CS 526

Advanced Compiler Construction

https://charithm.web.illinois.edu/cs526/sp2024/

(slides adapted from Sasa and Vikram)

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*).

Constructing SSA Form

Simple algorithm

1. insert φ-functions for every variable at every join (the beginning of CFG nodes with multiple incoming edges)
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?

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

     W = U + 1;
}
```

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

Where do we place φ-functions?

```
V0=...; U0=...; W0=...;
if (...) then {
      V1 = ...;
      if (...) {
         U1 = V1 + 1;
      } else {
         U2 = V1 + 2;
      }
      V2=φ(V1, V1);U3=φ(U1, U2);W1=φ(W0, W0)
      W1 = U3 + 1;
}
V3=φ(V0, V1); U4=φ(U0, U3); W2=φ(W0, W1)
```

- 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, 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 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.

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 φ-functions,

Pruned SSA: As few as possible φ-functions and no dead φ-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 φ-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, d1, d2, d3, 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,

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 *

DF(X) = {Y | ∃ P ∈ Pred(Y) : X dom P and not (X pdom Y)}

We can split DF(X) in two groups of sets:

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

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

One can show that:

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

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

LEMMA 1. *The dominance frontier equation* (4) *is correct.*

PROOF. Because dominance is reflexive, $DF_{local}(X) \subseteq DF(X)$. Because dominance is transitive, each child Z of X has $DF_{up}(Z) \subseteq DF(X)$. We must still show that everything in $DF(X)$ has been accounted for. Suppose $Y \in DF(X)$, and let $U \rightarrow Y$ be an edge such that X dominates U but does not strictly dominate Y. If $U = X$, then $Y \in DF_{local}(X)$, and we are done. If $U \neq X$, on the other hand, then there is a child Z of X that dominates U but cannot strictly dominate Y because X does not strictly dominate Y. This implies that $Y \in DF_{up}(Z)$. □

From Cytron et al.

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) ← Ø

for each Y ∈ succ(X) /* local */
    if not X idom Y then
        DF(X) ← DF(X) U {Y}

for each Z ∈ children(X) /* up */
    for each Y ∈ DF(Z)
        if not X idom Y then
            DF(X) ← DF(X) U {Y}
```

(The paper also has the argument for correctness)

Dominance and LLVM

LLVM mainline

Main Page | Related Pages | Modules | Namespaces | Classes | Files

File List | File Members

Dominators.h

Go to the documentation of this file.

```
//===- 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.

```
//===- 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.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_ANALYSIS_DOMINANCEFRONTIER_H
#define LLVM_ANALYSIS_DOMINANCEFRONTIER_H

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

namespace llvm {

//===----------------------------------------------------------------------===//
/// DominanceFrontierBase - Common base class for computing forward and inverse
/// dominance frontiers for a function.
///
template <class BlockT>
class DominanceFrontierBase {
public:
  typedef std::set<BlockT *> DomSetType;                // Dom set for a bb
  typedef std::map<BlockT *, DomSetType> DomSetMapType; // Dom set map

protected:
  typedef GraphTraits<BlockT *> BlockTraits;

```

SSA Construction Algorithm

Steps:

1. Compute the dominance frontiers
2. Insert φ-functions
3. Rename the variables

Insert φ-functions

```
for each variable V
    HasAlready ← ∅
    EverOnWorkList ← ∅
    WorkList ← ∅
    for each node X that may modify V
        EverOnWorkList ← EverOnWorkList ⋃ {X}
        WorkList ← WorkList ⋃ {X}
```

Insert φ-functions

```
for each variable V
    HasAlready ← ∅
    EverOnWorkList ← ∅
    WorkList ← ∅
    for each node X that may modify V
        EverOnWorkList ← EverOnWorkList ⋃ {X}
        WorkList ← WorkList ⋃ {X}

    while WorkList ≠ ∅
        remove X from WorkList
        for each Y ∈ DF(X)
            if Y ∉ HasAlready then
                insert a ϕ-node for V at Y
                HasAlready ← HasAlready ⋃ {Y}
                if Y ∉ EverOnWorkList then
                    EverOnWorkList ← EverOnWorkList ⋃ {Y}
                    WorkList ← WorkList ⋃ {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

```
B1: X = 0
    if ( Cond1) {
        do {
B2:         X = X + f1()
            do {
B3:             X = X + f2()
            } while (cond2)
B4:         X = X + f3()
        } while (cond3)
    }
B5: return X
```

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

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

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

CFG?

DT?

DF(B3)?

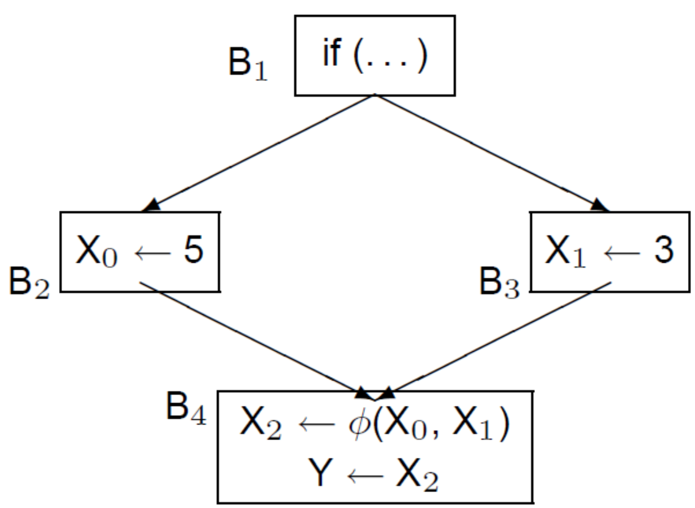
Insert phi at B3?

Translating Out of SSA Form

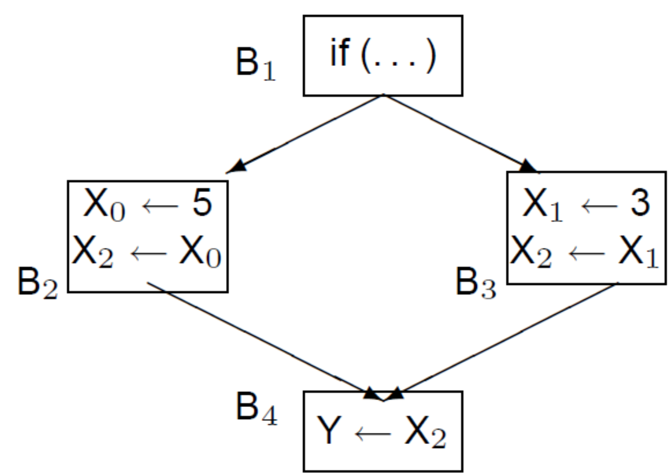
Overview:

1. Dead-code elimination (prune dead φs)
2. Replace φ-functions with copies in predecessors
3. Register allocation with copy coalescing

Before Step 2



After Step 2



Control Dependence

Def. Postdomination: node *p* postdominates a node *d* iff 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 path **B**,N1,N2, ...,Nk, **Y**, such that Y postdominates N1 . . .Nk (possibly empty), and
- Y does not postdominate B

Def. **Reverse Control Flow Graph** (RCFG) of a CFG has the same nodes as CFG and has edge Y → X if X → 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* $B \in DF(Y)$ in RCFG.

Algorithm:

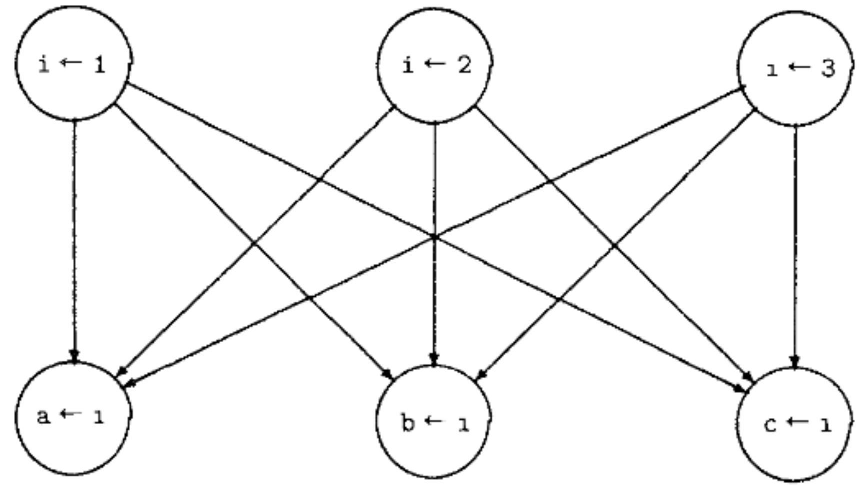
1. Build RCFG
2. Build dominator tree for RCFG
3. Compute dominance frontiers for RCFG
4. Compute $CD(B) = \{Y \mid B \in DF(Y)\}$.

CD(B) contains ***nodes that are control-dependent*** on B.

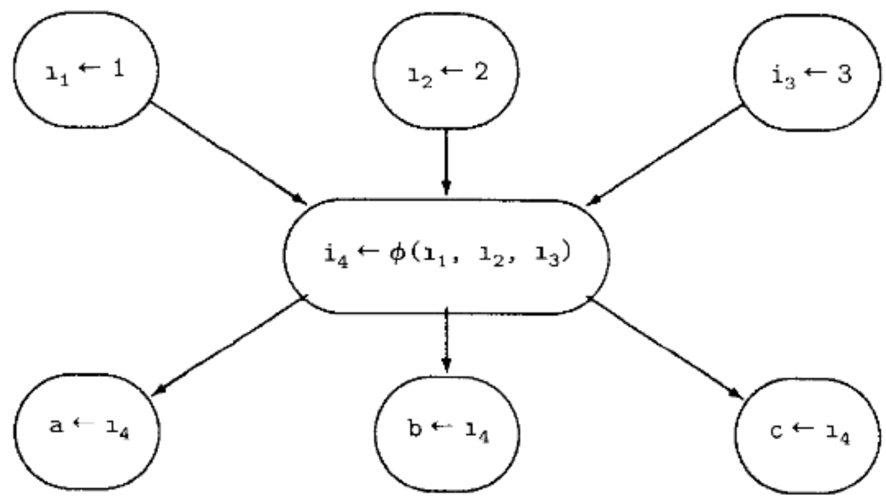
Def-use and Use-def (SSA vs no-SSA)

```
select j
  when x {i ← 1}
  when y {i ← 2}
  when z {i ← 3}
end
select k
  when x {a ← i}
  when y {b ← i}
  when z {c ← i}
end
```

Original Program



Def-Use Chains for Previous Program



SSA Graph for Previous Program

From Wegman et al.

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 ϕ-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 ϕ-function contributes 1 to A_{tot}. Placing ϕ-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
- Algorithms for computing the static single assignment form (JACM 2003)

Further reading:

- Tiger Book, Chapter 19
- On History: http://citi2.rice.edu/WS07/KennethZadeck.pdf