Building SSA Form

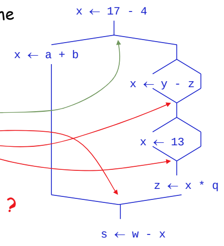
Slides mostly based on Keith Cooper's set of slides
(COMP 512 class at Rice University, Fall 2002).
Used with kind permission.

Why have SSA?

SSA-form
- Each name is defined exactly once, thus
- Each use refers to exactly one name

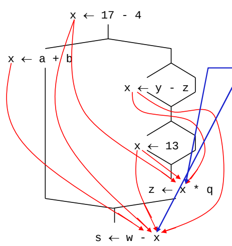
What's hard?
- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

Building SSA Form
- Insert Φ-functions at birth points ?
- Rename all values for uniqueness



Birth Points (*a notion due to Tarjan*)

Consider the flow of values in this example

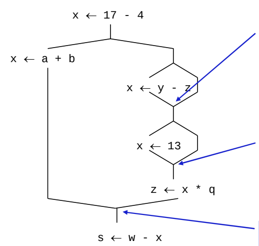


The value x appears everywhere
It takes on several values.
- Here, x can be 13, y-z, or 17-4
- Here, it can also be a+b

If each value has its own name …
- Need a way to merge these distinct values
- Values are "born" at merge points

Birth Points (cont)

Consider the flow of values in this example



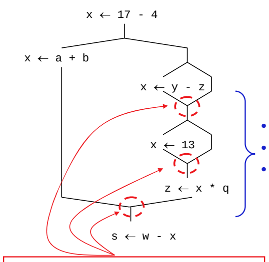
New value for x here
17 - 4 or y - z

New value for x here
13 or (17 - 4 or y - z)

New value for x here
a+b or ((13 or (17-4 or y-z))

Birth Points (cont)

Consider the flow of values in this example



- All birth points are join points
- Not all join points are birth points
- Birth points are value-specific …

These are all birth points for values

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 a move instruction that selects one of its parameters.

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

$y_1 \leftarrow \ldots$ $y_2 \leftarrow \ldots$

$y_3 \leftarrow \Phi(y_1, y_2)$

However, real machines do not implement a Φ-function in hardware.

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 …

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*)

Reaching Definitions

Domain is |DEFINITIONS|, same as number of operations

The equations

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

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

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

Computing 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.

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

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

     Compute list of blocks where each name is assigned. Use this list as the worklist.

     ∀ global name n
       ∀ block b in which n is defined
         ∀ block d in b's dominance frontier
           insert a Φ-function for n in d
           add d to n's list of defining blocks

     This adds to the worklist !

     Creates the iterated dominance frontier

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

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)

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

   Reset the state

   Need the end-of-block name for this path

Aside on Terminology: Dominators

Definitions

x <u>dominates</u> *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* <u>dominates</u> *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 Dom(*x*) such that y is closest to *x*
- We call this *y* the <u>immediate dominator</u> of *x*
- As a matter of notation, we write this as IDom(*x*)
- By convention, IDom(x_0) is not defined for the entry node x_0

Dominators (cont)

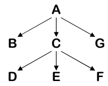
Dominators have many uses in program analysis & transformation

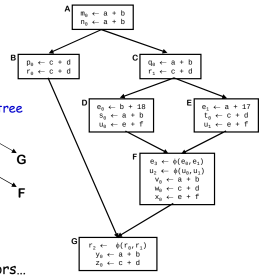
- Finding loops
- Building SSA form
- Making code motion decisions

Dominator sets

Block	Dom	IDom
A	A	–
B	A,B	A
C	A,C	A
D	A,C,D	C
E	A,C,E	C
F	A,C,F	C
G	A,G	A

Dominator tree





Let's look at how to compute dominators…

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 <u>immediate dominator</u> is its closest dominator, IDOM(*n*)†

$$\text{DOM}(n_0) = \{\, n_0 \,\}$$

Initially, Dom(n) = *N*, $\forall\, n \neq n_0$

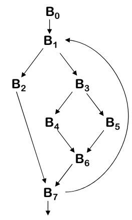
$$\text{DOM}(n) = \{\, n \,\} \cup \left(\cap_{p \in preds(n)} \text{DOM}(p)\right)$$

Computing DOM

- These equations form a <u>rapid</u> 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

†IDOM(*n*) ≠ *n*, unless *n* is n_0, by convention.

Example



Flow Graph

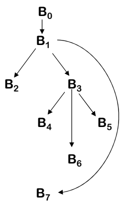
Progress of iterative solution for DOM

Iter-ation	DOM(*n*) 0	1	2	3	4	5	6	7
0	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	1	1	3	3	3	1

Example



Dominance Tree

Progress of iterative solution for DOM

Iter-ation	DOM(*n*) 0	1	2	3	4	5	6	7
0	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

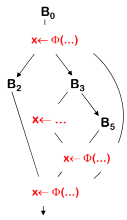
	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
IDOM		0	1	1	3	3	3	1

There are asymptotically faster algorithms.

With the right data structures, the iterative algorithm can be made faster.

See Cooper, Harvey, and Kennedy.

Example



Dominance Frontiers

Dominance Frontiers & Φ-Function Insertion

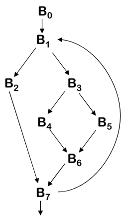
- A definition at *n* forces a Φ-function at *m* iff *n* ∉ DOM(*m*) but *n* ∈ DOM(p) for some *p* ∈ *preds(m)*
- DF(*n*) is fringe just beyond region *n* dominates

	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

- DF(4) is {6}, so ← in 4 forces a Φ-function in 6
- ← in 6 forces a Φ-function in DF(6) = {7}
- ← in 7 forces a Φ-function in DF(7) = {1}
- ← in 1 forces a Φ-function in DF(1) = Ø (*halt*)

For each assignment, we insert the Φ-functions

Example



Dominance Frontiers

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* <u>between</u> *x* and IDOM(*x*)

	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	–	1	7	7	6	6	7	1

- For some applications, we need <u>post-dominance</u>, the <u>post-dominator tree</u>, and <u>reverse dominance frontiers</u>, RDF(*n*)
  - > Just dominance on the reverse CFG
  - > Reverse the edges & add unique exit node
- We will use these in dead code elimination using SSA

SSA Construction Algorithm (Reminder)

1. Insert Φ-functions at some join points

   a.) calculate dominance frontiers

   b.) find global names — Needs a little more detail

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

   c.) insert Φ-functions

   ∀ global name *n*
   - ∀ block *b* in which *n* is defined
     - ∀ block *d* in *b*'s dominance frontier
       - insert a Φ-function for *n* in *d*
       - add *d* to *n*'s list of defining blocks

SSA Construction Algorithm

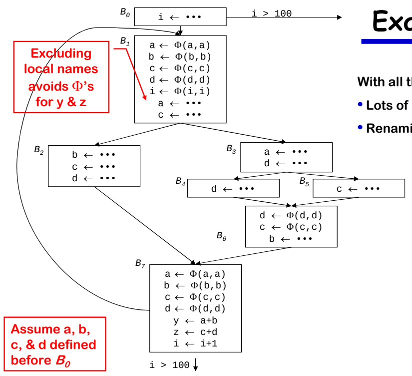
Finding global names

- 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*)

Otherwise, we do not need a Φ-function

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

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)

   Staring 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

SSA Construction Algorithm (Less high-level)

Adding all the details ...

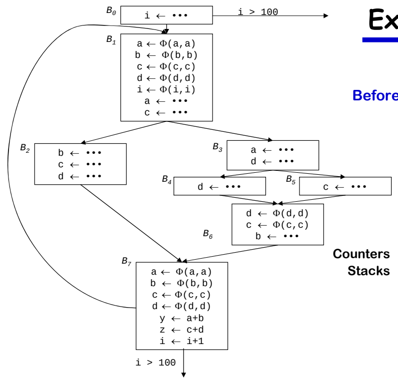
```
for each global name i
    counter[i] ← 0
    stack[i] ← Ø
call Rename(n0)
```

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

```
Rename(b)
    for each Φ-function in b, x ← Φ(…)
        rename x as NewName(x)

    for each operation "x ← y 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 Φ parameters

    for each successor s of b in dom. tree
        Rename(s)

    for each operation "x ← y op z" in b
        pop(stack[x])
```

Example

Before processing B_0



Assume a, b, c, & d defined before B_0

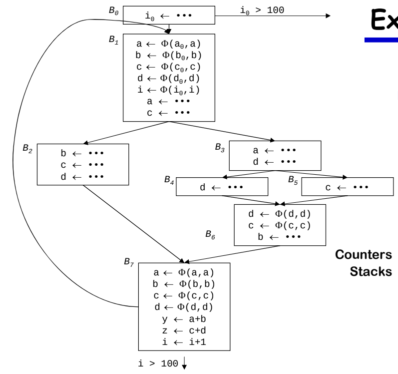
	a	*b*	*c*	*d*	*i*
Counters	1	1	1	1	0
Stacks	a_0	b_0	c_0	d_0	

i has not been defined

*

Example

End of B_0

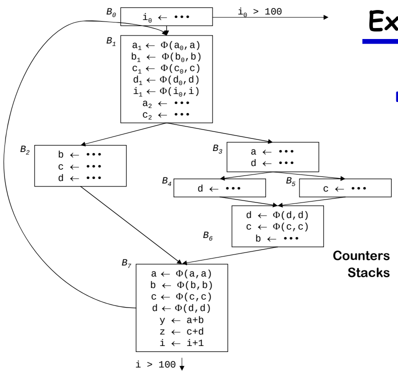


	a	*b*	*c*	*d*	*i*
Counters	1	1	1	1	1
Stacks	a_0	b_0	c_0	d_0	i_0

*

Example

End of B_1

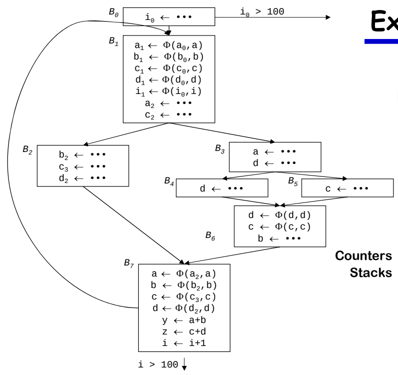


	a	*b*	*c*	*d*	*i*
Counters	3	2	3	2	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

*

Example

End of B_2

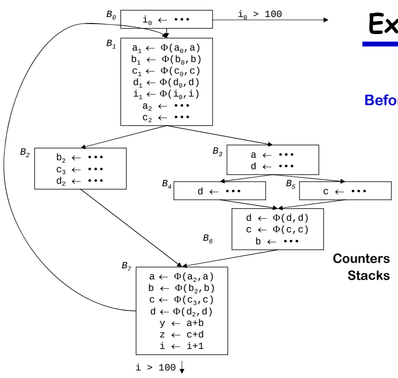


	a	*b*	*c*	*d*	*i*
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_2	c_2	d_2	
			c_3		

*

Example

Before starting B_3

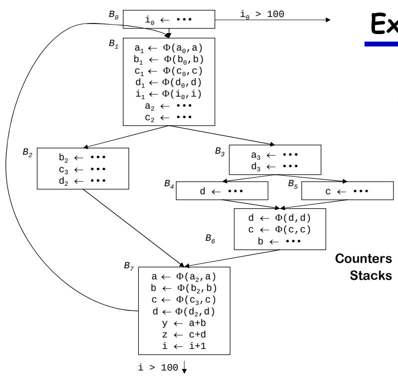


	a	*b*	*c*	*d*	*i*
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

*

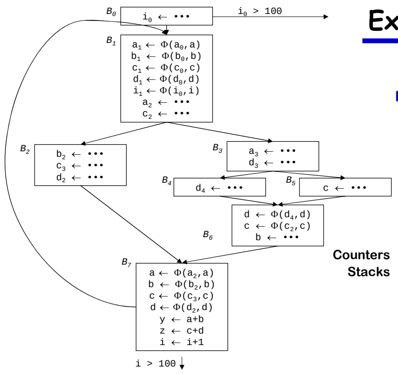
Example

End of B_3



	a	*b*	*c*	*d*	*i*
Counters	4	3	4	4	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3				

*

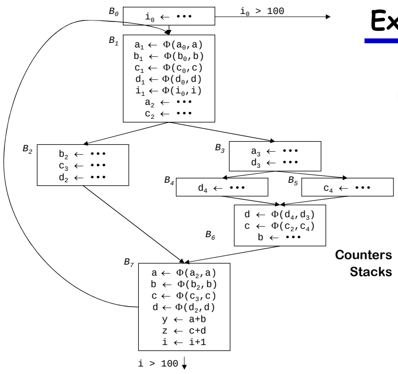


Example

End of B_4

	a	*b*	*c*	*d*	*i*
Counters	4	3	4	5	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3			d_4	

*

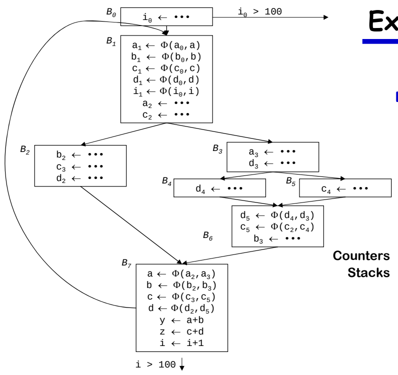


Example

End of B_5

	a	*b*	*c*	*d*	*i*
Counters	4	3	5	5	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_3	
	a_3		c_4		

*

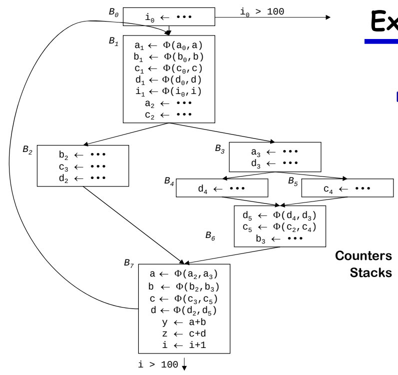


Example

End of B_6

	a	*b*	*c*	*d*	*i*
Counters	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_3	c_2	d_3	
	a_3		c_5	d_5	

*

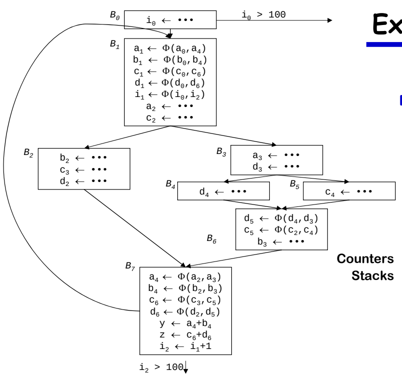


Example

Before B_7

	a	*b*	*c*	*d*	*i*
Counters	4	4	6	6	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

*

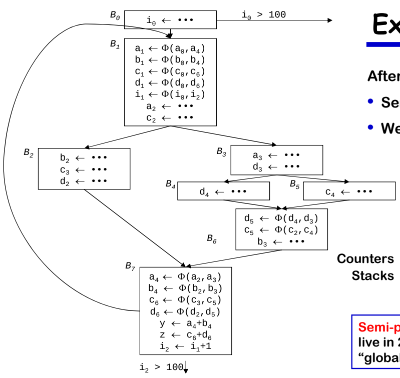


Example

End of B_7

	a	*b*	*c*	*d*	*i*
Counters	5	5	7	7	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_4	c_2	d_6	i_2
	a_4		c_6		

*



Example

After renaming

- Semi-pruned SSA form
- We're done …

Counters
Stacks

Semi-pruned ⇒ only names live in 2 or more blocks are “global names”.

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.

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

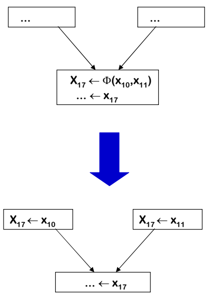
This is another good use for a scoped hash table

- 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

SSA Deconstruction

At some point, we need executable code

- Real machines do not implement Φ functions
- 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