Static Single-Assignment Form and Dataflow Analysis
Roadmap

Last time:
- Optimization overview
  • Soundness and completeness
- Simple optimizations
  • Peephole
  • LICM

This time:
- Data structures (and data) used to determine when it is safe (i.e., sound) to perform an optimizing transformation
  • Review dominators
  • SSA form
  • Dataflow analysis
DOMINATOR REVIEW
Dominator terms

Domination (A dominates B):
– to reach block B, you must have gone through block A

Strict Domination (A strictly dominates B)
– A dominates B and A is not B

Immediate Domination (A immediately dominates B)
– A immediately dominates B if A dominates B and has no intervening dominators
Dominator Example
Dominance Frontier

Definition: For a block X, the set of nodes Y such that X dominates an immediate predecessor of Y but does not strictly dominate Y
STATIC SINGLE ASSIGNMENT FORM (SSA FORM)
Goal of SSA Form

Build an intermediate representation of the program in which each variable is assigned a value in at most 1 program point:

<table>
<thead>
<tr>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- x = 1
- z = 2
- y = 3
- x = y
- z = y
- w = z
- x = 1
- i = 0;
- while( i < 10){
  k = i + 1;
}

**Statically:** There is at most one assignment statement that assigns to k

**Dynamically:** k can be assigned to multiple times
Conversion

We make new variables to carry over the effect of the original program

\[
x = 1 \\
x = x \\
y = x
\]

\[
x_1 = 1 \\
x_2 = x_1 \\
y_1 = x_2
\]
Benefits of SSA Form

There are some obvious advantages to this format for program analysis:

- Easy to see the live range of a given variable $x$ assigned to in statement $s$:
  - The region from "$x = \ldots;$" until the last use(s) of $x$ before $x$ is redefined.
  - In SSA form, from "$x_i = \ldots;\) to all uses of $x_i$, e.g., "$\ldots = f(\ldots, x_i, \ldots);$".

- Easy to see when an assignment is useless:
  - We have "$x_i = \ldots;$" and there are no uses of $x_i$ in any expression or assignment RHS.
  - ""$x_i = \ldots;$’ is a useless assignment’
  - ""$x_i = \ldots;$’ is dead code’

In other words, some useful information is pre-computed, or at least easily recoverable from SSA form.

Warning 1: Dead code = useless assignments + unreachable code

Warning 2: There is another concept called “live variables.”

- When variable $x$ is “not live,” a convenient shorthand is “Variable $x$ is dead.”
- When $x$ is dead just after a statement $s$, that does not imply that $s$ is dead code. (E.g., suppose $s$ assigns to $y$.)
- When $s$ is a useless assignment to $x$
  - Statement $s$ is dead code (because dead = useless or unreachable).
  - $x$ is not live just after $s$ (”Variable $x$ is dead just after $s$”)
  - Because variable $x$ is dead, $s$ is a useless assignment, and thus statement $s$ is dead code.

Warning 2: There is another concept called “live variables.”

- When variable $x$ is “not live,” a convenient shorthand is “Variable $x$ is dead.”
- When $x$ is dead just after a statement $s$, that does not imply that $s$ is dead code. (E.g., suppose $s$ assigns to $y$.)
- When $s$ is a useless assignment to $x$
  - Statement $s$ is dead code (because dead = useless or unreachable).
  - $x$ is not live just after $s$ (”Variable $x$ is dead just after $s$”)
  - Because variable $x$ is dead, $s$ is a useless assignment, and thus statement $s$ is dead code.
Optimizations Where SSA Helps

Dead-Code Elimination

At “if (b < 4)”, b is only reached by “b = 2;” Therefore, the else branch is unreachable (dead), and can be removed.

```c
int a = 9;
int b = 2;
if (g < 12){
a = 1;
} else {
if (b < 4){
a = 2;
} else {
    a = 3;
}
}

b = a;
return 2;
```
Optimizations Where SSA Helps

Constant-propagation/constant-folding

```c
int a = 30;
int b = 3; // (a / 5);
int c;
c = b * 4;
if (c > 10) {
c = c - 10;
}
return c * (60 / a);
```
What About Conditionals?

Which y to use?
Phi Functions ($\phi$)

- We introduce a special symbol $\Phi$ at such points of confluence
- $\Phi$’s arguments are all the instances of variable $y$ that might be the most recently assigned variant of $y$
- Returns the “correct” one
- Do we need a $\Phi$ for $x$?
  - No!
Computing Phi-Function Placement

Intuitively, we want to figure out cases where there are multiple assignments that can reach a node. To be safe, we can place a $\Phi$ function for each assignment at every node in the \textit{dominance frontier}.
Pruned Phi Functions

This criterion causes a bunch of useless $\Phi$ functions to be inserted
– Cases where the result is never used “downstream” (useless)

*Pruned SSA* is a version where useless $\Phi$ nodes are suppressed
Other Advantages of SSA Form

Flow dependences
$4 \times 4$ edges
Other Benefits of SSA Form

\[ x_1 = \ldots \]
\[ x_2 = \ldots \]
\[ x_3 = \ldots \]
\[ x_4 = \ldots \]

\[ x_5 = \phi(x_1, x_2, x_3, x_4) \]

\[ v = 3x_5 \]
\[ w = x_5 \]
\[ y = 7x_5 \]
\[ z = w \times x_5 \]

Multiplicative representation → Additive representation

4 × 4 edges → 4 + 4 edges
DATAFLOW ANALYSIS
Dataflow-Analysis Example 1

Reaching definitions

Before p1: ∅
After p1: {<p1, x>}

Before p2: {<p1, x>, ...}
After p2: {<p2, x>, ...}

Before p3: {<p2, x>, ...}
After p3: {<p2, x>, <p3, y>, ...}

Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.

Transfer function:
\[ \lambda S. (S - \{<p_i, x>\}) \cup \{<p_2, x>\} \]

Data: sets of <program-point, variable> pairs
Dataflow-Analysis Example 1

Reaching definitions

Before p1: \( \emptyset \)
After p1: \{<p1, x>\}

Before p2: \{<p1, x>, \ldots\}
After p2: \{<p2, x>, \ldots\}

Before p3: \{<p2, x>, <p4, x>, \ldots\}
After p3: \{<p2, x>, <p3, y>, <p4, x>, \ldots\}

Meet operation: Union of sets (of <program-point, variable> pairs)

Before p4: \( \emptyset \)
After p4: \{<p4, x>\}

p1: \( x = 1; \)
\[ \ldots \]
p2: \( x = 2; \)
\[ \ldots \]
p3: \( y = x; \)
p4: \( x = 7; \)

Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.
Dataflow-Analysis Example 1

Reaching definitions: Why is it useful?
Answers the question “Where could this variable have been defined?”

Before p1: ∅
After p1: {<p1, x>}

Before p2: {<p1, x>, ...}
After p2: {<p2, x>, ...}

Before p3: {<p2, x>, <p4, x> ...}
After p3: {<p2, x>, <p3, y>, <p4, x>, ...}

p1: x = 1;

... p2: x = 2;

Before p4: ∅
After p4: {<p4, x>
Dataflow-Analysis Example 2

Live Variables

Before p1: $\emptyset$
After p1: $\{x\}$

Before p2: $\{x\}$
After p2: $\{x, y\}$

Before p3: $\{x, y\}$
After p3: $\emptyset$

Before p4: $\emptyset$
After p4: $\{x\}$

Before p5: $\{x\}$
After p5: $\{x\}$

Before p6: $\{x\}$
After p6: $\emptyset$

Transfer function:
$\lambda S. (S - \{z\}) \cup \{x, y\}$

p1: $x = 1$;
if (...) {
  p2: $y = 0$;
  p3: $z = x + y$;
}

p4: $x = 2$;
p5: $z = 3$;
p6: cout $\ll x$;

Data: sets of variables

z is not live after p5, and thus p5 is a useless assignment (= dead code)
There are some obvious advantages to this format for program analysis:
- Easy to see the live range of a given variable \( x \) assigned to in statement \( s \)
  - The region from “\( x = \ldots; \)” until the last use(s) of \( x \) before \( x \) is redefined
  - In SSA form, from “\( x_i = \ldots; \)” to all uses of \( x_i \), e.g., “\( \ldots = f(\ldots, x_i, \ldots); \)”
- Easy to see when an assignment is useless
  - We have “\( x_i = \ldots; \)” and there are no uses of \( x_i \) in any expression or assignment RHS
  - “\( x_i = \ldots; \)” is a useless assignment
  - “\( x_i = \ldots; \)” is dead code

In other words, some useful information is pre-computed, or at least easily recoverable from SSA form.

**Warning 1:** Dead code = useless assignments + unreachable code

**Warning 2:** There is another concept called “live variables.”
- When variable \( x \) is “not live,” a convenient shorthand is “Variable \( x \) is dead.”
- When \( x \) is dead just after a statement \( s \), that does not imply that \( s \) is dead code. (E.g., suppose \( s \) assigns to \( y \).)
- When \( s \) is a useless assignment to \( x \)
  - Statement \( s \) is dead code (because dead = useless or unreachable)
  - \( x \) is not live just after \( s \) (“Variable \( x \) is dead just after \( s \)”)
  - Because variable \( x \) is dead, \( s \) is a useless assignment, and thus statement \( s \) is dead code.

---

**Warning 2:** There is another concept called “live variables.”
- When variable \( x \) is “not live,” a convenient shorthand is “Variable \( x \) is dead.”
- When \( x \) is dead just after a statement \( s \), that does not imply that \( s \) is dead code. (E.g., suppose \( s \) assigns to \( y \).)
- When \( s \) is a useless assignment to \( x \)
  - Statement \( s \) is dead code (because dead = useless or unreachable)
  - \( x \) is not live just after \( s \) (“Variable \( x \) is dead just after \( s \)”)
  - Because variable \( x \) is dead, \( s \) is a useless assignment, and thus statement \( s \) is dead code.

---

**Warning 1:** Dead code = useless assignments + unreachable code