

Building SSA Form

Slides mostly based on Keith Cooper's set of slides
(COMP 512 class at Rice University, Fall 2002).
Used with kind permission.

Why have SSA?

SSA-form

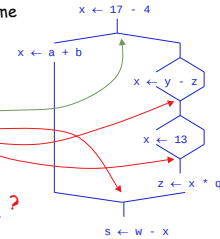
- Each name is defined exactly once
- Each use refers to exactly one name

What's hard?

- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

Building SSA Form

- Insert ϕ -functions at birth points ?
- Rename all values for uniqueness

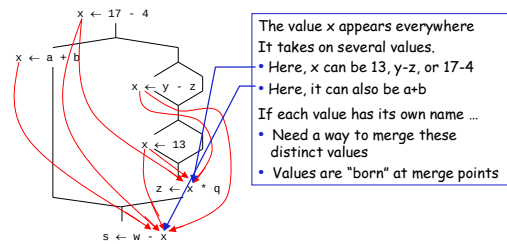


KT2, 2004

* 2

Birth Points (a notion due to Tarjan)

Consider the flow of values in this example

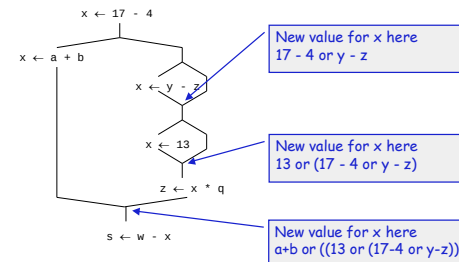


KT2, 2004

* 3

Birth Points (cont)

Consider the flow of values in this example

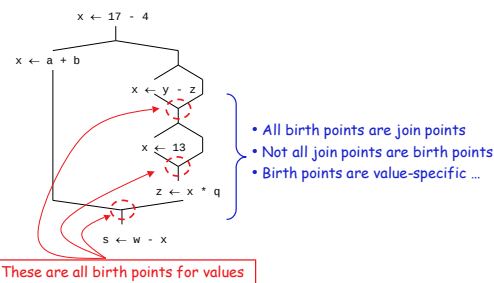


KT2, 2004

4

Birth Points (cont)

Consider the flow of values in this example



KT2, 2004

5

Static Single Assignment Form

SSA-form

- Each name is defined exactly once
- Each use refers to exactly one name

What's hard

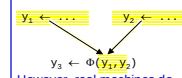
- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

Building SSA Form

- Insert ϕ -functions at birth points
- Rename all values for uniqueness

A ϕ -function is a special kind of copy that selects one of its parameters.

The choice of parameter is governed by the CFG edge along which control reached the current block.



KT2, 2004

* 6

SSA Construction Algorithm (High-level sketch)

1. Insert Φ -functions
2. Rename values

... that's all ...

... of course, there is some bookkeeping to be done ...

KT2, 2004

* 7

SSA Construction Algorithm (Less high-level)

1. Insert Φ -functions at every join for every name
2. Solve *reaching definitions*
3. Rename each use to the def that reaches it (will be unique)

KT2, 2004

8

Reaching Definitions

Domain is $\{\text{DEFINITIONS}\}$, same as number of operations

The equations

$$\text{REACHES}(n_0) = \emptyset$$

$$\text{REACHES}(n) = \bigcup_{p \in \text{preds}(n)} \text{DEFOUT}(p) \cup (\text{REACHES}(p) \cap \text{SURVIVED}(p))$$

- $\text{REACHES}(n)$ is the set of definitions that reach block n
- $\text{DEFOUT}(n)$ is the set of definitions in n that reach the end of n
- $\text{SURVIVED}(n)$ is the set of defs not obscured by a new def in n

Computing $\text{REACHES}(n)$

- Use any data-flow method (i.e., the iterative method)
- This particular problem has a very-fast solution (Zadeck)

F.K. Zadeck, "Incremental data-flow analysis in a structured program editor," *Proceedings of the SIGPLAN 84 Conf. on Compiler Construction*, June, 1984, pages 132-143.

KT2, 2004

* 9

SSA Construction Algorithm (Less high-level)

1. Insert Φ -functions at every join for every name
2. Solve *reaching definitions*
3. Rename each use to the def that reaches it (will be unique)

Builds maximal SSA

What's wrong with this approach

- Too many Φ -functions (precision)
- Too many Φ -functions (space)
- Too many Φ -functions (time)
- Need to relate edges to Φ -functions parameters (bookkeeping)

To do better, we need a more complex approach

KT2, 2004

10

SSA Construction Algorithm (Less high-level)

1. Insert Φ -functions

a.) calculate dominance frontiers

Moderately complex

b.) find global names

for each name, build a list of blocks that define it

c.) insert Φ -functions

$\forall$ global name n

$\forall$ block b in which n is defined

$\forall$ block d in b 's dominance frontier

insert a Φ -function for n in d

add d to b 's list of defining blocks

Creates the iterated dominance frontier

Compute list of blocks where each name is assigned & use as a worklist

This adds to the worklist!

Use a checklist to avoid putting blocks on the worklist twice; keep another checklist to avoid inserting the same Φ -function twice.

KT2, 2004

*11

SSA Construction Algorithm (Less high-level)

2. Rename variables in a pre-order walk over dominator tree

(use an array of stacks, one stack per global name)

Starting with the root block, b 1 counter per name for subscripts

a.) generate unique names for each Φ -function and push them on the appropriate stacks

b.) rewrite each operation in the block

i. Rewrite uses of global names with the current version (from the stack)

ii. Rewrite definition by inventing & pushing new name

c.) fill in Φ -function parameters of successor blocks

d.) recurse on b 's children in the dominator tree

Reset the state

e.) <on exit from block b > pop names generated in b from stacks

Need the end-of-block name for this path

KT2, 2004

*12

Aside on Terminology: Dominators

Definitions

- x **dominates** y if and only if every path from the entry of the control-flow graph to the node for y includes x
- By definition, x **dominates** x
- We associate a Dom set with each node
- $|\text{Dom}(x)| \geq 1$

Immediate dominators

- For any node x , there must be a y in $\text{Dom}(x)$ closest to x
- We call this y the **immediate dominator** of x
- As a matter of notation, we write this as $\text{IDom}(x)$

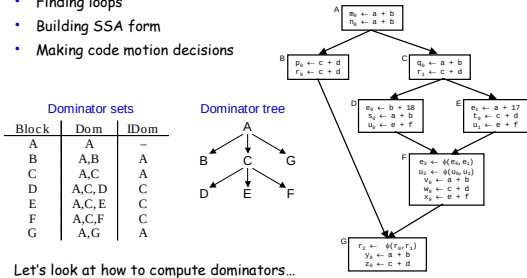
KT2, 2004

13

Dominators (cont)

Dominators have many uses in program analysis & transformation

- Finding loops
- Building SSA form
- Making code motion decisions



Let's look at how to compute dominators...

KT2, 2004

* 14

SSA Construction Algorithm (Low-level detail)

Computing Dominance

- First step in Φ -function insertion computes dominance
- A node n dominates m iff n is on every path from n_0 to m
 - Every node dominates itself
 - n 's **immediate dominator** is its closest dominator, $\text{IDom}(n)$ [†]

$$\text{Dom}(n_0) = \{n_0\}$$

Initially, $\text{Dom}(n) = n$, $\forall n \neq n_0$

$$\text{Dom}(n) = \{n\} \cup (\bigcap_{p \in \text{preds}(n)} \text{Dom}(p))$$

Computing DOM

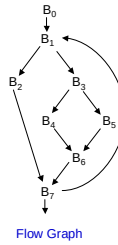
- These equations form a **rapid** data-flow framework
- Iterative algorithm will solve them in $d(G) + 3$ passes
 - Each pass does N unions & E intersections,
 - E is $O(N^2) \Rightarrow O(N^2)$ work

KT2, 2004

[†] $\text{IDom}(n) \neq n$, unless n is n_0 , by convention.

15

Example



Progress of iterative solution for Dom

Iter.	0	1	2	3	4	5	6	7
Dom	0	N	N	N	N	N	N	N
1	0	0.1	0.1.2	0.1.3	0.1.3.4	0.1.3.5	0.1.3.6	0.1.7
2	0	0.1	0.1.2	0.1.3	0.1.3.4	0.1.3.5	0.1.3.6	0.1.7

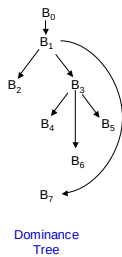
Results of iterative solution for Dom

	0	1	2	3	4	5	6	7
Dom	0	0.1	0.1.2	0.1.3	0.1.3.4	0.1.3.5	0.1.3.6	0.1.7
IDom	0	0	1	1	3	3	3	1

KT2, 2004

*16

Example



Progress of iterative solution for Dom

Iter.	0	1	2	3	4	5	6	7
Dom	0	N	N	N	N	N	N	N
1	0	0.1	0.1.2	0.1.3	0.1.3.4	0.1.3.5	0.1.3.6	0.1.7
2	0	0.1	0.1.2	0.1.3	0.1.3.4	0.1.3.5	0.1.3.6	0.1.7

Results of iterative solution for Dom

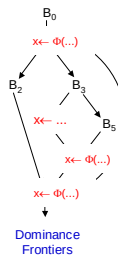
	0	1	2	3	4	5	6	7
Dom	0	0.1	0.1.2	0.1.3	0.1.3.4	0.1.3.5	0.1.3.6	0.1.7
IDom	0	0	1	1	3	3	3	1

There are asymptotically faster algorithms.
With the right data structures, the iterative algorithm can be made faster.
See Cooper, Harvey, and Kennedy.

KT2, 2004

17

Example



Dominance Frontiers & Φ -Function Insertion

- A definition at n forces a Φ -function at m iff $n \in \text{Dom}(m)$ but $n \notin \text{Dom}(p)$ for some $p \in \text{preds}(m)$
- $\text{DF}(n)$ is fringe just beyond region n dominates

	0	1	2	3	4	5	6	7
Dom	0	0.1	0.1.2	0.1.3	0.1.3.4	0.1.3.5	0.1.3.6	0.1.7
DF	-	-	7	7	6	6	7	1

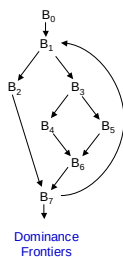
- $\text{DF}(4)$ is $\{6\}$, so $\leftarrow$ in 4 forces a Φ -function in 6
- $\leftarrow$ in 6 forces a Φ -function in $\text{DF}(6) = \{7\}$
- $\leftarrow$ in 7 forces a Φ -function in $\text{DF}(7) = \{1\}$
- $\leftarrow$ in 1 forces a Φ -function in $\text{DF}(1) = \emptyset$ (halt)

For each assignment, we insert the Φ -functions

KT2, 2004

* 18

Example



Computing Dominance Frontiers

- Only join points are in $DF(n)$ for some n
- Leads to a simple, intuitive algorithm for computing dominance frontiers
 - For each join point x (i.e., $|preds(x)| > 1$)
 - For each CFG predecessor of x
 - Run up to $IDOM(x)$ in the dominator tree, adding x to $DF(n)$ for each n between x and $IDOM(x)$
- For some applications, we need post-dominance, the post-dominator tree, and reverse dominance frontiers, $RDF(n)$
 - > Just dominance on the reverse CFG
 - > Reverse the edges & add unique exit node
- We will use these in dead code elimination

	0	1	2	3	4	5	6	7
DOM	0	0.1	0.1.2	0.1.3	0.1.3.4	0.1.3.5	0.1.3.6	0.1.7
DF	—	—	7	7	6	6	7	1

KT2, 2004

* 19

SSA Construction Algorithm (Reminder)

1. Insert Φ -functions at every join for every name

- calculate dominance frontiers
- find global names
for each name, build a list of blocks that define it
- insert Φ -functions
 - $\forall$ global name n
 - $\forall$ block b in which n is defined
 - $\forall$ block d in b 's dominance frontier
 - insert a Φ -function for n in d
 - add d to n 's list of defining blocks

KT2, 2004

* 20

SSA Construction Algorithm

Finding global names

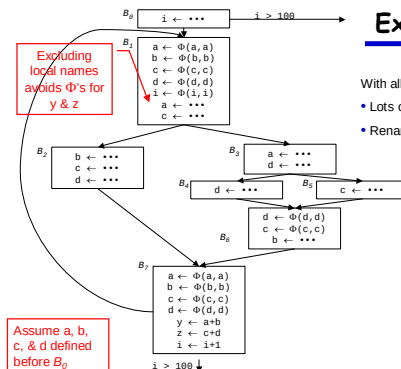
- Different between two forms of SSA
 - Minimal uses all names
 - Semi-pruned SSA uses names that are *live* on entry to some block
 - > Shrinks name space & number of Φ -functions
 - > Pays for itself in compile-time speed
- For each "global name", need a list of blocks where it is defined
 - > Drives Φ -function insertion
 - > b defines x implies a Φ -function for x in every $c \in DF(b)$

Pruned SSA adds a test to see if x is live at insertion point

KT2, 2004

21

Example



With all the Φ -functions

- Lots of new ops
- Renaming is next

SSA Construction Algorithm (Less high-level)

2. Rename variables in a pre-order walk over dominator tree (use an array of stacks, one stack per global name)

Starting with the root block, b 1 counter per name for subscripts

- generate unique names for each Φ -function and push them on the appropriate stacks
- rewrite each operation in the block
 - Rewrite uses of global names with the current version (from the stack)
 - Rewrite definition by inventing & pushing new name
- fill in Φ -function parameters of successor blocks
- recurse on b 's children in the dominator tree
- on exit from block b : pop names generated in b from stacks

Reset the state

Need the end-of-block name for this path

KT2, 2004

23

SSA Construction Algorithm (Less high-level)

Adding all the details ...

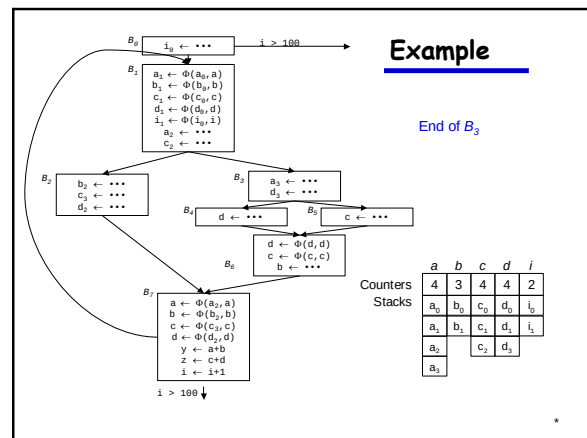
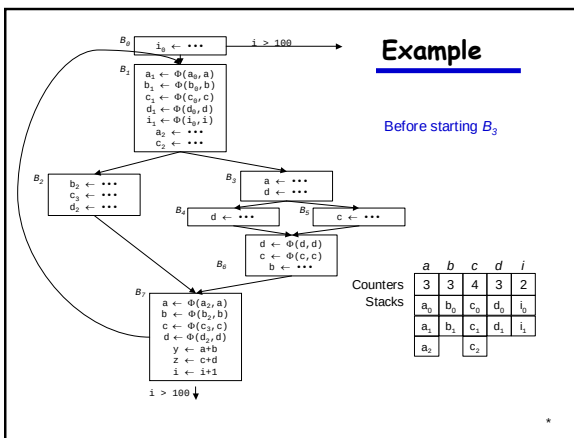
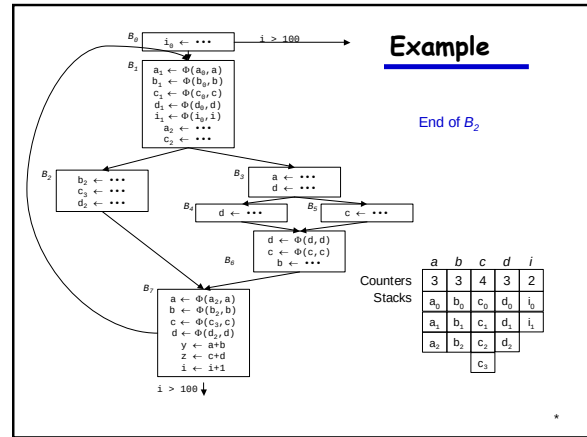
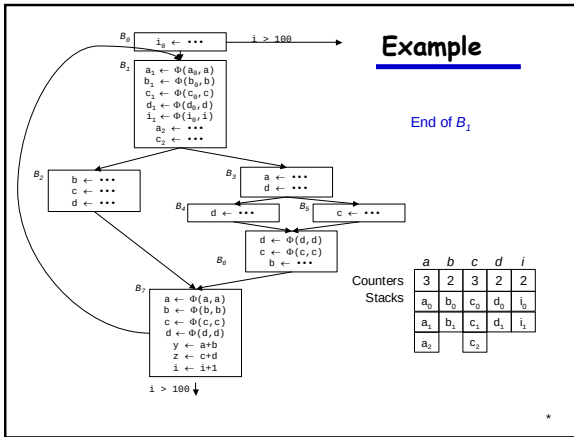
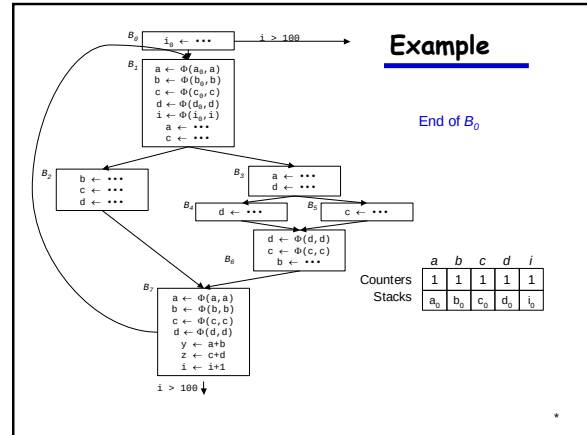
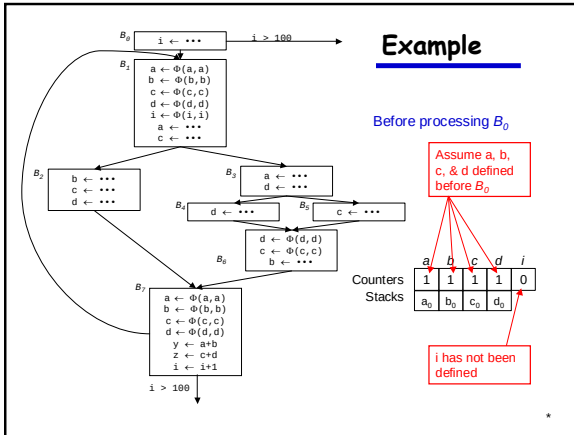
```
for each global name i
  counter[i] ← 0
  stack[i] ← ∅
  call Rename(n0)
```

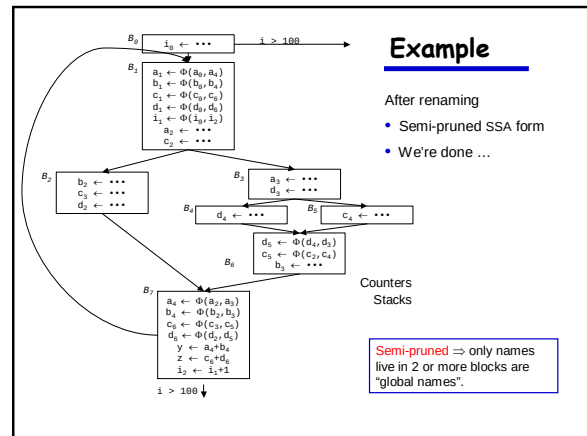
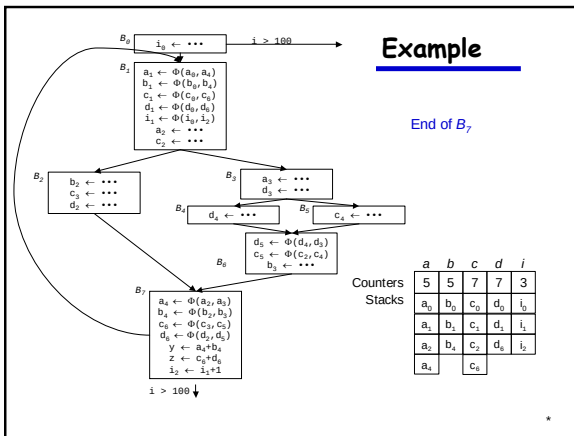
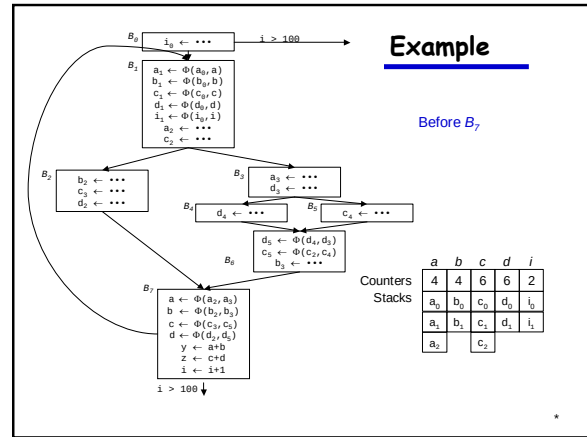
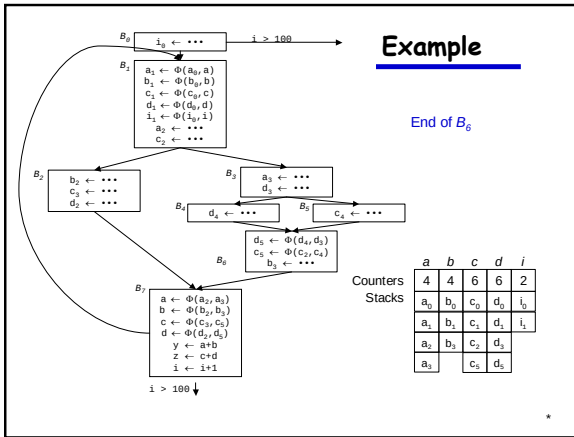
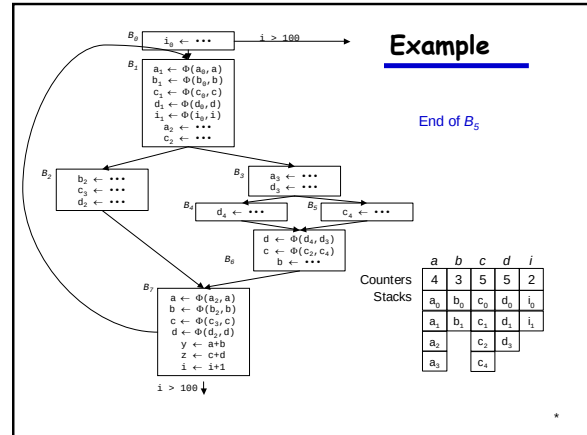
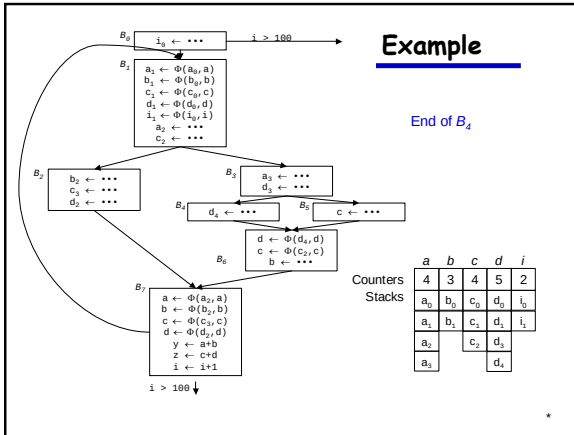
```
NewName(n)
  i ← counter[n]
  counter[n] ← counter[n] + 1
  push ni onto stack[n]
  return ni
```

```
Rename(b)
  for each  $\Phi$ -function in  $b$ ,  $x \leftarrow \Phi(\dots)$ 
    rename  $x$  as  $NewName(x)$ 
  for each operation " $x \leftarrow y \text{ op } z$ " in  $b$ 
    rewrite  $y$  as  $top(stack[y])$ 
    rewrite  $z$  as  $top(stack[z])$ 
    rewrite  $x$  as  $NewName(x)$ 
  for each successor of  $b$  in the CFG
    rewrite appropriate  $\Phi$  parameters
  for each successor  $s$  of  $b$  in dom. tree
    Rename(s)
  for each operation " $x \leftarrow y \text{ op } z$ " in  $b$ 
    pop(stack[x])
```

KT2, 2004

24





SSA Construction Algorithm (Pruned SSA)

What's this "pruned SSA" stuff?

- Minimal SSA still contains extraneous Φ -functions
- Inserts some Φ -functions where they are dead
- Would like to avoid inserting them

Two ideas

- **Semi-pruned SSA**: discard names used in only one block
 - > Significant reduction in total number of Φ -functions
 - > Needs only local liveness information (*cheap to compute*)
- **Pruned SSA**: only insert Φ -functions where their value is live
 - > Inserts even fewer Φ -functions, but costs more to do
 - > Requires global live variable analysis (*more expensive*)

In practice, both are simple modifications to step 1.

KT2, 2004

37

SSA Construction Algorithm

We can improve the stack management

- Push at most one name per stack per block (save push & pop)
- Thread names together by block
- To pop names for block b , use b 's thread

This is another good use for a scoped hash table

- Significant reductions in pops and pushes
- Makes a minor difference in SSA construction time
- Scoped table is a clean, clear way to handle the problem

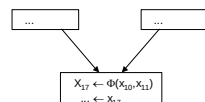
KT2, 2004

38

SSA Deconstruction

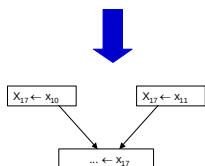
At some point, we need executable code

- Few machines implement Φ operations
- Need to fix up the flow of values



Basic idea

- Insert copies Φ -function pred's
- Simple algorithm
 - > Works in most cases
- Adds lots of copies
 - > Most of them coalesce away



KT2, 2004

* 39