Data Flow Analysis

Want to compute some info about program

- at **program points**, i.e. edges in CFG/DFG/...
- to identify opportunities for improving transformations

Can model data flow analysis as solving system of **constraints**

- each node in graph imposes constraints relating info at predecessor and successor points
- solution to constraints is result of analysis

Solution *must* be **safe** a.k.a. **sound**
Solution *can* be **conservative**

Key issues:

- how to represent info efficiently?
- how to represent & solve constraints efficiently?
  - how long does constraint solving take? does it terminate?
- what if multiple solutions are possible?
- how do transformations interact with analyses?
- how to reason about whether analyses & transformations are sound, i.e., semantics-preserving?

Example: reaching definitions

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

- reach: are the last definition of some variable

Info ≡ set of $var \rightarrow stmt$ bindings
E.g.:
$\{x \rightarrow s_1,\ y \rightarrow s_5,\ y \rightarrow s_8\}$

Can use reaching definition info to:

- build def-use chains
- do constant & copy propagation
- detect references to undefined variables
- present use/def info to programmer
- ...

Safety rule (for these intended uses of this info):
can have more bindings than the "true" answer,
but can't miss any

Constraints for reaching definitions

Main constraints:

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

- **strong update**

s: `X := ...`:

$$info_{succ} = info_{pred} - \{X \rightarrow s' \mid \forall s'\} \cup \{X \rightarrow s\}$$

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

- **weak update**

s: `*P := ...`:

$$info_{succ} = info_{pred} \cup \{X \rightarrow s \mid \forall X \in \text{may-point-to}(P)\}$$

Other statements: do nothing

$info_{succ} = info_{pred}$

Constraints for reaching definitions, continued

Branches pass through reaching defs to both successors

$info_{succ[i]} = info_{pred},\ \forall i$

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: "definitions" of formals

$info_{entry} = \{X \rightarrow entry \mid \forall X \in \text{formals}\}$

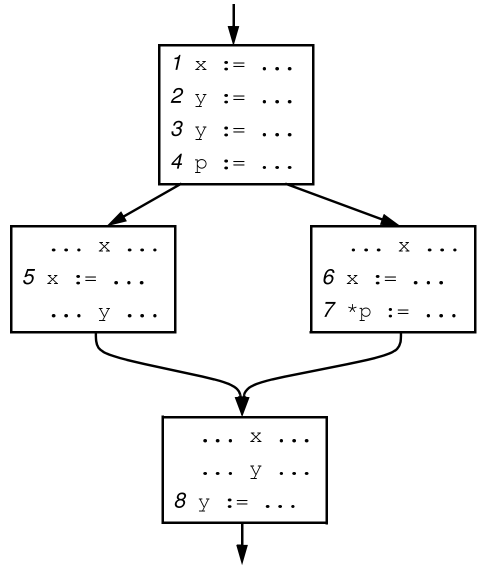
Solving constraints

A given program yields a system of constraints
Need to solve constraints

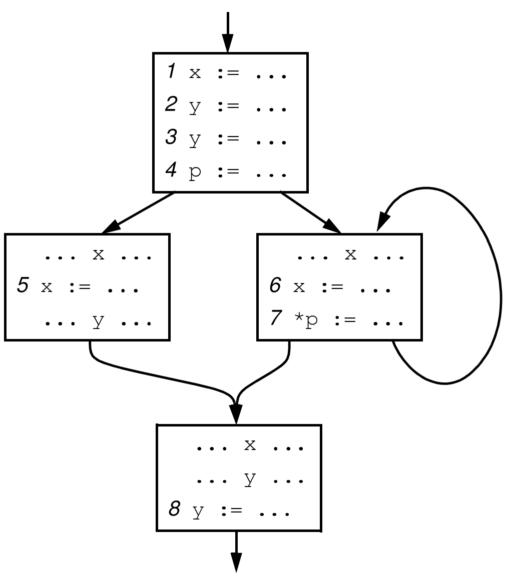
For reaching definitions,
can traverse instructions in forward topological order, computing successor info from predecessor info

- because of how the constraints are defined

Example



Another example



Topological order not defined!

Loop terminology

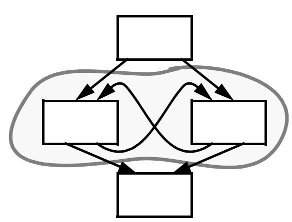
loop: strongly-connected component in CFG with single entry

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

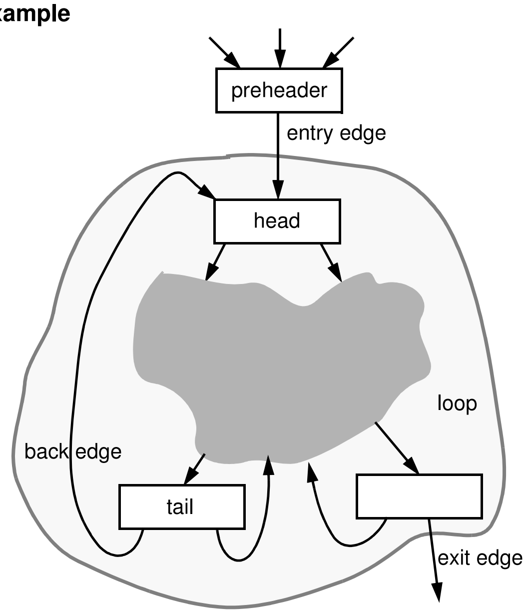
loop head node: target of loop entry edge
loop tail node: source of back edge
loop preheader node:
single node that's source of loop entry edge

nested loop: loop whose head is inside another loop

reducible flow graph: all SCC's have single entry



Example



Analysis of loops

If CFG has a loop, data flow constraints are recursively defined:

$info_{loop\text{-}head} = info_{loop\text{-}entry} \cup info_{back\text{-}edge}$
$info_{back\text{-}edge} = \ldots\ info_{loop\text{-}head} \ldots$

Substituting definition of $info_{back\text{-}edge}$:

$info_{loop\text{-}head} = info_{loop\text{-}entry} \cup (\ldots\ info_{loop\text{-}head} \ldots)$

Summarizing r.h.s. as F:

$info_{loop\text{-}head} = F(info_{loop\text{-}head})$

A legal solution to constraints is a **fixed-point** of *F*

Recursive constraints can have many solutions

- want **least** or **greatest** fixed-point, whichever corresponds to the most precise answer

How to find least/greatest fixed-point of *F*?

- for restricted CFGs can use specialized methods
  - e.g. **interval analysis** for **reducible** CFGs
- for arbitrary CFGs, can use **iterative** approximation

Solving constraints by iterative approximation

1. Start with initial guess of info at loop head:
   $info_{loop\text{-}head} = guess$

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

3. Test if found fixed-point:
   $info_{loop\text{-}head}' = info_{loop\text{-}head}$?

   A. if same, then done

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

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

When does iterating work?

Sufficient conditions:

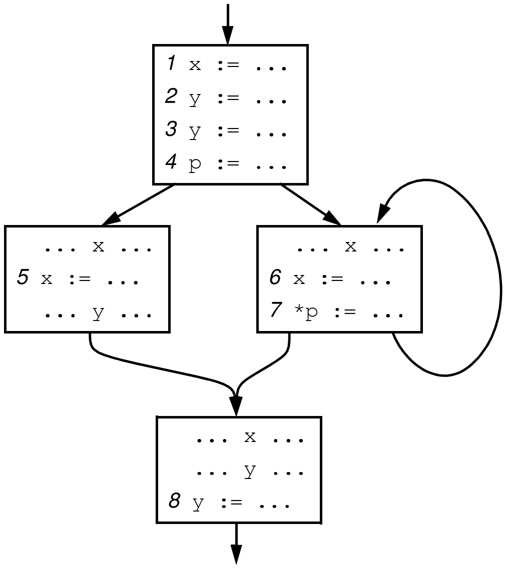
1. need to be able to make an initial guess
2. $info^{n+1}$ must be closer to the fixed-point than $info^{n}$
   (true if F_{body} is **monotonic**)
3. must eventually reach the fixed-point
   in a finite number of iterations
   (true if info drawn from a **finite-height domain**)

To reach best fixed-point, initial guess for loop head should be **optimistic**

- easy choice: $info_{loop\text{-}head} = info_{loop\text{-}entry}$

(Even if guess is overly optimistic, iteration will ensure we won't stop analysis until the answer is safe.)

The example, again



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

Directional constraints often called **flow functions**

- often written as functions on input info to compute output

  $RD_{s:\ X\ :=\ ...}(in) = in - \{X \to s' \mid \forall s'\} \cup \{X \to s\}$

  $RD_{s:\ *P\ :=\ ...}(in) = in \cup \{X \to s \mid \forall X \in \text{may-point-to}(P)\}$

For greatest solving efficiency:

- analyze acyclic subgraphs in topological order
- analyze loops till convergence before analyzing downstream of loops

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 reading defs:

$RD_{s:\ X\ :=\ ...}(in) = in - \{X \to s' \mid \forall s'\} \cup \{X \to s\}$

$RD_{s:\ *P\ :=\ ...}(in) = in \quad \cup \{X \to s \mid \forall X \in \text{mpt}(P)\}$

Bit vectors

For efficiency,
can sometimes represent info/KILL/GEN sets as **bit vectors**

- if can express abstractly as set of things (e.g. statements, vars), drawn from a statically known set of things, each thing getting a statically determined bit position
- bitvector encodes **characteristic function** of set

E.g., for reaching defs:

info = bitvector over statements,
each stmt getting a distinct bit position

- statement implies which variable is defined

Bit vectors compactly represent sets
Bit-vector operations efficiently perform set difference, union, ...

Flow function may be able to be represented simply by a pair of bit vectors, if they don't depend on input bit vector

- can merge the KILL and GEN bit vectors of a whole basic block of instructions into a single overall KILL and GEN set, for faster iterating

Another example: constant propagation

What info computed for each program point?

I is a conservative approximation to “true” info I_{true} iff:

Direction of analysis?

Initial info?

$CP_{X\ :=\ N}(\text{in})$ =

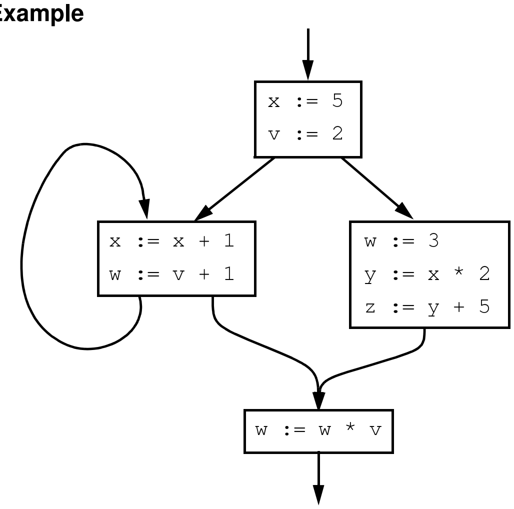
$CP_{X\ :=\ Y\ +\ Z}(\text{in})$ =

$CP_{*P\ :=\ *Q\ +\ *R}(\text{in})$ =

Merge function?

Can use bit vectors?

Example



May vs. must info

Some kinds of info imply guarantees: **must** info
Some kinds of info imply possibilities: **may** info
- the complement of **may** info is **must not** info

	May	Must
desired info	small set	big set
safe	overly big set	overly small set
GEN	add everything that might be true	add only if guaranteed true
KILL	remove only if guaranteed wrong	remove everything possibly wrong
MERGE	$\cup$	$\cap$

Another example: live variables

Want the set of variables that are **live** at each pt. in program
- live: *might* be used *later* in the program

Supports dead assignment elimination, register allocation

What info computed for each program point?
May or must info?
I is a conservative approximation to I_{true} iff:

Direction of analysis?
Initial info, at what program point(s)?

$LV_{X\ :=\ Y\ +\ Z}(\text{in})$ =

$LV_{*P\ :=\ *Q\ +\ *R}(\text{in})$ =

Merge function?

Can use bit vectors?

Example