Data flow analysis

Want to compute some info about program

- at **program points**
- to determine opportunities for improving transformations

Can model data flow analysis as solving a system of constraints

- each node in CFG places a constraint relating info at predecessor and successor points
- solution to constraints is result of analysis

Solution must be **safe**/**sound**
Solution can be **conservative**

Key issues:

- how to know if constraint system defines the analysis correctly?
- how to represent info efficiently?
- how to represent & solve constraints efficiently?
  - how long does constraint solving take? finite time?
- what if multiple solutions are possible?
- how to synchronize transformations with analysis?

Example: reaching definitions

For each program point,
want to compute set of definitions (statements) that *may reach* that point

- reach: are the last definition of some variable
- can use info to build def-use chains, do constant propagation, etc.

Info ≡ set of *var*→*rtl* bindings
E.g.:
$\{x \to s_1,\ y \to s_5,\ y \to s_8\}$

Safety rule:
can have more bindings than the "true" answer,
but can't miss any

Reaching definitions, continued

Main constraints:

A simple assignment removes any old reaching defs for the lhs and replaces them with this stmt:

- **strong update**

s: `x := ...`:
$info_{succ} = info_{pred} - \{x \to *\} \cup \{x \to s\}$

A pointer assignment may modify anything, but doesn't definitely replace anything

- **weak update**

s: `*p := ...`:
$info_{succ} = info_{pred} \cup \{* \to s\}$

Other statements: do nothing
$info_{succ} = info_{pred}$

Reaching definitions, still continued

Branches pass through reaching defs to both successors
$info_{succ[i]} = info_{pred}$

Merges take the union of all incoming reaching defs

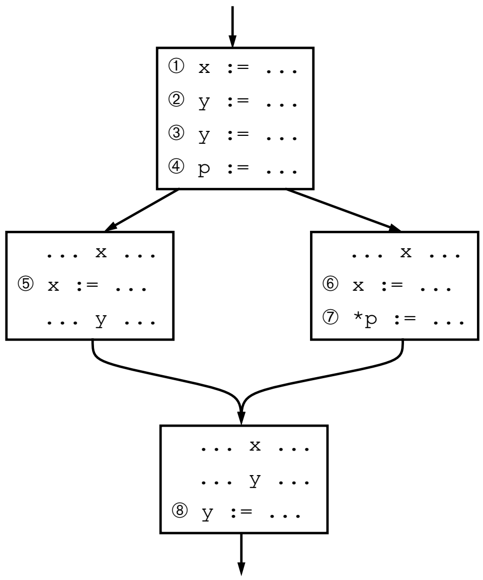
- we don't know which path is being taken at run-time ⇒ be conservative

$info_{succ} = \bigcup_i info_{pred[i]}$

Conditions at entry to CFG: {}

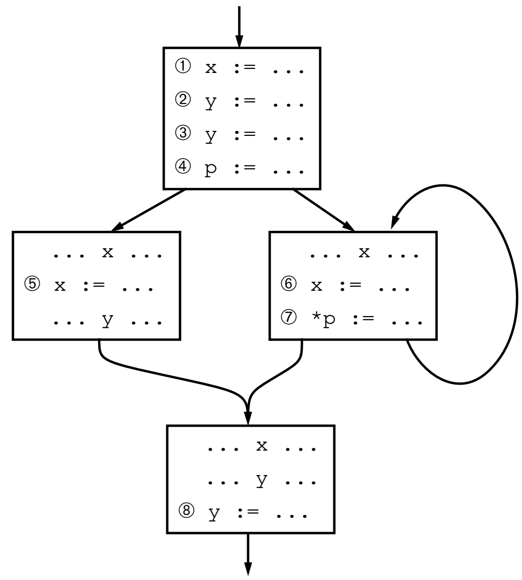
- or maybe bindings for formals

Example



Simple constraint-solving algorithm:
traverse instructions in forward topological order,
computing successor info from predecessor info

Another example



Topological order not defined!

Loops

If CFG has a loop, data flow constraints are recursively defined:
$\text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}}$
$\text{info}_{\text{back-edge}} = \ldots\ \text{info}_{\text{loop-head}} \cdots$

Summarizing set of constraints in loop body as F:
$\text{info}_{\text{loop-head}} = F(\text{info}_{\text{loop-head}})$

Legal solution to constraints is a **fixpoint** of F
- want **least**/**greatest** fixpoint,
  corresponding to most precise answer

How to find fixpoint?
- for restricted CFGs can use specialized methods
  - e.g. **interval analysis** for **reducible** CFGs
- for arbitrary CFGs, can use **iterative** approximation

Loop terminology

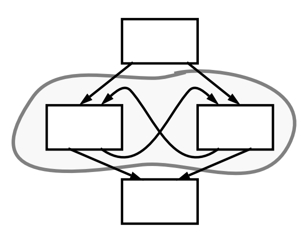
loop: strongly-connected component in CFG with single entry

loop entry edge: source not in loop, target in loop
loop exit edge: the reverse
back edge: target is loop head node

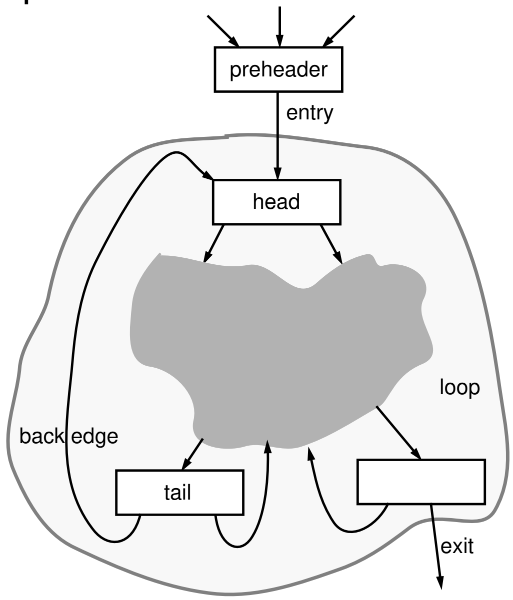
loop head node: target of loop entry edge
loop tail node: source of back edge
loop preheader node:
single node that's source of loop entry edge

nested loop: loop whose head is inside another loop

reducible flow graph: all SCC's have single entry



Example



Iterative data flow analysis

1. Start with initial guess of info at end of loop:
   $\text{info}_{\text{back-edge}} = \textit{guess}$
   $\text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}}$

2. Solve equations for loop:
   $\text{info}_{\text{back-edge}}' = F_{body}\,(\text{info}_{\text{loop-head}})$
   $\text{info}_{\text{loop-head}}' = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}}'$

3. Test if found fixpoint:
   $\text{info}_{\text{loop-head}}' = \text{info}_{\text{loop-head}}$?

   A. if so, then done

   B. if not, then adopt result as (better) guess and repeat:

   $\text{info}_{\text{back-edge}}'' = F_{body}\,(\text{info}_{\text{loop-head}}')$
   $\text{info}_{\text{loop-head}}'' = \text{info}_{\text{loop-entry}} \cup \text{info}_{\text{back-edge}}''$
   $\text{info}_{\text{loop-head}}'' = \text{info}_{\text{loop-head}}'$?
   ...

When does iterating work?

1. need to be able to make an initial guess
2. info^{n+1} must be closer to the fixpoint than info^{n}
   (constraints must be **monotonic**)
3. must eventually reach the fixpoint in a finite # of iterations
   (info must be drawn from a **finite-height domain**)

To reach best fixpoint, initial guess for loop head should be **optimistic**

- standard choice: $\text{info}_{\text{loop-head}} = \text{info}_{\text{loop-entry}}$

To speed iterative analysis, want to test guess ASAP

- ideal: solve constraints along shortest path from loop head to loop tail
- practical: don't solve any constraints outside of loop until fixpoint is reached

Direction of dataflow analysis

In what order are constraints solved, in general?

Constraints are declarative, not directional/procedural, so may require mixing forward & backward solving, or other more global solution methods

But often constraints can be solved by (directional) propagation & iteration

- may be **forward** or **backward** propagation of info
- topological traversals of acyclic graph components are best

Constraints of simple statements called **flow functions**

- often written as functions on input info to compute output

$\text{RD}_{s\texttt{: x := ...}}(\text{in}) = \text{in} - \{\texttt{x}{\to}{*}\} \cup \{\texttt{x}{\to}s\}$

$\text{RD}_{s\texttt{: *p := ...}}(\text{in}) = \text{in} \cup \{{*}{\to}s\}$

GEN and KILL sets

Can often think of flow functions in terms of each's GEN set and KILL set

- GEN = new information added
- KILL = old information removed

Then

$F_{instr}(in) = in - KILL_{instr} \cup GEN_{instr}$

E.g., for reaching defs:

$RD_{s\text{: x := ...}}(in) = in - \{x \rightarrow *\} \cup \{x \rightarrow s\}$

$RD_{s\text{: *p := ...}}(in) = in \cup \{* \rightarrow s\}$

Bit vectors

Can sometimes represent info/KILL/GEN sets as **bit vectors**

- if can express abstractly as set of things (e.g. statements, vars), each thing getting a bit position

Each flow function is then represented by a pair of bit vectors, and bit vector operations are used for set difference & union

Can summarize effect of basic block as single flow function by combining KILL & GEN sets for flow functions in block, for faster iteration

- perhaps can do this even if not using bit vectors