Local Optimizations

Prof. Alex Aiken

(minor edits by Prof. David Dill)
Lecture Outline

• Wrap up operational semantics: Dispatch

• Optimization
  - Introduction
  - Intermediate code
  - Local optimizations
Dispatch

e_0.f(e_1, \ldots, e_n)

1. Eval args in order
2. Eval e_0

Order seems odd.
Can it affect the result?
Dispatch

\[ e_0 . f(e_1, \ldots, e_n) \]

1. Eval args in order
2. Eval \( e_0 \)
3. Let \( X \) be the dynamic type of \( e_0 \) value
4. Get definition of \( f \) from \( X \)
5. Create \( n \) new locations for args
6. Update \( E \) to map formals to new locations
7. Update \( S \) to map new locs to arg values
8. Set self to \( e_1 \) value
9. Eval body of \( f \)
Notation: $\text{Impl}(A, f) = (x_1, x_2, \ldots, x_n, e_{\text{body}})$

$x_i$ - formal parameters

$e_{\text{body}}$ - body of function
so, E, S ⊢ e₁ : V₁, S₁

... ...

so, E, S_{n-1} ⊢ eₙ : Vₙ, Sₙ

so, E, Sₙ ⊢ e₀ : V₀, Sₙ₊₁

← e₀ evaluated with all side effects of e₁ ... eₙ

so, E, S ⊢ e₀ . f(e₁, ..., eₙ) : V, S_{n+3}
\[ s_0, E, S \vdash e_1 : V_1, S_1 \]

\[ \cdots \]

\[ s_0, E, S_{n-1} \vdash e_n : V_n, S_n \]

\[ s_0, E, S_n \vdash e_0 : V_0, S_{n+1} \]

\[ V_0 = X(a_1 = l_1, \ldots, a_m = l_m) \]

---

\[ s_0, E, S \vdash e_0 \cdot f(e_1, \ldots, e_n) : V, S_{n+3}. \]
\[ s_0, E, S \vdash e_1 : V_1, S_1 \]

\[ \cdots \]

\[ s_0, E, S_{n-1} \vdash e_n : V_n, S_n \]

\[ s_0, E, S_n \vdash e_0 : V_0, S_{n+1} \]

\[ V_0 = X(a_1 = e_1, \ldots, a_m = e_m) \]

\[ \text{impl} (X, f) = (x_1, x_2, \ldots, x_n) \in \text{body} \]

\[ \uparrow \text{formals \\ body of } f \]

\[ s_0, E, S \vdash e_0 \cdot f(e_1, \ldots, e_n) : V, S_{n+3} \]
$s_0, E, S \vdash e_1 : V_1, S_1$

\[ \cdots \]

$so, E, S_{n-1} \vdash e_n : V_n, S_n$

$s_0, E, S_n \vdash e_0 : V_0, S_{n+1}$

$V_0 = \mathit{X}(a_1 = l_1, \ldots, a_m = l_m)$

$\mathit{impl}(X, f) = (x_1, x_2, \ldots, x_n, e_{\mathit{body}})$

$\forall x_i = \mathit{newloc}(S_{n+1})$ for $i = 1, \ldots, n$

\[ \uparrow \text{new locations for arguments} \]

\[ so, E, S \vdash e_0 \cdot f(e_1, \ldots, e_n) : V, S_{n+3} \]
\[ s_0, E, S \vdash e_1 : V_1, S_1 \]

\[ \cdots \]

\[ s_0, E, S_{n-1} \vdash e_n : V_n, S_n \]

\[ s_0, E, S_n \vdash e_0 : V_0, S_{n+1} \]

\[ V_0 = \chi(a_1 = l_1, \ldots, a_m = l_m) \]

\[ \text{impl}(X, f) = (x_1, x_2, \ldots, x_n, e_{\text{body}}) \]

\[ \forall x_i = \text{newloc}(S_{n+1}) \text{ for } i = 1, \ldots, n \]

\[ E' = [a_1 : l_1, \ldots, a_m : l_m][x_1 / l_{x_1}, \ldots, x_n / l_{x_n}] \]

\[ \text{start with } X \text{ attributes only} \]

\[ \text{then bind forms to new locations} \]

\[ s_0, E, S \vdash e_0 \cdot f(e_1, \ldots, e_n) : V, S_{n+3} \]
\[ s_0, E, S \vdash e_1 : V_1, S_1 \]
\[ \quad \ldots \]
\[ s_0, E, S_{n-1} \vdash e_n : V_n, S_n \]
\[ s_0, E, S_n \vdash e_0 : V_0, S_{n+1} \]
\[ V_0 = X(a_1 = l_1, \ldots, a_m = l_m) \]
\[ \text{impl}(X, f) = (x_1, x_2, \ldots, x_n, e_{\text{body}}) \]
\[ l_{x_i} = \text{newloc}(S_{n+1}) \quad \text{for } i = 1, \ldots, n \]
\[ E' = [a_1 : l_1, \ldots, a_m : l_m][x_1/l_{x_1}, \ldots, x_n/l_{x_n}] \]
\[ S_{n+2} = S_{n+1}[V_1/l_{x_1}, \ldots, V_n/l_{x_n}] \]
\[ \uparrow \text{assign actuals to formals} \]

\[ s_0, E, S \vdash e_0 \cdot f(e_1, \ldots, e_n) : V_0, S_{n+3} \]
12

so, E, S ⊢ e₁ : v₁, S₁

... 

so, E, S_{n-1} ⊢ eₙ : vₙ, Sₙ

so, E, Sₙ ⊢ e₀ : v₀, S_{n+1}

v₀ = \mathcal{X}(a₁ = l₁, \ldots, aₘ = lₘ)

\text{impl}(x_j f) = (x_1, x_2, \ldots, x_n, e_{\text{body}})

\text{lxi} = \text{newloc}(S_{n+1}) \text{ for } i = 1, \ldots, n

E' = [a₁ : l₁, \ldots, aₘ : lₘ][\text{x₁}/\text{lxi}, \ldots, \text{xₙ}/\text{l₂n}]

S_{n+2} = S_{n+1}[v₁/\text{lxi}, \ldots, vₙ/\text{l₂n}]

v₀, E' ⊢ e_{\text{body}} : v₂, S_{n+3}

so, E, S ⊢ e₀ \cdot f(e₁, \ldots, eₙ) : v₂, S_{n+3}
Optimization
OPTIMIZATION
Optimization Overview

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

• Optimization should not alter what the program computes
  - The answer must still be the same
Optimization

• Optimization is our last compiler phase

• Most complexity in modern compilers is in the optimizer
  - Also by far the largest phase

• First, we need to discuss intermediate languages
INTERMEDIATE LANGUAGES
Why Intermediate Languages?

• 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 retargeting
  - On an intermediate language
    • Pro: Machine independent
    • Pro: Exposes optimization opportunities
Intermediate Languages

• 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
Three-Address Intermediate Code

- Each instruction is of the form
  \[ x := y \text{ op } z \]
  \[ x := \text{ op } y \]
  - \( y \) and \( z \) are registers or constants
  - Common form of intermediate code
- The expression \( x + y \ast z \) is translated
  \[ t_1 := y \ast z \]
  \[ t_2 := x + t_1 \]
  - Each subexpression has a “name”
Generating Intermediate Code

- Similar to assembly code generation
- But use any number of IL registers to hold intermediate results
Generating Intermediate Code (Cont.)

- $\text{igen}(e, t)$ 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
  $$\Rightarrow$$ simple code generation
An Intermediate Language

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

- id’s are register names
- Constants can replace id’s
- Typical operators: +, -, *
Definition. 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)
  - A basic block is a single-entry, single-exit, straight-line code segment
Basic Block Example

• 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 \( w := 3 \times x \)
Definition. Control-Flow Graphs

- 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 `jump L_B`
    - E.g., execution can fall-through from block A to block B
Example of Control-Flow Graphs

- 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
LOCAL OPTIMIZATIONS
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), few 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 compilation time.
  - Some optimizations have low benefit.
  - Many fancy optimizations are all three.

• Goal: Maximum benefit for minimum 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
Algebraic Simplification

• Some statements can be deleted
  \[ x := x + 0 \]
  \[ x := x * 1 \]

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

• Operations on constants can be computed at compile time
  - If there is a statement $x := y \text{ op } z$
  - And $y$ and $z$ are constants
  - Then $y \text{ op } z$ can be computed at compile time

• Example: $x := 2 + 2 \Rightarrow x := 4$
• Example: if $2 < 0$ jump L can be deleted
• Can do many of these on the AST.
Flow of Control Optimizations

• 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

• 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

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

  (\(b\) is a fresh register)

  - More complicated in general, due to loops
Common Subexpression Elimination

• 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:

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

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

(the values of \(x\), \(y\), and \(z\) do not change in the ... code)
Copy Propagation

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

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

• Only useful for enabling other optimizations
  - Constant folding
  - Dead code elimination
Copy Propagation and Constant Folding

- Example:

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

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

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

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

• *Algebraic optimization:*
  
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
An Example

- Copy propagation:
  
  a := x * x
  b := 3
  c := x
  d := c * c
  e := b << 1
  f := a + d
  g := e * f
An Example

• Copy propagation:
  a := x * x
  b := 3
  c := x
  d := x * x
  e := 3 << 1
  f := a + d
  g := e * f
An Example

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

- **Constant folding:**
  - \( a := x \times x \)
  - \( b := 3 \)
  - \( c := x \)
  - \( d := x \times x \)
  - \( e := 6 \)
  - \( f := a + d \)
  - \( g := e \times f \)
An Example

- *Common subexpression elimination:*

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

- Copy propagation:
  
  \[
  \begin{align*}
  a & := x \times x \\
  b & := 3 \\
  c & := x \\
  d & := a \\
  e & := 6 \\
  f & := a + a \\
  g & := 6 \times f
  \end{align*}
  \]
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
Peephole Optimizations on Assembly Code

• These optimizations work on intermediate code
  - Target independent
  - But they can be applied on assembly language also

• 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)
Peephole Optimizations (Cont.)

• Write peephole optimizations as replacement rules
  \[ i_1, \ldots, i_n \rightarrow j_1, \ldots, j_m \]
  where the rhs is the improved version of the lhs

• Example:
  \[
  \text{move $a$ $b, move $b$ $a \rightarrow move $a$ $b}
  \]
  Works if move $b$ $a$ is not the target of a jump

• Another example
  \[
  \text{addiu $a$ $a i, addiu $a$ $a j \rightarrow addiu $a$ $a i+j}
  \]
Peephole Optimizations (Cont.)

• Many (but not all) of the basic block optimizations can be cast as peephole optimizations
  - Example: `addiu $a $b 0` → `move $a $b`
  - Example: `move $a $a` →
  - These two together eliminate `addiu $a $a 0`

• As for local optimizations, peephole optimizations must be applied repeatedly for maximum effect
Local Optimizations: Notes

• Intermediate code is helpful for many optimizations

• Many simple optimizations can still be applied on assembly language

• “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is not optimal in any reasonable sense
  - “Program improvement” is a more appropriate term

• Next time: global optimizations