263-2810: Advanced Compiler Design

2.0 Static Single Assignment Form

Thomas R. Gross

Computer Science Department
ETH Zurich, Switzerland
2.0 Static Single Assignment IR

- SSA: Static single assignment
- There is one assignment statement that writes a variable/field/memory location
  - Assignment for short: statement, expression, ....
  - Static: in the source/IR
  - Single: each statement writes a different variable

- SSA makes data dependences *explicit*
Example

\[ x_1 = a_0 + b_0 \]
\[ d_1 = x_1 + 1 \]
\[ b_1 = a_0 + c_0 \]
Example

\[ x_1 = a_0 + b_0 \]
\[ d_1 = x_1 + 1 \]
\[ b_1 = a_0 + c_0 \]

- **Statement 1 produces a value for statement 2**
- **Only true dependences recorded**
  - No constraints due to variable *names*
Outline

SSA form is used in many production compilers

- SSA form for straight line code
  - How to turn a (JavaLi/C/Java/...) program into SSA form
- Conditional statements
- Benefits of SSA form
- SSA for well-structured programs
  - Only selected subset of control flow constructs allowed
    - “goto”-free programs
    - C – longjmp & friends
- 3.0 SSA for arbitrary programs
2.1 SSA for a basic block

- **Assumption:** Program(method) translated into basic blocks, forest of IR trees for each basic block
  - E.g., AST or similar IR
  - All sources and destinations of operations visible
  - Program written without consideration of SSA format

- **Goal:** transform one basic block into SSA format

- **Simplification**
  - Only method-local scalar variables matter
  - Arrays & fields handled later

- **Approach:** consider all statements (IR trees) in sequence
For each variable $X$: Counter $C_X$
  - $C_X$ initialized to 0 at start of method

$C_X$ indicates the “current” version

Given a statement or expression

$$D = S \otimes T \text{ or } S \otimes T$$

1. **Lookup $C_S$ and $C_T$**
   1. Yields current version, say $S_n$ and $T_m$

2. **Increment $C_D$**
   1. Yields new version ($D_k$)

3. **Replace $S$, $T$, $D$**
   $$D_k = S_n \otimes T_m \text{ or } S_n \otimes T_m$$
- Each basic block can be handled that way ....
- Only the variable names are changed
  - Otherwise use trees as before
  - Could use any other IR

- Need right value for counters $C_x$ at the start of a basic block
2.2 Conditional statements

- Simple example
  
  ```
  a = 1;
  if (b ≠ 0) {
    a = 0;
  }
  x = a;
  ```

Counters

- \(C_a = \)
- \(C_b = \)
- \(C_x = \)
2.2 Conditional statements

- Simple example

```java
a = 1;
if (b ≠ 0) {
a = 0;
}
x = a;
```

Counters

- $C_a = $
- $C_b = $
- $C_x = $
Finding the current version

- Simple example

```plaintext
a = 1;
if (b \neq 0) {
    a = 0;
}
x = a;
```

```plaintext
a_1 = 1;
if (b_0 \neq 0) {
    a_2 = 0;
}
x_1 = a_\ldots;
```

Can’t use \( a_1 \)
Can’t use \( a_2 \)
Solution: $\phi$ function

- Introduce a “magic” function $\phi$
- $\phi$ function delivers the correct version
  - (in the example)
  - If $b_0 = 0$ : returns $a_1$
  - If $b_0 \neq 0$ : returns $a_2$
- We discuss later how to implement the $\phi$ function efficiently

- The function picks the correct version depending on the path taken to reach BB2
  - More precisely: the point where we need to use either $a_1$ or $a_2$
**ϕ function**

- **Result (return value) depends on path taken**
  - Result value assigned to a new version of variable

- **Arguments are possible return values**
  - Different versions of the same (conceptually) variable

\[
a_1 = 1; \\
\text{if } (b_0 \neq 0) \{ \\
\quad a_2 = 0; \\
\} \\
a_3 = \phi(a_2, a_1) \\
x_1 = a_3; \\
\]
Paths

\[ a_1 = 1; \]
\[ \text{if } (b_0 \neq 0) \{ \]
\[ \quad a_2 = 0; \]
\[ \} \]
\[ a_3 = \phi(a_2, a_1) \]
\[ x_1 = a_3; \]
\( \phi \) function -- Notes

- \( \phi \) function appears only on the right hand side of an assignment
- \( \phi \) function placed at beginning of basic block
  - Not mandatory but simplifies reading examples
- How many arguments should the \( \phi \) function have?
if-then

Case, Switch

\( N \) predecessors. \( n \) arguments

\[ x_{new} = \phi(x_1, x_2, \ldots, x_n) \]
\( \phi \) function -- Notes

- \( \phi \) function appears only on the right hand side of an assignment

- \( \phi \) function placed at beginning of basic block
  - Not mandatory but simplifies reading examples

- How many arguments should the \( \phi \) function have?
  - Depends on the number of predecessor nodes in the control flow graph
    - Depends on programming language constructs/program
φ function -- Notes

- φ function does not evaluate *all* arguments
  - Only the single argument that is returned is evaluated
  - Why does this matter?
  - Precise and fine-grained information
φ function -- Notes

- φ function reads arguments in predecessor basic block

```plaintext
if (b_0 ≠ 0) {
  a_2 = 0;  \text{ BB1}
}
else {
  a_1 = 1;  \text{ BB2}
}

a_3 = φ(a_2, a_1)  \text{ BB2}
```
φ function -- Notes

- φ function reads arguments in predecessor basic block

```plaintext
if (b_0 \neq 0) {
  a_2 = 0;
} else {
  a_1 = 1;
}
a_3 = \phi(a_2, a_1)
```
- a2 is read (and therefore live) at the end of BB1
- a1 is read (and therefore live) at the end of BB2
- Neither a1 nor a2 is read (live) in BB3
  - No need to find register
  - Only a3 a candidate for register allocation
Converting to SSA

- Given a CFG with START node, forest of IR trees or AST
- Start with START node
  - Convert the operations in this basic block
- Insert $\phi$ functions as needed
  - More on this later
- Process next basic block until all blocks have been processed
2.3 Benefits of SSA form

1. Some optimizations are easy resp. obvious
2. Efficient representation of dependences
SSA-based optimizations

- Elimination of common subexpressions (CSE)
- The evaluation of an expression \( a + b \) at point \( P \) can be eliminated if \( a + b \) is evaluated on all paths leading to \( P \)
  - Must consider all paths
  - There cannot be an assignment to \( a \) or \( b \) along any paths after \( a+b \) has been evaluated
SSA-based optimizations

- Elimination of common subexpressions (CSE)
  - First step: identify common subexpressions
CSE

- Must consider complete program
  - "All paths ...."
  - "No assignments to operands ..."
SSA-based optimizations

- SSA form immediately provides the answer

\[ t = a_i + b_j ; \]

\[ v = a_m + b_n ; \]

If \( m=i \) and \( n=j \) the expression “\( a+b \)” is the same

- Candidate for removal
Example

\[
= a + b;
\]

```c
if ( ... ) {
    x = 0;
}
```

\[
= a + b;
\]
Example

\[ a_i + b_j; \]

\[
\text{if ( ... ) } \{
    x_2 = 0;
\}
\]

\[ a_i + b_j; \]
Example

\[ a_i + b_j; \]

\[ = a_i + b_j; \]

if ( \ldots ) {
    x_2 = 0;
}

\[ = a + b; \]

\[ = a + b; \]

if ( \ldots ) {
    a = 0;
}

\[ = a + b; \]

\[ = a + b; \]
Example

\[ = a_i + b_j; \]

if ( \ldots ) {
    x_2 = 0;
}

\[ = a_i + b_j; \]

\[ a_m = \phi(a_k, a_i) \]
\[ = a_m + b_j; \]
Use-Def (ud) chains

- Given a “use” of a variable, would like to know where the value was written (“defined”)
  - Useful to identify register allocation candidates
- SSA form makes it easy – there is one definition for each variable
  - Easily maintained mapping
Def-Use (du) chains

- For a given definition, find all uses of the value computed
- SSA form makes it easy: can easily identify uses
  - No extra work needed

\[ a_i = \ldots \]
\[ = a_i + \ldots \]
\[ = a_i + \]

\[ a_k = \ldots \]
\[ = a_k \]
\[ = a_i + \ldots \]
SSA efficient representation

- Consider ud-chains and du-chains
- Storage space for links directly proportional to the number of uses
SSA efficient representation

- Global dataflow equations are solved by iteration
- Use a bit vector to represent
  - Variables
  - Definitions
  - Expressions ...
Available Expressions: Finding IN(B) and OUT(B)

- $\text{gen}_B$ and $\text{kill}_B$ capture what happens inside a basic block
  - Sets of expressions "generated" and "killed"!

- We need IN and OUT for each basic block
  - $\text{IN}(B) = \bigcap_{B_i, B_i \text{ is predecessor of } B \text{ in CFG}} \text{OUT}(B_i)$
  - $\text{OUT}(B) = \text{gen}_B \cup (\text{IN}(B) - \text{kill}_B)$

- $N$ basic blocks, $2 \times N$ sets IN / OUT
Finding IN(B) and OUT(B)

- N basic blocks, 2×N sets IN / OUT
- System with 2×N unknowns
  - Solve by iterating until a fixed point is found

- How to start iteration?
  
  Safe assumption OUT[START] = ∅
Finding IN(B) and OUT(B)

- Safe assumption \( \text{OUT}[\text{START}] = \emptyset \)
- What about \( \text{OUT}[\text{Bi}] \) for \( \text{Bi} \neq \text{START} \)?
  - For reaching definitions, we wanted smallest set of definitions that “reach”
    - OK if we say d reaches but it does not
  - For available expressions, we want largest set of expressions that “reach”
    - OK if expr is available but not included in set
- So start with a large approximation and remove expressions that are clearly not available
  - \( \text{OUT}[\text{Bi}] = \mathcal{U} \)
  - \( \mathcal{U} \) is the set of all expressions that appear in the program
Finding available expressions

\[ \text{OUT}[\text{START}] = \emptyset \]

Initialize \( \text{OUT}[B] = \cup \) for \( \forall \ B \neq \text{START} \)

while (changes to any \( \text{OUT}(B) \)) {
  for (each basic block \( B \neq \text{START} \)) {
    \[ \text{IN}(B) = \bigcap_{B_i, B_i \text{ is predecessor of } B \text{ in } \text{CFG}} \text{OUT}(B_i) \]
    \[ \text{OUT}(B) = \text{gen}_B \cup (\text{IN}(B) - \text{kill}_B) \]
  }
}
Comments

- The order of visiting nodes of the control flow graph matters
  - For speed of convergence, not correctness
- Needs sets to hold all expressions that appear in function/method
  - Possibly large
  - Bit vector representation allows fast implementation of set operations
    - Need multi-word set representation
    - Compiler may limit size of bit vector
      - Not all instructions will be considered
SSA efficient representation

- Common subexpressions identified easily
- Cost of representation reduced *in practice*
2.4 SSA for well-structured programs

- Well-structured programs contain only “nice” control flow constructs
  - Programming language enforces property that program is well-structured

- Syntax-directed translation to SSA format
  - Insert $\phi$ functions
    - We say “insert $\phi$ node” – a tree node with the $\phi$ function
  - Rename variables
    - Use correct version~
Preparation

- Augment symbol table to record for each variable
  - Current version (integer)
  - Next version (integer)
  - We sometimes use the term “version number”
Syntax-directed translation

Given a CFG

Process program starting with START node

1. Handle straight-line code (basic blocks)
2. Handle conditional statements (if-then, if-then-else)
3. Handle loops
2.4.1 Straight-line code

- Process block from first statement (first IR node) to last statement, in source program order

- One statement $S$ at a time:
  - Consider the right-hand-side (assume no side effects)
    - For a variable $V$ on the RHS, use the current version
    - Found in symbol table
  - Rewrite RHS
  - Consider the left-hand-side (effect of the statement)
    - For variable $D$, use next version
    - Update symbol table, increment “next” version

- Same applies to expression $E$
  - No need to deal with “left-hand-side” unless there are side effects

- $V, D$ must be scalar method-local variables
  - Others unchanged
S1: \( x_1 = 6 \);
S2: \( y_1 = x_5 \);
S3: \( x_2 = 9 \);
S4: \( z_1 = y_1 \);
S5: \( y_2 = z_1 \);

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Init</th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( y )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( z )</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>