EDA045F: Program Analysis

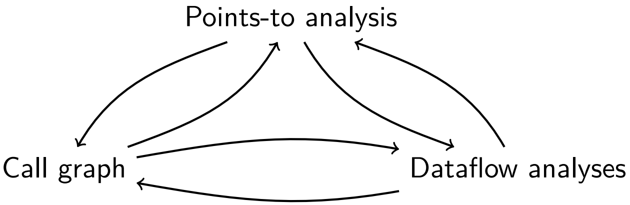
LECTURE 6: DATALOG

Christoph Reichenbach

In the last lecture...

- Pointer Analysis
  - Points-To Analysis
  - Alias Analysis
- Concrete Heap Graphs
- Abstract Heap Graphs
- Access Paths
- Heap Summarisation
  - Call-site
  - Variable-based
  - k-Limiting
- Steensgard's Analysis
- Andersen's Analysis
- Call graphs

Dependencies



- Mutual dependencies across program analyses
  - **Either:** loss of precision/soundness
    - Ignore dependence, run sequentially
    - Conservative/optimistic assumptions
  - **Or:** complex engineering
    - Each analysis may have to feed worklists of other analyses

Solving Complex Interdependency

- Engineering OO/imperative code for re-use of mutually dependent worklist analyses is complex
- Alternative: *Declarative specification* of analyses
  - Specify algorithms declaratively
  - Declarative language compiler automates handling of mutual dependencies
- Approaches:
  - Attribute Grammars
  - SAT / SMT solving
  - Prolog
  - **Datalog**

Facts

- **Object**: *any entity that we care about*
  - Analogous to primitive value, unique object
- **Relation**: *set of tuples* that encode relationships between objects

Example:

- Elements $= \{\mathrm{H}, \mathrm{He}, \mathrm{Li}, \mathrm{Be}, \ldots\}$
- **Objects** $=$ **Elements** $\cup \mathbb{N}$
- $\textsc{MassNumber} \subseteq$ Element $\times \mathbb{N}$

$$\begin{array}{|c|c|} \mathrm{H} & 1 \\ \mathrm{H} & 2 \\ \mathrm{H} & 3 \\ \mathrm{He} & 2 \\ \ldots & \ldots \end{array}$$

- $\textsc{Elements}$ is also a (unary) relation

Relations and Predicate Symbols

$\textsc{MassNumber} \subseteq \mathsf{Element} \times \mathbb{N} =$

H	1
H	2
H	3
He	2
...	...

- We use the terms **Relation**, **Predicate**, and **Table** interchangeably
- A **Predicate Symbol** is the name that we assign to a relation:
  - $\textsc{MassNumber}$ is a predicate symbol
  - The following *tuples* make up the relation bound to $\textsc{MassNumber}$:

$$\{\langle \mathrm{H}, 1\rangle, \langle \mathrm{H}, 2\rangle, \langle \mathrm{H}, 3\rangle, \langle \mathrm{He}, 2\rangle, \ldots\}$$

- An **atom** is a predicate symbol plus parameters:
  - $\textsc{MassNumber}(\mathrm{H}, 1)$

Datalog Programs

- A Datalog program is a collection of *Horn Clauses*:

$$H \leftarrow B_1 \wedge \ldots \wedge B_k.$$

  written as

$$H \text{ :- } B_1, \ldots, B_k.$$

- $H, B_1, \ldots, B_k$ are called *literals*
- H: Head
- $B_1, \ldots, B_k$: Body
- Semantics: if $B_1, \ldots, B_k$ are true:

$\Rightarrow$ H is also true

- Order of the rules is irrelevant
- Order of the conjuncts in the body (literals) is irrelevant

Rules in Detail

Literals may take parameters:

$$\textsc{Head}(v_1, \ldots, v_j) \text{ :- } \textit{Body}.$$

- where $\textit{Body} = \mathrm{B}_1(v_1^1, \ldots, v_{j_1}^1), \ldots, \mathrm{B}_k(v_1^k, \ldots, v_{j_k}^k)$
- $v_1, \ldots, v_j$ (etc.) are variables
- $v_1, \ldots, v_j$ *must* also appear in *Body*
- Semantics:
  - For all tuples $\langle o_1, \ldots, o_k \rangle$ for which we can show that

$$\textit{Body}[v_1 \mapsto o_1, \ldots, v_k \mapsto o_k]$$

  - we add $\langle o_1, \ldots, o_k \rangle \in \textsc{Head}$
  - Requires a mechanism to solve *unification*
  - **Set semantics**: Each tuple added at most once

Extracting Information

$\text{CONNECTION} =$

from	to	km	shortest train ride
Lund	Malmö	18.8	11
Lund	Eslöv	21.7	10
Lund	Landskrona	33.0	16
Lund	Helsingborg	54.5	27
Lund	Staffanstorp	10.7	-1
Staffanstorp	Malmö	15.6	-1

Set of all places:

$$\text{PLACE}(x) \quad :- \quad \text{CONNECTION}(x, y, \mathit{distance}, \mathit{traintime}).$$
$$\text{PLACE}(y) \quad :- \quad \text{CONNECTION}(x, y, \mathit{distance}, \mathit{traintime}).$$
$$\text{PLACE}(x) \quad :- \quad \text{CONNECTION}(x, \_, \_, \_).$$
$$\text{PLACE}(y) \quad :- \quad \text{CONNECTION}(\_, y, \_, \_).$$

$$\text{PLACE} = \{\text{Lund}, \text{Staffanstorp}, \text{Malmö}, \text{Eslöv}, \text{Landskrona}, \text{Helsingborg}\}$$

Filtering

$\textsc{Connection} =$

Lund	Malmö	18.8	11
Lund	Eslöv	21.7	10
Lund	Landskrona	33.0	16
Lund	Helsingborg	54.5	27
Lund	Staffanstorp	10.7	-1
Staffanstorp	Malmö	15.6	-1

All train connections:

$$\textsc{TrainConnection}(x, y, t) \mathrel{:\!-} \textsc{Connection}(x, y, \_, t), t \geq 0.$$

- A,B means that both A and B must be true
- Variables (x, y, t) are shared across each rule

$$\textsc{TrainConnection} = \{ \langle \text{Lund}, \text{Malmö}, 11 \rangle, \langle \text{Lund}, \text{Eslöv}, 10 \rangle, \langle \text{Lund}, \text{Landskrona}, 16 \rangle, \langle \text{Lund}, \text{Helsingborg}, 27 \rangle \}$$

Primitive Relations

$$\textsc{TrainConnection}(x, y, t) \text{ :- } \textsc{Connection}(x, y, \_, t), t \geq 0.$$

- $\geq$ denotes a relation, too:

$$(\geq)(t, 0)$$

- The ‘table’ underlying $\geq$ is infinite
- Challenge: computing table for

$$\textsc{Positive}(x) \text{ :- } x \geq 0.$$

Parents and Ancestors

$\textsc{Connection} =$

Lund	Malmö	18.8	11
Lund	Eslöv	21.7	10
Lund	Landskrona	33.0	16
Lund	Helsingborg	54.5	27
Lund	Staffanstorp	10.7	-1
Staffanstorp	Malmö	15.6	-1
Sylt	Malmö	-1	334

All places reachable by car:

$$
\begin{array}{lcl}
\textsc{Reachable}(x, y) & \text{:-} & \textsc{Connection}(x, y, d, \_), d \geq 0. \\
\textsc{Reachable}(y, x) & \text{:-} & \textsc{Reachable}(x, y). \\
\textsc{Reachable}(x, z) & \text{:-} & \textsc{Reachable}(x, y), \textsc{Reachable}(y, z). \\
\textsc{Reachable}(x, x) & \text{:-} & \textsc{Place}(x).
\end{array}
$$

- Can each place reach itself?

Datalog Literals and Terms

- **Literals** in Datalog communicate about tuples in a relation:

$$\textsc{Connection}(\text{Lund}, \text{Malmö}, 18.8, 11)$$

- The parameters of the literal are called *Terms*, must be:
  - Variable, or
  - Constant
- **Ground literals** (like the above) have only constants as terms
- The below is a literal, but not a *ground* literal:

$$\textsc{Connection}(\text{Lund}, x, 18.8, y)$$

Datalog Programs: Syntax

$$
\begin{array}{lcl}
\textit{Program} & ::= & \langle \textit{Rule} \rangle \star \\
\textit{Rule} & ::= & \langle \textit{Atom} \rangle \ \texttt{:-} \ \langle \textit{Literal} \rangle \star \ \ . \\
\textit{Atom} & ::= & \langle \textit{PredicateSymbol} \rangle \ (\ \langle \textit{Terms} \rangle \mathbf{?}) \\
 & | & \langle \textit{Term} \rangle = \langle \textit{Term} \rangle \\
 & | & \langle \textit{Term} \rangle \leq \langle \textit{Term} \rangle \\
\textit{Terms} & ::= & \langle \textit{Term} \rangle \\
 & | & \langle \textit{Terms} \rangle \ , \ \langle \textit{Term} \rangle \\
\textit{Term} & ::= & \langle \textit{Variable} \rangle \mid \langle \textit{Constant} \rangle \\
\textit{Literal} & ::= & \langle \textit{Atom} \rangle \\
 & | & \neg \langle \textit{Atom} \rangle \\
 & & \\
\textit{PredicateSymbol} & ::= & \textit{id} \\
\textit{Variable} & ::= & \textit{id} \\
\textit{Constant} & ::= & \textit{number} \mid \textit{string} \ldots
\end{array}
$$

Negation

- Negation is a popular extension to pure Datalog:

$$\textsc{Accessible}(\mathit{room}) \text{ :- } \textsc{Doors}(\mathit{room}, \mathit{door}), \neg\textsc{Locked}(\mathit{door}).$$

- Paradoxical rules may be **disallowed**:

$$\textsc{Accessible}(\mathit{room}) \text{ :- } \neg\textsc{Accessible}(\mathit{room}).$$

- Variables that only occur negatively and in the head may be **disallowed**:

$$\textsc{Available}(\mathit{room}) \text{ :- } \neg\textsc{Reserved}(\mathit{room}).$$

IDB and EDB

- Two types of database tables:
- **EDB** = Extensional Database
  - Elements explicitly enumerated
  - In Datalog: Input relations
- **IDB** = Intensional Database
  - Elements described by their properties
  - In datalog: Derived from rules
- Output marked explicitly in typical Datalog implementations

Interesting Properties

- *Monotonicity*:
  - Datalog *without negation* is *monotonic*
    - Adding EDB tuples can only ever add IDB tuples
- *Complexity*:
  - Consider Datalog with the following properties:
    - Negation of EDB relations only
    - Numeric constants in bodies
    - $(=)$ and $(\leq)$ (can be simulated through EDBs)
  - This extension of Datalog can express *exactly* all problems in the complexity class **P**.

Summary

- **Datalog** programs are sets of **Horn clauses**:

$$\textsc{Head}(v) \mathrel{:\!-} \textsc{Body}_1(\ldots), \ldots, \textsc{Body}_k(\ldots)$$

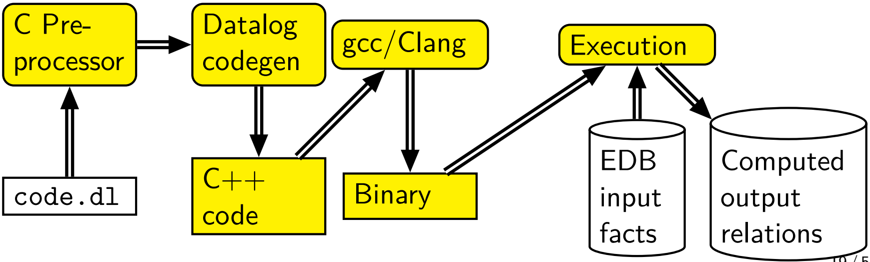
- The rule **Head** and the conjuncts of the **Body** are **Literals**
- Literals consist of a **Predicate Symbol** and **Terms**
- **Terms** can be varibales or constants
- **Negation** is permitted in some extensions
- Datalog reasons over relations that are bound to the predicate symbols
- Relations can be **IDB** (derived) or **EDB** (enumerated, typically input)

The Soufflé System

Soufflé

- Datalog implementation
- UPL licence (Open Source)
- Extends Datalog both syntactically and semantically
- Reads/emits various file formats (sqlite, csv, ...)

Running `souffle code.dl`:



Soufflé Example

```
.decl Place(placename: symbol)
.decl Distance(from: symbol, to: symbol, dist: number)
.decl Reachable(source: symbol, destination: symbol)

Reachable(s, d) :- Distance(s, d, _).
Reachable(s, d) :- Reachable(s, i, _), Reachable(i, d, _).

// Rome is reachable from anywhere:
Reachable(s, "Rome") :- Place(s).

.decl Unreachable(place: symbol)
Unreachable(place) :- Place(place), !Reachable(_, place).
```

- Predicates must be declared with **.decl**
- Comments can be written in C/C++ style
- Parameters are *typed*. Two primitive types:
  - `symbol`: A string
  - `number`: A 32 bit signed integer

Input Relations

```
.decl Distance(from: symbol, to: symbol, dist: number)
.input Distance(IO=file, filename="distance.csv", delimiter=",")
```

- **.input** directive marks relation as EDB
  - Read from external file
- Here, the input file is a text file of comma-separated inputs

distance.csv:

```
Lund,Malmö,19
Lund,Eslöv,22
Lund,Landskrona,33
Lund,Helsingborg,55
Lund,Staffanstorp,11
```

Equivalent Soufflé code:

```
Distance("Lund", "Malmö", 19).
Distance("Lund", "Eslöv", 22).
Distance("Lund", "Landskrona", 33).
Distance("Lund", "Helsingborg", 55).
Distance("Lund", "Staffanstorp", 11).
```

Output Relations

```
.decl Distance(from: symbol, to: symbol, dist: number)
.output Distance(IO=file, filename="distance.csv", delimiter=",")
```

- Analogous to **.input**
- Default settings write to `Distance.csv` as tab-separated values:

```
.decl Distance(from: symbol, to: symbol, dist: number)
.output Distance
```

Built-In Predicates

- Soufflé provides built-in infix predicates on `number` × `number`:

  >, >, <=, >=

- The following predicates are defined for all types:

  =, !=

```
ShoppingList(name, price) :-
    AvailableItem(name, price),
    price < 20,
    name = "Chocolate".
```

Conjuncive Heads

- Soufflé allows joining clauses that share a body:

$$H_1, \ldots, H_k \text{ :- } B.$$

- Semantically equivalent to:

$$\begin{array}{ccc} H_1 & \text{:-} & B. \\ & \vdots & \\ H_k & \text{:-} & B. \end{array}$$

```
Place(from),
Place(to),
Reachable(from, to) :- Distance(from, to, _).
```

Disjunction

- Soufflé allows disjunctions (‘A or B’) in a body:

$$H \text{ :- } B_p, (B_1; \ldots; B_k), B_s.$$

- Semantically equivalent to:

$$\begin{array}{lcl} H & \text{:-} & B_p, B_1, B_s. \\ & \vdots & \\ H & \text{:-} & B_p, B_k, B_s. \end{array}$$

```
Poisonous(a) :-
    InKitchen(a),
    (Expired(a) ;
     Contains(a, b), Poisonous(b)).
```

Terms and Functions

- Soufflé extends Datalog's *Terms* to *Expressions*:

```
Area(obj, height*width) :- Rectangle(obj, height, width).
Volume(obj, edge^3) :- Cube(obj, edge).
```

- Expressions do not participate in unification
  Not allowed (`x` cannot be bound in body):

```
C(a, x) :- B(a, x + 1).
```

- Expressions break the termination guarantee:

```
Number(x + 1) :- Number(x).
```

Functions

- `ord(s:symbol):number`
  Globally unique ID for string *s*
- `strlen(s:symbol):number`
  String length
- `to_number(s:symbol):number`
- `to_string(n:number):string`
- `lnot(n:number):number`
  Logical negation
- `bnot(n:number):number`
  Bitwise negation
- `(x:number + y:number):number`
- `(x:number - y:number):number`
- `(x:number * y:number):number`
- `(x:number / y:number):number`
- `(x:number % y:number):number`
  Remainder of the division $\frac{x}{y}$
- `band(x:number, y:number):number`
  Bitwise *and*
- `bor(x:number, y:number):number`
  Bitwise *or*
- `bxor(x:number, y:number):number`
  Bitwise *exclusive or*
- `land(x:number, y:number):number`
  Logical *and*
- `lor(x:number, y:number):number`
  Logical *or*
- `max(x:number, y:number):number`
- `min(x:number, y:number):number`
- `cat(x:symbol, y:symbol):symbol`
  String concatenation
- `substr(s:symbol, from:number, to:number):symbol`
  Substring extraction

Aggregation

```
TrafficHub(place) :-
    Place(place),
    connections = count:
        Reachable(place, _),
    connections >= 100.
```

- *Aggregation* merges a set of values into a single value.
- Soufflé supports four *aggregation operators*:
  - `count:` E
  - `min` x`:` E
  - `max` x`:` E
  - `sum` x`:` E
- x can be an expression with (possibly) multiple variables.
- For `min`, `max`, `sum`, if E is empty, the program aborts.

```
CheapestProducts(product, cost) :-
    Product(product),
    cost = min price:
        Price(product, _, price).
```

Types

- Soufflé allows custom types:
  - **.symbol_type** `st`: `st` inherits all `symbol` built-ins
  - **.number_type** `nt`: `nt` inherits all `number` built-ins
- Tagged `union` type construction:

$$\texttt{.type t = t}_1 | \ldots | \texttt{t}_k$$

- Values of different types are never equal:

$$("x" : \texttt{tomato})\texttt{!=}("x" : \texttt{cabbage})$$

```
.symbol_type apple
.decl Apple(a:apple).
.decl Tomato(p:tomato).
.decl Cabbage(c:cabbage).
.type fruit = apple | tomato
.type vegetable = cabbage | tomato
Vegetable(x) :- Cabbage(x).
Vegetable(x) :- Tomato(x).
```

Summary

- Soufflé is an extension of Datalog
- Two built-in types: `symbol`, `number`
- **Built-in predicates** on numbers, strings
- Terms are extended to support built-in operations (addition, etc.)
- **Aggregation** operations for summing up or computing the minimum etc.
- **Conjunctive heads** and **Disjunctions** add syntactic sygar
  - Are also exploited for optimisation
- Explicit declaration for input and output behaviour

Evaluating Datalog

- Several evaluation strategies
- *Incremental on input*:
  - Exploit monotonicity: grow IDB facts as EDB grows
  - For negative literals:
    - Delete and re-derive
    - Optimisations available (counting, provenance tracking)
- *On-demand*:
  - *Forward-chaining*:
    - Find rule heads that match fact that we're checking
    - Recursively try to prove atoms in body
    - Memoise results

Evaluating Datalog Efficiently

- Populate all IDB tables according to rules
- State of the art for full evaluation: *Semi-Naive Evaluation*
  - Needs *dependency graph* between relations
  - X *depends on* Z iff:
    - there is a rule $\mathrm{X}(\ldots)$:- $\ldots\mathrm{Z}(\ldots)\ldots$, or
    - there is a rule $\mathrm{X}(\ldots)$:- $\ldots\mathrm{Y}(\ldots)\ldots$, and Y depends on Z

Nonrecursive Case

Example:

$$\mathrm{H}(x, y) \text{ :- } \mathrm{A}(x, \_, z), \mathrm{B}(x, y, z).$$

- Requirement: A, B do not depend on H
- Implementation idea: *nested loops*:

  for $\langle x_1, \_, y_1 \rangle \in$ `A` **do**
      for $\langle x_2, y_2, z_2 \rangle \in$ `B` **do**
          if $x_1 = x_2$ `and` $y_1 = y_2$ **then**
              `H` $:=$ `H` $\cup \{\langle x_1, y_1 \rangle\}$
      done
  done
- Faster looping possible by exploiting representation (e.g., sorted B-trees)

Nonrecursive Case with Test

Example:

$$\mathrm{H}(x, y) \text{ :- } \mathrm{A}(x, y), \mathrm{B}(x, y).$$

- Requirements:
  - A, B do not depend on H
  - All variables occurring in $\mathrm{B}(\ldots)$ are bound by literals to the left of $\mathrm{B}(\ldots)$
- Implementation idea: *contains-check instead of loop*:

  for $\langle x_1, y_1 \rangle \in \mathtt{A}$ **do**
      if $\langle x_1, y_1 \rangle \in \mathtt{B}$ **then**
          $\mathtt{H} := \mathtt{H} \cup \{\langle x_1, y_1 \rangle\}$
      done
  done

Simple Recursion

Example:

$$\mathrm{H}(x,z) \text{ :- } \mathrm{A}(x,y), \mathrm{H}(y,z).$$

- Implementation idea: *fixpoint*:

$R_{\mathtt{H}} := \mathtt{H}$
do
  $\Delta\mathtt{H} = \emptyset$
  for $\langle x_1, y_1\rangle \in \mathtt{A}$ **do**
    for $\langle y_2, z_2\rangle \in R_{\mathtt{H}}$ **do**
      if $y_1 = y_2$ **and** $\underline{\langle x_1, z_2\rangle \notin \mathtt{H}}$ **then begin**
        $\mathtt{H} := \mathtt{H} \cup \{\langle x_1, z_2\rangle\}$
        $\Delta\mathtt{H} := \Delta\mathtt{H} \cup \{\langle x_1, z_2\rangle\}$
      end
  done
  $R_{\mathtt{H}} := \Delta\mathtt{H}$
done
while $\Delta\mathtt{H} \neq \emptyset$

- $\Delta\mathrm{H}$ acts as *worklist*

Mutual Recursion

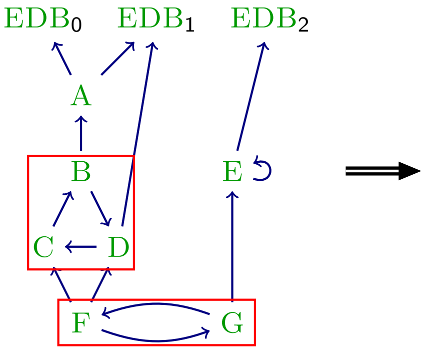
Example:

$$\begin{array}{lcl} \mathrm{H}(x,z) & \text{:-} & \mathrm{A}(x,y), \mathrm{K}(y,z). \\ \mathrm{K}(x,z) & \text{:-} & \mathrm{B}(x,y), \mathrm{H}(y,z). \end{array}$$

- Implementation idea: *fixpoint with multiple worklists*
- Analogous to simple recursion:
  - $\Delta\mathrm{H}$ for updates to H
  - $\Delta\mathrm{K}$ for updates to K
- Iterate rules until both $\Delta\mathrm{H}$ and $\Delta\mathrm{K}$ are empty

Evaluation Strata

- Strategy:
  - Evaluate dependencies first
  - Evaluate mutual dependencies $\boxed{\text{together}}$
  - Evaluate recursive dependencies with fixpoint
- *Stratify* predicates based on dependencies:



Strata:

1. Nonrecursive: A
2. **Fixpoint**: E
3. **Fixpoint**: B, C, D
4. **Fixpoint**: F, G

Negation and Aggregation

- Evaluating negative literals $\neg \mathrm{P}(v_1, \ldots, v_k)$:
  - Static check: all $v_1, \ldots, v_k$ must be bound before testing literal
  - Static check: P must be evaluated in earlier stratum
  - Use negated 'contains' check
- Evaluating aggregation:
  - Same stratification requirements as for negation

Optimisations

- Eliminate dead tables / rules
- Predicate reordering
- Optimised table representations
  - Sorted:
    - RB-Trees (ordered iteration)
    - B-Trees (ordered iteration, $O(n)$ joins with matching indices)
    - Tries (compression for common prefixes, *Leapfrog Triejoin*)
  - Hashsets ($O(1)$ contains checks)
  - Binary Decision Diagrams (BDDs, challenging to tune but can be very compact)

  ...
- Inlining
- *Magic Sets*
- *Leapfrog*

Predicate Positioning (Unsorted)

$$\mathrm{H}(x,y,z) \text{ :- } \overbrace{\textsc{Small}(x,y,z)}^{\text{Iterate over small tables first}}, \underbrace{y > 0}_{\text{Primitive tests early}}, \textsc{Big}(z,t,y), \overbrace{\neg\mathrm{Q}(t)}^{\text{Negated atoms must be } \textit{ground}}.$$

- Negation, aggregation, built-in tests:
  - Position after free variables bound
- Fail fast
- Exploit representations where possible
  - Testing usually faster than iteration
- Faster / more selective tests earlier
- Optimal positioning NP-hard, but:
  - many rules are short
  - heuristics help

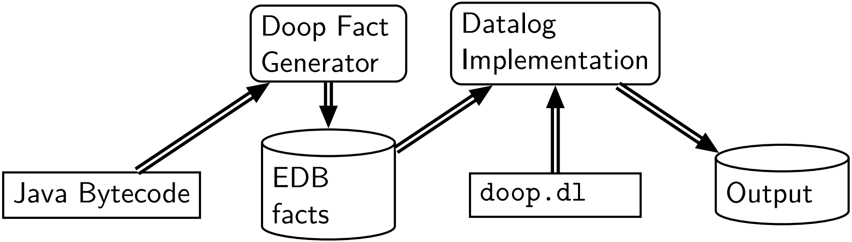
Summary

- Different evaluation strategies for Datalog
- **Semi-Naive Evaluation** is state-of-the art for full evaluation
  - Find dependencies
  - Cluster rules by dependencies
  - **Stratify** evaluation
  - Iterate with **Deltas** (equivalent to worklists)
- Practical implementations use further optimisation strategies

Doop

- Points-to analysis framework
- Core analysis implemented in Datalog
  - Based on Andersen's Analysis (last lecture)
- Supports different forms of *x-sensitivity*

Doop Overview

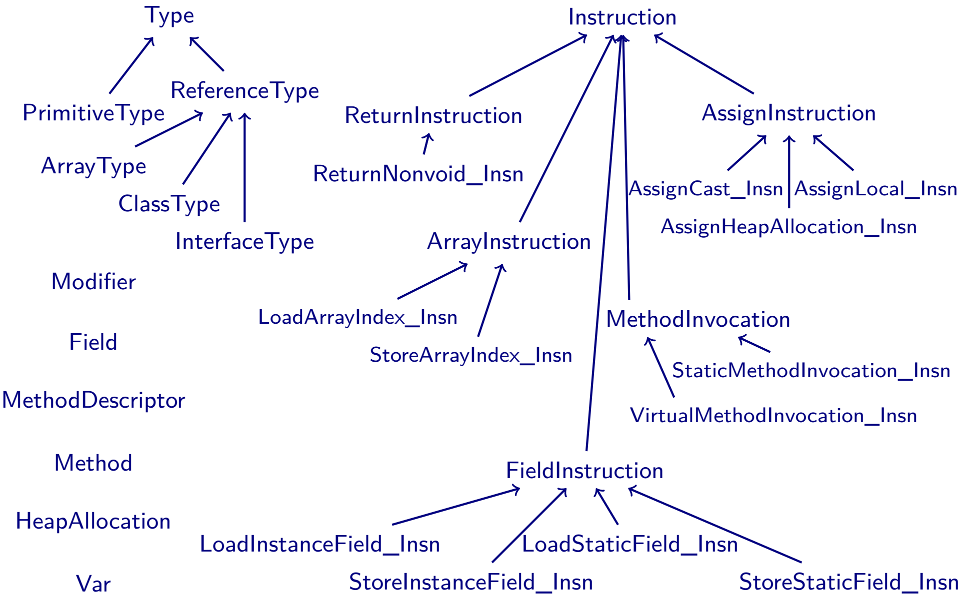


- Doop first generates EDB facts by scanning programs
  - Uses Soot (can also use Wala, other tools)
- Then analyses the facts using Datalog code
- Different (Datalog-based) analyses available
- Output:
  - Call graph
  - Points-to graph

Doop Key Types

- Type: Java type
- Var: Java variable (local or parameter)
  VAR_TYPE(?var:`Var`, ?type:`Type`)
- Method: Defined method
  FORMALPARAM(?index:`number`, ?method:`Method`, ?var:`Var`)
  (lists all parameter `Var`s for the given method, in order)
- MethodDescriptor: Method signature (parameter & return tyeps)
  BASIC.METHODLOOKUP(?simplename:`symbol`,
  ?descriptor:`MethodDescriptor`,
  ?type:`Type`,
  ?method:`Method`)
  (Resolve method name + signature + type to the invoked `Method`)
- Instruction: Soot instruction
  INSTRUCTION_METHOD(?insn:Instruction, ?inMethod:Method)
  (Connect instructions to the method that they occur in)
- HeapAllocation: Allocation site
- Field: Static or dynamic field

Doop Types (all)



Doop Outputs

- ASSIGN(?to:`Var`, ?from:`Var`)
  Variable assignment may take place
- VARPOINTSTO(?heap:`HeapAllocation`, ?var:`Var`)
  Variable may point to object from given heap allocation site
- INSTANCEFIELDPOINTSTO(?heap:`HeapAllocation`,
  ?fld:`Field`,
  ?baseheap:`HeapAllocation`)
  The field ?baseheap.?fld may point to the object allocated at ?heap
- STATICFIELDPOINTSTO(?heap:`HeapAllocation`, ?fld:`Field`)
- CALLGRAPHEDGE(?invocation:`MethodInvocation`, ?meth:`Method`)
  Instruction ?invocation may call method ?meth
- ARRAYINDEXPOINTSTO(?baseheap:`HeapAllocation`,
  ?heap:`HeapAllocation`)
- REACHABLE(?method:`Method`)
  The given method may be reached when executing the program

Doop Call Graph (1/2)

- As example, consider the computation of:

  CALLGRAPHEDGE(`?invocation`, `?tomethod`)

- 'May the instruction `?invocation` invoke method `?method`'?

CALLGRAPHEDGE(?invocation, ?tomethod) :-
  REACHABLE(?inmethod),
  STATICMETHODINVOCATION(?invocation, ?tomethod, ?inmethod).

CALLGRAPHEDGE(?invocation, ?tomethod) :-
  REACHABLE(?inmethod),
  INSTRUCTION_METHOD(?invocation, ?inmethod),
  METHODINVOCATION_METHOD(?invocation, ?tomethod).

Doop Call Graph (2/2)

```
CallGraphEdge(?invocation, ?toMethod) :-
    Reachable(?inMethod),
    Instruction_Method(?invocation, ?inMethod),
    VirtualMethodInvocation_Base(?invocation, ?base),
    VarPointsTo(?heap, ?base),
    HeapAllocation_Type(?heap, ?heaptype),
    VirtualMethodInvocation_SimpleName(?invocation, ?simplename),
    VirtualMethodInvocation_Descriptor(?invocation, ?descriptor),
    basic.MethodLookup( ?simplename, ?descriptor,
                        ?heaptype, ?toMethod).
```

x-Sensitivity in Doop

- Program analyses can be made more precise by adding different forms of *sensitivity*:
  - Flow-sensitivity
  - Context-sensitivity (which Doop calls *Call-site sensitivity*)
  - Object-sensitivity

  ...
- Doop calls them all *context sensitivity*
- Doop is designed to be easy to extend for such sensitivities

$$
\begin{tikzcd}
& \textsc{PointsTo}(v, h) \arrow[d, Rightarrow] & \\
& \textsc{PointsToCtx}(c, v, h) & \\
\textsc{PointsToCtx}(\texttt{"c0"}, v, h) \arrow[ur] & \textsc{PointsToCtx}(\texttt{"c1"}, v, h) \arrow[u] & \textsc{PointsToCtx}(\texttt{"c2"}, v, h) \arrow[ul]
\end{tikzcd}
$$

Summary

- **Doop** is a points-to analysis framework based on Datalog
  1. First extracts program facts (via Soot) into tables
  2. Then analyses tables with Datalog code
- Flow-insensitive
- Datalog code computes many facts of interest
- Extensible to support different forms of x-Sensitivity

Review

- Datalog
- Soufflé
- Doop

Homework

1. Basic Datalog
2. Implement call graph analyses
3. Add object-sensitivity to Doop
4. Build fact database

To be continued...

- Break for two weeks, then return on WEDNESDAYS
  2018-11-07 15:15