Using Static Single Assignment Form

Last Time
- Constructing SSA form

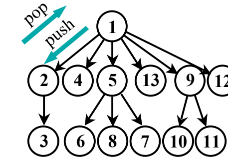
Today
- Finish naming algorithm
- Transforming from SSA
- Using SSA for program optimization
  - Dead-code elimination
  - Constant propagation
  - Register allocation
  - Copy propagation
  - Induction variables

Variable Renaming (cont)

Data Structures
- Stacks[v] ∀v
  Holds the subscript of most recent definition of variable v, initially empty
- Counters[v] ∀v
  Holds the current number of assignments to variable v; initially 0

Auxiliary Routine

procedure GenName(variable v)
 i := Counters[v]
 push i onto Stacks[v]
 Counters[v] := i + 1



Use the Dominance Tree to remember the most recent definition of each variable

Variable Renaming Algorithm

procedure Rename(block b)
 if b previously visited return
 for each ϕ-function p in b
  GenName(LHS(p)) and replace v with v_i, where i=Top(Stack[v])
 for each statement s in b (in order)
  for each variable v ∈ RHS(s)
   replace v by v_i, where i = Top(Stacks[v])
  for each variable v ∈ LHS(s)
   GenName(v) and replace v with v_i, where i=Top(Stack[v])
 for each s ∈ succ(b) (in CFG)
  j ← position in s's ϕ-function corresponding to block b
  for each ϕ-function p in s
   replace the jth operand of RHS(p) by v_i, where i = Top(Stack[v])
 for each s ∈ child(b) (in DT)
  Rename(s)
 for each ϕ-function or statement t in b
  for each v_i ∈ LHS(t)
   Pop(Stack[v])

Call Rename(entry-node)

Φ(, ,)

} Recurse using Depth First Search

} Unwind stack when done with this node

Transformation from SSA Form

Proposal
- Restore original variable names (*i.e.*, drop subscripts)
- Delete all ϕ-functions

Complications
- What if versions get out of order? (simultaneously live ranges)

```
x0 =
x1 =
     = x0
     = x1
```

Alternative
- Perform dead code elimination (to prune ϕ-functions)
- Replace ϕ-functions with copies in predecessors
- Rely on register allocation coalescing to remove unnecessary copies

Dead Code Elimination for SSA

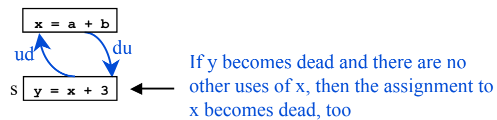
Dead code elimination

while ∃ a variable v with no uses and whose def has no other side effects

Delete the statement s that defines v

for each of s's ud-chains

Delete the corresponding du-chain that points to s



– Contrast this approach with one that uses liveness analysis
  – This algorithm updates information incrementally
  – With liveness, we need to invoke liveness and dead code elimination iteratively until we reach a fixed point

Constant Propagation

Goal

– Discover constant variables and expressions and propagate them forward through the program

Uses

– Evaluate expressions at compile time instead of run time
– Eliminate dead code (*e.g.,* debugging code)
– Improve efficacy of other optimizations (*e.g.,* value numbering and software pipelining)

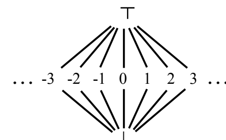
Data-Flow Analysis for Simple Constant Propagation

Simple constant propagation

– V: {All constants} ∪ {⊤,⊥}

– *:
  c * ¨ = c
  c * ⊥ = ⊥
  c * d = ⊥ if c ≠ d
  c * d = c if c = d

Using tuples of lattices



– F:
  – $F_{x\leftarrow c}(In) =$ **c**
  – $F_{x\leftarrow y\oplus z}(In) =$ **if $c_y = In_y$ & $c_z = In_z$, then $c_y \oplus c_z$, else ⊤ or ⊥**
  – . . .

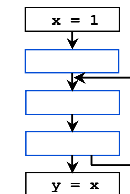
Implementing Simple Constant Propagation

Standard worklist algorithm

– Identifies simple constants
– For each program point, maintains one constant value for each variable
– O(EV) (E is the number of edges in the CFG; V is number of variables)

Problem

– Inefficient, since constants may have to be propagated through irrelevant nodes



Solution

– Exploit a sparse dependence representation (*e.g.,* SSA)

Sparse Simple Constant Propagation

Reif and Lewis algorithm

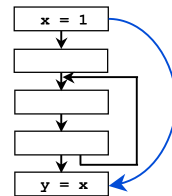
- Identifies simple constants
- Faster than Simple Constants algorithm

SSA edges

- Explicitly connect defs with uses
- How would you do this?

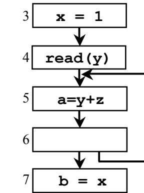
Main Idea

- Iterate over SSA edges instead of over all CFG edges



Sparse Simple Constants Algorithm (Ch. 19 in Appel)

worklist = all statements in SSA
while worklist ≠ ∅
    Remove some statement S from worklist
    if S is x = phi(c,c,...,c) for some constant c
        replace S with v = c
    if S is x=c for some constant c
        delete s from program
        for each statement T that uses v
            substitute c for x in T
            worklist = worklist union {T}



Sparse Simple Constants

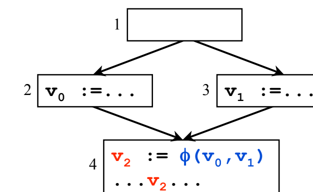
Complexity

- O(E') = O(EV), E' is number of SSA edges
- O(N) in practice

Backward Analyses vs. Forward Analyses

For forward data-flow analysis, at phi node apply meet function

For backward data-flow analysis?



Static Single Information Form (SSI)

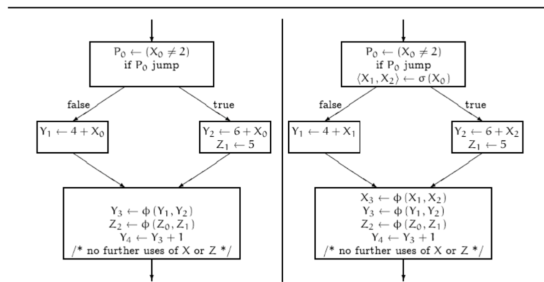


Figure 5.1: A comparison of SSA (left) and SSI (right) forms.

Ananian's Masters Thesis, 1997 MIT

Register Allocation

Constructing the interference graph

- Algorithm 19.17 in Appel
- walk backward from each use until find def for that use
- any defs found along the way cause interference
- when use is in a phi function, only traverse the edge in the flow graph for the relevant predecessor

Is it faster?

- Liveness O(VE), where v is the # of variables and E is # of edges in CFG
- Interference graph construction from Liveness is O(VE) + O(E+I), where I is the size of the interference graph.
- Using SSA, interference graph construction is O(E+I) with certain restrictions.

Copy Propagation

Algorithm

worklist = all statements in SSA
while worklist ≠ ∅
  Remove some statement S from worklist
  if S is x = phi(y) or x = y
    for each statement T that uses x
      replace all use of x with y
      worklist = worklist union {T}
    delete S

Induction Variable Identification

Types of Induction Variables

- **Basic** induction variables
  - Variables that are defined once in a loop by a statement of the form, **`i=i+c`** (or **`i=i*c`**), where **`c`** is a constant integer
- **Derived** induction variables
  - Variables that are defined once in a loop as a linear function of another induction variable
    - $j = c_1 * i + c_2$
    - $j = i / c_1 + c_2$, where c_1 and c_2 are loop invariant

Induction Variable Identification (cont)

Informal SSA-based Algorithm

- Build the SSA representation
- Iterate from innermost CFG loop to outermost loop
  - Find SSA cycles
    - Each cycle **may** be a **basic** induction variable if a variable in a cycle is a function of loop invariants and its value on the current iteration
  - Find **derived** induction variables as functions of loop invariants and basic induction variables

Induction Variable Identification (cont)

Informal SSA-based Algorithm (cont)

- Determining whether a variable is a function of loop invariants and its value on the current iteration
  - The ϕ -function in the cycle will have as one of its inputs a def from inside the loop and a def from outside the loop
  - The def inside the loop will be part of the cycle and will get one operand from the ϕ -function and all others will be loop invariant
  - The operation will be plus, minus, or unary minus

Next Time

Reading

- Value Numbering chapter

Lecture

- Value Numbering