CS 526
Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526
STATIC SINGLE ASSIGNMENT

The slides adapted from Vikram Adve

Muchnick, Section 8.11 (*partially covered*).

Engineering a Compiler, Section 5.4.2 (*partially covered*).
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
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:  \( A=1; B=A+1 \)
- Antidependence:  \( A=1; B=A+1; A=2 \)
- Output dependence:  \( A=1; A=2; B=A+1 \)

Cannot reorder

Explicit merging of values (\(\phi\)): key additional information

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

Simple algorithm
1. insert $\phi$-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 $\phi$-functions (precision)
2. too many $\phi$-functions (space)
3. too many $\phi$-functions (time)
Where do we place $\varphi$-functions?

\[ V = \ldots; \quad U = \ldots; \quad W = \ldots; \]
\[
\text{if (\ldots) then } \{
    V = \ldots;
    \text{if (\ldots) }
    \{
        U = V + 1;
    \}
    \text{else }
    \{
        U = V + 2;
    \}
\}
\]
\[
W = U + 1;
\]

- For $V$?
- For $U$?
- For $W$?
Where do we place $\varphi$-functions?

$V_0=\ldots$; $U_0=\ldots$; $W_0=\ldots$; 
if (...) then {
    $V_1 = \ldots$;
    if (...) {
        $U_1 = V_1 + 1$;
    }
} else {
    $U_2 = V_1 + 2$;
}

$V_2 = \varphi(V_1, V_1)$; $U_3 = \varphi(U_1, U_2)$; $W_1 = \varphi(W_0, W_0)$

$W_1 = U_3 + 1$;

$V_3 = \varphi(V_0, V_1)$; $U_4 = \varphi(U_0, U_3)$; $W_2 = \varphi(W_0, W_1)$

• For $V$?
• For $U$?
• For $W$?
Intuition for SSA Construction

**Informal Conditions**

If a block $X$ contains an assignment to a variable $V$, then a $\varphi$-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 the entry block(s) to $Z$ that does not go through $X$,

3. $Z$ is the first node on the path from $X$ that satisfies point 2.
Intuition for SSA Construction

**Informal Conditions**

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

1. there is a non-empty path between $X$ and $Z$, and
   the value of $V$ computed in $X$ reaches $Z$

2. there is a path from the 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 $\phi$

3. $Z$ is the first node on the path from $X$ to $Z$ that satisfies point 2
   the $\phi$ 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 $\phi$ is inserted at $Z$, the above process must be repeated for $Z$ because the $\phi$ is effectively a new definition of $V$.
- For each block $X$ and variable $V$, there must be at most one $\phi$ 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.
Minimal SSA

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 i.e., the use can refer to a unique name.

Minimal SSA: As few as possible $\phi$-functions,

Pruned SSA: As few as possible $\phi$-functions and no dead $\phi$-functions (i.e., the defined variable is used later)
• One needs to compute liveness information
• More precise, but requires additional time
SSA Construction Algorithm

Steps:
1. Compute the dominance frontiers*
2. Insert $\phi$-functions
3. Rename the variables

**Thm.** Any program can be put into minimal SSA form using the previous algorithm. [Refer to the paper for proof]
Dominance in Flow Graphs (review)

Let $d, d_1, d_2, d_3, n$ be nodes in $G$.

$d$ dominates $n$ ("$d$ dom $n$") iff every path in $G$ from $s$ to $n$ contains $d$

$d$ properly dominates $n$ ("$d$ pdom $n$") if $d$ dominates $n$ and $d \neq n$

$d$ is the immediate dominator of $n$ ("$d$ idom $n$") if $d$ is the last proper dominator on any path from initial node to $n$,

$\text{DOM}(x)$ denotes the set of dominators of $x$,

Dominator tree*: the children of each node $d$ are the nodes $n$ such that "$d$ idom $n$" (d immediately dominates n)
Dominance Frontier

The dominance frontier of node $X$ is the set of nodes $Y$ such that $X$ dominates a predecessor of $Y$, but $X$ does not properly dominate $Y$.

$$\text{DF}(X) = \{Y \mid \exists P \in \text{Pred}(Y) : X \text{ dom } P \text{ and not } (X \text{ pdom } Y)\}$$

We can split $\text{DF}(X)$ in two groups of sets:

$$\text{DF}_{\text{local}}(X) \equiv \{Y \in \text{Succ}(X) \mid \text{not } X \text{ idom } Y\}$$

$$\text{DF}_{\text{up}}(Z) \equiv \{Y \in \text{DF}(Z) \mid \exists W. W \text{ idom } Z \text{ and not } W \text{ pdom } Y\}$$

Then:

$$\text{DF}(X) = \text{DF}_{\text{local}}(X) \cup \bigcup_{Z \in \text{Children}(X)} \text{DF}_{\text{up}}(Z)$$

* child, parent, ancestor, and descendant always refer to the dominator tree. predecessor, successor, and path always refer to CFG
Dominance Frontier Algorithm

for each $X$ in a bottom-up traversal of the dominator tree (visit the node $X$ in the tree after visiting its children):

$$DF(X) \leftarrow \emptyset$$

for each $Y \in \text{succ}(X)$ /* local */

if not $X idom Y$ then

$$DF(X) \leftarrow DF(X) \cup \{Y\}$$

for each $Z \in \text{children}(X)$ /* up */

for each $Y \in DF(Z)$

if not $X idom Y$ then

$$DF(X) \leftarrow DF(X) \cup \{Y\}$$
Dominance and LLVM

**Dominators.h**

---

```
00001 //--- Dominators.h - Dominator Info Calculation -------------------*
00002 //
00003 //
00004 //
00005 // This file is distributed under the University of Illinois Open Source
00006 // License. See LICENSE.TXT for details.
00007 //
00008 //------------------------------------------------------------------------
00009 //
00010 // This file defines the DominatorTree class, which provides fast and efficient
00011 // dominance queries.
00012 //
00013 //------------------------------------------------------------------------
00014 //
```

**DominanceFrontier.h**

---

```
00001 //--- llvm/Analysis/DominanceFrontier.h - Dominator Frontiers --* C++ --*--
00002 //
00003 //
00004 //
00005 // This file is distributed under the University of Illinois Open Source
00006 // License. See LICENSE.TXT for details.
00007 //
00008 //------------------------------------------------------------------------
00009 //
00010 // This file defines the Dominancefrontier class, which calculate and holds the
00011 // dominance frontier for a function.
00012 //
00013 // This should be considered deprecated, don't add any more uses of this data
00014 //
00015 //
```

```cpp
00016 namespace llvm {
00017
00018 #ifndef LLVM_ANALYSIS_DOMINANCEFRONTIER_H
00019 #define LLVM_ANALYSIS_DOMINANCEFRONTIER_H
00020
00021 #include "LLVM/IR/Dominators.h"
00022 #include <map>
00023 #include <set>
00024
00025 namespace llvm {
00026
00027 #template <class BlockT>
00028 class DominanceFrontierBase {
00029 public:
00030   typedef std::set<BlockT *> DomSetType; // Dom set for a bb
00031   typedef std::map<BlockT *, DomSetType> DomSetMapType; // Dom set map
00032   protected:
00033   typedef GraphTraits<BlockT> BlockTraits;
00034
```
SSA Construction Algorithm

Steps:
1. Compute the dominance frontiers
2. Insert $\varphi$-functions
3. Rename the variables
Insert \( \varphi \)-functions

for each variable \( V \)

\[
\text{HasAlready} \leftarrow \emptyset \\
\text{EverOnWorkList} \leftarrow \emptyset \\
\text{WorkList} \leftarrow \emptyset
\]

for each node \( X \) that may modify \( V \)

\[
\text{EverOnWorkList} \leftarrow \text{EverOnWorkList} \cup \{X\} \\
\text{WorkList} \leftarrow \text{WorkList} \cup \{X\}
\]
Insert $\varphi$-functions

for each variable $V$

$$\text{HasAlready} \leftarrow \emptyset$$
$$\text{EverOnWorkList} \leftarrow \emptyset$$
$$\text{WorkList} \leftarrow \emptyset$$

for each node $X$ that may modify $V$

$$\text{EverOnWorkList} \leftarrow \text{EverOnWorkList} \cup \{X\}$$
$$\text{WorkList} \leftarrow \text{WorkList} \cup \{X\}$$

while $\text{WorkList} \neq \emptyset$

remove $X$ from $\text{WorkList}$

for each $Y \in \text{DF}(X)$

if $Y \notin \text{HasAlready}$ then

insert a $\varphi$-node for $V$ at $Y$

$$\text{HasAlready} \leftarrow \text{HasAlready} \cup \{Y\}$$

if $Y \notin \text{EverOnWorkList}$ then

$$\text{EverOnWorkList} \leftarrow \text{EverOnWorkList} \cup \{Y\}$$
$$\text{WorkList} \leftarrow \text{WorkList} \cup \{Y\}$$
Renaming Variables*

Renaming definitions is easy – just keep the counter for each variable.

To rename each use of V:

(a) **Use in non-φ-functions:** Refer to immediately dominating definition of V (+ φ nodes inserted for V).

   preorder on Dominator Tree!

(b) **Use as a φ-function operand:** Refer to the definition that immediately dominates the node with the incoming CFG edge (not the node with the φ-function)

   rename the φ-operand when processing the predecessor basic block!

* For the full algorithm refer to the paper
j=1;

while (j < X)
  ++j;

N = j;

A: j = 1;

B: if (j >= X) goto E;

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

E:

N = j;
\texttt{j=1;}

\texttt{while (j < X)}
\begin{itemize}
    \item \texttt{\texttt{++j;}}
\end{itemize}

\texttt{N = j;}

\texttt{A: j0 = 1;}

\texttt{B: if (j0 >= X) goto E;}

\texttt{S: j1 = \varphi(j0, j2)}
\begin{itemize}
    \item \texttt{j2 = j1+1;}
    \item if (j2 < X) goto S;
\end{itemize}

\texttt{E: j3 = \varphi(j0, j2)}
\begin{itemize}
    \item \texttt{N = j3;}
\end{itemize}
Translating Out of SSA Form

Overview:
1. Dead-code elimination (prune dead \( \phi \)s)
2. Replace \( \phi \)-functions with copies in predecessors
3. Register allocation with copy coalescing
Control Dependence

**Def.** Postdomination: node $p$ postdominates a node $d$ if all paths to the exit node of the graph starting at $d$ must go through $p$.

**Def.** In a CFG, node $Y$ is control-dependent on node $B$ if

- There is a non-empty path $N_0 = B,N_1,N_2, ..., N_k = Y$ such that $Y$ postdominates $N_1 \ldots N_k$, and
- $Y$ does not strictly postdominate $B$.

**Def.** The Reverse Control Flow Graph (RCFG) of a CFG has the same nodes as CFG and has edge $Y \rightarrow X$ if $X \rightarrow Y$ is an edge in CFG.

- $p$ is a postdominator of $d$ iff $p$ dominates $d$ in the RCFG.
Computing Control Dependence

**Key observation:** Node Y is control-dependent on B iff B ∈ DF(Y) in RCFG.

**Algorithm:**
1. Build RCFG
2. Build dominator tree for RCFG
3. Compute dominance frontiers for RCFG
4. Compute CD(B) = {Y | B ∈ DF(Y)}.

CD(B) gives the nodes that are control-dependent on B.
Summary

Complexity:

The conversion to SSA form is done in three steps:

1. The *dominance frontier* mapping is constructed from the control flow graph $CFG$ (Section 4.2). Let $CFG$ have $N$ nodes and $E$ edges. Let $DF$ be the mapping from nodes to their dominance frontiers. The time to compute the dominator tree and then the dominance frontiers in $CFG$ is $O(E + \sum X | DF(X)|)$.

2. Using the dominance frontiers, the locations of the $\phi$-functions for each variable in the original program are determined (Section 5.1). Let $A_{tot}$ be the total number of assignments to variables in the resulting program, where each ordinary assignment statement $LHS \leftarrow RHS$ contributes the length of the tuple $LHS$ to $A_{tot}$, and each $\phi$-function contributes 1 to $A_{tot}$. Placing $\phi$-functions contributes $O(A_{tot} \times avrgDF)$ to the overall time, where $avrgDF$ is the weighted average (7) of the sizes $|DF(X)|$.

3. The variables are renamed (Section 5.2). Let $M_{tot}$ be the total number of mentions of variables in the resulting program. Renaming contributes $O(M_{tot})$ to the overall time.

Follow up works:

- A linear time algorithm for placing phi-nodes (POPL 1995)
  
  [https://dl.acm.org/citation.cfm?id=199464](https://dl.acm.org/citation.cfm?id=199464)

- Algorithms for computing the static single assignment form (JACM 2003)

Further reading:

- Tiger Book, Chapter 19