Other Optimizations

March 7, 2001
CS 132: Compiler Design

Constant Folding

• Compile-time evaluation of expressions with constant values.

\[
\begin{align*}
\text{Const 1:} & \quad x \leftarrow 3 + 5 \quad \rightarrow \quad x \leftarrow 8 \\
\text{Const 2:} & \quad x \leftarrow \sqrt{4.0} \quad \rightarrow \quad x \leftarrow 2.0
\end{align*}
\]

• Results:
  - Eliminates run-time operations

Constant Folding

• Warnings:
  - Need to make sure that the transformation preserves meaning
  - If computation results in error, need to generate code that has the error
    • The code \( x \leftarrow 3/0 \) shouldn't cause compiler to terminate with a divide-by-zero error.
    • Need to detect overflow errors if in the language spec.
  - Must be accurate
    • Compiler's floating-point arithmetic must match the processor's floating-point arithmetic.

Constant Propagation

• Replaces uses of a variable whose value is known with the value itself.

\[
\begin{align*}
\text{Prop 1:} & \quad i \leftarrow 8 \\
\text{Prop 2:} & \quad y \leftarrow (x + i) \quad \rightarrow \quad y \leftarrow (x + 8)
\end{align*}
\]

• Results:
  - Moves constants to where they are used.
  - May eliminate variables (fewer registers required)
  - Permits better code on RISC machines
Copy Propagation

- Attempts to eliminate assignments of the form $x \leftarrow y$.

\[
\begin{align*}
x & \leftarrow y \\
z & \leftarrow 2^x \\
w & \leftarrow 2^y
\end{align*}
\]

- Results:
  - May eliminate variables, run-time moves
  - May expose opportunities for other optimizations

Dead Code Elimination

- Delete code whose results are never used

\[
\begin{align*}
x & \leftarrow y \\
z & \leftarrow 2^x \\
w & \leftarrow 2^y
\end{align*}
\]

\[
\begin{align*}
x & \leftarrow 2^y \\
z & \leftarrow 2^x \\
w & \leftarrow 2^y
\end{align*}
\]

(assuming $x$ not otherwise used)

- Results:
  - Shrinks code size, may reduce instruction count

Dead Code Elimination

- Warnings:
  - Must be sure that instruction being eliminated will either never be executed, or has no side-effects
  - e.g., generally can't delete function calls even if their return value never used.
  - Side-effects may include setting condition codes

Inlining

- Replace calls to procedures with copies of the body.

\[
\begin{align*}
\text{let} & \text{ fun } f(x,y) = \\
& \text{...x...y...} \\
in & f(3,17) \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{let} & \text{ fun } f(x,y) = \\
& \text{...x...y...} \\
in & \text{...3...17...} \\
\text{end}
\end{align*}
\]
Inlining

- Results:
  - Avoids overhead of function call
  - May expose many more optimizations in function body since the arguments are now known
- Warnings:
  - Potential for code-blowup (and i-cache misses)
    - Automatic inlining usually based on heuristics involving procedure size, number of calls to procedure, frequency of procedure calls, constancy of arguments, ...
  - May need to rename local variables to avoid name clashes

Specialization

- Generate code for a procedure specialized for certain inputs.

```
let
  fun f(x,y) =
    ...
  in
    f(3,z1);
    f(3,z2)
end
```

```
let
  fun f(x,y) =
    ...
  fun f3(y) =
    ...
  in
    f3(z1);
    f3(z2)
end
```

- Results:
  - Better optimization of procedure's body
  - Fewer arguments passed at run-time
- Warnings:
  - Like inlining, subject to code blow-up

Common Subexpression Elimination (CSE)

- Avoids repeating computations known to have been already computed.

- Results
  - Removes computations from code
- Warnings
  - Cannot avoid repeating computations that might have important side-effects (exceptions, assignments, condition codes)
**Common Subexpression Elimination (CSE)**

- $z \leftarrow a + 1$
- $x > 3$
- $a \leftarrow x \ast y$
- $y < 5$
- $z < 7$

**Loop Invariant Removal**

- Find computations that produce the same value on every occurrence of a loop, and move them out of the loop.

```java
for (i=0; i<100; i++) {
    for (j=0; j<100; j++) {
        w = i*(i+2);
        a[i][j] = 100*i + 10*w + j;
    }
}
```

**Code Motion**

- Many optimizations may change the order in which computations occur
  - For example, partial redundancy elimination
- Important to know that this doesn't change the code!
  - Swapped statements cannot both have visible side-effects
  - Swapped statements must be known not to interfere

```java
val n1 = x+y+1
val n2 = x/y
```

**Induction Variables**

- A variable $i$ is a basic induction variable in a loop if the only definitions of $i$ in the loop are of the form $i \leftarrow i+c$ or $i \leftarrow i-c$ where $c$ is loop-invariant.

- A derived induction variable is a variable whose value is a linear function of an induction variable
  - e.g., $j \leftarrow a \ast i + b$. 

---

**Entry**

- B1

**Exit**

- B2

**Common Subexpression Elimination**

- $z \leftarrow a + 1$
- $x > 3$
- $a \leftarrow x \ast y$
- $y < 5$
- $z < 7$

**Loop Invariant Removal**

- Find computations that produce the same value on every occurrence of a loop, and move them out of the loop.

```java
for (i=0; i<100; i++) {
    for (j=0; j<100; j++) {
        w = i*(i+2);
        a[i][j] = 100*i + 10*w + j;
    }
}
```

**Code Motion**

- Many optimizations may change the order in which computations occur
  - For example, partial redundancy elimination
- Important to know that this doesn't change the code!
  - Swapped statements cannot both have visible side-effects
  - Swapped statements must be known not to interfere

```java
val n1 = x+y+1
val n2 = x/y
```

**Induction Variables**

- A variable $i$ is a basic induction variable in a loop if the only definitions of $i$ in the loop are of the form $i \leftarrow i+c$ or $i \leftarrow i-c$ where $c$ is loop-invariant.

- A derived induction variable is a variable whose value is a linear function of an induction variable
  - e.g., $j \leftarrow a \ast i + b$. 

---

**Entry**

- B1

**Exit**

- B2

**Common Subexpression Elimination**

- $z \leftarrow a + 1$
- $x > 3$
- $a \leftarrow x \ast y$
- $y < 5$
- $z < 7$

**Loop Invariant Removal**

- Find computations that produce the same value on every occurrence of a loop, and move them out of the loop.

```java
for (i=0; i<100; i++) {
    for (j=0; j<100; j++) {
        w = i*(i+2);
        a[i][j] = 100*i + 10*w + j;
    }
}
```

**Code Motion**

- Many optimizations may change the order in which computations occur
  - For example, partial redundancy elimination
- Important to know that this doesn't change the code!
  - Swapped statements cannot both have visible side-effects
  - Swapped statements must be known not to interfere

```java
val n1 = x+y+1
val n2 = x/y
```

**Induction Variables**

- A variable $i$ is a basic induction variable in a loop if the only definitions of $i$ in the loop are of the form $i \leftarrow i+c$ or $i \leftarrow i-c$ where $c$ is loop-invariant.

- A derived induction variable is a variable whose value is a linear function of an induction variable
  - e.g., $j \leftarrow a \ast i + b$. 

---

**Entry**

- B1

**Exit**

- B2
**Strength Reduction**

- Replace more expensive operations with cheaper operations.
  - e.g., multiplication by addition

```c
i ← 0
L1: j ← 4*i
k ← a+j
x ← *k
sum ← sum+x
i ← i+1
if i<10 goto L1
```

**Induction Variable Elimination**

- Minimize instances of linearly dependent variables

```c
i ← 0
j ← -4
L1: j ← j+4
k ← a+j
x ← *k
sum ← sum+x
i ← i+1
if i<10 goto L1
```

(assuming i, j, k aren’t used afterwards)

**Loop Unrolling**

- Duplicate bodies of loops

```c
L1: x ← A[i]
s ← s+x
i ← i+4
if i<n goto L1
```

**Loop Unrolling**

- Duplicate bodies of loops

```c
if i >=n-8 goto L2
L1: x ← A[i]
s ← s+x
i ← i+4
if i<n goto L1
L2:
```
Loop Unrolling

• Results
  - Fewer loop termination tests
  - Fewer jumps executed
  - More potential parallelism (ILP)

• Warnings:
  - Increases code size

• Usually, loops are unrolled 2--4 times, occasionally up to 8.

Hoisting

• Find computations guaranteed to be executed, and move them as early as possible.

  y ← x+1
  w ← w-1
  z ← x+1

  x ← y*z
  w > 4

  w ← w+4
  z ← w+4

  y ← x+1
  w ← w-1
  z ← t

• Results:
  - More opportunities to do CSE
    • E.g., if a conditional evaluates the same expression in both branches it may be lifted out
  - May shrink code size, but little direct effect on time

• Warnings:
  - Tends to expand live ranges of variables, increasing register pressure.

Loop Fusion

• Merge multiple loops into one loop

  for (i=0; i<n; i++) {
    b[i] = a[i] + 1.0;
  }
  for (j=0; j<n; j++) {
    c[j] = a[j] / 2.0;
  }
  for (i=0; i<n; i++) {
    b[i] = a[i] + 1.0;
    c[i] = a[i] / 2.0;
  }
Loop Fusion

• Results:
  – Decreased loop overhead
  – In this case, an opportunity for CSE.

• Warnings:
  – Interleaves the execution of loop bodies; must know there are no bad dependencies.

Ordering Optimizations

• There is no "optimal" order for optimizations.
• One optimization may generate more opportunities for other optimizations
  – Either by happenstance or by design
  – Often simplifies optimizer to not worry about creating dead code or extra variable copies.
  – If we're going to run dead code elimination later, why worry?
• Order (and number of passes) a compiler heuristic
  – See Muchnick handout for suggestions