Compilers

Optimization Overview
• Optimization is our last compiler phase

• Most complexity in modern compilers is in the optimizer
  – Also by far the largest phase
- **When should we perform optimizations?**
  - **On AST**
    - **Pro:** Machine independent
    - **Con:** Too high level
  - **On assembly language**
    - **Pro:** Exposes optimization opportunities
    - **Con:** Machine dependent
    - **Con:** Must reimplement optimizations when retargetting
  - **On an intermediate language**
    - **Pro:** Machine independent
    - **Pro:** Exposes optimization opportunities
• **Intermediate language = high-level assembly**
  – 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
P → S P | S
S → id := id op id
    | id := op id
    | id := id
    | push id
    | id := pop
    | if id relop id goto L
    | L:
    | jump L

- Id’s are register names
- Constants can replace id’s
- Typical operators: +, -, *
• 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)
  – A basic block is a single-entry, single-exit, straight-line code segment
• Consider the basic block
  1. L:
  2. \( t := 2 \times x \)
  3. \( w := t + x \)
  4. if \( w > 0 \) goto L’

• (3) executes only after (2)
  – We can change (3) to \( w := 3 \times x \)
  – Can we eliminate (2) as well?
• A control-flow graph is a directed graph with
  – Basic blocks as nodes
  – An edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B
    • E.g., the last instruction in A is \textit{jump L}_B
    • E.g., execution can fall-through from block A to block B
An example control-flow graph

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

• Optimization seeks to improve a program’s resource utilization
  – Execution time (most often)
  – Code size
  – Network messages sent, memory usage, etc.

• Optimization should not alter what the program computes
  – The answer must still be the same
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), few do (3)
In practice, often a conscious decision is made not to implement the fanciest optimization known in research literature.

Why?
- Some optimizations are hard to implement
- Some optimizations are costly in compilation time
- Some optimizations have low payoff
- Many fancy optimizations are all three!

Goal: Maximum benefit for minimum cost
Compilers

Local Optimization
• The simplest form of optimization

• Optimize one basic block
  • No need to analyze the whole procedure body
Local Optimization

- 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!)

Algebraic Simplification
• Operations on constants can be computed at compile time
  – If there is a statement \( x := y \circ z \)
  – And \( y \) and \( z \) are constants
  – Then \( y \circ z \) can be computed at compile time

  Constant Folding

• Example: \( x := 2 + 2 \Rightarrow x := 4 \)
• Example: if \( 2 < 0 \) jump L can be deleted
• Constant folding can be dangerous.

Assume that the compiler is being run on modern machine X and code is going to be run on an old machine Y where X and Y are different machines (i.e., called a cross compiler).

Example: different floating point results

\[ a := 1.5 + 3.7 , \ a = 5.2 \text{ on X and } a = 5.19 \text{ on Y} \]
• **Eliminate unreachable basic blocks:**
  – Code that is unreachable from the initial block
  • E.g., basic blocks that are not the target of any jump or “fall through” from a conditional

• **Removing unreachable code makes the program smaller**
  – And sometimes also faster
  • Due to memory cache effects
  • Increased spatial locality
Why would unreachable basic blocks occur?

- Debug mode
  
  ```
  #define DEBUG 0
  If (DEBUG) then ...
  ```

- Libraries

- Result of other optimizations
Local Optimization

• Some optimizations are simplified if each register occurs only once on the left-hand side of an assignment

• Rewrite intermediate code in *single assignment* form

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

(b is a fresh register)

— More complicated in general, due to loops
• If
  – Basic block is in single assignment form
  – A definition \( x := \) is the first use of \( x \) in a block
• Then
  – When two assignments have the same rhs, they compute the same value
• Example:
  \[
  x := y + z \quad \Rightarrow \quad x := y + z \quad \text{Common Sub-expression Elimination}
  
  w := y + z \quad \Rightarrow \quad w := x
  \]
  (the values of \( x, y, \) and \( z \) do not change in the \( ... \) code)
Local Optimization

- If \( w := x \) appears in a block, replace subsequent uses of \( w \) with uses of \( x \)
  - Assumes single assignment form

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

- Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination
• Example:

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

\[
\begin{align*}
a & := 5 \\
x & := 10 \\
y & := 16 \\
t & := x \ll 4 \\
\text{Or} \\
t & := 160
\end{align*}
\]
If

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

Then

the statement \( w := \text{rhs} \) is dead and can be eliminated

– Dead = does not contribute to the program’s result

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

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

• Each local optimization does little by itself

• Typically optimizations interact
  – Performing one optimization enables another

• Optimizing compilers repeat optimizations until no improvement is possible
  – The optimizer can also be stopped at any point to limit compilation time
Local Optimization

- Initial code:

```plaintext
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```
• Algebraic optimization:

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

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := c \times c \\
e & := b \ll 1 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
• Copy propagation:

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := c \times c \\
e & := b \ll 1 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
• **Copy propagation:**

\[
a := x \times x \\
b := 3 \\
c := x \\
d := x \times x \\
e := 3 \ll 1 \\
f := a + d \\
g := e \times f
\]
• **Constant folding:**

\[
\begin{align*}
a & := x \times x \\
b & := 3 \\
c & := x \\
d & := x \times x \\
e & := 3 \ll 1 \\
f & := a + d \\
g & := e \times f
\end{align*}
\]
• Constant folding:

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

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

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

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

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

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

  \[ f := a + a \]
  \[ g := 6 \times f \]

• Many compilers find this (though we can optimize even further)
• Dead code elimination:
  
  \[ a := x \times x \]

  \[ f := 2 \times a \]
  \[ g := 12 \times a \]

• This is the final form
• Dead code elimination:
  \[ a := x \times x \]

  \[ f := 2 \times a \]
  \[ g := 12 \times a \]

• This is the final form
Which of the following are valid local optimizations for the given basic block? Assume that only $g$ and $x$ are referenced outside of this basic block.

- Copy propagation: Line 4 becomes $d := a \times b$.
- Common subexpression elimination: Line 5 becomes $e := d$.
- Dead code elimination: Line 3 is removed.
- After many rounds of valid optimizations, the entire block can be reduced to $g := 5$. 

```
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a := 1$</td>
</tr>
<tr>
<td>2</td>
<td>$b := 3$</td>
</tr>
<tr>
<td>3</td>
<td>$c := a + x$</td>
</tr>
<tr>
<td>4</td>
<td>$d := a \times 3$</td>
</tr>
<tr>
<td>5</td>
<td>$e := b \times 3$</td>
</tr>
<tr>
<td>6</td>
<td>$f := a + b$</td>
</tr>
<tr>
<td>7</td>
<td>$g := e - f$</td>
</tr>
</tbody>
</table>
```
Which of the following are valid local optimizations for the given basic block? Assume that only $g$ and $x$ are referenced outside of this basic block.

- Copy propagation: Line 4 becomes $d := a \times b$.
- Common subexpression elimination: Line 5 becomes $e := d$.
- Dead code elimination: Line 3 is removed.
- After many rounds of valid optimizations, the entire block can be reduced to $g := 5$. 

The code is as follows:

1. $a := 1$
2. $b := 3$
3. $c := a + x$
4. $d := a \times 3$
5. $e := b \times 3$
6. $f := a + b$
7. $g := e - f$
Loop Optimization

1. Code Motion

2. Reduction in Strength

3. Induction Variables elimination
• **Code Motion**

```
L:
  n := 2 + m
  t1 := i * 8
  t2 := A[t1]
  dp := dp + t2
  i := i + 1
  if i < n goto L
```

```
⇒
```

```
L:
  dp := 0
  i := 1
  n := 2 + m
```

```
⇒
```

```
L:
  t1 := i * 8
  t2 := A[t1]
  dp := dp + t2
  i := i + 1
  if i < n goto L
```

“n := 2 + m” can be moved out of the loop
• **Reduction in Strength**

```
L:
t1 := i * 8
t2 := A[t1]
dp := dp + t2
i := i + 1
if i < n goto L
```

⇒

```
L:
t1 := t1 + 8
t2 := A[t1]
dp := dp + t2
i := i + 1
if i < n goto L
```

- i is increased by 1
- t1 is increased by 8
- “*” can be replaced by “+”
Optimization Overview

Induction Variables Elimination

- "induction variables" are variables that in each iteration, are increased (or decreased) by a constant value c
- here, i and t1 are regarded as "induction variables"
- i can be removed and t1 is used instead
- "i := i + 1" is then a dead code and can be removed

```
L:
  t1 := t1 + 8
t2 := A[t1]
dp := dp + t2
i := i + 1
if i < n goto L
```

```
dp := 0
i := 1
n := 2 + m
t1 := 0
```

```
L:
  t1 := t1 + 8
t2 := A[t1]
dp := dp + t2
if t1 < t3 goto L
```

```
dp := 0
n := 2 + m
t3 := 8 * n
t1 := 0
```
Compilers

Peephole Optimization
- Optimizations can be directly applied to assembly code

- **Peephole optimization** is effective for improving assembly code
  - The “peephole” is a short sequence of (usually contiguous) instructions
  - The optimizer replaces the sequence with another equivalent one (but faster)
Removing Redundant Load and Stores

3AC
\[ a := b + c; \]
\[ d := a - e; \]

Machine Code

```
mov registera b
add registera c
mov a registera
mov registera a <- This load statement is not needed
sub registera e
mov d registera
```
- Use of Registers to store the most used variables because access time is much quicker than memory

- Use of specialized instructions

\[
\begin{align*}
\text{mov registera } & \text{ a} \\
\text{add registera } & \text{ 1} \\
\text{mov a registera} & \\
\Rightarrow & \quad \text{inc a}
\end{align*}
\]
Some machine codes can be deleted
   mult registera 1
   add registera 0

Using shift to left instead of multiplication by powers of 2

Using shift to right instead of division into powers of 2