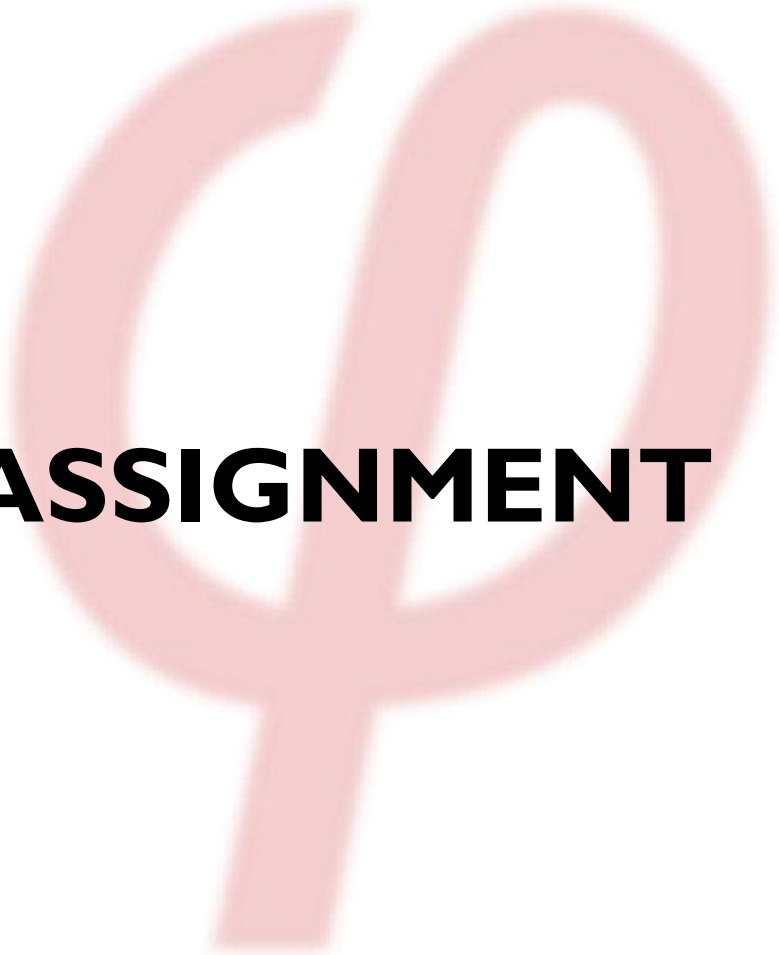
CS 526

Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526

STATIC SINGLE ASSIGNMENT



The slides adapted from Vikram Adve

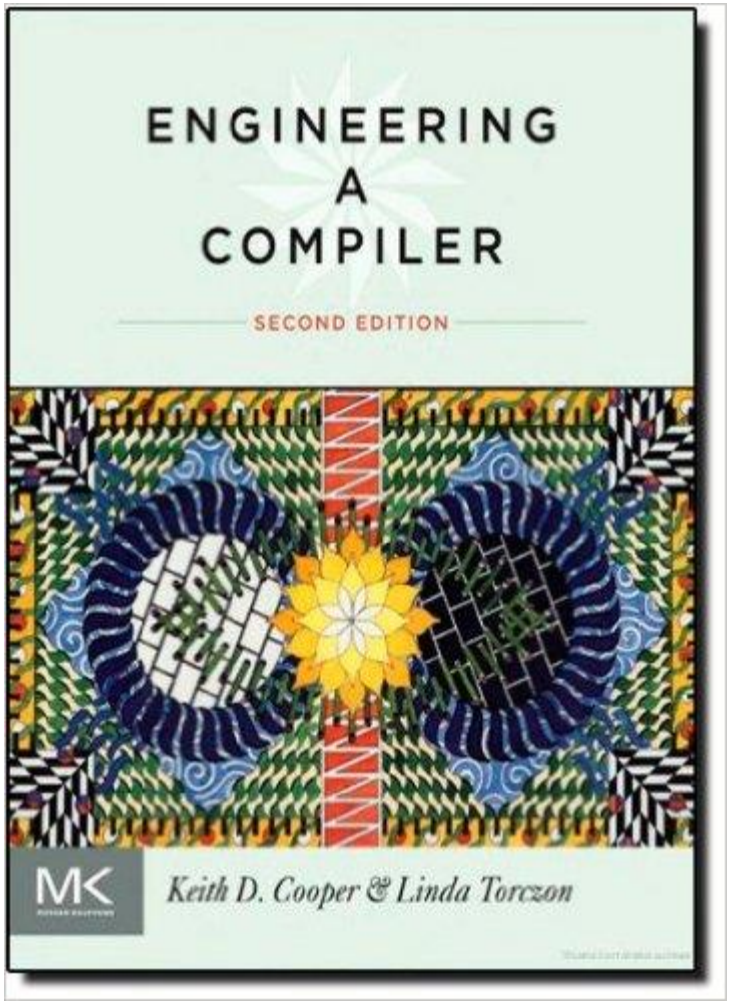
References

Cytron, Ferrante, Rosen, Wegman, and Zadeck,
"Efficiently Computing Static Single Assignment Form and the Control Dependence Graph,"
ACM Trans. on Programming Languages and Systems, 13(4), Oct. 1991, pp. 451–490.

Muchnick, Section 8.11 (*partially covered*).

Engineering a Compiler, Section 5.4.2 (*partially covered*).

Books (Updated)



Available Online:
(we have institutional access)

http://www.sciencedirect.com/science/book/9780120884780

(If accessing from home, use VPN)

Also see:
http://openlibrary.org

What is SSA Form?

What is intermediate language?

Design tradeoffs:

- Expressive enough to represent source code information unambiguously
- Efficient for (numerous) optimizations and analyses
- Can easily generate backend code from it

Why do we study SSA ?

What is SSA Form?

(Informally) A program can be converted into ***SSA form*** as follows:

- Each assignment to a variable is given a unique name
- All of the uses reached by that assignment are renamed.

Easy for straight-line code:

$$V \leftarrow 4$$
$$\ldots \leftarrow V + 5$$
$$V \leftarrow 6$$
$$\ldots \leftarrow V + 7$$

What is SSA Form?

(Informally) A program can be converted into ***SSA form*** as follows:

- Each assignment to a variable is given a unique name
- All of the uses reached by that assignment are renamed.

Easy for straight-line code:

$$\begin{aligned} \mathbf{v_0} &\leftarrow 4 \\ \ldots &\leftarrow \mathbf{v_0} + 5 \\ \mathbf{v_1} &\leftarrow 6 \\ \ldots &\leftarrow \mathbf{v_1} + 7 \end{aligned}$$

SSA Straight-line Code

```
X = 1;
X = X + 1;
Y = X;
```

SSA Straight-line Code

```
X0 = 1;
X1 = X0 + 1;
Y = X1;
```

SSA Straight-line Code

```
X = 1;
Y = f(X);
X = Y + X;
```

SSA and Branches

```
if (...)
   X = 42;
else
   X = 7*7;

Y = X;
```

SSA and Branches

```
X = 0;
if (...)
    X = 42;

Y = X;
```

SSA and Branches

```
X = 0;
if (...) {
   if (...)
      X = 42;
   else
      X = 7*7;

}

Y = X;
```

SSA and Loops

```
j=1;

while (j < X)
    ++j;

N = j;
```

```
j = 1;

if (j >= X) goto E;

S:
j = j+1;
if (j < X) goto S;

E:
N = j;
```

SSA and Switches

```
X = 0;
switch (...) of
  a: X = 1;
  b: X = 2;
  c: X = 3;

Y = X;
```

Definition of φ Function

In a basic block B with N predecessors, P1, P2, . . . , PN,

$$\mathbf{X = \varphi(V1, V2, \ldots, VN)}$$

assigns X = Vi if control enters block B from Pi , $1 \leq i \leq N$

Properties of φ-functions:

- φ is not an executable operation.
- φ has exactly as many arguments as the number of incoming basic block edges
- Think about the argument Vi as being evaluated on CFG edge from predecessor Pi to B

More Definitions

Value: expression that cannot be evaluated further (numbers, words, memory addresses, …)

Storage location: register or memory address

- Machine and virtual registers
- Stack and heap

Variable: named storage location (map from name to address)

Pointer: variable whose value is another memory location

Alias: an alternative name of an entity (a variable, a location, …)

More Definitions

Use of a variable: A use of variable X is a reference that may read the value stored in the location named X.

Definition of a variable: A definition (def) of a variable X is a reference that may store a value into the location named X. *Examples:* Assignment; input I/O

Ambiguity of definitions:

Unambiguous definition **(must)**: guaranteed to store to X

Ambiguous definition **(may)**: may store to X

Q. Where does ambiguity come from?

We define ambiguous/unambiguous use similarly.

Def-Use Chains

- **Def-use chain**: The set of uses reached by a particular definition.

- **Use-def chain**: The set of definitions reaching a particular use

Definition of SSA Form

A program is in SSA form if:

- each variable is assigned a value in **exactly one** statement
- each **use** of a variable is **dominated** by the **definition**

Which Variables to Convert?

Convert all variables to SSA form, ***except*** . . .

Arrays: Array elements do not have an explicit name

Variables that may have aliases: do not have a unique name

Volatile variables: can be modified "unexpectedly"

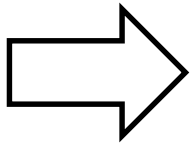
E.g., In LLVM, only variables in virtual registers are in SSA form.

LLVM: Mem2reg

-mem2reg: Promote Memory to Register

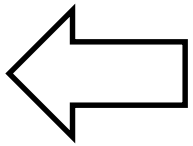
"This file promotes memory references to be register references. It promotes alloca instructions which only have loads and stores as uses. An alloca is transformed by using dominator frontiers to place phi nodes, then traversing the function in depth-first order to rewrite loads and stores as appropriate. This is just the standard SSA construction algorithm to construct "pruned" SSA form."

```
int f(int x) {
  int y = x + 1;
  return y
}
```



No optimizations

```
...
%6 = alloca i32, align 4
%7 = load i32, i32* %0, align 4
%8 = add nsw i32 1, %7
store i32 %8, i32* %6, align 4
%9 = load i32, i32* %6, align 4
ret i32 %9
```



After mem2reg

```
%3 = add nsw i32 %0, 1
ret i32 %3
```

Advantages of SSA Form

Makes def-use and use-def chains explicit:

These chains are foundation of many dataflow optimizations

- We will see some soon!

Compact, flow-sensitive* def-use information

- fewer def-use edges per variable: one per CFG edge

* Takes the order of statements into account

Advantages of SSA Form (cont.)

No anti- and output dependences on SSA variables

- Direct dependence:
- Antidependence:
- Output dependence:

Explicit merging of values (φ): key additional information

Can serve as **IR for code transformations** (see LLVM)

Disadvantages of SSA Form

Size of SSA program is $O(N^2)$ for an ordinary program with N variables.

Often Not used for structures and arrays

May not be used for scalar variables with aliases

If used as IR, must be converted back to code (Not bad)

Otherwise, must be recomputed frequently (Often bad)

Constructing SSA Form

Simple algorithm

1. insert φ-functions for every variable at every join
2. solve reaching definitions
3. rename each use to the def that reaches it (unique)

What's wrong with this approach?

1. too many φ-functions (precision)
2. too many φ-functions (space)
3. too many φ-functions (time)

Where do we place φ-functions?

```
if (...) then {
      V = ...;
      if (...) {
         U = V + 1;
      } else {
         U = V + 2;
      }
      W = U + 1;
}
```

- For V?
- For U?
- For W?

Intuition for SSA Construction

Informal Conditions

If block X contains an assignment to a variable V, then a φ-function must be inserted in each block Z such that:

1. there is a non-empty path between X and Z,

2. there is a path from entry block (s) to Z that does not go through X,

3. Z is the first node on the path from X to Z that satisfies point 2.

Intuition for SSA Construction

Informal Conditions

If block X contains an assignment to a variable V, then a φ-function must be inserted in each block Z such that:

1. there is a non-null path between X and Z, and

   the value of V computed in X reaches Z

2. there is a path from entry block (s) to Z that does not go through X

   there is a path that does not go through X, so some other value of V reaches Z along that path(ignore bugs due to uses of uninitialized variables). So, two values must be merged at X with a φ

3. Z is the first node on the path from X to Z that satisfies point 2

   the φ for the value coming from X is placed in Z and not in some earlier node on the path

Intuition for SSA Construction

Informal Conditions

Iterating the Placement Conditions:

- After a φ is inserted at Z, the above process must be repeated for Z because the φ is effectively a new definition of V.
- For each block X and variable V, there must be at most one φ for V in X. This means that the above iterative process can be done with a single worklist of nodes for each variable V , initialized to handle all original assignment nodes X simultaneously.