Organization

Dates: Lecture: Monday, 14:00-15:30
       Wednesday, 8:30-10:00
Tutorials: Tuesday/Wednesday, 10:00-12:00
Kalmer: Apinis: apinis@in.tum.de
Material: slides, recording :-)
          Moodle
          Program Analysis and Transformation
          Springer, 2012

Grades: Bonus for homeworks
         written exam
Organization

**Dates:** Lecture: Monday, 14:00-15:30
Wednesday, 8:30-10:00

**Tutorials:** Tuesday/Wednesday, 10:00-12:00

**Material:** slides, recording ·
Moodle
Program Analysis and Transformation
Springer, 2012

Proposed Content:

1. Avoiding redundant computations
   → available expressions
   → constant propagation/array-bound checks
   → code motion

2. Replacing expensive with cheaper computations
   → peep hole optimization
   → inlining
   → reduction of strength
   ...

Introduction

Observation 1: Intuitive programs often are inefficient.

Example:

```c
void swap (int i, int j) {
    int t;
    if (a[i] > a[j]) {
        t = a[j];
        a[j] = a[i];
        a[i] = t;
    }
}
```
Inefficiencies:
- Addresses $a[i], a[j]$ are computed three times
- Values $a[i], a[j]$ are loaded twice

Improvement:
- Use a pointer to traverse the array $a$
- store the values of $a[i], a[j]$

```c
void swap (int *p, int *q) {
    int t, ai, aj;
    ai = *p; aj = *q;
    if (ai > aj) {
        t = aj;
        *q = ai;
        *p = t;  // t can also be
    }        // eliminated!
}
```

```c
void swap (int *p, int *q) {
    int t, ai, aj;
    ai = *p; aj = *q;
    if (ai > aj) {
        t = aj;
        *q = ai;
        *p = t;  // t can also be
    }        // eliminated!
}
```
Inefficiencies:
- Addresses a[i], a[j] are computed three times :-(
- Values a[i], a[j] are loaded twice :-(

Improvement:
- Use a pointer to traverse the array a:
- Store the values of a[i], a[j]!

Observation 3:
Program Improvements need not always be correct :-(

Example:
\[ y = \text{f()} + \text{f}(); \quad \Rightarrow \quad y = 2 \ast \text{f}(); \]

Idea: Save second evaluation of f() ...

Consequences:

\[ \Rightarrow \quad \text{Optimizations have assumptions.}\]
\[ \Rightarrow \quad \text{The assumption must be:} \]
  - formalized,
  - checked :-(
\[ \Rightarrow \quad \text{It must be proven that the optimization is correct, i.e., preserves the semantics !!!} \]

Observation 4:
Optimization techniques depend on the programming language:

\[ \rightarrow \quad \text{which inefficiencies occur;} \]
\[ \rightarrow \quad \text{how analyzable programs are;} \]
\[ \rightarrow \quad \text{how difficult/impossible it is to prove correctness ...} \]

Example: Java
Observation 3:
Programm-Improvements need not always be correct :-(

Example:
\[ y = f() + f(); \quad \rightarrow \quad y = 2 \ast f(); \]

Idea: Save the second evaluation of \( f() \) ???

Problem: The second evaluation may return a result different from the first; (e.g., because \( f() \) reads from the input :-)

\begin{verbatim}
void swap (int *p, int *q) {
    int t, ai, aj;
    ai = *p; aj = *q;
    t = aj;
    *q = ai;
    *p = t;  // t can also be
    }  // eliminated!
\end{verbatim}

Correctness proofs:
+ more or less well-defined semantics;
- features, features, features;
- libraries with changing behavior ...

Correctness proofs:
+ more or less well-defined semantics;
- features, features, features;
- libraries with changing behavior ...
... in this course:

A simple imperative programming language with:

- `variables` // registers
- `R = c` // assignments
- `R = M[A]` // loads
- `M[A] = c` // stores
- `if (c) s_1` else `s_2` // conditional branching
- `goto L;` // no loops

Note:

- For the beginning, we omit procedures :-)  
- External procedures are taken into account through a statement `f()` for an unknown procedure `f`.

    intra-procedural

    kind of an intermediate language in which (almost) everything can be translated.

Example: `swap()`

Optimization 1: `1 * R` \(\longrightarrow\) `R`

Optimization 2: Reuse of subexpressions

\[
\begin{align*}
A_1 &= A_0 + 1 \ast i; & A_0 &= \&a
\end{align*}
\]

\[
\begin{align*}
R_1 &= M[A_1]; & R_1 &= a[i]
\end{align*}
\]

\[
\begin{align*}
A_2 &= A_0 + 1 \ast j;
\end{align*}
\]

\[
\begin{align*}
R_2 &= M[A_2]; & R_2 &= a[j]
\end{align*}
\]

\[
\begin{align*}
\text{if}(R_1 > R_2) \\
A_3 &= A_0 + 1 \ast j;
\end{align*}
\]

\[
\begin{align*}
t &= M[A_3];
\end{align*}
\]

\[
\begin{align*}
A_4 &= A_0 + 1 \ast j;
\end{align*}
\]

\[
\begin{align*}
A_5 &= A_0 + 1 \ast i;
\end{align*}
\]

\[
\begin{align*}
R_3 &= M[A_5];
\end{align*}
\]

\[
\begin{align*}
M[A_1] &= R_4;
\end{align*}
\]

\[
\begin{align*}
A_6 &= A_0 + 1 \ast i;
\end{align*}
\]

\[
\begin{align*}
M[A_6] &= t;
\end{align*}
\]

\[
\begin{align*}
R_1 &= R_3
\end{align*}
\]
0:  \[ A_1 = A_0 + 1 \times i \]  // \[ A_0 \rightarrow &a \]
1:  \[ R_1 = M[A_1]; \]  // \[ R_1 \rightarrow a[i] \]
2:  \[ A_2 = A_0 + 1 \times j; \]
3:  \[ R_2 = M[A_2]; \]  // \[ R_2 \rightarrow a[j] \]
4:  if \( R_1 > R_2 \) {
5:    \[ A_3 = A_0 + 1 \times j; \]
6:    \[ t = M[A_3]; \]
7:    \[ A_4 = A_0 + 1 \times j; \]
8:    \[ A_5 = A_0 + 1 \times i; \]
9:    \[ R_3 = M[A_5]; \]
10:  \[ M[A_4] = R_3; \]
11:  \[ A_6 = A_0 + 1 \times i; \]
12:  \[ M[A_6] = t; \]
}
Inefficiencies:
- Addresses \( a[i], a[j] \) are computed three times :-(
- Values \( a[i], a[j] \) are loaded twice :-(

Improvement:
- Use a pointer to traverse the array \( a \):
- store the values of \( a[i], a[j] \)!

Optimization 3: Contraction of chains of assignments :-)

Gain:

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>store</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Inefficiencies:
- Addresses \( a[i], a[j] \) are computed three times :-(
- Values \( a[i], a[j] \) are loaded twice :-(

Improvement:
- Use a pointer to traverse the array \( a \):
- store the values of \( a[i], a[j] \)!

Optimization 3: Contraction of chains of assignments :-)

Gain:

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>store</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>
1 Removing superfluous computations

1.1 Repeated computations

Idea:

If the same value is computed repeatedly, then
→ store it after the first computation;
→ replace every further computation through a look-up!

⇒ Availability of expressions
⇒ Memoization

Note:

B is a repeated computation of the value of \( y + z \) if:
1. A is always executed before B; and
2. y and z at B have the same values as at A  :-)  

⇒ We need:
→ an operational semantics  :-)  
→ a method which identifies at least some repeated computations ...

Problem: Identify repeated computations!

Example:

\[
\begin{align*}
\text{A:} & \quad x_1 = y + z; \\
\text{B:} & \quad x_2 = y + z; \\
\end{align*}
\]

Background 1: An Operational Semantics

we choose a small-step operational approach.
Programs are represented as control-flow graphs.
In the example:
Background 1: An Operational Semantics

we choose a small-step operational approach.
Programs are represented as control-flow graphs.
In the example:

\[
\begin{align*}
A_1 &= A_0 + 1 \times i; \\
B_1 &= M[A_1]; \\
A_2 &= A_0 + 1 \times j; \\
B_2 &= M[A_2]; \\
\text{Neg}(R_1 > R_2) &\quad \text{Pos}(R_1 > R_2) \\
A_3 &= A_0 + 1 \times f;
\end{align*}
\]

Thereby, represent:

<table>
<thead>
<tr>
<th>vertex</th>
<th>program point</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>program start</td>
</tr>
<tr>
<td>stop</td>
<td>program exit</td>
</tr>
<tr>
<td>edge</td>
<td>step of computation</td>
</tr>
</tbody>
</table>

Edge Labelings:

Test: Pos(e) or Neg(e)
Assignment: \( R = e; \)
Load: \( R = M[e]; \)
Store: \( M[e_1] = e_2; \)
Nop: ;

Thereby, represent:

Computations follow paths.
Computations transform the current state
\[ s = (\rho, \mu) \]
where:

| \( \rho \) : Vars \rightarrow int | contents of registers |
| \( \mu \) : N \rightarrow int | contents of storage |

Every edge \( k = (u, lab, v) \) defines a partial transformation
\[ [k] = [lab] \]

of the state:
\[ \llbracket \cdot \rrbracket (\rho, \mu) = (\rho, \mu) \]
\[ \llbracket \text{Pos} (e) \rrbracket (\rho, \mu) = (\rho, \mu) \quad \text{if } [e]_{\rho} \neq 0 \]
\[ \llbracket \text{Neg} (e) \rrbracket (\rho, \mu) = (\rho, \mu) \quad \text{if } [e]_{\rho} = 0 \]

// [e] : evaluation of the expression e, e.g.
// \[ x + y \{ x \mapsto 7, y \mapsto -1 \} = 6 \]
// \[ [x == 4] \{ x \mapsto 5 \} = 1 \]

\[ \llbracket \cdot \rrbracket (\rho, \mu) = (\rho, \mu) \]
\[ \llbracket \text{Pos} (e) \rrbracket (\rho, \mu) = (\rho, \mu) \quad \text{if } [e]_{\rho} \neq 0 \]
\[ \llbracket \text{Neg} (e) \rrbracket (\rho, \mu) = (\rho, \mu) \quad \text{if } [e]_{\rho} = 0 \]

// [e] : evaluation of the expression e, e.g.
// \[ x + y \{ x \mapsto 7, y \mapsto -1 \} = 6 \]
// \[ [x == 4] \{ x \mapsto 5 \} = 1 \]

\[ R = M[e]; \llbracket R \rrbracket (\rho, \mu) = (\rho \oplus \{ R \mapsto \mu([e]_{\rho}) \}, \mu) \]
\[ M[e_1] = e_2; \llbracket R \rrbracket (\rho, \mu) = (\rho, \mu \oplus \{ e_1 \mapsto [e_2]_{\rho} \}) \]

Example:
\[ [x = x + 1; \{ x \mapsto 5 \}, \mu) = (\rho, \mu) \quad \text{where:} \]
\[ \rho = \{ x \mapsto 5 \} \oplus \{ x \mapsto \{ x + 1 \{ x \mapsto 5 \} \} \}
\[ = \{ x \mapsto 5 \} \oplus \{ x \mapsto 6 \}
\[ = \{ x \mapsto 6 \} \]