Components of representation

Operations

Control flow: sequencing of operations

- evaluation of then & else arms depends on result of test
- side-effects of statements occur in right order

Data flow: flow of values from **definitions** to **uses**

- operands computed before operation

Also called control & data **dependence**

Ideal: represent just those dependences that matter

- dependences constrain transformations
- fewest dependences ⇒ most flexibility in implementation

Representing control dependences

Option 1: high-level representation

- control flow implicit in semantics of AST nodes

Option 2: control flow graph

- nodes are basic blocks
- edges represent branches
  (control flow between basic blocks)

Fancier:

- control dependence tree,
  part of program dependence graph (PDG)
- convert into data dependences on explicit state,
  in value dependence graph [Weise *et al. 94*]

Kinds of data dependences

read-after-write (RAW): true (a.k.a. flow) dependence

- reflects real data flow, operands to operation

write-after-read (WAR): anti-dependence
write-after-write (WAW): output dependence

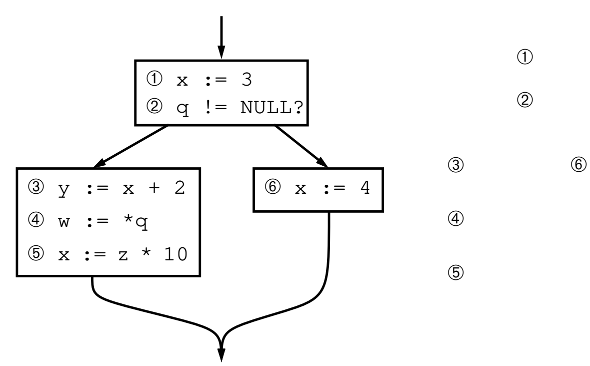
- reflects reuse of memory, not real data flow

read-after-read (RAR): no dependence

- can occur in any order

Example

```
x := 3
if q != NULL then
  y := x + 2
  w := *q
  x := z * 10
else
  x := 4
endif
```



Representing data dependences

Within basic block:

Option 1: sequence of instructions
(represent data flow as a kind of control dependence)
+ simple, source-like
+ fixed ordering supports easy analysis
– may overconstrain order of operations

Option 2: expression tree/DAG
+ natural, abstract
+ directly captures data dependences within basic block
+ DAG supports local CSE
+ can be compact
– conceptually harder to analyze, transform
– must linearize eventually

Representing data dependences, cont.

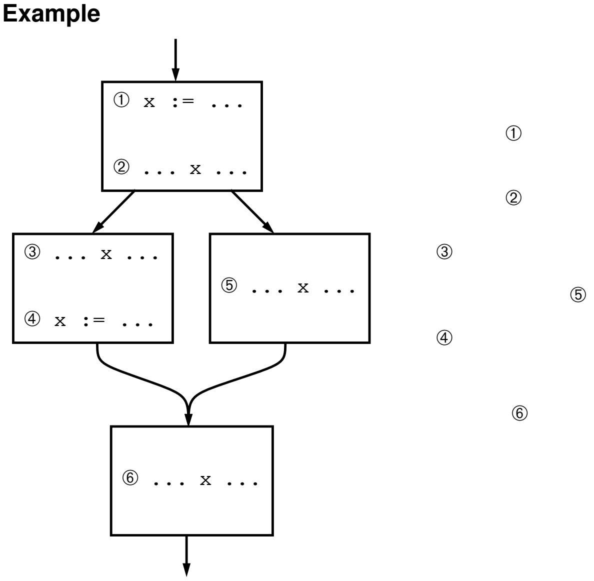
Across basic blocks:

Option 1: implicitly through variable defs/uses
+ simple
– analysis wants important things explicit

Option 2: def/use chains, linking each def with each use
+ explicit
– must be computed, maintained after transformations
– space consuming

Option 3: static single assignment (SSA) form
Option 4: value dependence graphs
...

Example



Data flow analysis

Want to compute some info about program
- at **program points**
- to determine opportunities for improving transformations

Compute info by setting up and solving **data flow equations**
- each node in CFG places a constraint on info at predecessor and successor points
- solution to constraints is result of analysis

Solution must be **safe**, possibly **conservative**

Key issues:
- how to know if constraints are safe?
- how to represent info efficiently?
- how to represent & solve constraints efficiently?
  - how long does constraint solving take? finite time?
- what if multiple solutions possible?
- how to synchronize transformations with analysis?

Example: reaching definitions

For each program point,
want to compute set of definitions that *reach* that point

- can use info to build def-use chains, do constant propagation, etc.

Info = set of ($var \rightarrow rtl$) bindings

Safe approximation:
can have more bindings than the “true” answer

Main constraints:

s: `x := ...`:

$info_{succ} \supseteq info_{pred} - \{x \rightarrow *\} \cup \{x \rightarrow s\}$

s: `*p := ...`:

$info_{succ} \supseteq info_{pred} \cup \{* \rightarrow s\}$

At branches:

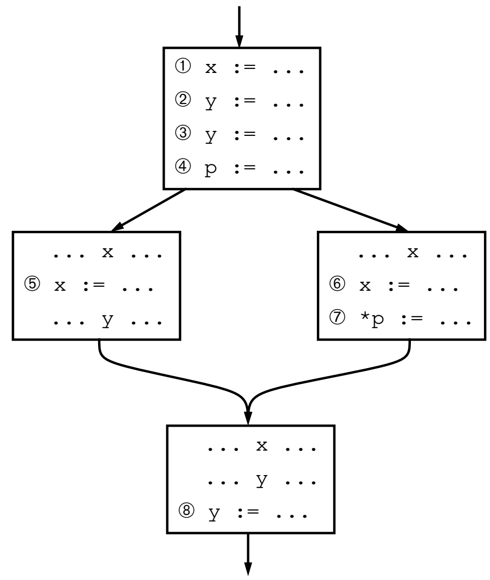
$info_{succ[i]} \supseteq info_{pred}$

At merges:

$info_{succ} \supseteq \cup_i\, info_{pred[i]}$

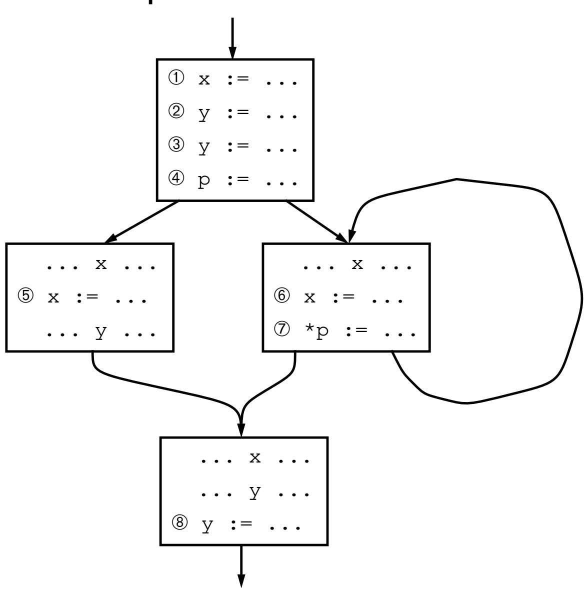
Initial conditions: {}

Example



Simple 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:

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

$info_{back\text{-}edge} \supseteq F\,(info_{loop\text{-}head})$

Want to solve for **fixpoint** of these equations

- want **least**/**greatest** fixpoint, corresponding to most precise answer

For restricted CFGs can use specialized methods

- **interval analysis** for **reducible** CFGs

For arbitrary CFGs, can use **iterative** approximation

Iterative data flow analysis

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

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

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

   A. if so, then done

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

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

When does iterating work?

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

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 that's a proper subset of another loop

reducible flow graph: all "loops" have single entry

Example