One-Slide Summary

- An optimization changes a program so that it computes the same answer in less time (or using less of some other resource).
- We represent the program using a special intermediate form.
- Each method is viewed as a control flow graph where the nodes as basic blocks of instructions with known entry and exit points. The instructions have been changed so that a single assignment defines each variable.

Lecture Outline

- Intermediate code
- Local optimizations
- Next time: larger-scale program analyses
When To Optimize?

- When to perform optimizations
  - On AST (just like type checking)
    - Pro: Machine independent
    - Cons: Too high level
  - On assembly language (compilers only)
    - Pro: Exposes optimization opportunities
    - Cons: Machine dependent
    - Cons: Must reimplement optimizations when retargeting
  - On an intermediate language
    - Pro: Machine independent
    - Pro: Exposes optimization opportunities
    - Cons: One more language to worry about

Intermediate Languages

- Each compiler uses its own intermediate language
  - IL design is still an active area of research
- Intermediate language = high-level assembly language
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
  - Uses opcodes but some are higher level
    - e.g., push translates to several assembly instructions
    - Most opcodes correspond directly to assembly opcodes

Three-Address Intermediate Code

- Each instruction is of the form
  \[ x := y \text{ op } z \]
  - y and z can be only registers, variables or constants
- Common form of intermediate code
- The AST expression \( x + y \cdot z \) is translated as
  \[ t_1 := y \cdot z \]
  \[ t_2 := x + t_1 \]
  - Each subexpression lives in a temporary
Generating Intermediate Code

- **igen(e, t)** function generates code to compute the value of *e* in register *t*

- **Example:**
  
  \[
  \text{igen}(e_1 + e_2, t) = \\
  \text{igen}(e_1, t_1) \quad (t_1 \text{ is a fresh register}) \\
  \text{igen}(e_2, t_2) \quad (t_2 \text{ is a fresh register}) \\
  t := t_1 + t_2
  \]

- **Unlimited number of registers**
  
  ⇒ simple code generation

---

An Intermediate Language

\[
P \rightarrow S \mid e \\
S \rightarrow \text{id} := \text{id} \ op \ \text{id} \\
| \ id := \ op \ id \\
| \ id := \ id \\
| \ push \ id \\
| \ id := \ pop \\
| \ \text{id} \ relop \ \text{id} \ \text{goto} \ L \\
| \ L: \\
| \ \text{jump} \ L
\]

- **id**'s are register names
- Constants can replace **id**'s
- Typical operators: +, -, *

---

Basic Blocks

- A **basic block** is a maximal sequence of instructions with:
  - no labels (except at the first instruction), and
  - no jumps (except in the last instruction)

- **Idea:**
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - Each instruction in a basic block is executed after all the preceding instructions have been executed
Basic Block Example

- Consider the basic block
  1. L1:
  2. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. \( \text{if } w > 0 \text{ goto } L2 \)
- No way for (3) to be executed without (2) having been executed right before
  - We can change (3) to \( w := 3 \times x \)
  - Can we eliminate (2) as well?

Control-Flow Graphs

- A control-flow graph is a directed graph:
  - Basic blocks as nodes
  - An edge from block A to block B if the execution can flow from the last instruction in A to the first instruction in B
    - e.g., the last instruction in A is jump \( L_B \)
    - e.g., the execution can fall-through from block A to block B
- Frequently abbreviated as CFG

Control-Flow Graphs. Example.

- The body of a method (or procedure) can be represented as a control-flow graph
- There is one initial node
  - The “start node”
- All “return” nodes are terminal
Optimization Overview

- **Optimization** seeks to improve a program’s utilization of some resource
  - Execution time (most often)
  - Code size
  - Network messages sent
  - Battery power used, etc.
- Optimization should **not** alter what the program computes
  - The answer must still be the same

A Classification of Optimizations

- For languages like C and Cool there are three granularities of optimizations
  1. **Local optimizations**
     - Apply to a basic block in isolation
  2. **Global optimizations**
     - Apply to a control-flow graph (method body) in isolation
  3. **Inter-procedural optimizations**
     - Apply across method boundaries
- Most **compilers** do (1), many do (2) and very few do (3)
- Some **interpreters** do (1), few do (2), basically none do (3)
Cost of Optimizations

- In practice, a conscious decision is made **not** to implement the fanciest optimization known.
- Why?
  - Some optimizations are hard to implement.
  - Some optimizations are costly in terms of compilation/interpretation time.
  - The fancy optimizations are both hard and costly.
- The goal: maximum improvement with minimum of cost.

Local Optimizations

- The simplest form of optimizations.
- No need to analyze the whole procedure body.
  - Just the basic block in question.
- Example:
  - algebraic simplification
  - constant folding
  - Python 2.5 does stuff like this if you say “-O”.

Algebraic Simplification

- Some statements can be deleted:
  - $x := x + 0$
  - $x := x * 1$
- Some statements can be simplified:
  - $x := x * 0 \Rightarrow x := 0$
  - $y := y ** 2 \Rightarrow y := y * y$
  - $x := x * 8 \Rightarrow x := x << 3$
  - $x := x * 15 \Rightarrow t := x << 4; x := t - x$
(On some machines $<<$ is faster than $*$; but not on all!)
Constant Folding

- Operations on constants can be computed before the code executes.
- In general, if there is a statement
  \[ x := y \text{ op } z \]
  - And \( y \) and \( z \) are constants
  - Then \( y \text{ op } z \) can be computed early.
- Example: \( x := 2 + 2 \Rightarrow x := 4 \)
- Example: if \( 2 < 0 \) jump L can be deleted.
- When might constant folding be dangerous?

Flow of Control Optimizations

- Eliminating unreachable code:
  - Code that is unreachable in the control-flow graph.
  - Basic blocks that are not the target of any jump or “fall through” from a conditional.
  - Such basic blocks can be eliminated.
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller.
  - And sometimes also faster.
    - Due to memory cache effects (increased spatial locality).

Single Assignment Form

- Most optimizations are simplified if each assignment is to a temporary that has not appeared already in the basic block.
- Intermediate code can be rewritten to be in single assignment form:
  \[
  \begin{align*}
  x := a + y & \Rightarrow x := a + y \\
  a := x & \Rightarrow a_1 := x \\
  x := a \ast x & \Rightarrow x_1 := a_1 \ast x \\
  b := x + a & \Rightarrow b := x_1 + a_1 \\
  \end{align*}
  \]
  (\( x_1 \) and \( a_1 \) are fresh temporaries)
Single Assignment vs. Functional Programming

- In functional programming variable values do not change.
- Instead you make a new variable with a similar name.
- Single assignment form is just like that!

\[
\begin{align*}
x &:= a + y & \text{let } x = a + y \text{ in} \\
a_1 &:= x & \text{let } a_1 = x \text{ in} \\
x_1 &:= a_1 \ast x & \text{let } x_1 = a_1 \ast x \text{ in} \\
b &:= x_1 + a_1 & \text{let } b = x_1 + a_1 \text{ in}
\end{align*}
\]

Common Subexpression Elimination

- Assume:
  - Basic block is in single assignment form.
- Then all assignments with same rhs compute the same value *(why?)*

\[
\begin{align*}
x &:= y + z & \Rightarrow \quad w &:= x
\end{align*}
\]

- Why is single assignment important here?

Copy Propagation

- If \( w := x \) appears in a block, all subsequent uses of \( w \) can be replaced with uses of \( x \).

\[
\begin{align*}
b &:= z + y & \Rightarrow \quad b &:= z + y \\
a &:= b & \Rightarrow \quad a &:= b \\
x &:= 2 \ast a & \Rightarrow \quad x &:= 2 \ast b
\end{align*}
\]

- This does not make the program smaller or faster but might enable other optimizations
  - Constant folding
  - Dead code elimination (we’ll see this in a bit!)
- Again, single assignment is important here.
Copy Propagation and Constant Folding

- Example:
  
  \[
  \begin{align*}
  a & := 5 \\
  x & := 2 \times a \quad \Rightarrow \quad x := 10 \\
  y & := x + 6 \\
  t & := x \times y \\
  \end{align*}
  \]

  \[
  \begin{align*}
  y & := x + 6 \\
  t & := x \times y \\
  \end{align*}
  \]

Dead Code Elimination

If

- \( w := \text{rhs} \) appears in a basic block
- \( w \) does not appear anywhere else in the program

Then

- the statement \( w := \text{rhs} \) is dead and can be eliminated
  - \textit{Dead} = does not contribute to the program’s result

Example: (\( a \) is not used anywhere else)

\[
\begin{align*}
  x & := z + y \\
  a & := x \quad \Rightarrow \quad a := b \\
  x & := 2 \times a \\
  \end{align*}
\]

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations \textit{interact}
  - Performing one optimizations enables other opts
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
- Interpreters and JITs must be fast!
  - The optimizer can also be stopped at any time to limit the compilation time
An Example

• Initial code:
  \[
  a := x \cdot x \\
  b := 3 \\
  c := x \\
  d := c \cdot c \\
  e := b \cdot 2 \\
  f := a + d \\
  g := e \cdot f
  \]

An Example

• Algebraic optimization:
  \[
  a := x \cdot x \\
  b := 3 \\
  c := x \\
  d := c \cdot c \\
  e := b \cdot 2 \\
  f := a + d \\
  g := e \cdot f
  \]
An Example

• Copy propagation:
  a := x * x
  b := 3
  c := x
  d := c * c
  e := b + b
  f := a + d
  g := e * f

An Example

• Copy propagation:
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 3 + 3
  f := a + d
  g := e * f

An Example

• Constant folding:
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 3 + 3
  f := a + d
  g := e * f
An Example

• Constant folding:
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 6
  f := a + d
  g := e * f

An Example

• Common subexpression elimination:
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 6
  f := a + d
  g := e * f
An Example

• Copy propagation:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := a \]
  \[ e := 6 \]
  \[ f := a + d \]
  \[ g := e \times f \]

An Example

• Copy propagation:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := a \]
  \[ e := 6 \]
  \[ f := a + a \]
  \[ g := 6 \times f \]

An Example

• Dead code elimination:
  \[ a := x \times x \]
  \[ b := 3 \]
  \[ c := x \]
  \[ d := a \]
  \[ e := 6 \]
  \[ f := a + a \]
  \[ g := 6 \times f \]
An Example

- Dead code elimination:
  
  \[ a := x \times x \]

  \[ f := a + a \]

  \[ g := 6 \times f \]

- This is the final form

Cool and Intermediate Form

- Cool does not have `goto`
- Cool does not have `break`
- Cool does not have `exceptions`

- How would you make basic blocks from a Cool AST?

Local Optimization Notes

- Intermediate code is helpful for many optimizations
  - Basic Blocks: known entry and exit
  - Single Assignment: one definition per variable
- “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is not optimal in any reasonable sense
  - “Program improvement” is a more appropriate term
- Next: larger-scale program changes
Homework

- PA4 due this Friday March 30th (3 days)
- Midterm 2 - Thursday April 12 (17 days)