Istruzioni Loop-Invariant L'algoritmo per individuare le istruzioni loop-invariant è il seguente:

Per ogni istruzione del tipo A=B+C marcare l'istruzione come Loop-INVARIANT se:

- Tutte le definizioni di B e C che raggiungono l'istruzione A=B+C si trovano fuori dal loop
  - Se B e/o C fossero costanti?
- **Oppure** c'è esattamente una *reaching definition* per B (e C), e si tratta di un'istruzione del loop che è stata marcata *loop-invariant*

Nell'applicarlo, si sono prese in esame tutte le binary operations, per semplicità. Il campo d'azione è stato poi ristretto, tagliando fuori le PHI-instructions. Queste, infatti, non possono per loro natura essere loop-invariant, poiché il loro valore dipende strettamente dal flusso del codice all'interno del loop. Anche in questo caso poi, si potrebbe non considerare una gamma più ampia di istruzioni non loop-invariant (es. load, store, ecc...).
Si è affrontato il problema con il seguente algoritmo, espresso in pseudo-codice:

```
map<Instruction, bool>

for BasicBlock BB in Loop:
    for Instruction I in BB:
        if I is binary && I is not PHI:
            isLoopInvariant = True
            for Operand O in I:
                if O is constant:
                    continue
                else
                    if map[reaching_def(O)] == False
                        isLoopInvariant = False
                    else
                        continue
            if isLoopInvariant:
                map[I] = True
            else:
                map[I] = False
        else
            map[I] = False
```

Per memorizzare le istruzioni e marcarle come loop-invariant, ci si è serviti di una map inizializzata vuota. Si sono poi scorse sequenzialmente tutte le istruzioni di tutti i basic block appartenenti al loop: quelle che corrispondono ai criteri visti sopra sono state prese in esame, le altre marcate direttamente come non loop-invariant.
Per determinare se un'istruzione fosse loop-invariant si è usato un flag booleano inizializzato a True, che cambia a False solo in caso di lookup negativo (istruzione non loop-invariant interna al loop).I lookup della map che non danno risultati (istruzioni esterne al loop), così come quelli positivi (istruzioni lop-invariant interne al loop), non alterano il suo valore.
Si è supposto infatti che nel caso in cui un lookup della map non dia risultati, allora l'istruzione non si è ancora incontrata all'interno del loop (dato che il controllo sulle istruzioni è sequenziale), quindi deve provenire necessariamente dall'esterno del loop.
Anche in questo caso, la forma SSA ci viene incontro, garantendo l'esistenza di una sola reaching definition per ciascun operando. Così, il secondo punto dell'algoritmo formalizzato a lezione, è stato sostanzialmente accorpato al primo.

Calcolare il Dominance Tree Il dominance tree è stato necessario al momento di determinare le istruzioni candidate alla code motion (passo descritto più avanti). Per questo,

si è eseguita un'analisi preventiva con l'API di LLVM, preservandone le informazioni, e poi da questa ottenuto il dominance tree.

```
virtual void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<DominatorTreeWrapperPass>();
    ...
}

virtual bool runOnLoop(Loop *L, LPPassManager &LPM) override {
    ...
    DominatorTree *DT = &getAnalysis<DominatorTreeWrapperPass>().getDomTree();
    ...
}
```

Trovare le uscite del loop Si è usato il seguente metodo, a cui è stato passato uno SmallVector che viene riempito con i blocchi che rappresentano le uscite del loop. Usare questo tipo di struttura dati impatta positivamente le performance del passo di ottimizzazione.

```
llvm::SmallVector<llvm::BasicBlock*> ExitingBlocks;
L->getExitingBlocks(ExitingBlocks);
```

Trovare le istruzioni candidate alla Code Motion Per realizzare questo passo, si è partiti dalla map contenente le istruzioni del loop. Per ciascuna istruzione marcata come loop-invariant, si è controllato che:

- Il blocco di appartenenza dominasse tutte le uscite del loop (raccolte nel vector di cui sopra);
- Il blocco di appartenenza dominasse tutti i blocchi nel loop che ne facevano uso;

Non è stato necessario controllare se le variabili fossero definite in altri blocchi del loop, sempre per la forma SSA.
Come accennato, in questa fase ci si è serviti del dominance tree a nostra disposizione, per verificare le proprietà di dominanza.

Eseguire la Code Motion In quest'ultimo punto, si è visitato il dominance tree secondo l'ordine dettato da una DFS, per muovere una ad una le istruzioni idonee alla code motion nel preheader.
L'esistenza di quest'ultimo è garantita dal controllo eseguito all'inizio dell'ottimizzazione, in cui si verifica che il loop sia in forma canonica:

```
if(!L->isLoopSimplifyForm()){
    std::cout << "Loop is not in canonical form!" << std::endl;
    return false;
}
```

Per la visita del dominance tree si è usato un comodo costrutto messo a disposizione dall'API di LLVM:

```
for (
  auto node = GraphTraits<DominatorTree *>::nodes_begin(DT);
  node != GraphTraits<DominatorTree *>::nodes_end(DT);
  ++node
  ) {
    ...
}
```

Le istruzioni sono state individuate nell'ordine corretto in un primo sweep, ed inserite in un vector. Dopodiché, iterando sul vector le si è svincolate dal basic block di appartenenza, ed aggiunte al preheader.
L'operazione è stata eseguita in due fasi, perché spostare le istruzioni nel preheader in fase di iterazione (primo sweep) avrebbe dato origine ad un ciclo infinito.