Lecture 21

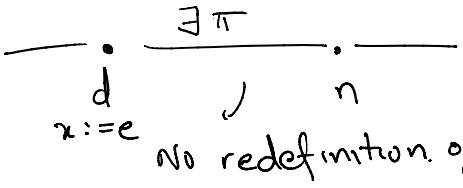
Wednesday, April 8, 2020 4:39 PM

- Hello!
- Please turn in Assignment 2 by April 15.
  - Do: Blindly apply Tseitin's transform for Q2

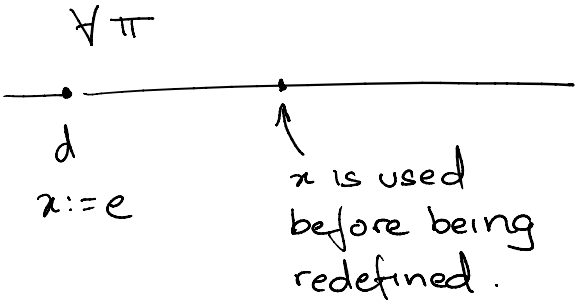
- Chaotic iteration algorithm to compute reaching definitions

  Compute-RD

- Relatives of reaching definitions

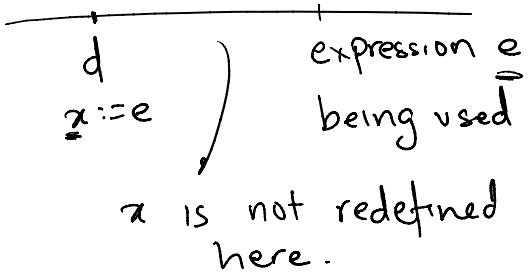


Very busy expressions

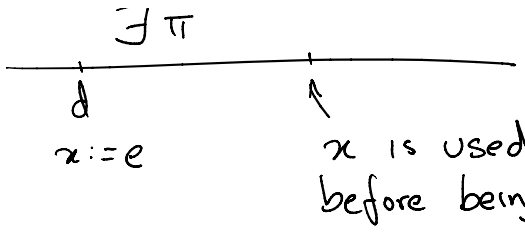


Available expressions

$\exists x. \forall \pi$



Live Variable (Useful in dead code elimination)



Elementary observations about RD

Claim (Lower bounds on IN & OUT)

(2') $\forall^{\wedge} i$ $OUT[n_i] \subseteq IN[n_i]$

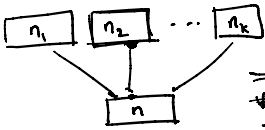
$\forall^{\wedge} i$ d, $d \in OUT[n_i] \Rightarrow d \in IN[n_i]$

(3) $\forall n$, $(IN[n] \setminus KILL[n]) \cup GEN[n] \subseteq OUT[n]$

(4')
$\forall n$ d, $d \in IN[n] \setminus KILL[n]$
$\Rightarrow d \in OUT[n]$ ✓

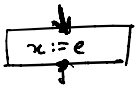
$\forall n$ d, $d \in GEN[n] \Rightarrow d \in OUT[n]$ ✓

Claim: ① $OUT[Entry] = \emptyset$

② 

$IN[n] = \bigcup_i OUT[n_i]$

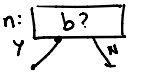
~~$\forall i$. $OUT[n_i] \subseteq IN[n]$~~

③ 

$OUT[n] = (IN[n] \setminus KILL[n]) \cup GEN[n]$

Every dataflow fact involving x → $KILL[n] = \{d \mid d : x := e\}$

The statement $x := e$ → $GEN[n] = \{n\}$

④ 

$OUT[n] = (IN[n] \setminus KILL[n]) \cup GEN[n]$

Conditional statements don't kill any facts; don't give rise to any dataflow facts → $KILL[n] = \emptyset$, $GEN[n] = \emptyset$

↑ System of constraints, very similar to a system of linear equations.

Q1. Does it admit a solution?

Q2. Does it admit a unique solution?

Q3. How are these solutions related to each other?

↓

At least the trivial solution

- Admits at least the trivial solution

$\forall n . IN[n] = OUT[n] = \{$ all assignment statements $d\}$

- Compute-RD also produces a solution.

Thm: $\vec{a}$ is a solution, $\vec{b}$ is a solution to a system of linear equations, $A\vec{x} = \vec{0}$

then $\forall \alpha, \beta \in R$, $\alpha\vec{a} + \beta\vec{b}$ is also a solution.

Prop: If $(IN[n_1], IN[n_2], \ldots, IN[n_k]$
$OUT[n_1], OUT[n_2], \ldots, OUT[n_k])$ is a solution to

(2') (3') (4')

& if $(IN'[n_1], IN'[n_2], \ldots, IN'[n_k],$
$OUT'[n_1], OUT'[n_2], \ldots, OUT'[n_k])$ is also a solution,

Conj 1: $(IN[n_1] \cup IN'[n_1], IN[n_2] \cup IN'[n_2], \ldots, IN[n_k] \cup IN'[n_k],$
$OUT[n_1] \cup OUT'[n_1], OUT[n_2] \cup OUT'[n_2], \ldots, OUT[n_k] \cup OUT'[n_k])$
is also a solution.

Conj 2: $(IN[n_1] \cap IN'[n_1], IN[n_2] \cap IN'[n_2], \ldots, IN[n_k] \cap IN'[n_k],$
$OUT[n_1] \cap OUT'[n_1], OUT[n_2] \cap OUT'[n_2], \ldots, OUT[n_k] \cap OUT'[n_k])$

$OUT[n_1] \cap OUT'[n_1], OUT[n_2] \cap OUT'[n_2], \ldots, OUT[n_K] \cap OUT'[n_K])$
is also a solution.

Proof of conj 1: If (IN, OUT) satisfies (2') (3') (4')
$(IN' OUT')$ satisfies (2') (3') (4')

then $(IN \cup IN', OUT \cup OUT')$ also satisfies (2') (3') (4)

Proof (2'): $\overset{TST}{\forall} n_i$ n d s.t n_i is an immediate predecessor of n

if $d \in OUT[n_i] \cup OUT'[n_i]$ then $d \in IN[n] \cup IN'[n]$.

Case 1: $d \in OUT[n_i] \Rightarrow d \in IN[n]$ by hypothesis

Case 2: $d \in OUT'[n_i] \Rightarrow$ ——————

Proofs of (3'), (4'): Similar.

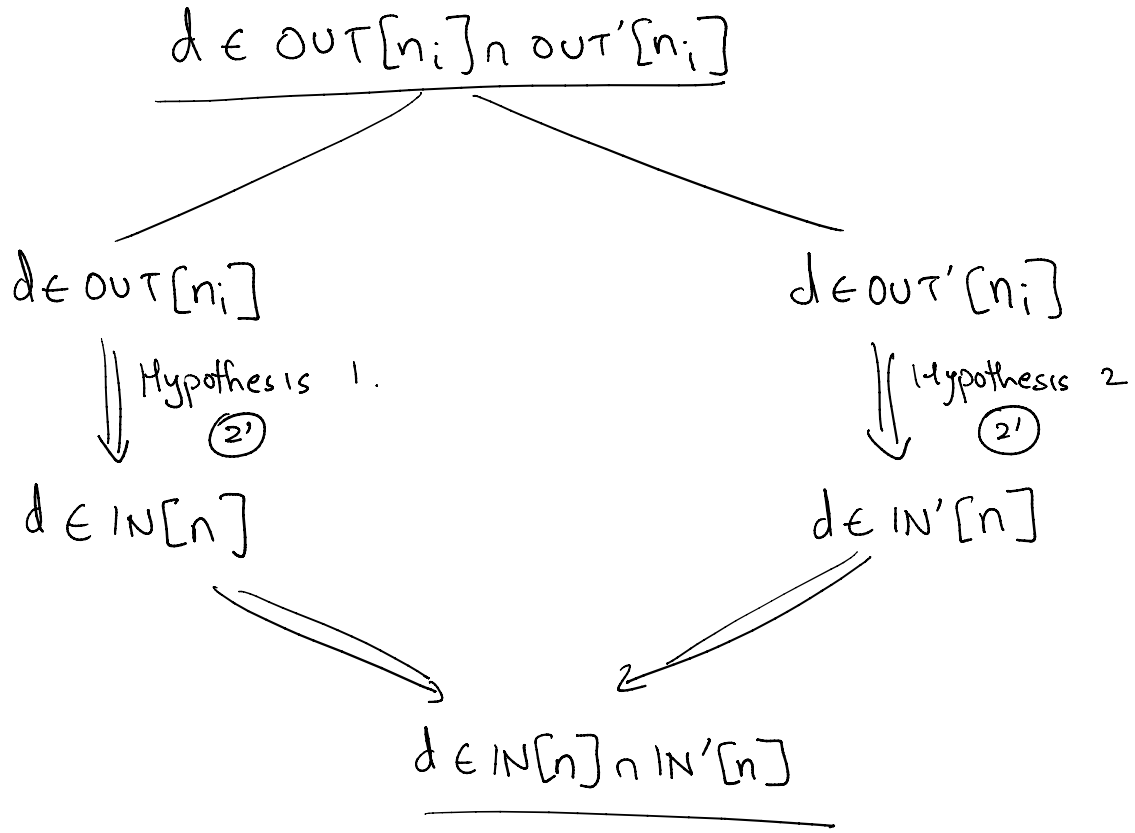
Proof of conj 2: If $(IN \quad OUT)$ both satisfy (2') (3') (4')
$(IN' \quad OUT')$

then $(IN \cap IN' \quad OUT \cap OUT')$ also satisfies (2') (3') (4').

Proof of (2'): $\forall n_i$ n d if n_i is an imm. pred. of n

then $d \in OUT[n] \cap OUT'[n] \Rightarrow d \in IN[n] \cap IN'[n]$

then $d \in OUT[n_i] \cap OUT'[n_i] \Rightarrow d \in IN[n] \cap IN'[n]$.



Prop: Compute-RD(P) $= \bigcap$ (IN, OUT).

Program — all solutions (IN OUT)

In other words, compute-RD produces the smallest soln to (2') (3') (4').

Algorithm Compute-RD (Chaotic Iteration Algorithm)

(1) For each statement n, Invariant: Always

① For each statement n,

initialize $IN[n], OUT[n] := \emptyset$.

Invariant: Always

$(IN, OUT)_{\text{Compute-RD}} \subseteq (IN, OUT)_{\text{every soln.}}$

② Until fixpoint : (iter # k)

For each non-entry sfmt n:

- Update $IN^{(k)}[n] := \bigcup_{i} OUT^{(k-1)}[n_i]$

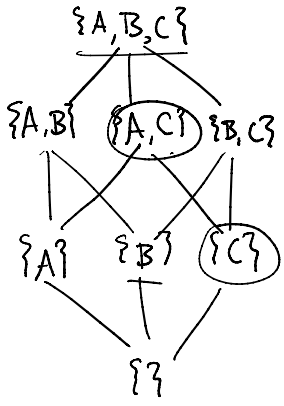
  i ← ranging over all predecessors

- Update $OUT^{(k)}[n] := (IN^{(k)}[n] \setminus KILL[n]) \cup GEN[n]$

Specified as part of the input; Computed at the beginning of time.

Mathematically speaking,

the solutions to ② ③ ④ form a lattice.



Claim (Lower bounds on IN & OUT)

②' $\forall^{n}_{i}$ $OUT[n_i] \subseteq IN[n_i]$

$\forall^{n}_{i}$ d, $d \in OUT[n_i] \Rightarrow d \in IN[n]$

$\forall$ n n_i d

$IN(n, d) :- OUT(n_i, d), pred(n_i, n)$

$\forall i\ d,\ d \in OUT[n_i] \Rightarrow d \in IN[n]$

$IN(n,d) :- OUT(n_i, d), pred(n_i, n)$

③ $\forall n, (IN[n] \setminus KILL[n]) \cup GEN[n] \subseteq OUT[n]$

④

$\forall n\ d,\ d \in IN[n] \setminus KILL[n] \Rightarrow d \in OUT[n]$ ✓

$\forall n\ d,\ d \in GEN[n] \Rightarrow d \in OUT[n]$

$\forall n\ d$

$OUT(n,d) :- IN(n,d), !KILL(n,d)$

$\forall n$

$OUT(n,d) :- GEN(n,d)$

Datalog

$[\forall n\ n_i\ d]$

$IN(n,d) :- OUT(n_i, d), pred(n_i, n)$

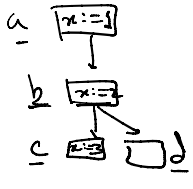
$[\forall n\ d]$

$OUT(n,d) :- IN(n,d), !KILL(n,d)$

$[\forall n]$

$OUT(n,d) :- GEN(n,d)$

pred	
a	b
b	c
b	d



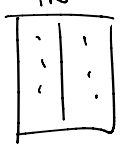
Input relations

IO analysis →

Program → Control-flow graph → Datalog evaluator → Dataflow facts

output relation

IN



OUT

KILL

b	a
c	b
c	b

GEN

a	a
b	b
c	c

Datalog solver produces the least solution to the set of rules.

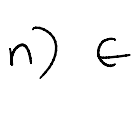
set of rules.

IN(n d) :- OUT(n_i d) pred(n_i n)

" For all CFG nodes n_i , n ,

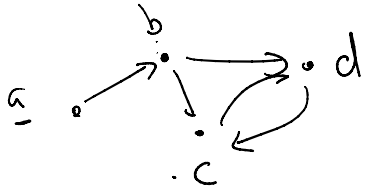
for all assignment statements d

if (n_i d) $\in$ OUT (d $\in$ OUT[n_i])

& if (n_i n) $\in$ pred ( in CFG)

then (n, d) $\in$ IN (d $\in$ IN[n]) "

Ex: Reachability in graphs.



edge ← Adjacency relation

a	b
b	d
b	c
c	d
d	c

Input table

Input relation

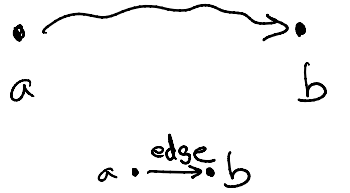
To compute : path (Output relation)

s.t (a, b) $\in$ path iff there is a path from a to b.

R_1 path (a, b) :- edge (a, b).

R_1 path (a, b) :- edge (a, b).

R_2 path (a, b) :- edge (a, c), path (c, b).



To compute : strongly connected components

(a, b) ∈ scc iff a, b are in the same scc.

R_3 scc (a, b) :- path (a, b), path (b, a).

Programmer doesn't worry about

1. Order of computation
2. Order of loops
3. Optimizations
4. Parallelism
5. Provenance tracking
   ⋮

Rule R_1 (imperative)

∀ (a, b) ∈ edge

path := path ∪ {(a, b)}

Rule R_2 (imperative)

R2 path (a, b) :- edge(a, c), path(c, b).

for all (a, c) ∈ edge : for all vertices b :

if (c, b) ∈ path :

path := path ∪ {(a, b)}

Problem : We are redefining path in the middle of the loop.

∀ (a, c) ∈ edge

newPaths := ∅

∀ vertices b, if (c, b) ∈ path :

newPaths := newPaths ∪ {(a, b)}

paths := paths ∪ newPaths.

Problem : Inner loop is wasting time.

∀ (a, c) ∈ edge

np := ∅

∀ paths (d, b) where c = d

np := np ∪ {(a, b)}

p := p ∪ np.

Problem: How to index these sets?
How to parallelize?
How to compute provenance?

- Souffle (open-source, started life in Oracle labs)
- LogicBlox (commercial)
- GraphQL (graph query languages)
- Semmle (static analysis. Acquired by GitHub / Microsoft).