CSE P 501 – Compilers

SSA

Hal Perkins

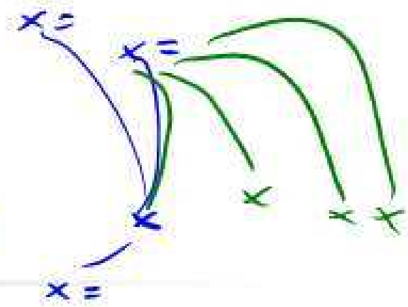
Autumn 2009

Agenda

- Overview of SSA IR
  - Constructing SSA graphs
  - SSA-based optimizations
  - Converting back from SSA form

- Source: Appel ch. 19, also an extended discussion in Cooper-Torczon sec. 9.3

Def-Use (DU) Chains



- Common dataflow analysis problem: Find all sites where a variable is used, or find the definition site of a variable used in an expression
- Traditional solution: def-use chains – additional data structure on the dataflow graph
  - Link each statement defining a variable to all statements that use it
  - Link each use of a variable to its definition

DU-Chain Drawbacks

- Expensive: if a typical variable has N uses and M definitions, the total cost is O(N * M)
  - Would be nice if cost were proportional to the size of the program
- Unrelated uses of the same variable are mixed together
  - Complicates analysis

```
for (k= —){
  a[k]=0
}
for (k= ){
  b[k]= —
}
```

SSA: Static Single Assignment

- IR where each variable has only one definition in the program text
  - This is a single *static* definition, but it may be in a loop that is executed dynamically many times

a_{17} =

SSA in Basic Blocks

We've seen this before when looking at value numbering

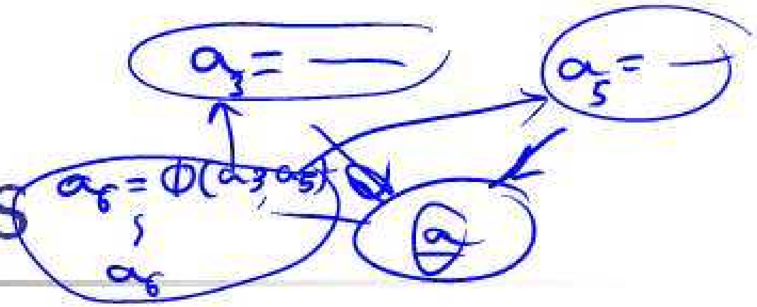
- Original

  a := x + y
  b := a – 1
  a := y + b
  b := x * 4
  a := a + b

- SSA

  a_1 := x + y
  b_1 := a_1 – 1
  a_2 := y + b_1
  b_2 := x * 4
  a_3 := a_2 + b_2

Merge Points

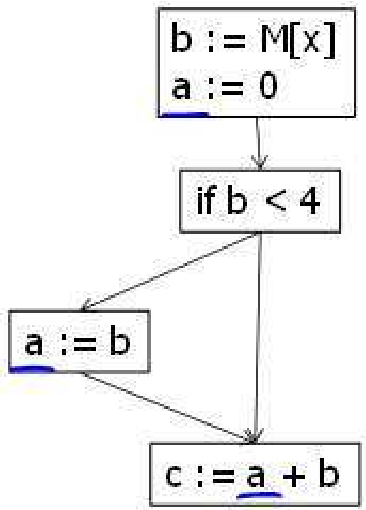


- The issue is how to handle merge points
- Solution: introduce a Φ-function

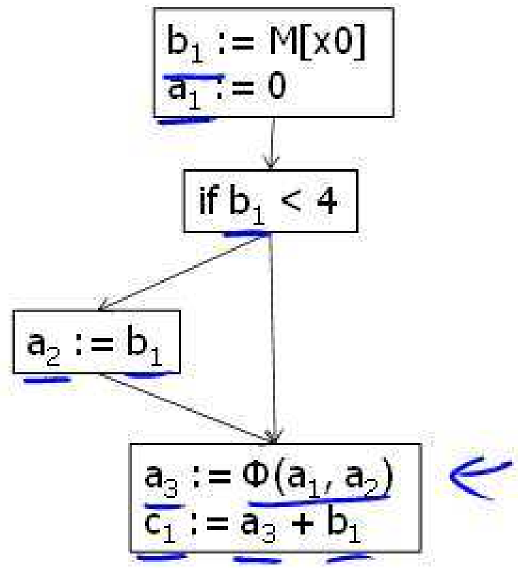
  $a_3 := \Phi(a_1, a_2)$
- Meaning: a_3 is assigned either a_1 or a_2 depending on which control path is used to reach the Φ-function

Example

Original



SSA



How Does Φ "Know" What to Pick?

- It doesn't
  - When we translate the program to executable form, we can add code to copy either value to a common location on each incoming edge
  - For analysis, all we may need to know is the connection of uses to definitions – no need to "execute" anything

Example With Loop

Original

live c

```
a := 0
```

```
b := a + 1
c := c + b
a := b * 2
if a < N
```

```
return c
```

c_0

SSA

$a_1 := 0$

$a_3 := \Phi(a_1, a_2)$
$\cancel{b_1 := \Phi(b_0, b_2)}$
$c_2 := \Phi(c_0, c_1)$
$b_2 := a_3 + 1$
$c_1 := c_2 + b_2$
$a_2 := b_2 * 2$
if $a_2 < N$

return c_1

Notes:

- a_0, b_0, c_0 are initial values of a, b, c on block entry
- b_1 is dead – can delete later
- c is live on entry – either input parameter or uninitialized

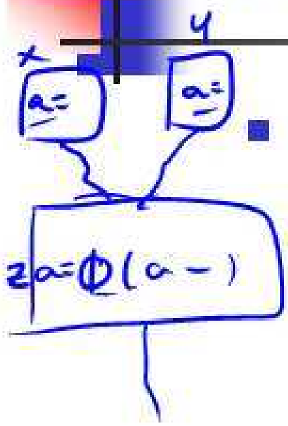
Converting To SSA Form

- Basic idea
  - First, add Φ-functions
  - Then, rename all definitions and uses of variables by adding subscripts

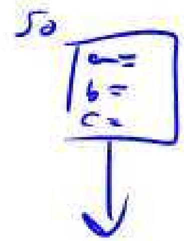
Inserting Φ-Functions

- Could simply add Φ-functions for every variable at every join point(!)
- But
  - Wastes *way* too much space and time
  - Not needed

Path-convergence criterion



- Insert a Φ-function for variable a at point z when
  - There are blocks x and y, both containing definitions of a, and x ≠ y
  - There are nonempty paths from x to z and from y to z
  - These paths have no common nodes other than z
  - z is not in both paths prior to the end (it may appear in one of them)



Details

- The start node of the flow graph is considered to define every variable (even if to "undefined")
- Each Φ-function itself defines a variable, so we need to keep adding Φ-functions until things converge

Dominators and SSA

$x_i := -$

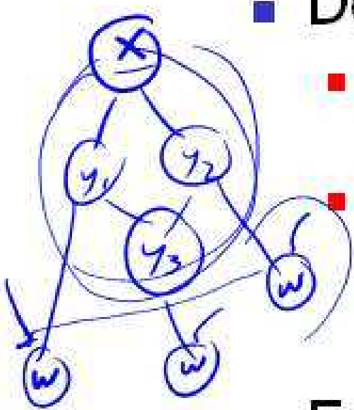
$x := \Phi - x_i -$

- One property of SSA is that definitions dominate uses; more specifically:
  - If $x := \Phi(\ldots,x_i,\ldots)$ in block n, then the definition of x dominates the ith predecessor of n
  - If x is used in a non-Φ statement in block n, then the definition of x dominates block n

Dominance Frontier (1)

- To get a practical algorithm for placing Φ-functions, we need to avoid looking at all combinations of nodes leading from x to y
- Instead, use the dominator tree in the flow graph

Dominance Frontier (2)



- Definitions
  - x *strictly dominates* y if x dominates y and x $\neq$ y
  - The *dominance frontier* of a node x is the set of all nodes w such that x dominates a predecessor of w, but x does not strictly dominate w
- Essentially, the dominance frontier is the border between dominated and undominated nodes

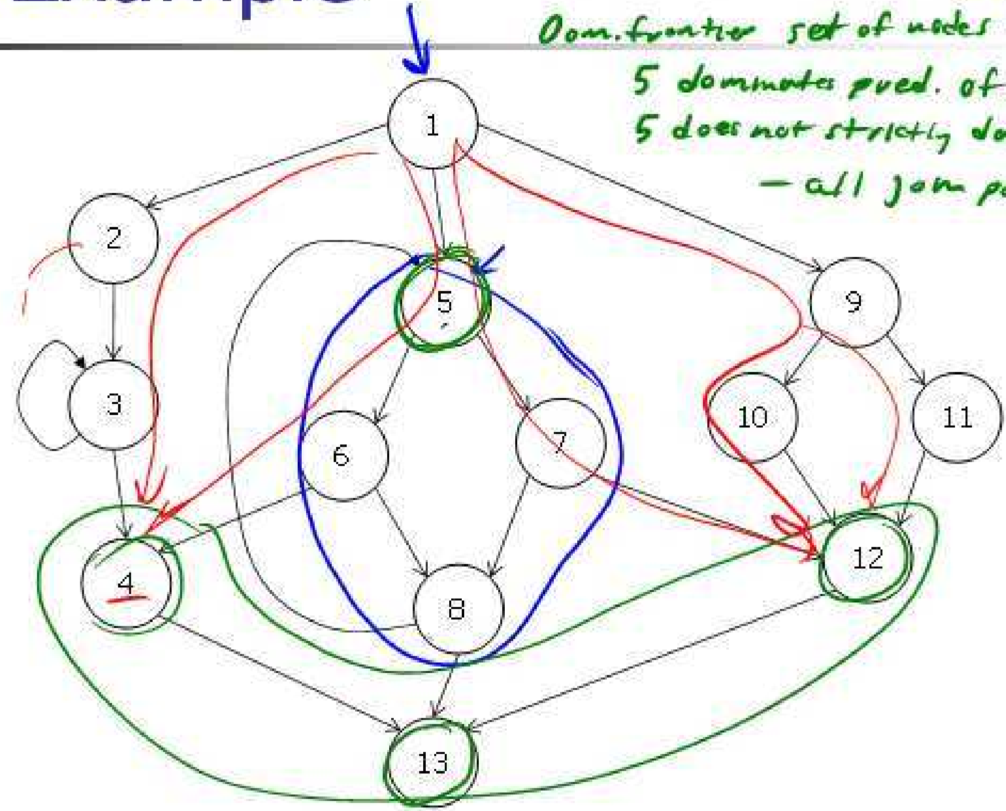
Example

Dom. frontier of 5

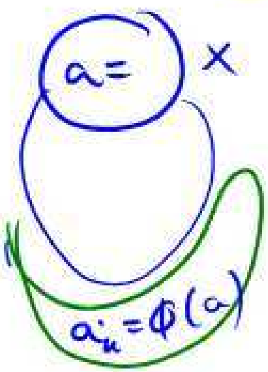
5 dominates 6, 7, 8, 5

5 strictly dominates 6, 7, 8

Dom. frontier set of nodes x st.
5 dominates pred. of x but
5 does not strictly dom. x
– all join points



Dominance Frontier Cirterion



- If a node x contains the definition of variable a, then every node in the dominance frontier of x needs a Φ-function for a
  - Since the Φ-function itself is a definition, this needs to be iterated until it reaches a fixed-point
- Theorem: this algorithm places exactly the same set of Φ-functions as the path criterion given previously

Placing Φ-Functions: Details

- The basic steps are:
  1. Compute the dominance frontiers for each node in the flowgraph
  2. Insert just enough Φ-functions to satisfy the criterion. Use a worklist algorithm to avoid reexamining nodes unnecessarily
  3. Walk the dominator tree and rename the different definitions of variable a to be a_1, a_2, a_3, …

Efficient Dominator Tree Computation

- Goal: SSA makes optimizing compilers faster since we can find definitions/uses without expensive bit-vector algorithms
- So, need to be able to compute SSA form quickly
- Computation of SSA from dominator trees are efficient, but…

Lengauer-Tarjan Algorithm

- Iterative set-based algorithm for finding dominator trees is slow in worst case
- Lengauer-Tarjan is near linear time
  - Uses depth-first spanning tree from start node of control flow graph
  - See books for details

SSA Optimizations

- Given the SSA form, what can we do with it?
- First, what do we know? (i.e., what information is kept in the SSA graph?)

SSA Data Structures

- Statement: links to containing block, next and previous statements, variables defined, variables used.
  - Statement kinds are: ordinary, Φ-function, fetch, store, branch
- Variable: link to definition (statement) and use sites
- Block: List of contained statements, ordered list of predecessors, successor(s)

Dead-Code Elimination

$v_{17} = x_{35}$

- A variable is live iff its list of uses is not empty(!)
- Algorithm to delete dead code:

```
while there is some variable v with no uses
    if the statement that defines v has no
        other side effects, then delete it
```

  - Need to remove this statement from the list of uses for its operand variables – which may cause those variables to become dead

Simple Constant Propagation

- If c is a constant in $v := c$, any use of v can be replaced by c
  - Then update every use of v to use constant c
- If the c_i's in $v := \Phi(c_1, c_2, \ldots, c_n)$ are all the same constant c, we can replace this with $v := c$
- Can also incorporate copy propagation, constant folding, and others in the same worklist algorithm

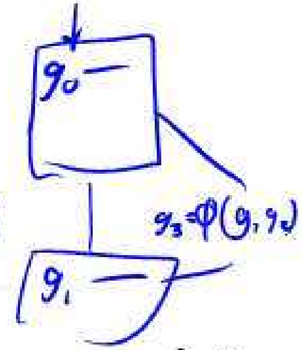
Simple Constant Propagation

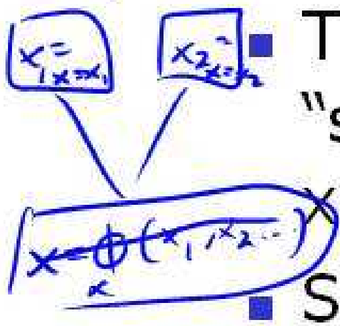
```
W := list of all statements in SSA program
while W is not empty
  remove some statement S from W
  if S is v:=Φ(c, c, ..., c), replace S with v:=c
  if S is v:=c
     delete S from the program
     for each statement T that uses v
          substitute c for v in T
          add T to W
```

Converting Back from SSA

- Unfortunately, real machines do not include a Φ instruction
- So after analysis, optimization, and transformation, need to convert back to a "Φ-less" form for execution

Translating Φ-functions





- The meaning of $x := \Phi(x_1, x_2, \ldots, x_n)$ is "set $x := x_1$ if arriving on edge 1, set $x := x_2$ if arriving on edge 2, etc."
- So, for each i, insert $x := x_i$ at the end of predecessor block i
- Rely on copy propagation and coalescing in register allocation to eliminate redundant moves

SSA Wrapup

- More details in recent compiler books (but not the new dragon book!)
- Allows efficient implementation of many optimizations
- Used in many new compiler (e.g. llvm) & retrofitted into many older ones (gcc)
- Not a silver bullet – some optimizations still need non-SSA forms