Static Analysis in Datalog

Gang Tan
CSE 597
Spring 2019
Penn State University
DATALOG INTRO
Logic Programming

• Logic programming
  – In a broad sense: the use of mathematical logic for computer programming

• Prolog (1972)
  – Use logical rules to specify how mathematical relations are computed
  – Turing complete
  – Dynamically typed
Logic Programming Overview

• Programming based on logical rules
  – A prolog program is a database of logical rules
  – Example:
    1) seattle is rainy
    2) state college is rainy
    3) state college is cold
    4) If a city is both rainy and cold, then it is snowy
  – Search for solutions based on rules
    • Query: which city is snowy?
What has Logic Programming Been Used For?

• Knowledge representation as a deductive system
  – Rule representation: if A is to the left of B, B is to the left of C, then A is to the left of C
• Expert systems, deductive databases
  – E.g., expert systems to assist doctors: symptoms -> diagnosis
• Logic problems
  – State searching (Rubik’s cube)
• Natural language processing
• Theorem provers
• Reasoning about safety and security
Datalog

• Every Datalog program is a Prolog program
• Enforce restrictions
  – Require well-formed rules
  – Negation must be stratified
  – Disallows function symbols as arguments of predicates
• As a result, Datalog is pure declarative programming
  – All Datalog programs terminate
  – Ordering of rules do not matter
  – Not Turing complete
  – Efficient implementations typically based on databases
Environment: Souffle

- We will use Souffle
  - https://souffle-lang.github.io/
- Demo for the snowy program

```plaintext
.decl rainy(c:symbol)
.decl cold(c:symbol)
.decl snowy(c:symbol)
.output snowy

rainy("seattle").
rainy("stateCollege").
cold("stateCollege").
snowy(c) :- rainy(c), cold(c).
```
Predicates

- Predicates: parameterized propositions
  - pred(x, y, z, ...)
  - Also called an atom
- Examples
  - rainy(x), cold(x), snowy(x): city x is rainy, cold, snowy, respectively
  - italianFood(x): x is italian food
  - square(x, y): y is the square of x
  - xor(x, y, z): the xor of x and y is z
  - parent(x, y): x is y’s parent
  - speaks(x, a): x speaks language a
  - brother (x, y): x is y’s brother
Semantics of Predicates: Relations

• Each predicate specifies a relation: a set of tuples for which the predicate is true
  – The parent predicate: \{(sam,mike), (sussan,mike),(don,sam), (rosy,sam), ... \}
  – The xor predicate: \{(t,t,f), (t,f,t), (f,t,t), (f,f,f)\}

• Notes:
  – Relations are n-ary, not just binary
  – Relations may not be functions
    • E.g., Parent is not a function, since it can map “sam” to two different children of sam
“Directionality” of Relations

• Parameters are not directional
  – No input/output in parameters
  – Prolog programs can be run “in reverse”

• parent: {(sam,mike), (sussan,mike),(don,sam),
  (rosy,sam), ... }
  – parentOfMike(x) :- parent(x,mike).
    • Who are the parents of Mike?
  – childrenOfSussan(c) :- parent(sussan, c).
    • Who are the children of Sussan?
Specify Relations

- Cannot enumerate a relation for large sets
- Specify it using a finite number of logical rules, AKA Horn clauses
Horn Clauses

- A Horn clause has a **head** $h$, which is a predicate, and a **body**, which is a list of literals $l_1, l_2, ..., l_n$, written as
  - $h \leftarrow l_1, l_2, ..., l_n$
  - Souffle syntax: $h :- l_1, l_2, ..., l_n$.
  - $l_i$ is either a predicate or the negation of a predicate
    - That is, either $p$ or $!p$

- This means, “$h$ is true when $l_1, l_2, ..., l_n$ are simultaneously true.”
  - snowy(c) :- rainy(c), cold(c).
    - says, “it is snowy in city c if it is rainy in city c and it is cold in city c.”
  - parent(x, y) :- father(x, y).
  - parent(x, y) :- mother(x, y).

- Note: a clause can have no assumptions; just a head
  - Called facts/axioms; rainy(“seattle”).
Datalog Programming Model

• A program is a database of (Horn) clauses
• The snowy program has 3 facts and 1 rule (or 4 rules)
• Notes:
  – The rule holds for any instantiation of its variables
    • For example, c= “seattle”, or c=“stateCollege”
  – Closed-world assumption: anything not declared is not true
  – Ordering of rules does not matter for results
    • One difference between Datalog and Prolog
    • In Prolog, ordering of rules matters
EDB versus IDB Predicates

• Typically, a Datalog program does not put facts in Datalog programs
  – They are put in an external database

• *Extensional database predicates* (**EDB**)
  – Predicates whose facts are imported from external databases

• *Intensional database predicates* (**IDB**)
  – Predicates whose results are derived from the rules in the program
The snowy Program, Revisited

.decl rainy(c:symbol)
.decl cold(c:symbol)
.decl snowy(c:symbol)
.input rainy, cold
.output snowy

snowy(c) :- rainy(c), cold(c).

• EDB predicates: rainy, cold
• IDB predicates: snowy
The snowy Program, Revisited

- Input: rainy.facts
  - seattle
  - stateCollege
- Input: cold.facts
  - stateCollege
- Output: snowy.csv
  - stateCollege
Datalog Review

• A program is a collection of logical rules
  – $h :\!-\! l_1, l_2, \ldots, l_n$.
  – $l_i$ is either a predicate or the negation of a predicate
    • That is, either $p$ or $!p$
  – Semantics: $h$ is true when $l_1, l_2, \ldots,$ and $l_n$ are simultaneously true

• EDB predicates
  – Predicates whose facts are imported from external databases

• IDB predicates
  – Predicates whose results are derived from the rules in the program
Souffle Datalog

• Two kinds of constants
  – Signed integer numbers: 3, 4, -3
  – Symbols (in quotes): “stateCollege”, “hello”

• Variables
  – e.g. x, y, X, Y, Food

• Predicates
  – e.g. indian(Food), date(year,month,day), Indian(food)
Recursive Rules

• Consider the encoding of a directed graph
  .decl link(n1:number, n2:number)
  .input link

• reachable(i,j): node i can reach node j
  .decl reachable(n1:number, n2:number)
  .output reachable
  reachable(n1,n2) :- link(n1,n2).
  reachable(n1,n3) :- link(n1,n2), reachable(n2,n3).

Rule 1: “For all nodes n1 and n2, if there is a link from n1 to n2, then n1 can reach n2”.
Rule2 (recursive): “For all nodes n1 and n3, if there exists a node n2 so that there is a link from n1 to n2, AND n2 can reach n3, then n1 can reach n3”.
Negation

• Negation is allowed
  – We may put !(NOT) before a predicate
    • E.g., !link(n1,n2)
• Example
  .decl moreThanOneHop(n1:number, n2:number)
  .output moreThanOneHop
  moreThanOneHop(n1,n2) :-
    reachable(n1,n2), !link(n1,n2).
• Restrictions
  – Negation only in the body of a rule; not in the head
    Invalid rule: !reachable(n1,n2) :- !link(n1,n2).
  – Further, Datalog places more restriction than Prolog on negation; more on this later
Well-Formed Datalog

• A rule is well-formed if all variables that appear in the head also appear in the **positive** form of a predicate in the body
  – Ensure that the results are finite and depend only on the actual contents of the database
• Examples of well-formed rules
  reachable(n1,n3) :- link(n1,n2), reachable(n2,n3).
  moreThanOneHop(n1,n2) :-
    reachable(n1,n2), !link(n1,n2).
• Examples of non-well-formed rules
  reachable(n1,n3) :- link(n1,n2), reachable(n2,n1).
  moreThanOneHop(n1,n2) :- !link(n1,n2).
• A Datalog program is well-formed if all of its rules are well-formed
Positive Datalog

• A Datalog Program is positive if all of its rules do not contain negation
Positive Datalog: the “Naïve” Evaluation Algorithm

Idea:
• Start with the empty IDB database
• Keep evaluating rules with EDB and the previous IDB, to get a new IDB
• End when there is no change to IDB

IDB := empty;
repeat
   IDB_{old} := IDB;
   IDB := ApplyAllRules(IDB_{old}, EDB);
until (IDB == IDB_{old})
Naïve Evaluation

Implementation: joining the database tables of link and reachable

reachable(n1,n2) :- link(n1,n2).
reachable(n1,n3) :- link(n1,n2), reachable(n2,n3).

* Slide from “Datalog and Emerging Applications: an Interactive Tutorial”
Semi-Naïve Evaluation

• Observation: each round produces new IDB tuples; the next round we need to only join the new IDB tuples and the EDB
  – No need to perform the join on old IDB tuples

• That is, evaluate the following rule instead
  – reachable(n1,n3) :- link(n1,n2), Δreachable(n2,n3).

* Slide from “Datalog and Emerging Applications: an Interactive Tutorial”
reachable(n1,n2) :- link(n1,n2).
reachable(n1,n3) :- link(n1,n2),
Δreachable(n2,n3).
What about Negation?

• For positive Datalog, we have **monotonicity**
  – We only keep deriving new tuples, **never removing tuples**
  – Pure; functional

• However, with negation, the story changes
  E.g., “unReachable(n1,n2) :-
  \[\text{node}(n1), \text{node}(n2), !\text{reachable}(n1,n2).\]”
  – We cannot trigger this rule, **until all reachable tuples have been derived**
  – In the middle of generating reachable tuples, we cannot possibly know what new reachable tuples might be generated in the future
Precedence Graph

- Nodes: predicates
- Edges \( q \leftarrow p \), if there is a rule in which \( q \) is the head predicate and \( p \) appears in the body
  - That is, \( q \)'s computation depends on \( p \)
  - Label the edge “!” if the predicate \( p \) is negated in the rule
- Note Souffle can produce this graph with “-r”
  - souffle -r graph.html graph.dl
Precedence Graph Example

reachable(n1,n2) :- link(n1,n2).
reachable(n1,n3) :- link(n1,n2), reachable(n2,n3).
unReachable(n1,n2) :-
    node(n1), node(n2), !reachable(n1,n2).
Stratified Negation

- A Datalog program has stratified negation if its precedence graph does not have cycles involving edges labeled with !.
- When the Datalog program is stratified, we can evaluate
  - IDB predicates lowest-stratum-first
  - Once evaluated, treat it as EDB for higher strata.
- Example of non-stratified negation:
  - \( p(x) :- q(x), !p(x) \).

* Slide adapted from “Datalog and Emerging Applications: an Interactive Tutorial”
LOGICAL INTERPRETATION OF DATALOG AND FORALL EMULATION
Math Preliminaries: Propositional Logic

• True, False
• p1, p2, ...: for atomic sentences
  - p1 = x > 3
  - p2 = x < 10
• p1 \& p2
  - e.g., x > 3 \& x < 10
• p1 \lor p2
  - E.g., x > 3 \lor x < 10
• \neg p1
  - \neg (x > 3)
• p1 \rightarrow p2
  - (x > 3) \rightarrow (x < 10)
  - p1 \rightarrow p2 = \neg p1 \lor p2
  - (p1 \rightarrow p2) \& p1 \rightarrow p2 vs. (p1 \rightarrow p2) \rightarrow p1 \rightarrow p2
  - P \rightarrow True
  - False \rightarrow P
• p1 \leftrightarrow p2
Math Preliminaries: Predicate Logic

• \( \forall x. P(x) \)
  – e.g. \( \forall x. x < 10 \rightarrow x < 3 \)

• \( \exists x. P(x) \)
  – e.g. \( \exists x. x > 10 \)
  – e.g. \( \exists y. x = y \times y \)

• Examples
  – \( \forall p. p \lor \neg p. \)
  – \( \forall x. \exists y. y > x. \)
  – For all square numbers, they are greater than or equal to zero
    • \( \forall x. (\exists y. x = y \times y) \rightarrow x \geq 0 \)
Math Preliminaries: Some Logical Equivalences

• $p_1 \rightarrow p_2 \equiv \neg p_1 \lor p_2$

• $\neg(p_1 \land p_2) \equiv \neg p_1 \lor \neg p_2$

• $\neg(p_1 \lor p_2) \equiv \neg p_1 \land \neg p_2$

• $\neg(\forall x. P(x)) \equiv \exists x. \neg P(x)$

• $\neg(\exists x. P(x)) \equiv \forall x. \neg P(x)$

• Exercise
  $\neg(\forall x. (\exists y. x = y \cdot y) \rightarrow x \geq 0) \equiv ?$
Datalog Rule Logical Interpretation

reachable(n1,n2) :- link(n1,n2).
reachable(n1,n3) :- link(n1,n2), reachable(n2,n3).

– Rule 1: ∀n1 n2. link(n1,n2) → reachable(n1,n2).

– Rule 2:
  • ∀n1 n3. (∃n2. link(n1,n2) ∧ reachable(n2,n3)) → reachable(n1,n3)

– In general, for rule “head :- p1, ..., pn.”
  • Interpreted as “∀ x1 ... xi (∃ y1 ... yj. p1 ∧ ... ∧ pn) → head”
  • Variables in the head are universally quantified
  • Variables that appear only in the body are existentially quantified
Datalog Rule Logical Interpretation: the Reverse Direction

reachable(n1,n2) :- link(n1,n2).
reachable(n1,n3) :- link(n1,n2), reachable(n2,n3).

– If we have reachable(n1,n2), then it must be derived by one of the two above rules
  • \( \forall n1 \ n3. \text{reachable}(n1,n3) \rightarrow (\text{link}(n1,n3) \lor \exists n2. \text{link}(n1,n2) \land \text{reachable}(n2,n3)) \).
Forall Emulation

• As we have seen
  – Variables that appear only in the body are existentially quantified
    • \( \forall n_1 \ n_3. (\exists n_2. \text{link}(n_1,n_2) \land \text{reachable}(n_2,n_3)) \rightarrow \text{reachable}(n_1,n_3) \)

• What if we need to a Datalog rule that needs a universal quantification for those variables
Example

- link(n1,n2) and reachable(n1,n2) are as before
- Suppose some nodes are return nodes
  .decl return(n:number)
  .input return
Example: canReachAllReturns(n)

• Goal: define canReachAllReturns(n)
  – It holds when n can reach all return nodes
  – Logically, the body of its rule needs
    \( \forall n1. \text{return}(n1) \rightarrow \text{reachable}(n,n1) \)
  – Want a rule like this:
    
    \[
    \text{canReachAllReturns}(n) :\neg \\
    \quad \text{Forall } n1, \text{return}(n1) \rightarrow \text{reachable}(n,n1).
    \]
  – However, not supported in typical Datalog
Forall Emulation via Negation

• Define a new predicate representing the negation of canReachAllReturns
  – cannotReachAllReturns(n) holds when
    \( \exists n_1. \text{return}(n_1) \land \neg \text{reachable}(n,n_1) \)
  – This is definable in Datalog; it uses only stratified negation

```datalog
.decl cannotReachAllReturns(n:number)
cannotReachAllReturns(n) :-
    node(n), \text{return}(n_1), \neg \text{reachable}(n,n_1).

.decl canReachAllReturns(n:number)
canReachAllReturns(n) :-
    node(n), \neg \text{cannotReachAllReturns}(n).
```
Example: allSuccCanReachAReturn(n)

- **Goal:** define allSuccCanReachAReturn(n)
  - It holds when any successor of n can reach one of the return nodes
  - Logically, the body of the rule needs
    \[
    \forall n_1. \text{link}(n,n_1) \rightarrow \\
    \exists n_2. \text{return}(n_2) \land \text{reachable}(n_1,n_2)
    \]
  - The negation technique won’t work
    - The negation of the above formula gets \(\forall\) when negating \(\exists\)
Forall Emulation via Iteration

• Notice in
  – $\forall n_1. \text{link}(n,n_1) \rightarrow$
    $\exists n_2. \text{return}(n_2) \land \text{reachable}(n_1,n_2))$
  – Predicate link(n,n1) is already known
  – The number of successors of n is finite
  – The successors of n can be ordered

• When these conditions hold, we can apply the technique of forall emulation via iteration
  – Iterate over all successors
Ordering the Successors

// the first successor of n1 is n2
.decl firstSucc(n1:number, n2:number)
firstSucc(n1,n2) :- link(n1,_), n2 = min n: link(n1,n).

// n3 is a greater successor of n1 than n2
.decl succGreaterThan(n1:number, n2:number, n3:number)
succGreaterThan(n1,n2,n3) :- link(n1,n2), link(n1,n3), n2<n3.

// n3 is the next successor of n1 after n2
.decl nextSucc(n1:number, n2:number, n3:number)
nextSucc(n1,n2,n3) :- link(n1,n2), n3=min n: succGreaterThan(n1,n2,n).
Iterate Over Successors

// all successors of n1, up to n2, can reach a return
.decl allSuccCanReachAReturnUpTo(n1:number, n2:number)
allSuccCanReachAReturnUpTo(n1,n2) :-
  firstSucc(n1,n2), return(r), reachable(n2,r).
allSuccCanReachAReturnUpTo(n1,n3) :-
  nextSucc(n1,n2,n3), allSuccCanReachAReturnUpTo(n1,n2),
  return(r), reachable(n3,r).

.decl allSuccCanReachAReturn(n1:number)
allSuccCanReachAReturn(n1) :-
  allSuccCanReachAReturnUpTo(n1,n2), !nextSucc(n1,n2,_).
allSuccCanReachAReturn(n1) :- node(n1), !link(n1,_).
Datalog Review

• A Datalog program is a set of Horn clauses
  – h:- l1, l2, ..., ln.
  – Well-formed rules
• Positive Datalog
  – Semi-naïve evaluation
• Stratified Negation
  – The precedence graph does not have cycles involve negation
• Logical interpretation of Datalog
  – Variables in the head are universally quantified
  – Variables that appear only in the body are existentially quantified
• Forall emulation
  – Via negation
  – Via iteration
IMPLEMENTING DATAFLOW ANALYSIS IN DATALOG
Representing the Program: the CFG

- The control flow
- Initial and final labels

```
.number_type Label

// control flow from l1 to l2
.decl flow(l1:Label, l2:Label)

// l is a initial label
.decl initLabel(l:Label)

// l is a final label
.decl finalLabel(l:Label)

.input flow, initLabel, finalLabel
```
Example

\[x:=5\]^1; \[y:=1\]^2;\nwhile \[x>1\]^3 do (\[y:=x*y\]^4; \[x:=x-1\]^5)\n
flow.facts:

\[
\begin{array}{ll}
1 & 2 \\
2 & 3 \\
3 & 4 \\
4 & 5 \\
5 & 3 \\
\end{array}
\]
Example

\[ [x:=5]^1; [y:=1]^2; \]
while \([x>1]^3\) do ([y:=x*y]^4; [x:=x-1]^5)

initLabel.facts:
1

finalLabel.facts:
5
Representing the Program: Instruction Information

- Representing what vars are used and assigned in each instruction
- Note that this represents only partial info about instructions
  - Sufficient for reaching definition analysis and liveness analysis
  - But insufficient for other analysis

.type Var

// in the instruction with label l, x is assigned
.decl assign(l:Label, x:Var)

// in the instruction with label l, x is read (used)
.decl read(l:Label, x:Var)

.input assign, read
Example

\[x:=5\]; \[y:=1\];
while \[x>1\] do ((\[y:=x\cdot y\]; \[x:=x-1\])

assign.facts:
1  x
2  y
4  y
5  x

read.facts:
3  x
4  x
4  y
5  x
Reaching Definition Analysis

• Recall that
  – It determines at each point what definitions can reach here
Recall the Equations

**kill and gen functions**

\[
\begin{align*}
\text{kill}_{RD}([x := a]^{\ell}) &= \{(x, ?)\} \\
&\quad \cup \{(x, \ell') \mid B^{\ell'} \text{ is an assignment to } x \text{ in } S_+\}
\end{align*}
\]

\[
\begin{align*}
\text{kill}_{RD}([\text{skip}]^{\ell}) &= \emptyset \\
\text{kill}_{RD}([b]^{\ell}) &= \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{gen}_{RD}([x := a]^{\ell}) &= \{(x, \ell)\} \\
\text{gen}_{RD}([\text{skip}]^{\ell}) &= \emptyset \\
\text{gen}_{RD}([b]^{\ell}) &= \emptyset
\end{align*}
\]

**data flow equations:**

\[
\begin{align*}
\text{RD}_{entry}(\ell) &= \begin{cases} 
\{(x, ?) \mid x \in \text{FV}(S_+)\} & \text{if } \ell = \text{init}(S_+) \\
\cup \text{RD}_{exit}(\ell') & \text{otherwise}
\end{cases} \\
\text{RD}_{exit}(\ell) &= (\text{RD}_{entry}(\ell) \setminus \text{kill}_{RD}(B^{\ell})) \cup \text{gen}_{RD}(B^{\ell}) \\
\text{where } B^{\ell} &\in \text{blocks}(S_+)
\end{align*}
\]
Some Auxiliary Rules

.decl initLabelDef(l:Label)

// assume at entry every var is initialized at a special label (? in PPA slides)
initLabelDef(999).

.decl isVar(x:Var)

// collect all variables in assign and read relations
isVar(x) :- assign (_,x).
isVar(x) :- read (_,x).
Datalog Rules for \( RD_{\text{entry}} \) and \( RD_{\text{exit}} \)

// definition of x at label def can reach the point before block with label l
.decl rdEntry(l:Label, x:Var, def:Label)
// definition of x at label def can reach the point after block with label l
.decl rdExit(l:Label, x:Var, def:Label)
.output rdEntry, rdExit

// at entry, every var is assigned at site ?
rdEntry(l,x,def) :- initLabel(l), isVar(x), initLabelDef(def).
// rdEntry of l2 is the union of \{rdExit(l1) | flow(l1,l2)\}
rdEntry(l2,x,def) :- rdExit(l1,x,def), flow(l1,l2).

// def (x,l) can reach the end of block l
rdExit(l,x,l) :- assign(l,x).
// def (x,def) can reach the end of block l, if l doesn't assign x
rdExit(l,x,def) :- rdEntry(l,x,def), !assign(l,x).
Liveness Analysis in Datalog

• Similar to reaching definition analysis
  – Both are may analysis
  – Difference: liveness goes backward; reaching definition analysis goes forward

• Left as homework
Available Expression Analysis

• Challenges
  – Need to represent the program in a different way
    • It’s insufficient to just know what variables are assigned and used in each statement
    • Need to represent expressions: what expressions are used in instructions
  – During available expression analysis, AE sets decrease during analysis
    • Not monotone
Representing the Program: Expressions

Idea: representing the abstract syntax tree of an expression by giving each node in the tree a unique ID

// syntax: e ::= n | x | e1 op e2
//       op ::= + | *
.type Exp = ConstExp | VarExp | OpExp
.type ConstExp
.type VarExp
.type OpExp
.type Var
.type Op

// relations for representing expressions
.decl constExp(id:ConstExp, n:number)
.decl varExp(id:VarExp, x:Var)
.input constExp, varExp, opExp
Example

• “x*y” represented by

varExp.facts:
  e10  x
  e11  y

opExp.facts:
  e20  *  e10  e11
Representing the Program: Blocks

• An assignment or a conditional test

.decl assignStmt(l:Label, x:Var, e:Exp)
.input assignStmt, testCond
Example

\[ [x:=5]^1; [y:=1]^2; \]
\[ \text{while } [x>1]^3 \text{ do (} [y:=x*y]^4; [x:=x-1]^5) \]

assignStmt.facts:
1 \quad x \quad e1
2 \quad y \quad e2
4 \quad y \quad e20
5 \quad x \quad e21

testCond.facts:
3 \quad > \quad e10 \quad e2
Recall Available Expression Analysis

\[ \text{kill and gen functions} \]

\[
\begin{align*}
\text{kill}_{AE}([x := a]^{\ell}) &= \{ a' \in \text{AExp}_{*} \mid x \in FV(a') \} \\
\text{kill}_{AE}([\text{skip}]^{\ell}) &= \emptyset \\
\text{kill}_{AE}([b]^{\ell}) &= \emptyset \\
\text{gen}_{AE}([x := a]^{\ell}) &= \{ a' \in \text{AExp}(a) \mid x \not\in FV(a') \} \\
\text{gen}_{AE}([\text{skip}]^{\ell}) &= \emptyset \\
\text{gen}_{AE}([b]^{\ell}) &= \text{AExp}(b)
\end{align*}
\]

\[ \text{data flow equations: } AE^= \]

\[
\begin{align*}
\text{AE}_{\text{entry}}(\ell) &= \begin{cases} \\
\emptyset & \text{if } \ell = \text{init}(S_{*}) \\
\bigcap \{ \text{AE}_{\text{exit}}(\ell') \mid (\ell', \ell) \in \text{flow}(S_{*}) \} & \text{otherwise}
\end{cases} \\
\text{AE}_{\text{exit}}(\ell) &= (\text{AE}_{\text{entry}}(\ell) \setminus \text{kill}_{AE}(B^{\ell})) \cup \text{gen}_{AE}(B^{\ell}) \\
&\text{where } B^{\ell} \in \text{blocks}(S_{*})
\end{align*}
\]
Datalog Rules for Kill/Gen Sets

// e is in the kill set of block l
.decl killAE(l:Label, e:Exp)
killAE(l,e) :- assignStmt(l,x,_), isComplexExp(e), freeVar(e,x).

// e is in the gen set of block l
.decl genAE(l:Label, e:Exp)
genAE(l,e) :- assignStmt(l,x,a), subExp(a,e), !freeVar(e,x).
genAE(l,e) :- testCond(l,_,a1,a2), (subExp(a1,e); subExp(a2,e)).
Computing the AE Sets

• Challenge: AE sets decrease during computation
• Idea: the complement of AE sets always increase
Rules for Computing the Complement of AE Sets

// e may not be available at the entry of block l
.decl mayNotBeAvailableEntry (l:Label, e:Exp)
// e may not be available at the exit of block l
.decl mayNotBeAvailableExit (l:Label, e:Exp)

// at the entry, no expression is available
mayNotBeAvailableEntry(l,e) :- initLabel(l), isComplexExp(e).

// MNAE_e(l) := union {MNAE_x(l') | (l',l) in flow}
mayNotBeAvailableEntry(l2,e) :- mayNotBeAvailableExit(l1,e), flow(l1,l2).

// Since AE_x(l) = (AE_e(l) \ kill(l)) union gen(l), we have
// MNAE_x(l) = (MNAE_e(l) union kill(l)) \ gen(l)
mayNotBeAvailableExit(l,e) :- mayNotBeAvailableEntry(l,e), !genAE(l,e).
mayNotBeAvailableExit(l,e) :- killAE(l,e), !genAE(l,e).
Computing AE Sets

// e must be available at the exit of block l
.decl availableExpExit (l:Label, e:OpExp)
availableExpExit(l,e) :- isLabel(l),
isComplexExp(e), !mayNotBeAvailableExit(l,e).
Very Busy Expression Analysis

• Similar to available expression analysis
  – Both are must analysis
• Left as homework
Datalog Additional References

- Datalog and Emerging Applications: an Interactive Tutorial
  - [https://www.cis.upenn.edu/~boonloo/research/talks/sigmod11-tutorial-all.pptx](https://www.cis.upenn.edu/~boonloo/research/talks/sigmod11-tutorial-all.pptx)

- Program analysis with Datalog tutorial

- [http://infolab.stanford.edu/~ullman/dragon/w06/lectures/datalog.pdf](http://infolab.stanford.edu/~ullman/dragon/w06/lectures/datalog.pdf)

- [https://polybox.ethz.ch/index.php/s/s5BSzgE7lPe8tio?fbclid=IwAR0Yw9aYPrSEhMgR_Fn8Vz2K2TjzbZTrZGarrrvGGW3xztmbvuOQB_2QvJw](https://polybox.ethz.ch/index.php/s/s5BSzgE7lPe8tio?fbclid=IwAR0Yw9aYPrSEhMgR_Fn8Vz2K2TjzbZTrZGarrrvGGW3xztmbvuOQB_2QvJw)

- What you always wanted to know about Datalog (and never dared to ask)