Dataflow analysis

1

Dataflow analysis: what is it?

- A common framework for expressing algorithms that compute information about a program
- Why is such a framework useful?

2

Dataflow analysis: what is it?

- A common framework for expressing algorithms that compute information about a program
- Why is such a framework useful?
- Provides a common language, which makes it easier to:
  - communicate your analysis to others
  - compare analyses
  - adapt techniques from one analysis to another
  - reuse implementations (eg: dataflow analysis frameworks)

3

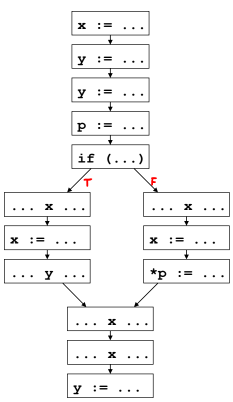
Control Flow Graphs

- For now, we will use a Control Flow Graph representation of programs
  - each statement becomes a node
  - edges between nodes represent control flow
- Later we will see other program representations
  - variations on the CFG (eg CFG with basic blocks)
  - other graph based representations

4

Example CFG

```
x := ...
y := ...
y := ...
p := ...
if (...) {
    ... x ...
    x := ...
    ... y ...
}
else {
    ... x ...
    x := ...
    *p := ...
}
... x ...
... y ...
y := ...
```

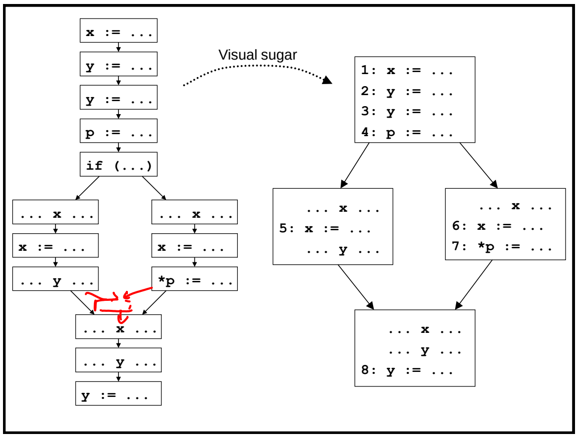


5

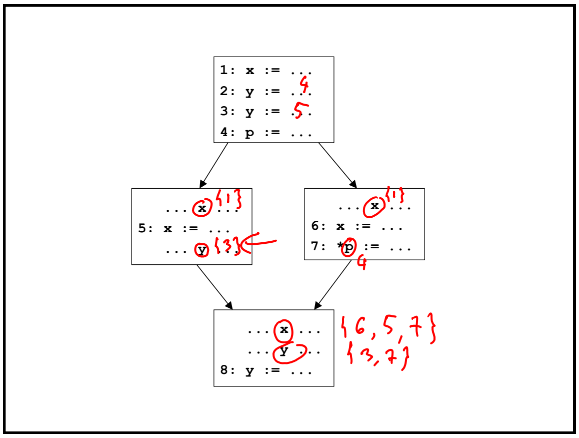
An example DFA: reaching definitions

- For each use of a variable, determine what assignments could have set the value being read from the variable
- Information useful for:
  - performing constant and copy prop
  - detecting references to undefined variables
  - presenting “def/use chains” to the programmer
  - building other representations, like the DFG
- Let’s try this out on an example

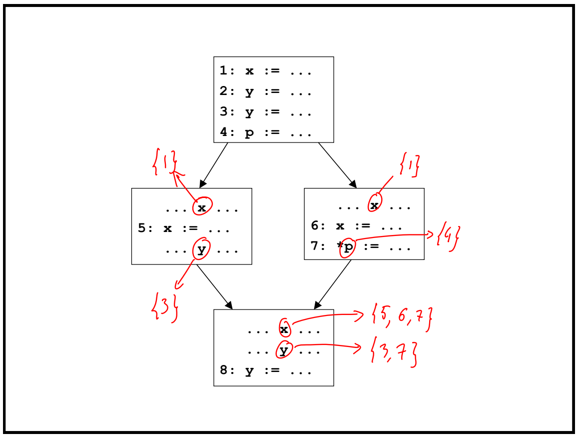
6



7



8



9

Safety

- When is computed info safe?
- Recall intended use of this info:
  - performing constant and copy prop
  - detecting references to undefined variables
  - presenting "def/use chains" to the programmer
  - building other representations, like the DFG
- Safety:
  - can have more bindings than the "true" answer, but can't miss any

10

Reaching definitions generalized

- DFA framework geared to computing information at each program point (edge) in the CFG
  - So generalize problem by stating what should be computed at each program point
- For each program point in the CFG, compute the set of definitions (statements) that may reach that point
- Notion of safety remains the same

11

Reaching definitions generalized

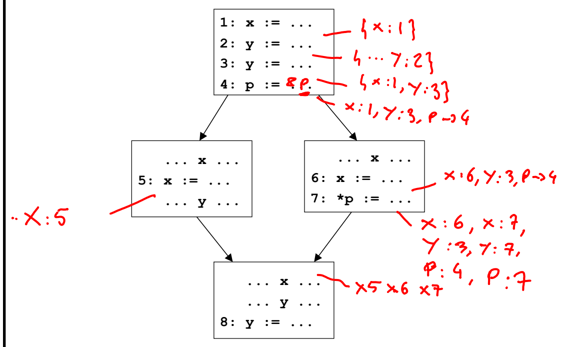
- Computed information at a program point is a set of var → stmt bindings
  - eg: $\{ x \rightarrow s_1, x \rightarrow s_2, y \rightarrow s_3 \}$
- How do we get the previous info we wanted?
  - if a var *x* is used in a stmt whose incoming info is *in*, then:

12

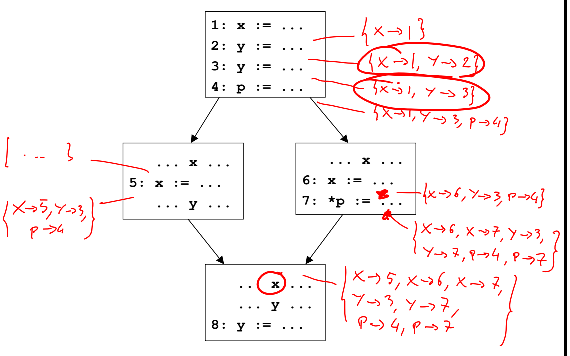
Reaching definitions generalized

- Computed information at a program point is a set of var → stmt bindings
  - eg: { x → s_1, x → s_2, y → s_3 }
- How do we get the previous info we wanted?
  - if a var *x* is used in a stmt whose incoming info is *in*, then: { *s* | (*x* → *s*) ∈ *in* }
- This is a common pattern
  - generalize the problem to define what information should be computed at each program point
  - use the computed information at the program points to get the original info we wanted

13



14



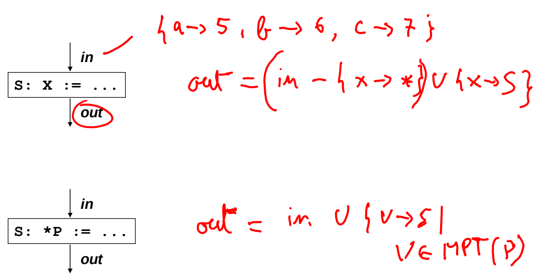
15

Using constraints to formalize DFA

- Now that we've gone through some examples, let's try to precisely express the algorithms for computing dataflow information
- We'll model DFA as solving a system of constraints
- Each node in the CFG will impose constraints relating information at predecessor and successor points
- Solution to constraints is result of analysis

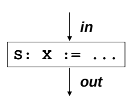
16

Constraints for reaching definitions

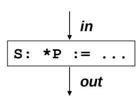


17

Constraints for reaching definitions



out = in – { X → S' | S' ∈ stmts } ∪ { X → S }



- Using may-point-to information:

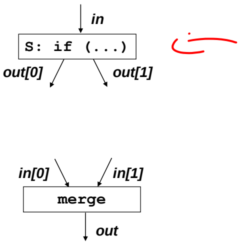
  out = in ∪ { X → S | X ∈ may-point-to(P) }

- Using must-point-to aswell:

  out = in – { X → S' | X ∈ must-point-to(P) ∧ S' ∈ stmts }
  ∪ { X → S | X ∈ may-point-to(P) }

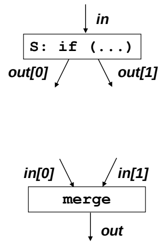
18

Constraints for reaching definitions



19

Constraints for reaching definitions



out [0] = *in* ∧
out [1] = *in*

more generally: $\forall\, i$. *out* [*i*] = *in*

out = *in* [0] ∪ *in* [1]

more generally: *out* = $\bigcup_i$ *in* [*i*]

20

Flow functions

- The constraint for a statement kind s often have the form: out = F_s(in)
- F_s is called a flow function
  - other names for it: dataflow function, transfer function
- Given information *in* before statement s, F_s(*in*) returns information after statement s
- Other formulations have the statement s as an explicit parameter to F: given a statement *s* and some information *in*, F(*s*,*in*) returns the outgoing information after statement s

21

Flow functions, some issues

- Issue: what does one do when there are multiple input edges to a node?
- Issue: what does one do when there are multiple outgoing edges to a node?

22

Flow functions, some issues

- Issue: what does one do when there are multiple input edges to a node?
  - the flow functions takes as input a tuple of values, one value for each incoming edge
- Issue: what does one do when there are multiple outgoing edges to a node?
  - the flow function returns a tuple of values, one value for each outgoing edge
  - can also have one flow function per outgoing edge

23

Flow functions

- Flow functions are a central component of a dataflow analysis
- They state constraints on the information flowing into and out of a statement
- This version of the flow functions is local
  - it applies to a particular statement kind
  - we'll see global flow functions shortly...

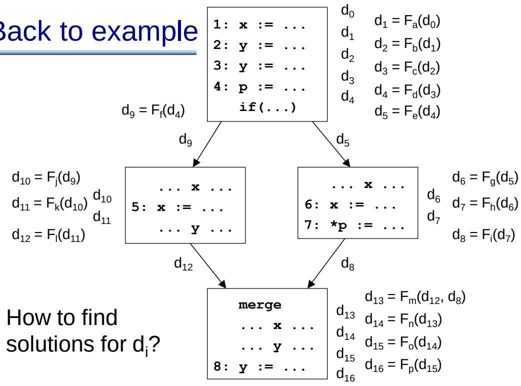
24

Summary of flow functions

- 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

25

Back to example



How to find solutions for d_i?

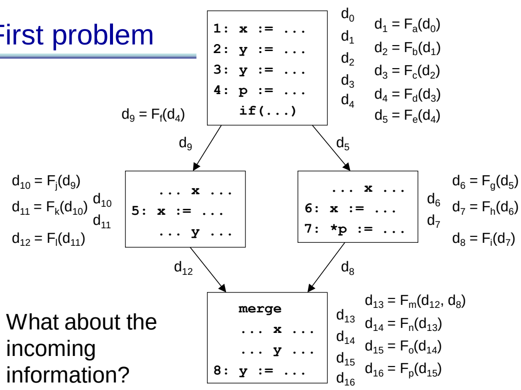
26

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?

27

First problem



What about the incoming information?

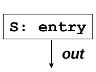
28

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)

29

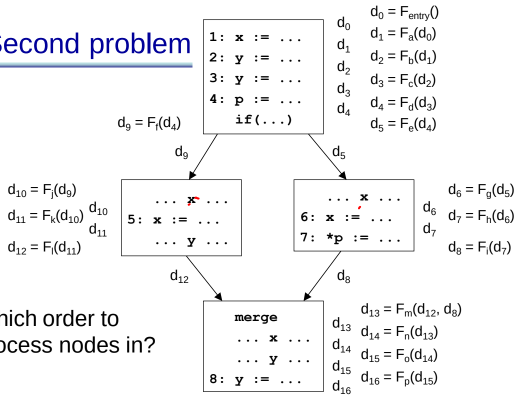
Entry node



out = $\{ X \to S \mid X \in \text{Formals} \}$

30

Second problem



Which order to process nodes in?

31

Second problem

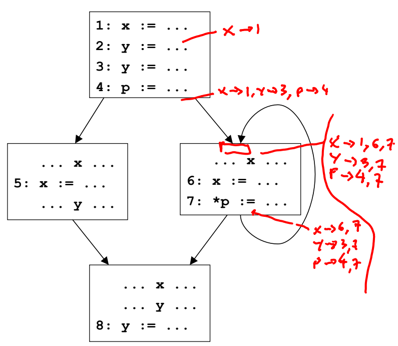
- Which order to process nodes in?
- Sort nodes in topological order
  - each node appears in the order after all of its predecessors
- Just run the flow functions for each of the nodes in the topological order
- What's the problem now?

32

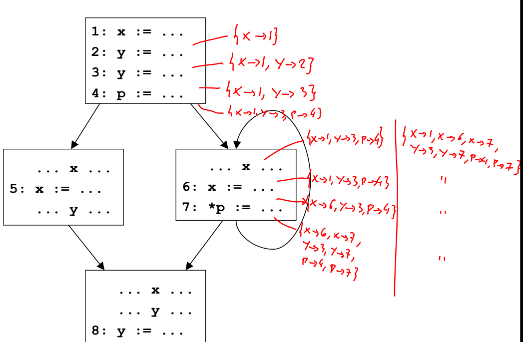
Second problem, prime

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

33



34



35

Worklist algorithm

- 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

36

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) := ∅

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-1 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);
```

37

Issues with worklist algorithm

38

Two issues with worklist algorithm

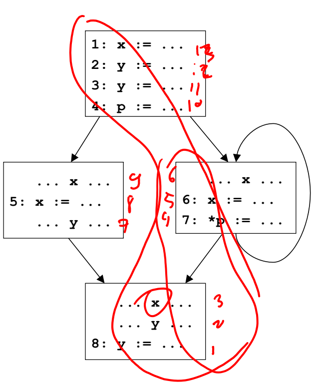
- Ordering
  - In what order should the original nodes be added to the worklist?
  - What order should nodes be removed from the worklist?
- Does this algorithm terminate?

39

Order of nodes

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

40



41

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...

42

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-1 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);
```

43

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-1 do
      let new_info := m(n.outgoing_edges[i]) ∪
                      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);
```

44

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

45