Local Optimizations

When I'm walking, I worry a lot about the efficiency of my path.
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
Why Optimize?

- What's the point?
- Do we care about this in real life?
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 retargetting
  - On an intermediate language
    • Pro: Machine independent
    • Pro: Exposes optimization opportunities
    • Cons: One more language to worry about

You do not have to know assembly language.
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 \( x + y \ast z \) is translated as
  \[ t_1 := y \ast z \]
  \[ t_2 := x + t_1 \]
  - 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:
  
  \[
  igen(e_1 + e_2, t) = \\
  igen(e_1, t_1) \quad (t_1 \text{ is a fresh register}) \\
  igen(e_2, t_2) \quad (t_2 \text{ is a fresh register}) \\
  t := t_1 + t_2
  \]

- Unlimited number of registers
  \[\Rightarrow\] simple code generation
An Intermediate Language

\[ P \rightarrow S P \mid \varepsilon \]
\[ S \rightarrow id := id \ op \ id \]
\[ \quad \mid id := op \ id \]
\[ \quad \mid id := id \]
\[ \quad \mid push \ id \]
\[ \quad \mid id := pop \]
\[ \quad \mid if \ id \ relop \ id \ goto \ L \]
\[ \quad \mid L: \]
\[ \quad \mid jump \ L \]

- 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. $t := 2 * x$
  3. $w := t + x$
  4. if $w > 0$ goto L2

• No way for (3) to be executed without (2) having been executed right before
Basic Block Example

• Consider the basic block
  1. L1:
  2. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) goto L2

• No way for (3) to be executed without (2) having been executed right before
  - We can change (3) to \( w := 3 \times x \)
Basic Block Example

• Consider the basic block
  1. L1:
  2. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) 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

```plaintext
x := 1
i := 1

L:
x := x * x
i := i + 1
if i < 10 goto L
```
CREATING AN AIM PROFILE:

HAVE FRIENDS?

NO

HAVE BOYFRIEND/GIRLFRIEND?

YES

A PROFILE TRIBUTE IS THE GREATEST POSSIBLE EXPRESSION OF LOVE.

LINK TO YOUR LIVEJOURNAL

NO

INSIDE JOKES!

YES

AND WANT TO ALIENATE EVERYONE ELSE

ANGSTY ABOUT IT?

YES

NO

YES YOU ARE

NO
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?

THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE'S COMPILING."

HEY! GET BACK TO WORK!

COMPILING!

OH. CARRY ON.
Cost of Optimizations

• In practice, a conscious decision is made \textit{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.
Q: Movies (363 / 842)

• This 1993 comedy film also starring Andie MacDowell "begins" with the following radio banter: "Rise and shine, campers, and don't forget your booties 'cause it's cooooold out there today. / It's cold out there every day. What is this, Miami Beach? / Not hardly. So the big question on everybody's lips / -- On their chapped lips -- / their chapped lips is, does Phil feel lucky?"
Q: Cartoons (674 / 842)

• This 1953 Warner Brothers' cartoon mouse is known for his cry of "Arriba! Arriba! Andele!"
CFG

• This CFG stuff sounds complicated ...

• Can't we skip it for now?
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
  \[ x := x + 0 \]
  \[ x := x \times 1 \]

• Some statements can be simplified
  \[ x := x \times 0 \quad \Rightarrow \quad x := 0 \]
  \[ y := y \times^2 \quad \Rightarrow \quad y := y \times y \]
  \[ x := x \times 8 \quad \Rightarrow \quad x := x \ll 3 \]
  \[ x := x \times 15 \quad \Rightarrow \quad t := x \ll 4; \; x := t - x \]
  (on some machines \( \ll \) is faster than \( \times \); 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:

\[
\begin{align*}
x & := a + y \\
a & := x \quad \Rightarrow \quad a_1 := x \\
x & := a \times x \\
b & := x + a \\
b & := x_1 + a_1
\end{align*}
\]

\((x_1 \text{ and } a_1 \text{ are fresh temporaries})\)
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!

\[
\begin{align*}
x & := a + y \\
a_1 & := x \\
x_1 & := a_1 \times x \\
b & := x_1 + a_1
\end{align*}
\]

\[
\begin{align*}
\text{let } x & = a + y \text{ in} \\
\text{let } a_1 & = x \text{ in} \\
\text{let } x_1 & = a_1 \times x \text{ in} \\
\text{let } b & = x_1 + a_1 \text{ in}
\end{align*}
\]
Common Subexpression Elimination

• Assume:
  - Basic block is in single assignment form

• Then all assignments with same rhs compute the same value \((\text{why?})\)

• Example:

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

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

• 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 \( x \)

• Example:

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

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

⇒

\[
\begin{align*}
a & := 5 \\
x & := 10 \\
y & := 16 \\
t & := x \ll 4
\end{align*}
\]
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

- **Dead** = does not contribute to the program’s result

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

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

- Each local optimization does very little by itself
- Typically optimizations *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 ** 2
  b := 3
  c := x
  d := c * c
  e := b * 2
  f := a + d
  g := e * f
An Example

- Algebraic optimization:
  
  \[
  \begin{align*}
  a &:= x^{**2} \\
  b &:= 3 \\
  c &:= x \\
  d &:= c \times c \\
  e &:= b \times 2 \\
  f &:= a + d \\
  g &:= e \times f
  \end{align*}
  \]
An Example

• Algebraic optimization:

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := c \times c \\
e & := b + b \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
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:

\[
\begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := x \times x \\
  e & := 3 + 3 \\
  f & := a + d \\
  g & := e \times f
\end{align*}
\]
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

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

• Copy propagation:

\[
\begin{align*}
    a & := x \times x \\
    b & := 3 \\
    c & := x \\
    d & := a \\
    e & := 6 \\
    f & := a + d \\
    g & := e \times f
\end{align*}
\]
An Example

• Copy propagation:

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

• Dead code elimination:

\[
\begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := a \\
  e & := 6 \\
  f & := a + a \\
  g & := 6 \times f
\end{align*}
\]
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 tomorrow
  - Use spiffy auto-testing feature
• Reading for Thursday (basic blocks, etc.)
• Midterm 2 - Tuesday April 15 (19 days)