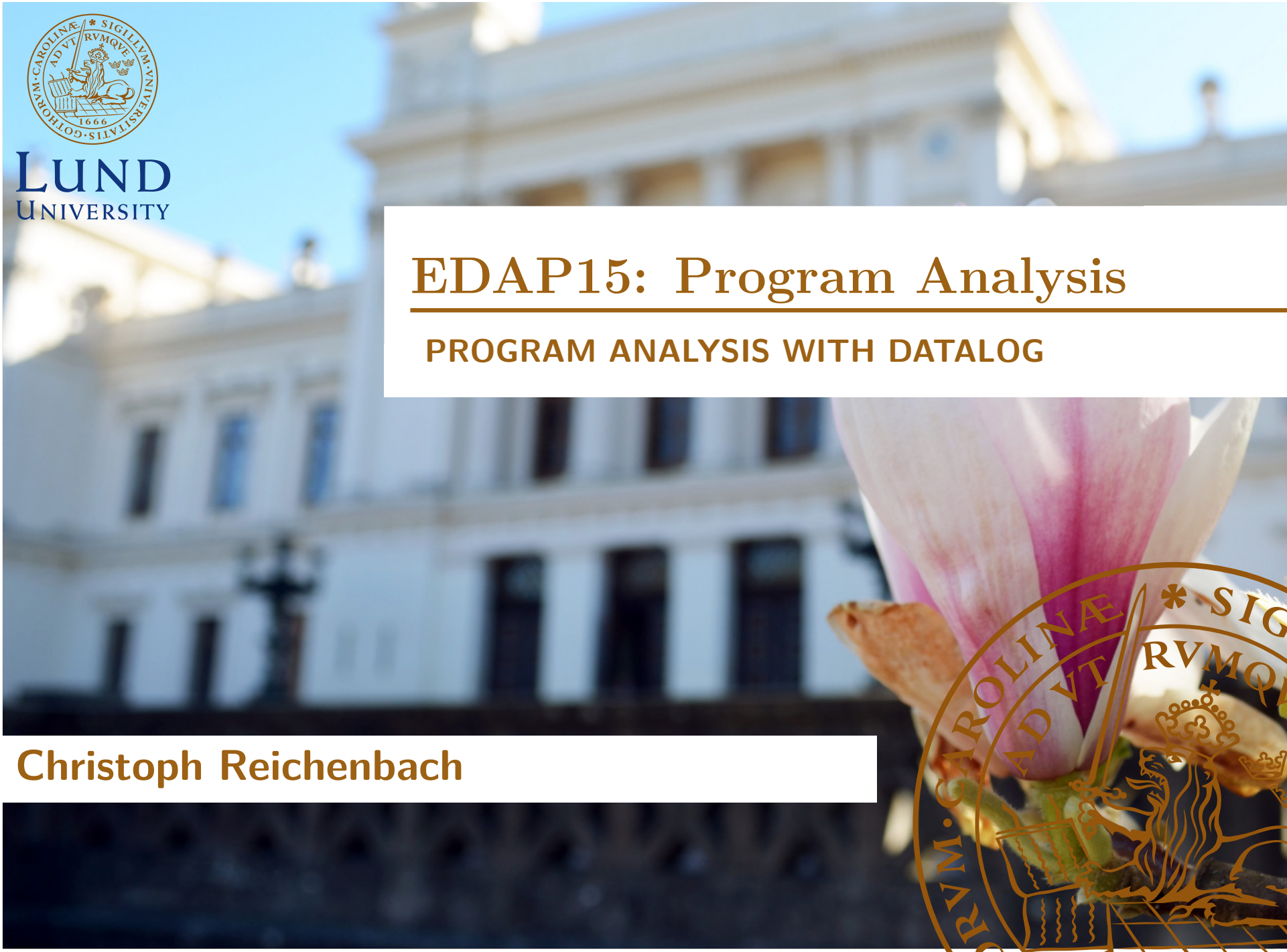
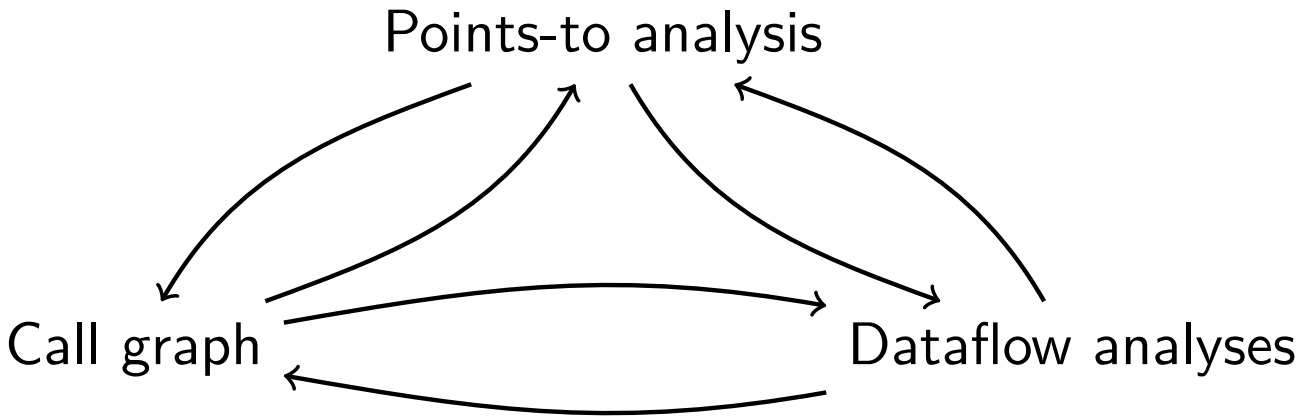


EDAP15: Program Analysis

PROGRAM ANALYSIS WITH DATALOG

Christoph Reichenbach

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**

Objects and Relations

- **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 $\times \mathbb{N}$

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

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

Relations and Predicate Symbols

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

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 $\textsc{MassNumber}$:

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

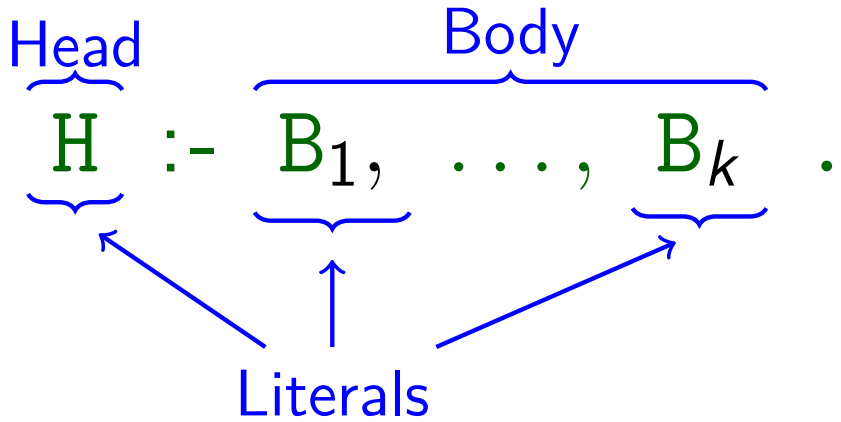
- An **atom** is a predicate symbol plus parameters:
  - $\textsc{MassNumber}(\mathrm{H}, 1)$

Datalog Programs: Syntax

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

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

written as



Datalog Programs: Semantics

$$\frac{\mathtt{B}_1 \quad \ldots \quad \mathtt{B}_k}{\mathtt{H}}$$

- Semantics: if $\mathtt{B}_1, \ldots, \mathtt{B}_k$ are true, $\mathtt{H}$ is also true
- Order of the conjuncts $\mathtt{B}_i$ in the body is irrelevant
- Order of the rules is irrelevant

Rules in Detail

Literals may take parameters:

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

- where $\mathit{Body} = \mathrm{B}_1(v_1^1, \ldots, v_{j_1}^1), \ldots, \mathrm{B}_k(v_1^k, \ldots, v_{j_k}^k)$
- $v_1, \ldots, v_j$ (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

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

  - 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

Datalog by Example

```
if ...  {
  a := 7;
  print(0);
} else {
  a := 8;
  a := 9;
}
print(a);
```

Datalog by Example

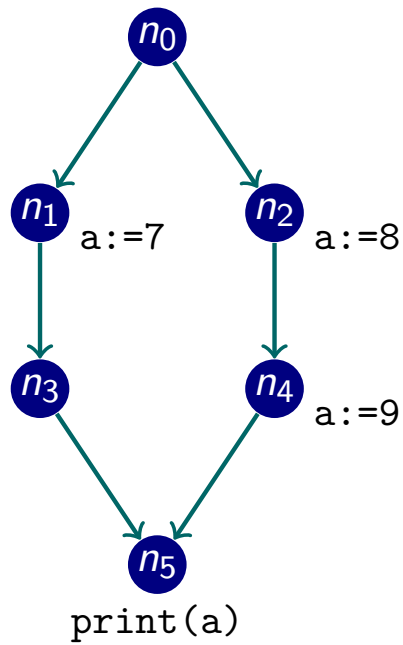
Input Facts

$\textsc{Next}(\texttt{"n\_0"}, \texttt{"n\_1"})$, $\textsc{Next}(\texttt{"n\_0"}, \texttt{"n\_2"})$, ...
$\textsc{Assign}(\texttt{"n\_1"}, \texttt{"a"})$, $\textsc{Assign}(\texttt{"n\_2"}, \texttt{"a"})$, ...

$\textsc{Reach}(n, v, d)$:- $\textsc{Assign}(d, v), \textsc{Next}(d, n)$.
$\textsc{Reach}(n, v, d)$:- $\textsc{Reach}(p, v, d), \textsc{Next}(p, n), \neg\textsc{Assign}(p, v)$.

Derived Facts

Reach		
"n_3"	"a"	"n_1"
"n_4"	"a"	"n_2"
"n_5"	"a"	"n_1"
"n_5"	"a"	"n_4"



Datalog Literals and Terms

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

$$\textsc{Assign}(\texttt{"n\_1"}, \texttt{"a"})$$

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

$$\textsc{Assign}(n, \texttt{"a"})$$

Datalog Programs: Syntax

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

Negation

- Negation is a popular extension to pure Datalog:

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

- 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**:

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

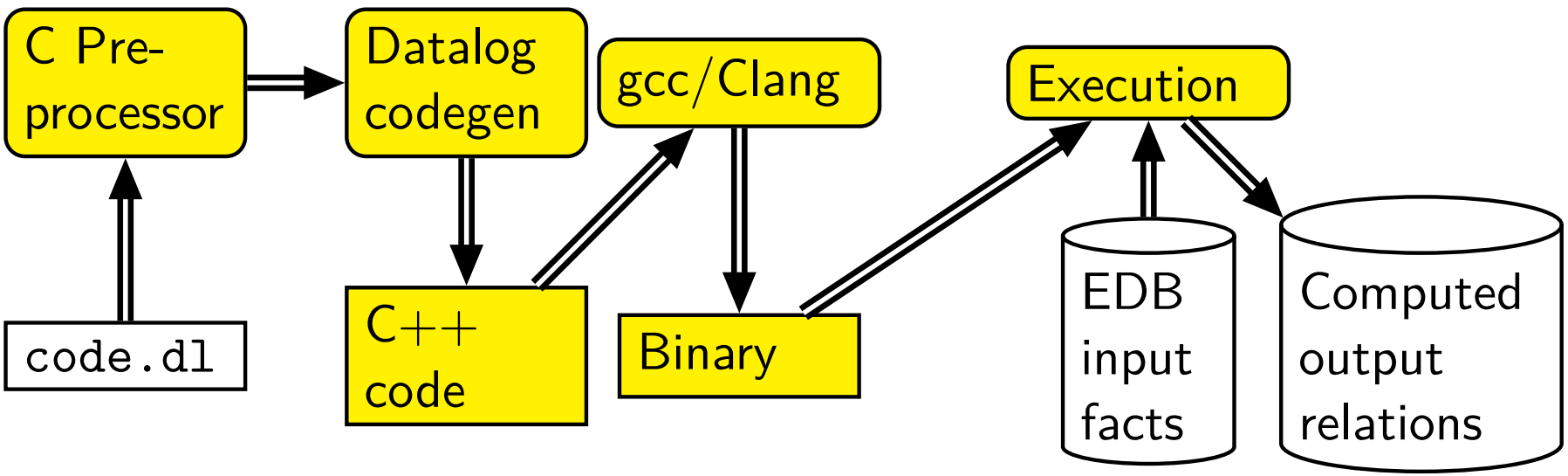
- The rule **Head** and the conjuncts of the **Body** are **Literals**
- Literals consist of a **Predicate Symbol** and **Terms**
- **Terms** can be variables or constants
- **Negation** is permitted in some extensions
- Datalog reasons over relations that are bound to the predicate symbols
- Relations 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 Next(node: symbol, succ: symbol)
.decl Assign(node: symbol, var: symbol)
.decl Reach(node: symbol, var: symbol, def: symbol)

Reach(n, v, d) :- Assign(d, v), Next(d, n).

Reach(n, v, d) :- Reach(p, v, d), Next(p, n),
                  !Assign(p, v).
```

- 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 Next(node: symbol, succ: symbol)
.input Next(IO=file, filename="next.csv", delimiter=",")
```

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

next.csv:

```
n_0,n_1
n_0,n_2
n_1,n_3
n_2,n_4
n_3,n_5
n_4,n_5
```

Equivalent Soufflé code:

```
Next("n_0", "n_1").
Next("n_0", "n_2").
Next("n_1", "n_3").
Next("n_2", "n_4").
Next("n_3", "n_5").
Next("n_4", "n_5").
```

Output Relations

```
.decl Reach(node: symbol, var: symbol, def: symbol)
.output Reach(IO=file, filename="reach.csv", delimiter=",")
```

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

```
.decl Reach(node: symbol, var: symbol, def: symbol)
.output Reach
```

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".
```

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).
```

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.)
- Explicit declaration for input and output behaviour
- ***Aggregation*** *operations for summing up or computing the minimum etc.*
- ***Conjunctive heads*** *and* ***Disjunctions*** *add syntactic sugar*

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 ⟨x1, _, y1⟩ ∈ A do
  for ⟨x2, y2, z2⟩ ∈ B do
    if x1 = x2 and y1 = y2 then
      H := H ∪ {⟨x1, y1⟩}
  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:

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

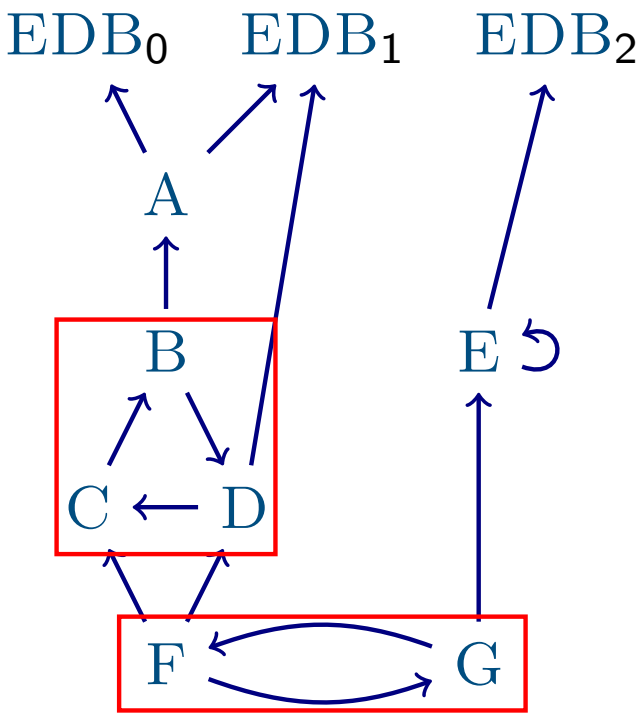
- Implementation idea: *fixpoint*:

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

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

Evaluation Strata

- Strategy:
  - Evaluate dependencies first
  - Evaluate mutual dependencies 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

- 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

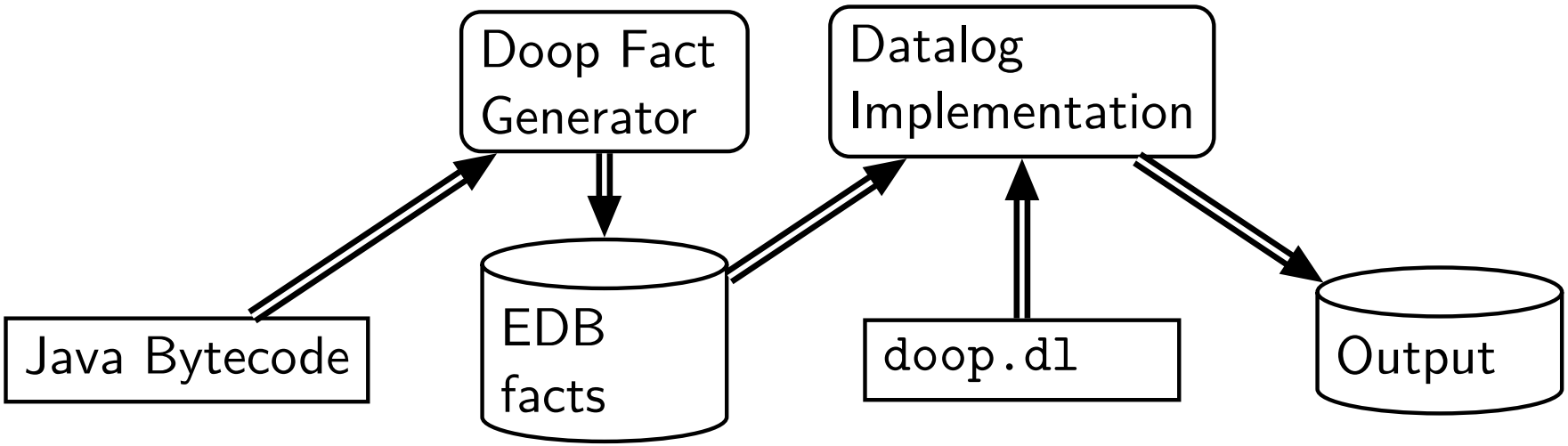
Summary

- 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

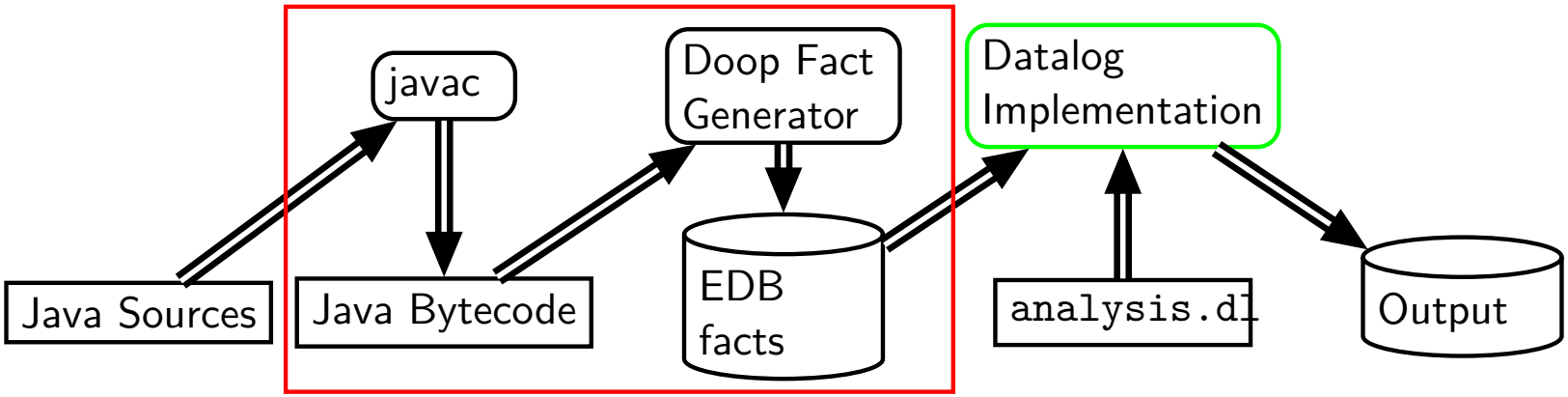
- Points-to analysis framework for Java bytecode
- Core analysis implemented in Datalog
  - Based on Andersen's Analysis
- Supports different forms of *x-sensitivity*: call-site, field, object.

Doop Overview



- Doop first generates EDB facts by scanning programs
- Then analyses the facts using Datalog code
- Different (Datalog-based) analyses available
- Output:
  - Call graph
  - Points-to graph

MetaDL

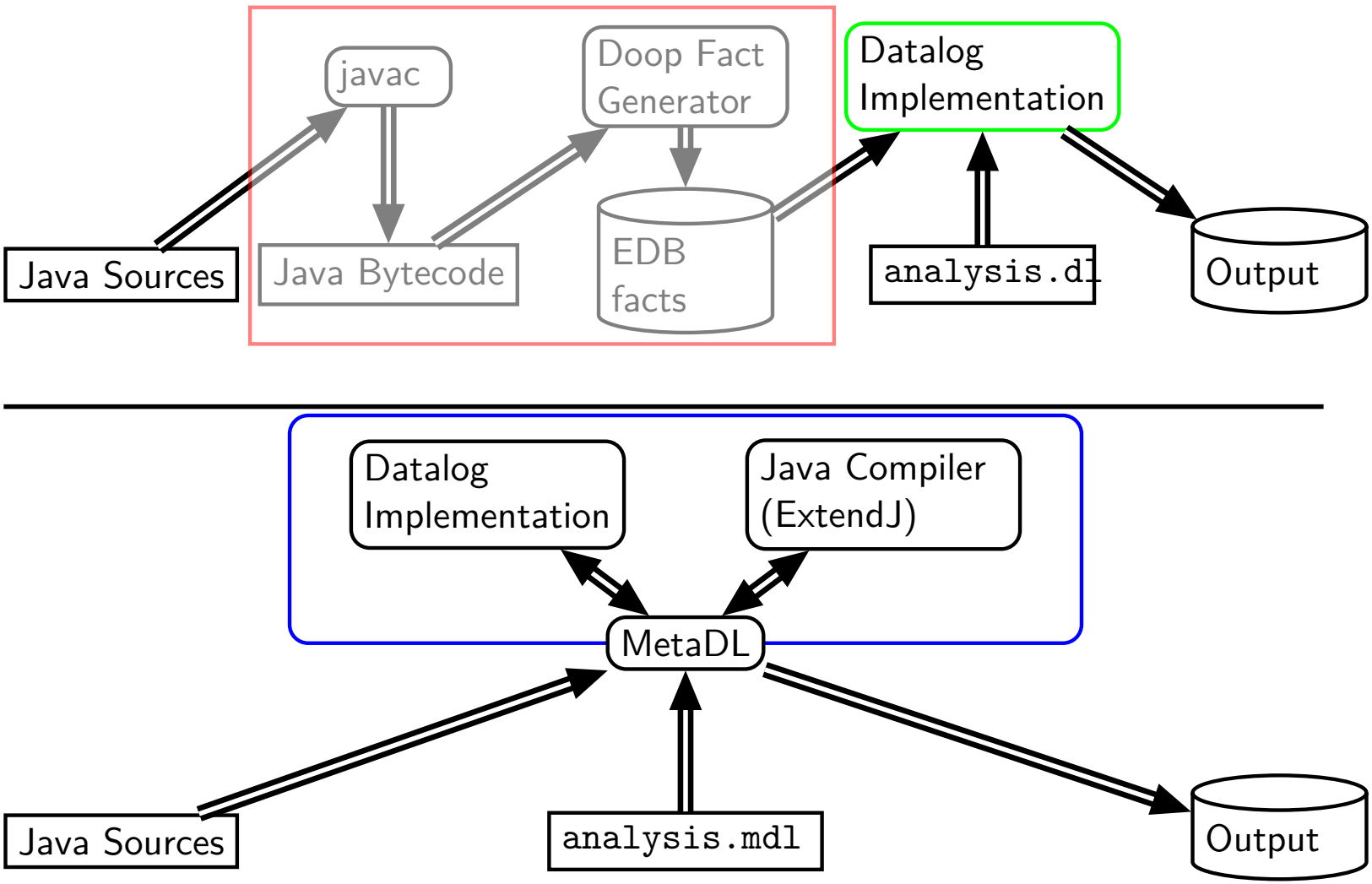


- Current Datalog program analyses (including Doop) work on an intermediate representation of the analysed program
  - The intermediate representation is produced by a fact extractor (or a chain of) implemented in an imperative language
  - The IR abstractions are typically far-removed from the source
- How do we enable software developers to write custom declarative analyses?

MetaDL

- How do we enable software developers to write custom declarative analyses?
- By providing an analysis language that uses abstractions of the analysed programming language
  - Syntactic patterns
- ... and a few other well-understood concepts:
  - types
  - declarations
  - source locations

MetaDL


javac
Doop Fact Generator
Datalog Implementation
Java Sources
Java Bytecode
EDB facts
analysis.dl
Output
Datalog Implementation
Java Compiler (ExtendJ)
MetaDL
Java Sources
analysis.mdl
Output

Extending Datalog

- Syntactic patterns - a kind of **atom**
  - `<: public class ‘a extends ‘b { .. } :>`
  - `<: ‘x = ‘b.‘m(.., ‘p, ..) :>`
  - quantified over the entire analysed program
- A new type of **object**: ASTNode
  - Represented by **metavariable** terms: `‘a`, `‘b`, `‘x`, `‘m`, ...
- Special predicates:
  - `DECL(n, d)`, connects a named term, `n`, to its declaration, `d`.
  - `TYPE(n, t)`, connects a term, `n`, to its type, `t`.

A simple nullness check

```
VarMayBeNull(v)      :-  <:  `v = null :>, DECL(`v, v).
VarMayBeNull(v)      :-  <:  `v = `w :>, DECL(`v, v),
                         DECL(`w, w), VarMayBeNull(w).
ArgMayBeNull(m, i)   :-  <:  `b.`f(.., `v, ..)  :>,
                         DECL(`f, m), DECL(`v, v),
                         VarMayBeNull(v), INDEX(`v, i).
ArgMayBeNull(m, i)   :-  <:  `b.`f(.., `n, ..)  :>
                         `n <:  null :>, INDEX(`n, i).
VarMayBeNull(`p)     :-  ArgMayBeNull(m, i),
                         m <:..`t `n(..,`p,..){..}:>,
                         INDEX(`p, i).
```

- We identify the null variables by their declaration, hence the use of the DECL predicate.

How does MetaDL perform?

- The language is sufficient for implementing static bug detectors from established bug checkers
  - The top 5 bug detectors (by number of reports generated) in Error Prone and SpotBugs
- In a declarative and compact way:
  - Reduced the number of SLOC/check by 10x for the SpotBugs detectors
  - And by 2x for Error Prone detectors
- Reasonably fast:
  - The MetaDL prototype is 2x slower than SpotBugs and 10x slower than Error Prone

Research

- MetaDL spans the following research threads:
  - Automatic derivation of program analysis languages
  - Representation of patterns and programs in Datalog
  - Parsing in the presence of ambiguity
  - Interaction between Datalog and attribute grammars