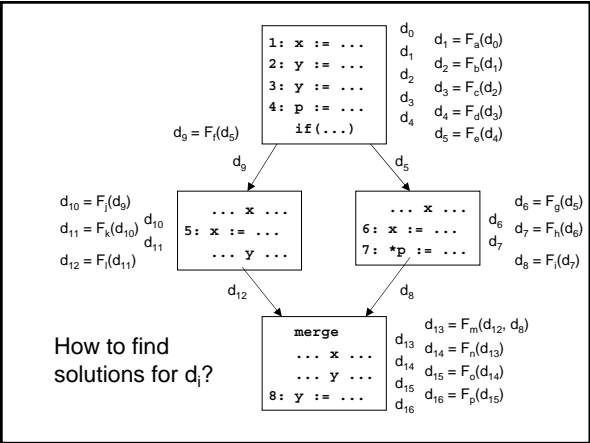


- ◆ New HW on the web
 - > Due Thurs!

- ### Last Time
- ◆ Tour of optimizations
 - ◆ Intro to dataflow
 - > dataflow analysis = dataflow framework + transfer functions

- ### Today
- ◆ Dataflow frameworks in detail
 - > Algorithms
 - > Theory background



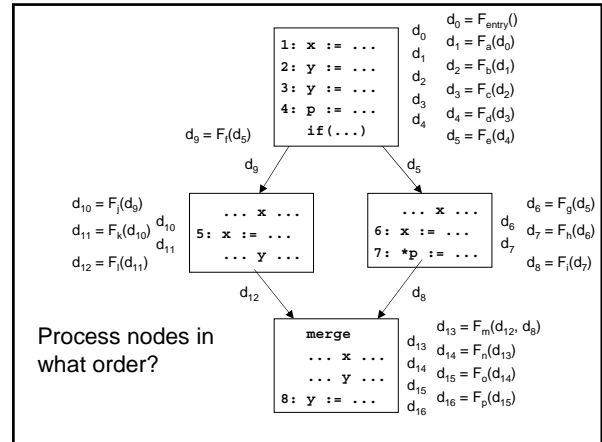
- ### How to find solutions for d_i ?
- ◆ This is a forward problem
 - > Given information flowing *in* to a node, we can determine the information flowing *out* of the node using a transfer function
 - ◆ To solve the global problem:
 - > Propagate information forward through the control flow graph, using the transfer functions
 - ◆ Does this work?

- ### First problem
- ◆ What about the initial conditions?
 - > d_0 is not constrained
 - > so where do we start?
 - ◆ Need to constrain d_0
 - ◆ Two options
 - > Explicitly state entry information
 - > Have an entry node whose transfer function sets the information on entry
 - Doesn't matter if entry node has an incoming edge, its transfer function ignores any input

Entry node



out = { X → S | X ∈ Formals }

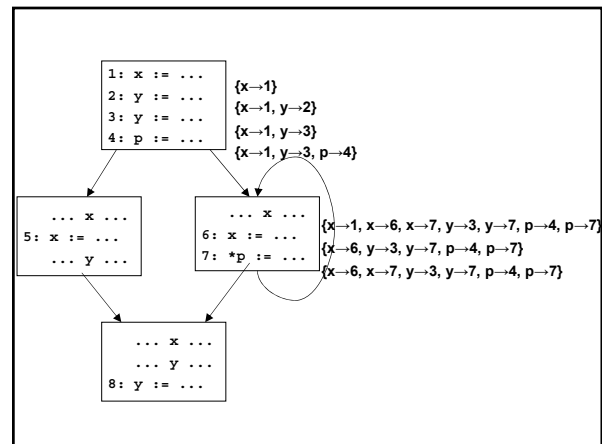
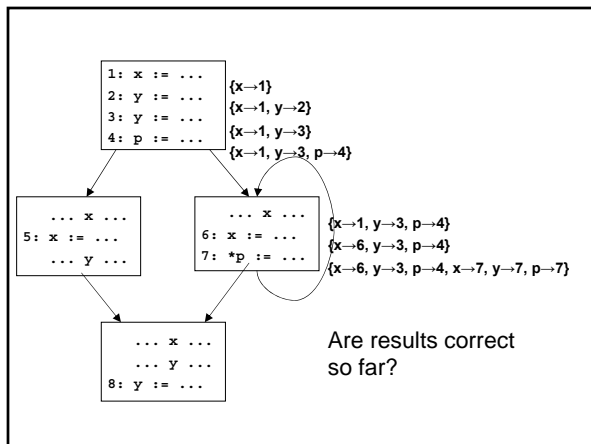


How to find solutions for d_i ?

- ◆ Sort nodes in topological order
 - Each node appears after all of its predecessors
 - Flattens the CFG
 - Topological sort algorithms are fast and easy
- ◆ Then just run the transfer functions for each of the nodes in the topological order

Second problem

- ◆ Only DAGs have a topological order
 - Programs with loops (either explicit or recursive) are not DAGs
- ◆ We'll ignore the loop and see what happens:
 1. Turn CFG into a DAG by removing back-edges
 2. Compute topological order
 3. Run transfer functions in this order



Solution to Loop Problem: Iterate

- ◆ Initialize all d_i to the empty set
- ◆ Store all nodes onto a worklist
 - Worklist stores the nodes whose dataflow equations need to be recomputed
- ◆ Now run this algorithm:
 - While worklist is not empty
 - Remove node n from worklist
 - Apply transfer function for node n
 - Update the appropriate d_i , and add nodes whose inputs have changed back onto worklist
- ◆ What is a worklist, really?
 - Just an optimized way to compute a solution to the set of dataflow equations

Worklist Algorithm

```
m: map from edge to computed value at edge
worklist: work list of nodes

for each edge e in CFG do
  m(e) = 0

for each node n do
  worklist.add(n)

while (!worklist.empty) do
  n = worklist.remove_any;
  info_in = m(n.incoming_edges);
  info_out = F(n, info_in);
  foreach i in info_out do
    if (m(n.outgoing_edges[i]) != info_out[i])
      m(n.outgoing_edges[i]) = info_out[i];
      worklist.add(n.outgoing_edges[i].dest);
```

Issues with Worklist Algorithm

- ◆ Ordering
 - In what order should the original nodes be added to the worklist?
 - What order should nodes be removed from the worklist?
- ◆ Does it terminate?
- ◆ Under what assumptions (if any) does it give correct results?

Order of nodes

- ◆ Order affects performance but not correctness
- ◆ Goal: Maximize performance by minimizing number of transfer function applications
- ◆ Use a worklist that maintains ordering
- ◆ Initial order: Reverse depth-first topological order after removing back-edges
 - Idea: Run transfer functions in something like the order in which the program executes
- ◆ In practice this gets pretty involved...

Termination

- ◆ Do we care? Can we stop the algorithm in the middle and just say we're done?
 - No: We need to run it to completion, otherwise the results are not safe
 - So: Establishing termination is important
- ◆ It is also possible to create analyses that can be stopped at any point, returning a safe result
 - These are "pessimistic" analyses
 - Often, optimistic analyses return better results

Termination

- ◆ Assuming we're doing reaching definitions, let's try to guarantee that the worklist loop terminates, regardless of what the transfer function F does

```
while (!worklist.empty) do
  n = worklist.remove_any;
  info_in = m(n.incoming_edges);
  info_out = F(n, info_in);
  foreach i in info_out do
    if (m(n.outgoing_edges[i]) != info_out[i])
      m(n.outgoing_edges[i]) = info_out[i];
      worklist.add(n.outgoing_edges[i].dest);
```

- ◆ We'll need to revise the algorithm a bit...

Termination

◆ Revised algorithm:

```
while (!worklist.empty) do
  n = worklist.remove_any;
  info_in = m(n.incoming_edges);
  info_out = F(n, info_in);
  foreach i in info_out do
    new_info = m(n.outgoing_edges[i]) U
              info_out[i];
  if (m(n.outgoing_edges[i]) ≠ new_info)
    m(n.outgoing_edges[i]) = new_info;
    worklist.add(n.outgoing_edges[i].dst);
```

Structure of the domain

◆ Termination argument exploits the structure of the domain

- Doesn't involve the transfer functions

◆ In general, it's useful to have a framework that formalizes this structure

- We will use lattices

Bonus Background Material

Relations

◆ A binary relation over a set S is a set $R \subseteq S \times S$

- This definition trivially extends to k-ary relations
- Shorthand: We write "a R b" for $(a,b) \in R$
- Note: S can be infinite

◆ A binary relation R is

- reflexive iff $\forall a \in S . a R a$
- transitive iff $\forall a \in S, b \in S, c \in S . a R b \wedge b R c \Rightarrow a R c$
- symmetric iff $\forall a, b \in S . a R b \Rightarrow b R a$
- anti-symmetric iff $\forall a, b \in S . a R b \Rightarrow \neg(b R a)$

Relation Exercises

◆ Name a unary relation

- (A unary relation is usually just called a property)

◆ Name a ternary relation

◆ Consider these binary relations over Z: the set of integers

- =
- >
- ≥
- ≠
- divides

◆ Which properties – reflexive, transitive, symmetric, anti-symmetric – hold for each?

Partial Orders

◆ An equivalence class is a relation that is

- Reflexive, transitive, and symmetric

◆ A partial order is a relation that is:

- Reflexive, transitive, and anti-symmetric

◆ A partially ordered set (poset) is a pair (S, \leq) of a set S and a partial order \leq over S

◆ Posets:

- $(2^S, \subseteq)$
- (Z, \leq)
- $(Z, \text{divides})$

lub and glb

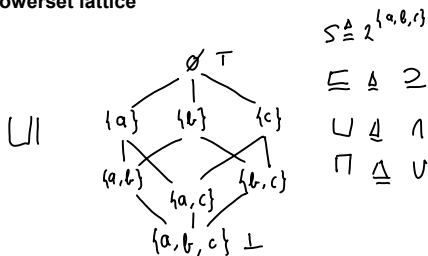
- ◆ Given a poset (S, \leq) , and two elements $a \in S$ and $b \in S$, then the:
 - > Least upper bound (lub) is an element $c \in S$ such that $a \leq c$, $b \leq c$, and $\forall d \in S. (a \leq d \wedge b \leq d) \Rightarrow c \leq d$
 - > Greatest lower bound (glb) is an element $c \in S$ such that $c \leq a$, $c \leq b$, and $\forall d \in S. (d \leq a \wedge d \leq b) \Rightarrow d \leq c$
- ◆ lub and glb don't always exist

Lattices

- ◆ A lattice is a tuple $(S, \sqsubseteq, \perp, \top, \sqcup, \sqcap)$ such that:
 - > (S, \sqsubseteq) is a poset
 - > $\forall a \in S. \perp \sqsubseteq a$
 - > $\forall a \in S. a \sqsubseteq \top$
 - > Every two elements from S have a lub and a glb
 - > \sqcup is the least upper bound operator, called a join
 - > \sqcap is the greatest lower bound operator, called a meet
- ◆ Sometimes this is called a "complete lattice"

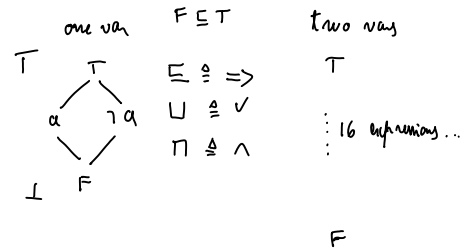
Example Lattice 1

- ◆ Powerset lattice



Example Lattice 2

- ◆ Boolean expressions



End of Background Material

- ◆ For more detail consult a book on abstract algebra or lattice theory
 - > I have a few of these

Back to our example

- ◆ We'll formalize the domain of reaching definitions using a lattice:
 - > Powerset of assignments found in the program
- ◆ Does it matter which end of the powerset is top in the lattice and which is bottom?

Back to our example

- ◆ We'll formalize the domain of reaching definitions using a powerset lattice
- ◆ Does it matter which end of the powerset is top and which is bottom?
 - > No – can formulate any analysis either way
 - > Yes – flipping the direction confuses people

Direction of Lattice

- ◆ Unfortunately...
 - > Dataflow analysis community has picked one direction
 - > Abstract interpretation community has picked the other
- ◆ So you'll see both in research papers
 - > In practice, you figure out which using context
- ◆ We will work with the abstract interpretation direction
- ◆ Bottom is the most precise (optimistic) answer, Top the most imprecise (conservative, pessimistic)
 - > I.e., for a powerset lattice using set S:
 - $\perp = \{\}$
 - $\top = S$

Direction of Lattice

- ◆ Always OK to return an element higher in the lattice
 - > Can safely turn any dataflow result to \top
 - > \top means "all assignments reach this program point"
- ◆ Need to be careful when returning an element lower in the lattice
- ◆ Bottom will be the empty set in reaching definitions
 - > Bottom means: No assignment reaches this program point

Worklist Algorithm w/Lattices

```
m: map from edge to computed value at edge
worklist: work list of nodes

for each edge e in CFG do
  m(e) =  $\perp$ 

for each node n do
  worklist.add(n)

while (!worklist.empty) do
  n = worklist.remove_any;
  info_in = m(n.incoming_edges);
  info_out = F(n, info_in);
  foreach i in info_out do
    let new_info = m(n.outgoing_edges[i])  $\sqcup$ 
                  info_out[i];
    if (m(n.outgoing_edges[i])  $\neq$  new_info)
      m(n.outgoing_edges[i]) = new_info;
      worklist.add(n.outgoing_edges[i].dst);
```

Termination of Dataflow?

- ◆ For reaching definitions, it terminates...
- ◆ Why?
 - > Lattice is finite
 - > lub operator always moves us up in the lattice
- ◆ Can we loosen this requirement?
 - > Yes: Only require the lattice to have a finite height
- ◆ Height of a lattice: Length of the longest ascending or descending chain
 - > Height of the powerset lattice?
- ◆ What's the worst-case running time of dataflow analysis?

Termination

- ◆ Still, it's annoying to have to perform a join in the worklist algorithm

```
while (!worklist.empty) do
  n = worklist.remove_any;
  info_in = m(n.incoming_edges);
  info_out = F(n, info_in);
  foreach i in info_out do
    let new_info = m(n.outgoing_edges[i])  $\sqcup$ 
                  info_out[i];
    if (m(n.outgoing_edges[i])  $\neq$  new_info)
      m(n.outgoing_edges[i]) = new_info;
      worklist.add(n.outgoing_edges[i].dst);
```
- ◆ It would be nice to get rid of it, if there is a property of the transfer functions that would allow us to do so

Even More Formal

- ◆ To reason more formally about termination and precision, we re-express our worklist algorithm mathematically
- ◆ We will use fixed points to formalize our algorithm

Fixed Points

- ◆ Recall: We are computing m , a map from edges to dataflow information
- ◆ Global transfer function F defined as follows:
 - F takes a map m as a parameter and returns a new map m' , in which individual local transfer functions have been applied



Fixed Points

- ◆ We want to find a fixed point of F
 - That is, a map m such that $m = F(m)$
- ◆ Approach to doing this?
- ◆ Define $\tilde{\perp}$, which is \perp lifted to be a map:
 - $\tilde{\perp} = \lambda e. \perp$
- ◆ Compute $F(\tilde{\perp})$, then $F(F(\tilde{\perp}))$, then $F(F(F(\tilde{\perp})))$, ... until the result doesn't change anymore

Fixed Points

- ◆ Formally:
 - $$\text{Soln} = \bigsqcup_{i=0}^{\infty} F^i(\tilde{\perp})$$
- ◆ We would like the sequence $F^i(\tilde{\perp})$ for $i = 0, 1, 2 \dots$ to be increasing, so we can get rid of the outer join
- ◆ The requirement that makes this possible is F being monotonic:
 - $\forall a, b. a \sqsubseteq b \Rightarrow F(a) \sqsubseteq F(b)$

Fixed Points

$$\begin{aligned} \tilde{\perp} &\sqsubseteq F(\tilde{\perp}) \\ F(\tilde{\perp}) &\sqsubseteq F(F(\tilde{\perp})) \\ F^k(\tilde{\perp}) &\sqsubseteq F^{k+1}(\tilde{\perp}) \\ F^{k+1}(\tilde{\perp}) &\sqsubseteq F^{k+2}(\tilde{\perp}) \end{aligned}$$

Back to Termination

- ◆ So if F is monotonic and lattice has finite height, then we have what we want: Termination without the outer join
- ◆ Also: If the local transfer functions are monotonic, then the global transfer function F is monotonic

Benefit of Monotonicity

- ◆ Usually, there are many fixpoints
 - E.g., could initialize the analysis with top at each program point instead of bottom
- ◆ Algorithm from this lecture gives the least fixpoint
 - The most precise one!

Summary

- ◆ Formalize domains as lattices
- ◆ Apply fixpoint reasoning
- ◆ This gives termination when transfer functions are monotonic
- ◆ What have we left out?