STATIC SINGLE ASSIGNMENT

The slides adapted from Vikram Adve
References


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: \( A=1; A=2; B=A+1 \)

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;\ U=\ldots;\ W=\ldots;\$

if (...) then {
    $V=\ldots;\$
    if (...) {
        $U=V+1;\$
    } else {
        $U=V+2;\$
    }
}

$W=U+1;\$

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

\[ V=\ldots; \quad U=\ldots; \quad W=\ldots; \]
\[
\text{if } (...) \text{ then } \{
\quad V_1 = \ldots;
\quad \text{if } (...) \{
\quad \quad U_1 = V_1 + 1;
\quad \} \text{ else } \{
\quad \quad U_2 = V_1 + 2;
\quad \}
\}
\quad V_2 = \phi(V_1, V_1); \ U_3 = \phi(U_1, U_2); \ W_1 = \phi(W_0, W_0); \ W_1 = U_3 + 1;
\}
\quad V_3 = \phi(V_0, V_1); \ U_4 = \phi(U_0, U_3); \ W_2 = \phi(W_0, W_1) \]
Intuition for SSA Construction

Informal Conditions

If 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 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-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 $\phi$

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 $\varphi$-functions
3. Rename the variables

Thm. Any program can be put into minimal SSA form using the previous algorithm. [Refer to 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:

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

for each $X$ in a bottom-up traversal of the dominator tree

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

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 $X$ pdom $Y$ then

$$DF(X) \leftarrow DF(X) \cup \{Y\}$$
**Dominance Frontier Properties**

**Thm. 1:** Dominance Frontier Algorithm is correct

**Set dominance frontier:** For a set \( \mathcal{P} \) of flow graph nodes,

\[
DF(\mathcal{P}) = \bigcup_{X \in \mathcal{P}} DF(X)
\]

**Iterated dominance frontier:** \( DF^+(\mathcal{P}) \) is a limit of the sequence

\[
DF_i = DF(\mathcal{P}) \\
DF_{i+1} = DF(\mathcal{P} \cup DF_i)
\]

**Thm. 2:** The set of nodes that need \( \varphi \)-functions for any variable \( V \) is the iterated dominance frontier \( DF^+(\mathcal{P}_X) \), where \( \mathcal{P}_X \) is the set of nodes that may modify \( V \).
Dominance and LLVM

LLVM mainline

Dominators.h

Go to the documentation of this file.

```c
/***** Dominators.h - Dominator Info Calculation *******- C++ -*-****/
/****
/**** The LLVM Compiler Infrastructure
/****
/**** This file is distributed under the University of Illinois Open Source
/**** License. See LICENSE.TXT for details.
/****
/**** This file defines the DominatorTree class, which provides fast and efficient
/**** dominance queries.
/****
/**** --------------- *****/
```

DominanceFrontier.h

Go to the documentation of this file.

```c
/***** LLVM/Analysis/DominanceFrontier.h - Dominator Frontiers --* C++ -*-****/
/****
/**** The LLVM Compiler Infrastructure
/****
/**** This file is distributed under the University of Illinois Open Source
/**** License. See LICENSE.TXT for details.
/****
/**** This file defines the DominanceFrontier class, which calculate and holds the
/**** dominance frontier for a function.
/****
/**** This should be considered deprecated, don't add any more uses of this data
/****
/**** structure.
/****
/**** --------------- *****/
```

```c
//ifndef LLVM_ANALYSIS_DOMINANCEFRONTIER_H
#define LLVM_ANALYSIS_DOMINANCEFRONTIER_H

#include "llvm/IR/Dominators.h"
#include <map>
#include <set>

namespace llvm {

namespace {

template <class BlockT>

class DominanceFrontierBase {

public:

typedef std::set<BlockT*> DomSetType; // Dom set for a bb

typedef std::map<BlockT*, DomSetMapType>; // Dom set map

protected:

typedef GraphTraits<BlockT> BlockTraits;
```

```c
```
SSA Construction Algorithm

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

for each variable $V$
  HasAlready $\leftarrow \emptyset$
  EverOnWorkList $\leftarrow \emptyset$
  WorkList $\leftarrow \emptyset$

for each node $X$ that may modify $V$
  EverOnWorkList $\leftarrow$ EverOnWorkList $\cup \{X\}$
  WorkList $\leftarrow$ WorkList $\cup \{X\}$
Insert $\varphi$-functions

for each variable $V$
\begin{align*}
&\text{HasAlready} \leftarrow \emptyset \\
&\text{EverOnWorkList} \leftarrow \emptyset \\
&\text{WorkList} \leftarrow \emptyset \\
&\text{for each node } X \text{ that may modify } V \\
&\quad \text{EverOnWorkList} \leftarrow \text{EverOnWorkList} \cup \{X\} \\
&\quad \text{WorkList} \leftarrow \text{WorkList} \cup \{X\}
\end{align*}

while $\text{WorkList} \neq \emptyset$
\begin{align*}
&\text{remove } X \text{ from } W \\
&\text{for each } Y \in \text{DF}(X) \\
&\quad \text{if } Y \not\in \text{HasAlready} \text{ then} \\
&\quad \quad \text{insert a } \varphi\text{-node for } V \text{ at } Y \\
&\quad \quad \text{HasAlready} \leftarrow \text{HasAlready} \cup \{Y\} \\
&\quad \text{if } Y \not\in \text{EverOnWorkList} \text{ then} \\
&\quad \quad \text{EverOnWorkList} \leftarrow \text{EverOnWorkList} \cup \{Y\} \\
&\quad \quad \text{WorkList} \leftarrow \text{WorkList} \cup \{Y\}
\end{align*}
Renaming Variables

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

To rename each use of $V$:

(a) Use in a non-$\varphi$-functions: Use immediately dominating definition of $V$ (+ $\varphi$ nodes inserted for $V$).

preorder on Dominator Tree!

(b) Use in a $\varphi$-function operand: Use the definition that immediately dominates incoming CFG edge (not $\varphi$)

rename the $\varphi$-operand when processing the predecessor basic block!
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 \( N_0 = B, N_1, N_2, \ldots, N_k = Y \) 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.
Computing Control Dependence

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

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

$CD(B)$ gives the nodes that are control-dependent on $B$. 
SSA-Based Optimizations

- Dead Code Elimination (DCE)
- Sparse Conditional Constant Propagation (SCCP)
- Loop-Invariant Code Motion (LICM)
- Global Value Numbering (GVN)
- Strength Reduction of Induction Variables
- Live Range Identification in Register Allocation
Conditional Constant Propagation: SCCP

Goals
Identify and replace SSA variables with constant values
Delete infeasible branches due to discovered constants

Safety
Analysis: Explicit propagation of constant expressions
Transformation: Most languages allow removal of computations

Profitability
Fewer computations, almost always (except pathological cases)

Opportunity
Symbolic constants, conditionally compiled code, …
Example 1

J = 1;
...
if (J > 0)
    I = 1; // Always produces 1
else
    I = 2;
Example 2

I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    I = J;  // Always produces 1
}

We need to proceed with the assumption that everything is constant until proved otherwise.
Example 3

I = 1;
...
while (...) {
    J = I;
    I = f(...);
    ...
    if (J > 0)
        I = J; // Always produces 1
}
Conditional Constant Propagation

**Advantage:**
Simultaneously finds constants + eliminates infeasible branches.

**Optimistic**
Assume every variable may be constant (T), until proven otherwise.
Pessimistic \( \equiv \) initially assume nothing is constant (\( \bot \)).

**Sparse**
Only propagates variable values where they are actually used or defined
(using def-use chains in SSA form).

**SSA vs. def-use chains**
Much faster: SSA graph has fewer edges than def-use graph
Paper claims SSA catches more constants (not convincing)