Assignment 2
Optimizations with SSA

Remi Meier
remi.meier@inf.ethz.ch
Assignment 2

Two tasks:

1. Apply the following optimizations
   a) Constant Folding (CF)
   b) Copy Propagation (CP)
   c) Common Subexpression Elimination (CSE)

→ cd.cfg.Optimizer

2. Find uninitialized variables
Constant Folding (CF)
Constant Folding

• Evaluation of expressions at **compile time**

• Operations with **constant** operands

  \[
  i_1 = 3 + 2 \\
  j_2 = i_1 + 40 \times 2 \\
  k_1 = \text{true} || \text{false}
  \]

  \[
  i_1 = 5 \\
  j_2 = i_1 + 80 \\
  k_1 = \text{true}
  \]

• Optional: Expression simplification & strength reduction

  1: \[i_1 = i_0 \times 1\]  \[\rightarrow\]  \[i_1 = i_0\]

  2: \[k_1 = i_0 \times 0\]  \[\rightarrow\]  \[k_1 = 0\]

  3: \[j_2 = i_1 - i_1\]  \[\rightarrow\]  \[j_2 = 0\]

  4: \[k_2 = 2 \times i_0\]  \[\rightarrow\]  \[k_2 = i_0 + i_0\]

• Optional: Operations on floats
Framework

• Work directly on the AST

→ cd.ir.AstRewriteVisitor
Copy Propagation (CP)
Copy Propagation

- Given a **definition** of a variable \( x = \text{RHS} \)
- Replace later **uses** of \( x \) with **RHS**
  - No other definitions of \( x \) in-between uses
  - **SSA**: only one definition per variable version
- **RHS** must be a
  - Constant
  - Local variable
  - A phi-node with all the same operands

\[
x = 5 \\
\ldots \\
\text{write}(x)
\]
Copy Propagation

\[ x_0 = 5 \]

\[ x_1 = y_1 \]

\[ x_2 = \text{phi}(x_0, x_1) \]
Copy Propagation

A \[ x_0 = 5 \]

B \[ \text{write}(x_0) \]

C \[ x_1 = y_1 \]

D \[ x_2 = \phi(x_0, x_1) \]

replace later uses
Copy Propagation

A \[ x_0 = 5 \]

B \[
\text{write}(x_0)\]

C \[ x_1 = y_1 \]

D \[ x_2 = \text{phi}(x_0, x_1) \]

optionally delete
useless definitions
Copy Propagation

A
\[ x_0 = 5 \]

B
\text{write}(5)

C
\[ x_1 = y_1 \]

D
\[ x_2 = \text{phi}(5, y_1) \]
Common Subexpression Elimination (CSE)
Common Subexpression Elimination

- Look for multiple occurrences of the \textit{same} expression
- \textbf{Evaluate} the expression \textit{once}
- \textbf{Replace} other uses of the expression with the \textit{cached result}

Locally (inside a basic block):

\[
\begin{align*}
i_1 &= a_1 + b_1 \\
j_1 &= a_1 + b_1
\end{align*}
\]
\[
\begin{align*}
\text{tmp}_1 &= a_1 + b_1 \\
i_1 &= \text{tmp}_1 \\
j_1 &= \text{tmp}_1
\end{align*}
\]
Common Subexpression Elimination

**Problem:** AST Equivalence

\[
\begin{align*}
\text{a + b} & \quad \text{b + a} \\
\text{a} & \quad \text{b} \\
\text{b} & \quad \text{a}
\end{align*}
\]

Commutativity
Common Subexpression Elimination

**Problem:** AST Equivalence

\[
\begin{align*}
&\text{a + (b + c)} \\
&\quad \downarrow \quad \downarrow \\
&\quad \text{a} \quad \text{b} \\
&\quad \downarrow \quad \downarrow \\
&\quad \text{b} \quad \text{c}
\end{align*}
\]

\[
\begin{align*}
&(\text{a + b}) + \text{c} \\
&\quad \downarrow \quad \\
&\quad \text{a} \quad \text{b}
\end{align*}
\]

\[
\begin{align*}
&\text{a + (b + c)} = ?
\end{align*}
\]

Associativity
Common Subexpression Elimination

- Map each expression to a **key**:
  - **Canonical form** of the expression
  - Using a string for the key is fine

```
a + b
```
```
b + a
```
```
a + c
```

\[
\begin{align*}
a + b & \rightarrow "a+b"
\end{align*}
\]

\[
\begin{align*}
b + a & \rightarrow "a+b"
\end{align*}
\]

\[
\begin{align*}
a + c & \rightarrow "a+c"
\end{align*}
\]
Common Subexpression Elimination

• Full points for detecting commutativity:
  \[ a + b = b + a \]

• Not necessary to handle associativity:
  \[ (a + b) + c = a + (b + c) \]

• Think about how larger expressions are handled:
  \[ a + (b - c) * (-d) \]
  \[ \{ \begin{align*}
    & a + (b - c) * (-d) \\
    & (b - c) * (-d) \\
    & b - c \\
    & -d
  \end{align*} \]
Common Subexpression Elimination

- Eliminate expressions locally (same basic block) and *globally* (same method)
  - How to **discover** identical expressions globally?
  - Where to **insert** the temporary variable?

```
A
  tmp₁ = b₁ + c₂

B
  b₁ + c₂

C
  tmp₂ = a₁ + d₂

D
  b₁ + c₂  a₁ + d₂

E

CSE

A
  tmp₁ = b₁ + c₂

B
  tmp₁

C
  tmp₂ = a₁ + d₂

D
  tmp₁  a₁ + d₂

E
  tmp₂
```
Common Subexpression Elimination

- Use the **dominator tree** (DT):

  ![Dominator Tree Diagram]

If an `expr` dominates another use of the same `expr`, later use can be replaced by cached result.
Common Subexpression Elimination

• Example:

Expression can be cached
Common Subexpression Elimination

• Example:

```
A
  ┌───────┐
  │       │
  │       │
  │       │
  └───────┘

B  b₁ + c₂
  └───┐
       │
       │
       D

C  b₁ + c₂
  └───┐
       │
       │
       │

None can be eliminated with this approach!
```
Big Picture

- **Individual** optimizations are not powerful
- **Combination** and **repeated application** may trigger additional opportunities
Uninitialized Variables
Uninitialized Variables

• Detect potentially uninitialized variables
  • Generate a semantic error
  • Be conservative and try to be precise

```c
int x;
if (...) {
    x = 3
}
y = x  ← possibly uninitialized use of x
```
→ cd.cfg.SSA
Uninitialized Variables

- For local variables, insert definitions like
  \[ x_0 = \text{UNINIT} \]
- Propagate \text{UNINIT} through phi-nodes
  \[ x_2 = \text{phi}(x_0, x_1) = \{\text{UNINIT}\} \]
- Throw error for uses of variables that may be \text{UNINIT}

\[ x_0 = \text{UNINIT} \]
\[ \text{if} (...) \{ \]
\[ \quad x_1 = 3 \]
\[ \} \]
\[ x_2 = \text{phi}(x_0, x_1) \]
\[ y_1 = x_2 \text{ ← possible error} \]
DeSSA Problems with Optimizations
SSA & Optimizations

$A$

$x_1 = \phi(x_0, x_2)$
$y_1 = \phi(y_0, y_2)$

$B$

$t = x_1$
$x_2 = y_1$
$y_2 = t$

swaps $x, y$

$A$

$x_1 = \phi(x_0, y_1)$
$y_1 = \phi(y_0, x_1)$

$B$

$t = x_1$
$x_2 = y_1$
$y_2 = x_1$
DeSSA: Resolve phi() with assignments

\[ x_1 = \text{phi}(x_0, y_1) \]
\[ y_1 = \text{phi}(y_0, x_1) \]

DeSSA: no longer swaps!

\[ x_1 = x_0 \]
\[ y_1 = y_0 \]
**SSA & Optimizations**

**Fixed DeSSA**

\[ x_1 = \phi(x_0, y_1) \]
\[ y_1 = \phi(y_0, x_1) \]

**Done by the provided framework. Use it!**
Summary

• Assignment 2:
  • Constant Folding, Constant Propagation, Common Subexpression Elimination
  • Uninitialized Variables

• Use our framework!

• You may ignore floats, fields, and array items
  • only local variables
  • only scalar values