

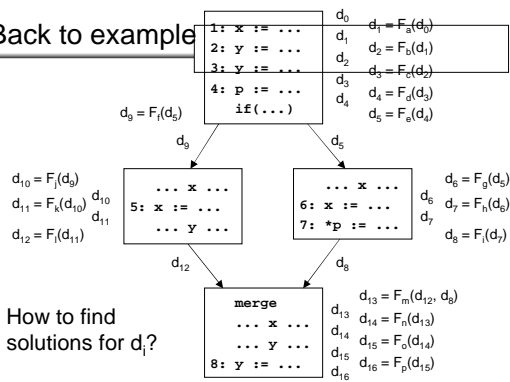
Administrative info

- Subscribe to the class mailing list!!!
 - instructions are on the class web page, which is accessible from my home page, which is accessible by searching for Sorin Lerner on google

From last lecture

- Flow functions: Given information *in* before statement *s*, $F_s(in)$ returns information after statement *s*
- Flow functions are a central component of a dataflow analysis
- They state constraints on the information flowing into and out of a statement

Back to example



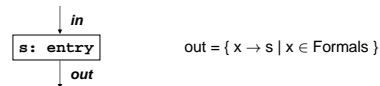
How to find solutions for d_i ?

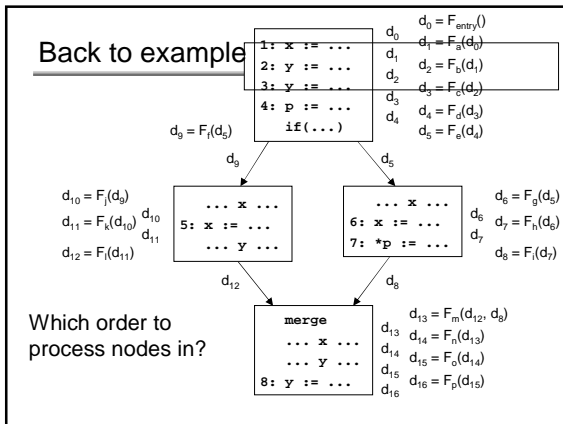
- This is a forward problem
 - given information flowing *in* to a node, can determine using the flow function the info flow *out* of the node
- To solve, simply propagate information forward through the control flow graph, using the flow functions
- What are the problems with this approach?

First problem

- What about the incoming information?
 - 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 flow function sets the information on entry (doesn't matter if entry node has an incoming edge, its flow function ignores any input)

Entry node



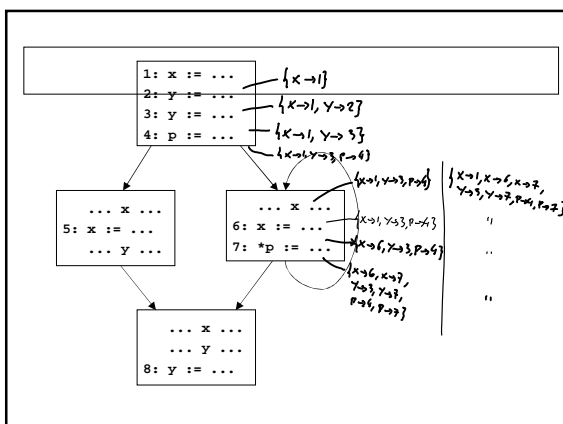
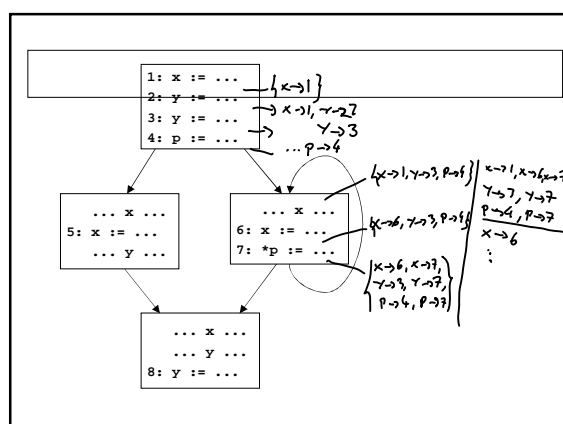


How to find solutions for d_i ?

- Sort nodes in topological order
 - each node appears in the order after all of its predecessors
- Just run the flow functions for each of the nodes in the topological order

Second problem

- When there are loops, there is no topological order!
- What to do?
- Let's try and see what we should do



Solution: iterate!

- Initialize all d_i to the empty set
- Store all nodes onto a worklist
- while worklist is not empty:
 - remove node n from worklist
 - apply flow function for node n
 - update the appropriate d_i , and add nodes whose inputs have changed back onto worklist

Worklist algorithm

```
let m: map from edge to computed value at edge
let 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.not) do
  let n := worklist.remove_any;
  let info_in := m(n.incoming_edges);
  let info_out := F(n, info_in);
  for i := 0 .. info_out.length do
    if (m(n.outgoing_edges[i]) ≠ info_out[i])
      m(n.outgoing_edges[i]) := info_out[i];
      worklist.add(n.outgoing_edges[i].dst);
```

Issues with worklist algorithm

Two 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 this algorithm terminate?

Order of nodes

- Topological order assuming back-edges have been removed
- Reverse depth first order
- Use an ordered worklist



Termination

- Why is termination important?
- 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...

Termination

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

```
while (worklist.empty.not) do
  let n := worklist.remove_any;
  let info_in := m(n.incoming_edges);
  let info_out := F(n, info_in);
  for i := 0 .. info_out.length do
    if (m(n.outgoing_edges[i]) ≠ info_out[i])
      m(n.outgoing_edges[i]) := info_out[i];
      worklist.add(n.outgoing_edges[i].dst);
```

Termination

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

```
while (worklist.empty.not) do
  let n := worklist.remove_any;
  let info_in := m(n.incoming_edges);
  let info_out := F(n, info_in);
  for i := 0 .. info_out.length do
    let 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

- We're using the structure of the domain outside of the flow functions
- In general, it's useful to have a framework that formalizes this structure
- We will use lattices

Background material

Relations

- A relation over a set S is a set $R \subseteq S \times S$
 - We write $a R b$ for $(a,b) \in R$
- A 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 $\forall a, b, \in S . a R b \Rightarrow \neg(b R a)$

Relations

- A relation over a set S is a set $R \subseteq S \times S$
 - We write $a R b$ for $(a,b) \in R$
- A 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 $\forall a, b, \in S . a R b \wedge b R a \Rightarrow a = b$

Partial orders

- An equivalence class is a relation that is:
- A partial order is a relation that is:

Partial orders

- An equivalence class is a relation that is:
 - reflexive, transitive, symmetric
- A partial order is a relation that is:
 - reflexive, transitive, anti-symmetric
- A partially ordered set (a poset) is a pair (S, \leq) of a set S and a partial order \leq over the set \leftarrow
- Examples of posets: $(2^S, \subseteq)$, (\mathbb{Z}, \leq) , $(\mathbb{Z}, \text{divides})$

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 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 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

- 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 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 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:

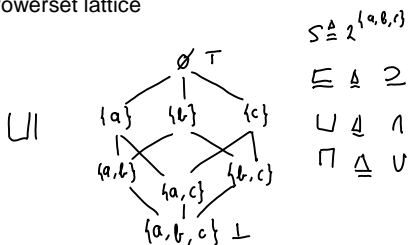


Lattices

- A lattice is a tuple $(S, \subseteq, \perp, \top, \sqcup, \sqcap)$ such that:
 - (S, \subseteq) is a poset
 - $\forall a \in S. \perp \subseteq a$
 - $\forall a \in S. a \subseteq \top$
 - 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

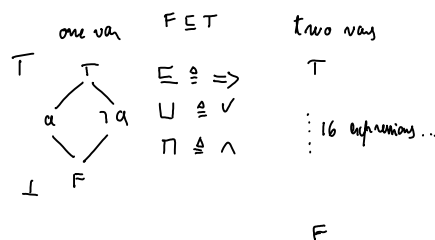
Examples of lattices

- Powerset lattice



Examples of lattices

- Booleans expressions



End of background material

Back to our example

- We formalize our domain with a powerset lattice
- What should be top and what should be bottom?
- Does it matter?

Back to our example

- We formalize our domain with a powerset lattice
- What should be top and what should be bottom?
- Does it matter?
 - It matters because, as we've seen, there is a notion of approximation, and we will use this notion to show up in the lattice

Direction of lattice

- Unfortunately:
 - dataflow analysis community has picked one direction
 - abstract interpretation community has picked the other
- We will work with the abstract interpretation direction
- Bottom is the most precise (optimistic) answer, Top the most imprecise (conservative)

$$\perp = \emptyset$$

Direction of lattice

- Always safe to go up in the lattice
- Can always set the result to \top
- Hard to go down in the lattice
- So ... Bottom will be the empty set in reaching defs



Worklist algorithm using lattices

```
let m: map from edge to computed value at edge
let 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.not) do
  let n := worklist.remove_any;
  let info_in := m(n.incoming_edges);
  let info_out := F(n, info_in);
  for i := 0 .. info_out.length 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 this algorithm?

- For reaching definitions, it terminates...
- Why?
 - lattice is finite
- Can we loosen this requirement?
 - Yes, we only require the lattice to have a finite height
- Height of a lattice: length of the longest ascending or descending chain
- Height of lattice $(2^S, \subseteq) = |S|$

Termination of this algorithm?

- For reaching definitions, it terminates...
- Why?
 - lattice is finite
- Can we loosen this requirement?
 - Yes, we only require the lattice to have a finite height
- Height of a lattice: length of the longest ascending or descending chain
- Height of lattice $(2^S, \subseteq) = |S|$

Termination

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

```
while (worklist.empty.not) do
  let n := worklist.remove_any;
  let info_in := m(n.incoming_edges);
  let info_out := F(n, info_in);
  for i := 0 .. info_out.length 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);
```

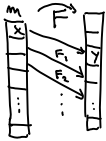
- It would be nice to get rid of it, if there is a property of the flow functions that would allow us to do so

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
- Define a global flow function F as follows: F takes a map m as a parameter and returns a new map m' , in which individual local flow functions have been applied



Fixed points

- We want to find a fixed point of F , that is to say 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:

$$S_{\text{den}} = \bigsqcup_{i=0}^{\infty} F^i(\tilde{I})$$



- We would like the sequence $F^i(\tilde{I})$ for $i = 0, 1, 2, \dots$ to be increasing, so we can get rid of the outer join
- Require that F be monotonic:
 - $\forall a, b. a \sqsubseteq b \Rightarrow F(a) \sqsubseteq F(b)$

Fixed points

$$\tilde{I} \sqsubseteq F(\tilde{I})$$

$$F(\tilde{I}) \sqsubseteq F(F(\tilde{I}))$$

$$F^k(\tilde{I}) \sqsubseteq F^{k+1}(\tilde{I})$$

$$F^{k+1}(\tilde{I}) \sqsubseteq F^{k+2}(\tilde{I})$$

Back to termination

- So if F is monotonic, we have what we want: finite height \Rightarrow termination, without the outer join
- Also, if the local flow functions are monotonic, then global flow function F is monotonic

Another benefit of monotonicity

- Suppose Marsians came to earth, and miraculously give you a fixed point of F , call it fp .
- Then:

$$\tilde{I} \sqsubseteq fp$$

$$F(\tilde{I}) \sqsubseteq F(fp)$$

$$F(\tilde{I}) \sqsubseteq fp$$

$$F(F(\tilde{I})) \sqsubseteq fp$$

$$fp \sqsubseteq fp$$

Another benefit of monotonicity

- We are computing the least fixed point...