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*).

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 reoder

Explicit merging of values (φ): key additional information

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

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?

```
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?

```
V=...; U=...; W=...;
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,

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 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.

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 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:

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

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

Then:

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

Dominance Frontier Algorithm

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

    DF(X) ← ∅

    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 *pdom* Y then

                DF(X) ← DF(X) U {Y}

Dominance Frontier Properties

Thm. 1: Dominance Frontier Algorithm is correct

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

$$\mathrm{DF}(\mathcal{P}) = \bigcup_{X \in \mathcal{P}} \mathrm{DF}(X)$$

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

$$\mathrm{DF}_i = \mathrm{DF}(\mathcal{P})$$

$$\mathrm{DF}_{i+1} = \mathrm{DF}(\mathcal{P} \cup \mathrm{DF}_i)$$

Thm. 2: The set of nodes that need φ-functions for any variable V is the iterated dominance frontier $\mathrm{DF}^{+}(\mathcal{P}_X)$, where $\mathcal{P}_{\mathrm{X}}$ is the set of nodes that may modify V.

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 W
        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 a non-φ-functions: Use immediately dominating definition of V (+ φ nodes inserted for V).

preorder on Dominator Tree!

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

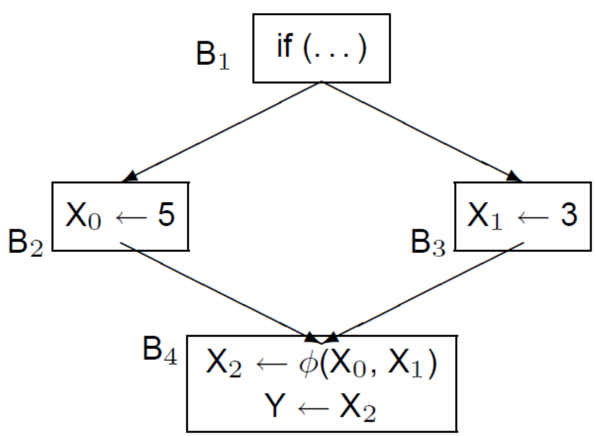
rename the φ-operand when processing the predecessor basic block!

Translating Out of SSA Form

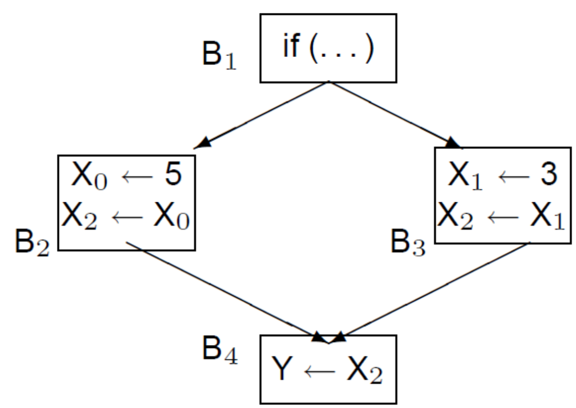
Overview:

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* 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 N0 = B,N1,N2, ...,Nk = Y such that Y postdominates N1 . . .Nk, 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 I

```
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
}
```

For Ex. 1, we could do constant propagation and condition evaluation separately, and repeat until no changes. This separate approach is not sufficient for Ex. 3.

Conditional Constant Propagation

Advantage:

Simultaneously finds constants + eliminates infeasible branches.

Optimistic

Assume every variable may be constant ($\top$), until proven otherwise.

Pessimistic ≡ initially assume nothing is constant ($\perp$).

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)