**Loop-invariant code motion**

Two steps: analysis & transformation

Step 1: find invariant computations in loop
- invariant: computes same result each time evaluated

Step 2: move them outside loop
- to top: **code hoisting**
  - if used within loop
- to bottom: **code sinking**
  - if only used after loop

**Example**

```
p := w + y
x := x + 1
q := q + 1
w := w + 5
z := x * y
q := y * y
w := y + 2
y := 4
x := 3
y := 5
```

**Detecting loop-invariant expressions**

An expression is invariant w.r.t. a loop $L$ iff:

**base cases:**
- it’s a constant
- it’s a variable use, **all of whose defs are outside $L$**

**inductive cases:**
- it’s an idempotent computation
  
    all of whose args are loop-invariant
- it’s a variable use **with only one reaching def**, and the rhs of that def is loop-invariant

**Computing loop-invariant expressions**

Option 1:
- repeat iterative dfa
  
    until no more invariant expressions found
- to start, optimistically assume all expressions loop-invariant

Option 2:
- build def/use chains,
  
    follow chains to identify & propagate invariant expressions

Option 3:
- convert to SSA form,
  
    then similar to def/use form
Example using def/use chains

Example using SSA form

Example using SSA form & preheader

Loop-invariant expression detection for SSA form

SSA form simplifies detection of loop invariants, since each use has only one reaching definition.

An expression is invariant w.r.t. a loop $L$ iff:

base cases:

- it’s a constant
- it’s a variable use whose single def is outside $L$

inductive cases:

- it’s an idempotent computation all of whose args are loop-invariant
- it’s a variable use whose single def’s rhs is loop-invariant

$\phi$ functions are not idempotent
Code motion

When find invariant computation \( S : z := x \, \text{op} \, y \), want to move it out of loop (to loop preheader)

When is this legal?

Sufficient conditions:

- \( S \) dominates all loop exits
  - \( A \) dominates \( B \) when all paths to \( B \) must first pass through \( A \)
  - otherwise may execute \( S \) when never executed otherwise
  - can relax this condition, if \( S \) has no side-effects or traps, at cost of possibly slowing down program

- \( S \) is only assignment to \( z \) in loop, & no use of \( z \) in loop is reached by any def other than \( S \)
- otherwise may reorder defs/uses and change outcome
- unnecessary in SSA form!

If met, then can move \( S \) to loop preheader

- but preserve relative order of invariant computations, to preserve data flow among moved statements

### Example of need for domination requirement

\[
\begin{align*}
x &:= 0 \\
y &:= 1 \\
z &!:= 0?
\end{align*}
\]

\[
\begin{align*}
x &:= a \times b \\
y &:= x / z \\
q &:= x + y
\end{align*}
\]

### Avoiding domination restriction

Requirement that invariant computation dominates exit is strict

- nothing in conditional branch can be moved
- nothing after loop exit test can be moved

Can be circumvented through other transformations such as loop normalization

- move loop exit test to bottom of loop (while-do ⇒ do-while)

### Example of data dependence restrictions

“\( S \) is only assignment to \( z \) in loop, & no use of \( z \) in loop is reached by any def other than \( S \)”

\[
\begin{align*}
z &:= 5 \\
z &:= z + 1
\end{align*}
\]
Example in SSA form

Restrictions unnecessary if in SSA form

- If reorderdefs/uses, generate code along merging arcs to implement $\phi$ functions

```
z_{1} := 5
z_{2} := \phi(z_{1}, z_{4})
z_{3} := z_{2} + 1
S: z_{4} := 0
... z_{4} ...
```

Loop-invariant code copying

Alternative to code motion:

- **copy** instruction to loop header, assigning to new temp, then do CSE & copy propagation to simplify in-loop version
- More modular design, leverage off of existing optimizations

Can always copy, unless instruction has side-effects

CSE & copy propagation will eliminate in-loop instruction exactly when (non-SSA) loop-invariant code motion would have, PLUS can replace invariant but unmovable instructions with copies

SSA-based code motion gets same effect

- Copies correspond to reified $\phi$ functions

Example

```
x := a * b
y := q * x
q := z * w
q := 0
y := 1
... y ...
... q ...
```