5.0 Optimization

Arg. from 3.0...inserts

Is this optimal?

if (...) {
    if (...) {
        a = 0;
        b = a * f;
    }
    else {
        a = 1;
    }
    continue;
}

then

a1 = 0

a2 = 1

a3 = φ(a0, a2)

b1 = a3 * f

continue:
5.1 Strict SSA

\( \Phi \)-nodes are inserted only into blocks where a variable is live.

Various strategies are used by compilers.

1. Know liveness of variables

   - Separate data flow analysis — poor choice
     include liveness information as the compiler finds versions

   - Variable on RHS of a statement:
     find version for variables
     mark assigned (could be a \( \Phi \) node)
     that generated version
Know if a variable is (possibly) in use in any other basic block.

- won't catch example on slide!
- Catch variables that are used only in a single block

Separate pass over CFG to remove unnecessary \( \phi \) nodes (based on live-range/usage information)
5.2 de SSA

mapping SSA → Code

key idea: ined copies

\[ a_1 = \ldots \]
\[ a_3 = a_1 \]

\[ a_2 = \ldots \]
\[ a_3 = a_2 \]

\[ a_3 = \phi(a_1, a_2) \]

Note: if you need an extension to assignment 1: mail to TA

done later with the copies
- copy propagation
- register allocation
Warning: don't map to IR without SSA by dropping the version number...

```c
while (...) {
    x = a;
    a = 7;
    y = a + y;
}
```

*Assigned

```
a_2 = 7
```

is loop invariant — a compiler may hoist it.
modified CFG

\[ a_2 = 7 \]

\[ a_3 = \phi(a_1, a_2) \]

\[ y_3 = d(y_4, y_2) \]

\[ x_1 = a_3 \]

\[ y_2 = a_2 + y_3 \]

must consider \( \phi \) nodes and versions...

copies are necessary... but what if there are conflicting copies?
Consider this setup:

\[ a_4 = \ldots \]

\[ a_5 = \phi(a_4, a_2) = a_5 \]

\[ a_2 = \]

\[ a_3 = a_5 \]

bad!

assigned \( a_5 = a_2 \) (copy)

is not necessary if there is another iteration — only an exit
5.2.1 Critical Edges

Inserting of copies into predecessor blocks causes problems if a predecessor has multiple successors.

An edge $A \rightarrow B$ such that $A$ has multiple successors and $B$ has multiple predecessors is called a critical edge.

Remedy: insert a new basic block into a critical edge.

Place copies into new block.
Splitting a critical edge causes problems if the critical edge is the back edge of a loop (especially if the loop is executed frequently).

- Many optimizations won't be effective (block contains nothing but copies)
- May need an extra branch/jump instruction
5.3 Common Subexpression elimination
inside a basic block (locale)

value numbering

for each expression (RHS)

break expr. into
expressions with
two operands

\[ x = a + b; \]
\[ y = a + b; \]
\[ x = 7; \]
\[ z = a + b; \]

with SSA

\[ x_1 = a_0 + b_0 \]
\[ y_1 = a_0 + b_0 \]
\[ x_2 = 7; \]
\[ z_1 = a_0 + b_0 \]

compute an integer

hash

hash_md5 followed by
String Compare

\[ x_1 = a_0 + b_0 \]
\[ y_1 = x_1 \]
\[ x_2 = 7; \]
\[ z_1 = x_1 \]

value number table

SSA exposes opportunities: \( a_0 + b_0 \), \( 17 \), \( x_1 \)
5.4 Constant folding

(copy propagation)

Idea: both operands are constants $\Rightarrow$ compute result at compile time

$v = 7$
$m = 6$

$x = u + v$

SSA

$v_1 = 7$
$m_3 = 6$

$x_2 = v_1 + m_3 = 13$

Easy to implement in SSA as there is a unique definition.
Optimization is beneficial because it enables other transformations/optimizations.

- Loops with constant loop bounds
  - Trip count known at compile time
    - Unroll loop completely
    - Unroll K times, eliminate branches...

- Identify array elements
  \[ A[C_{i,j}] = \ldots = A[C_{j}] \]

  \( i, j \) are constants

\( i \neq j \) / transformations

\( i = j \)
5.5 Removal of checks for non-null references

(Many) OO languages demand that compiler/runtime system check that a reference is non-null prior to invoking a method or accessing a field.

- eliminate some checks ...

Object  $O_a = \text{new Object}(\ldots)$

check $O_a \neq \text{null}$
$O_a. \text{foo}(\ldots) \leq$ eliminate check

$O_3 = \Phi (O_a, O_2)$

check $O_3 \neq \text{null}$
$O_3. \text{foo}()$  

check can be removed only if $O_1 \neq \text{null}$ and $O_2 \neq \text{null}$
5.6 Method resolution

Assumption: dynamic method resolution based on cached type of receiver object

class A {
    void foo() {
    }
}

class AX {
    void foo() {
    }
}

A p, q, r;

To = new A();
qo = new AX();
qo = fct();
To::foo();
pqo::foo();

A::foo();

no information dynamic resolution
SSA makes it easy to identify defining node (return value, assignment/copy, new operator)

→ may be able to identify type

a. No need for dynamic dispatch

Hint: "instance_of" predicate allows the program to check → SSA propagate information to place where a method is involved.
5.7 Analysis to warn users of possible errors

Warn user if the program uses an uninitialized variable.

Assume $x_0$ : pseudo assignment

use of $x_0$ implies use of a (potentially) uninitialized variable

$x_0$ in the RHS, $x_0$ in $\phi$ node (Consider issuing a warning)
Complication

\[ ? \]
\[ \frac{x}{?} = x \]
\[ ? \]
\[ \text{uninit.} \]

\[ ? \]
\[ \text{if (...) } \]
\[ \frac{x}{?} = x \]
\[ \text{(pot) uninit.} \]

\( \phi \) nodes are special (uses that deserve more analysis)

\[ \text{pseudo def } x_0 \]

\[ \text{if (...) } \]
\[ x = 1 \]
\[ \ldots \]
\[ \frac{x}{?} = x \]
\[ \text{no more \( dx \)} \]

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

\[ \text{then} \]

\[ x_1 = \ldots = x_1 \]

\[ \text{Should not trigger a warning.} \]