Verifying Datalog Reasoning with Lean

KR 2025 (extended abstract of ITP 2025 paper)
based on Johannes' Diploma Thesis

Johannes Tantow[1] Lukas Gerlach[2] Stephan Mennicke[2] Markus Krötzsch[2]

[1]TU Chemnitz, Germany
[2]Knowledge-Based Systems Group, TU Dresden, Germany

14.11.2025

International Center
for Computational Logic

Datalog primer - transitive closure of a relation 🔁

📢 Datalog is a database query language that allows for *recursive queries*.

Datalog primer - transitive closure of a relation 🔁

📢 Datalog is a database query language that allows for *recursive queries*.

Transitive Closure

```
e(a,b). e(b,c). e(c,d).

tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :-
  e(?X, ?Y), tr(?Y, ?Z).
```

Datalog primer - transitive closure of a relation 🔁

📢 Datalog is a database query language that allows for *recursive queries*.

Transitive Closure

```
e(a,b). e(b,c). e(c,d).

tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :-
  e(?X, ?Y), tr(?Y, ?Z).
```

Reasoning Steps

```
0. e(a,b). e(b,c). e(c,d).
1. tr(a,b). tr(b,c). tr(c,d).
2. tr(a,c). tr(b,d).
3. tr(a,d).
```

Datalog primer - transitive closure of a relation 🔁

📢 Datalog is a database query language that allows for *recursive queries*.

Transitive Closure

```
e(a,b). e(b,c). e(c,d).

tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :-
  e(?X, ?Y), tr(?Y, ?Z).
```

Reasoning Steps

```
0. e(a,b). e(b,c). e(c,d).
1. tr(a,b). tr(b,c). tr(c,d).
2. tr(a,c). tr(b,d).
3. tr(a,d).
```

👍 We like Datalog since it's explainable and the reasoning is trustworthy.

Datalog primer - transitive closure of a relation 🔁

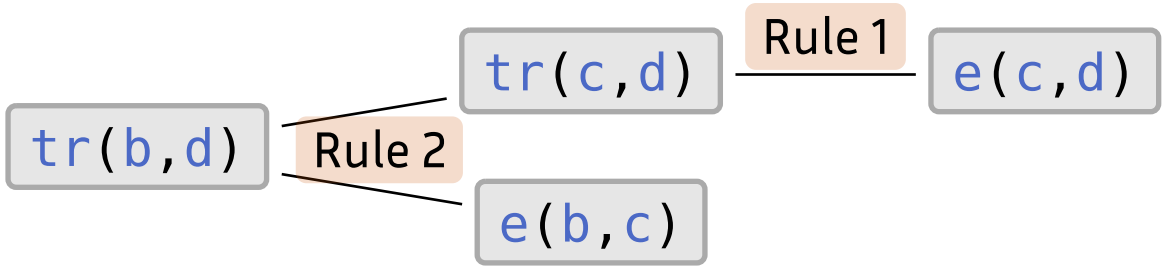
📢 Datalog is a database query language that allows for *recursive queries*.

Transitive Closure

```
e(a,b). e(b,c). e(c,d).

tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :-
  e(?X, ?Y), tr(?Y, ?Z).
```

A proof tree for `tr(b,d)`*:*



👍 We like Datalog since it's explainable and the reasoning is trustworthy.

Datalog primer - transitive closure of a relation 🔁

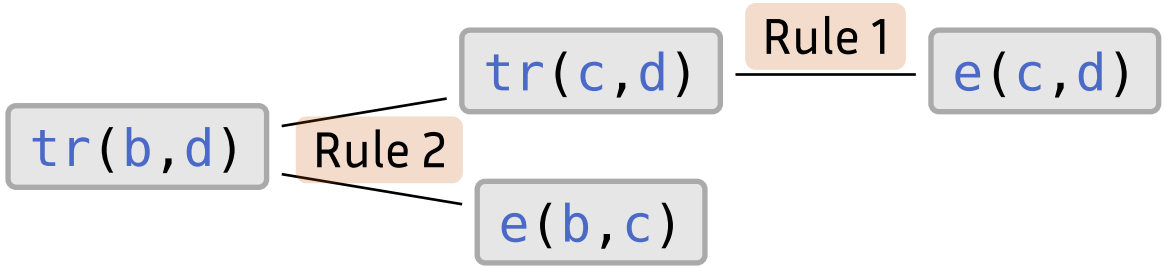
📢 Datalog is a database query language that allows for *recursive queries*.

Transitive Closure

```
e(a,b). e(b,c). e(c,d).

tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :-
  e(?X, ?Y), tr(?Y, ?Z).
```

A proof tree for `tr(b,d)`*:*



👍 We like Datalog since it's explainable and the reasoning is trustworthy.

🤔 When leaving the nice world of theory this might not be true though...

Why bother verifying reasoning ❓

🐞 Practical Datalog (dialect) implementations[1] aim to handle billions of logical facts and are highly optimized for this purpose. Bugs included!

[1] To name a few: Gringo (Gebser et al. 2019), Soufflé (Jordan et al. 2016), Nemo (Ivliev et al. 2024), RDFox (Nenov et al.), VLog (Urbani et al. 2016)

Why bother verifying reasoning ❓

🐞 Practical Datalog (dialect) implementations[1] aim to handle billions of logical facts and are highly optimized for this purpose. Bugs included!

🐌 Verifying the whole implementation[2] is (I) hard and (II) languages able to do so might produce less efficient executables. (Milliseconds matter!)

[1] To name a few: Gringo (Gebser et al. 2019), Soufflé (Jordan et al. 2016), Nemo (Ivliev et al. 2024), RDFox (Nenov et al.), VLog (Urbani et al. 2016)

[2] Existing formalization approaches: in Rocq (Benzaken et al.) or Isabelle/HOL (Schlichtkrull et al. 2024) but with different focus.

Why bother verifying reasoning ❓

🐞 Practical Datalog (dialect) implementations[1] aim to handle billions of logical facts and are highly optimized for this purpose. Bugs included!

🐌 Verifying the whole implementation[2] is (I) hard and (II) languages able to do so might produce less efficient executables. (Milliseconds matter!)

📋 Some Datalog engines can *trace* their inferences, thereby producing Datalog proof structures can act as *certificates*.[3]

[1] To name a few: Gringo (Gebser et al. 2019), Soufflé (Jordan et al. 2016), Nemo (Ivliev et al. 2024), RDFox (Nenov et al.), VLog (Urbani et al. 2016)

[2] Existing formalization approaches: in Rocq (Benzaken et al.) or Isabelle/HOL (Schlichtkrull et al. 2024) but with different focus.

[3] This approach of having an algorithm produce a certificate alongside the result is know as the *de Bruijn criterion*. (Barendregt and Wiedijk 2005)

Why bother verifying reasoning ❓

🐞 Practical Datalog (dialect) implementations[1] aim to handle billions of logical facts and are highly optimized for this purpose. Bugs included!

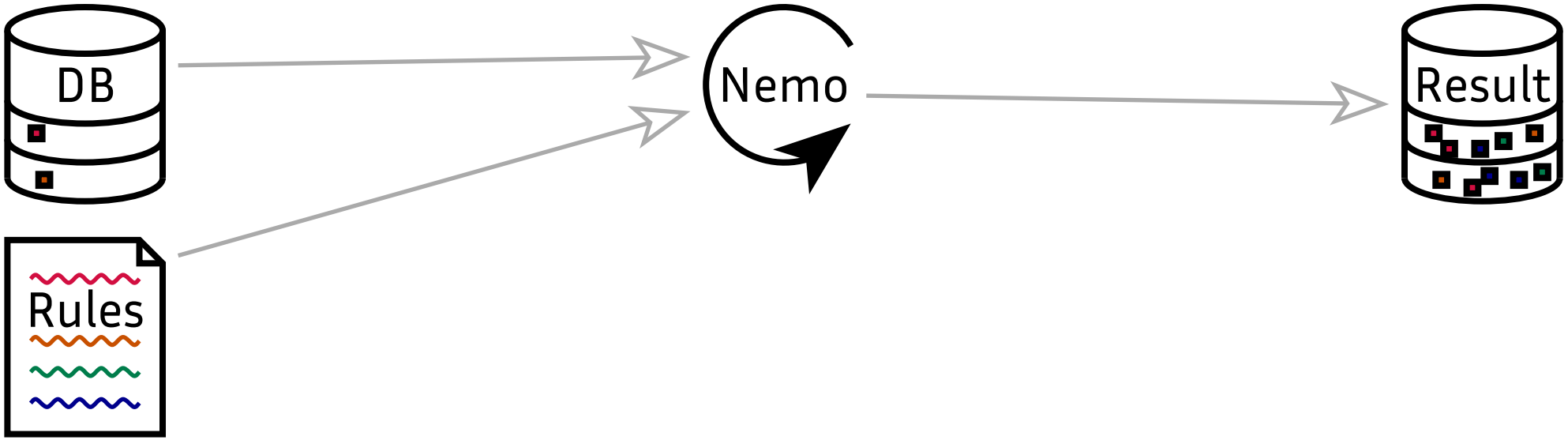
🐌 Verifying the whole implementation[2] is (I) hard and (II) languages able to do so might produce less efficient executables. (Milliseconds matter!)

📋 Some Datalog engines can *trace* their inferences, thereby producing Datalog proof structures can act as *certificates*.[3]

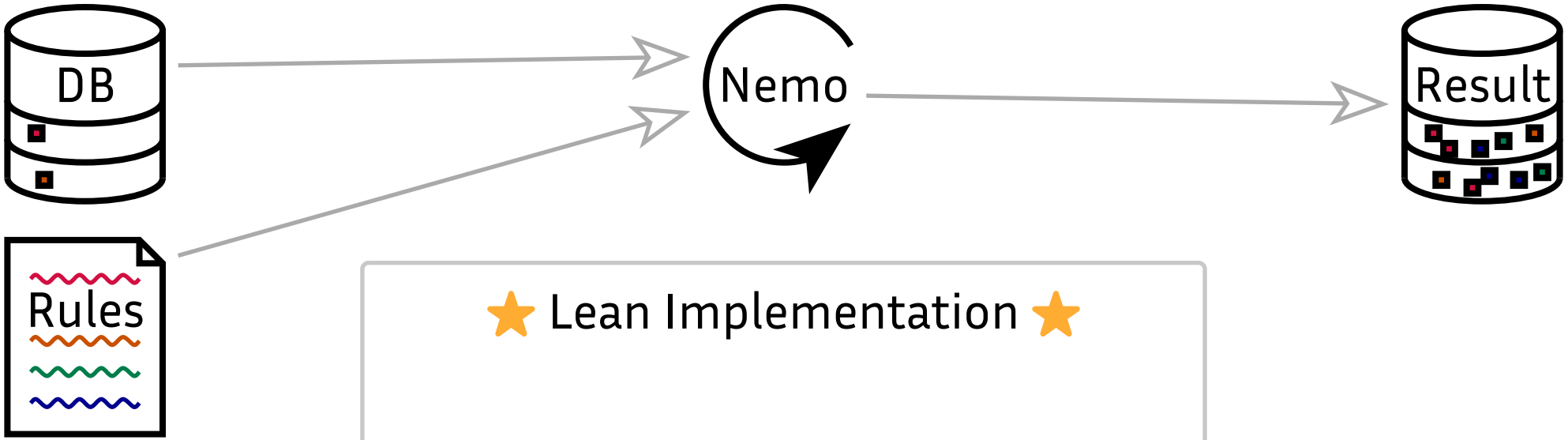
🏍️ It's easy to verify a certificate. Using *Lean*, a *purely functional language* and *interactive theorem prover*, we can implement an algorithm and prove its correctness as part of the code.

[1] To name a few: Gringo (Gebser et al. 2019), Soufflé (Jordan et al. 2016), Nemo (Ivliev et al. 2024), RDFox (Nenov et al.), VLog (Urbani et al. 2016)
[2] Existing formalization approaches: in Rocq (Benzaken et al.) or Isabelle/HOL (Schlichtkrull et al. 2024) but with different focus.
[3] This approach of having an algorithm produce a certificate alongside the result is know as the *de Bruijn criterion*. (Barendregt and Wiedijk 2005)

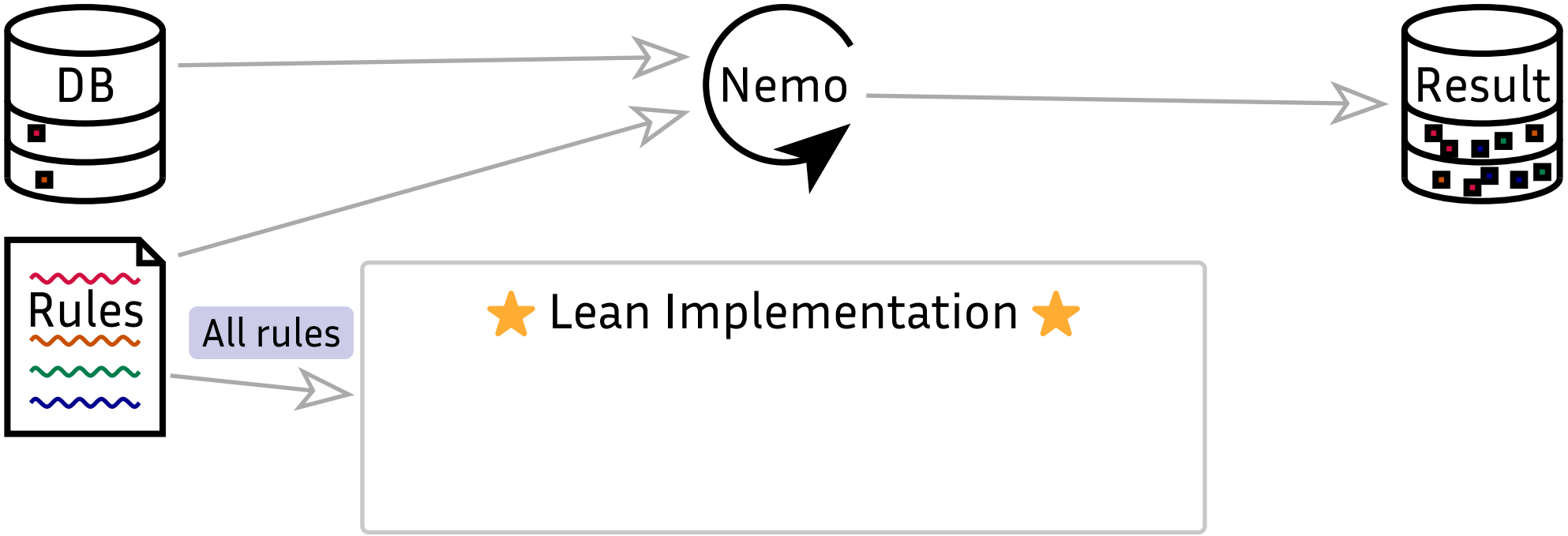
The pipeline



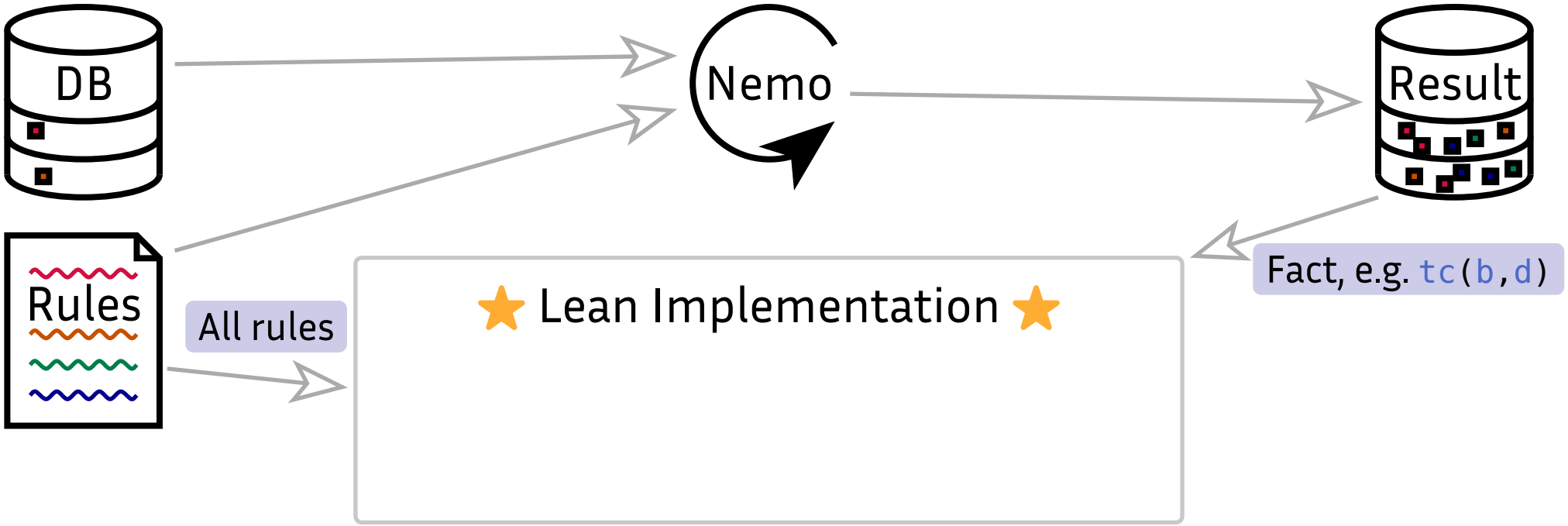
The pipeline



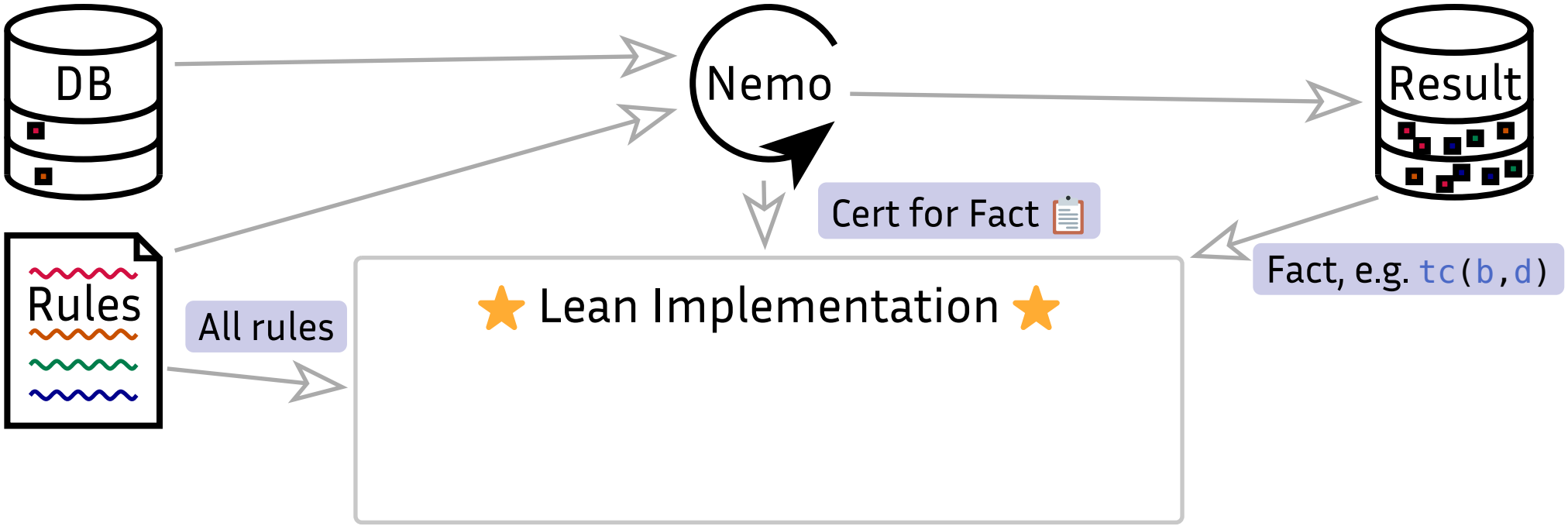
The pipeline



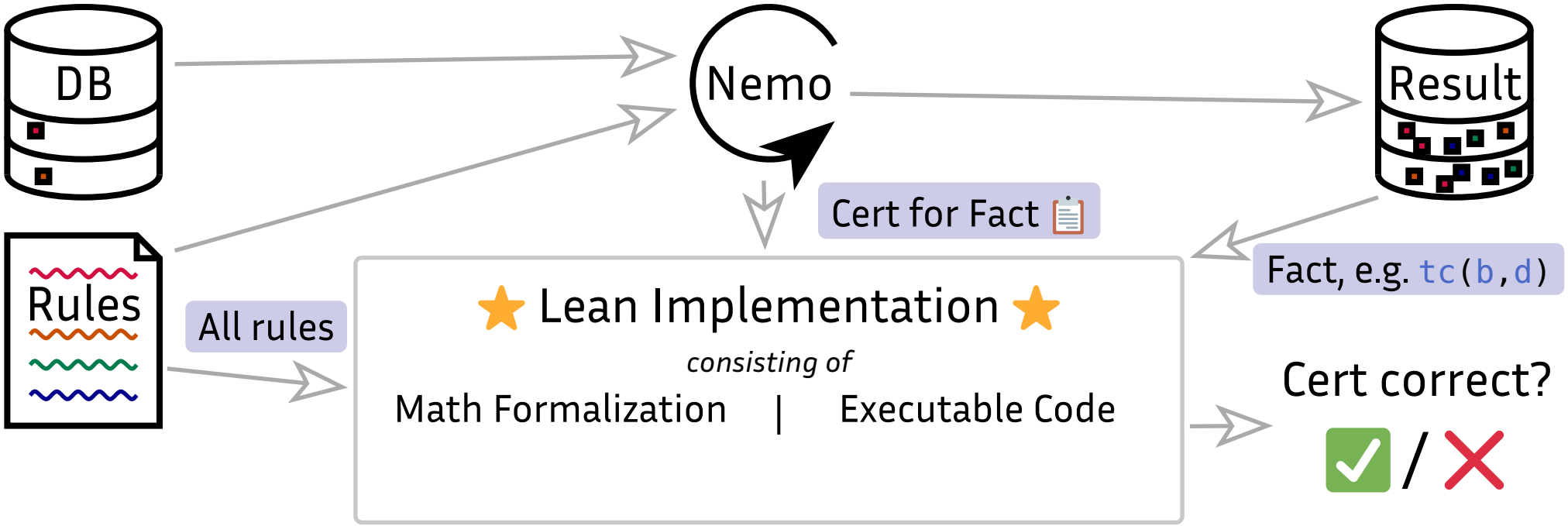
The pipeline



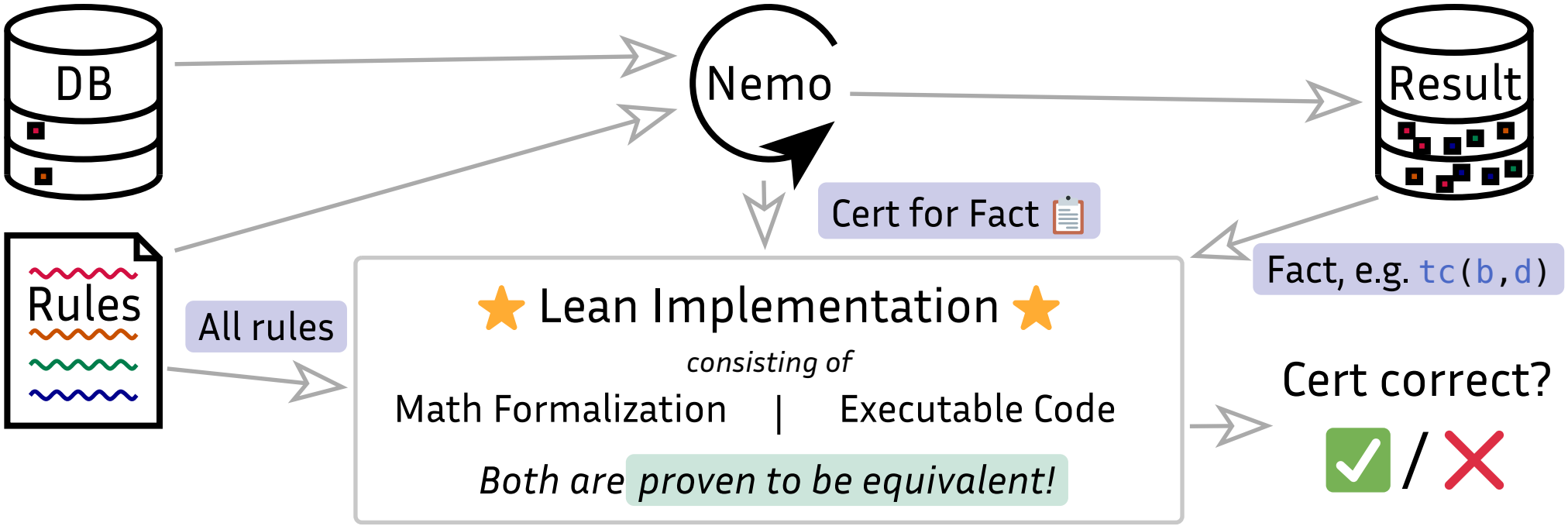
The pipeline



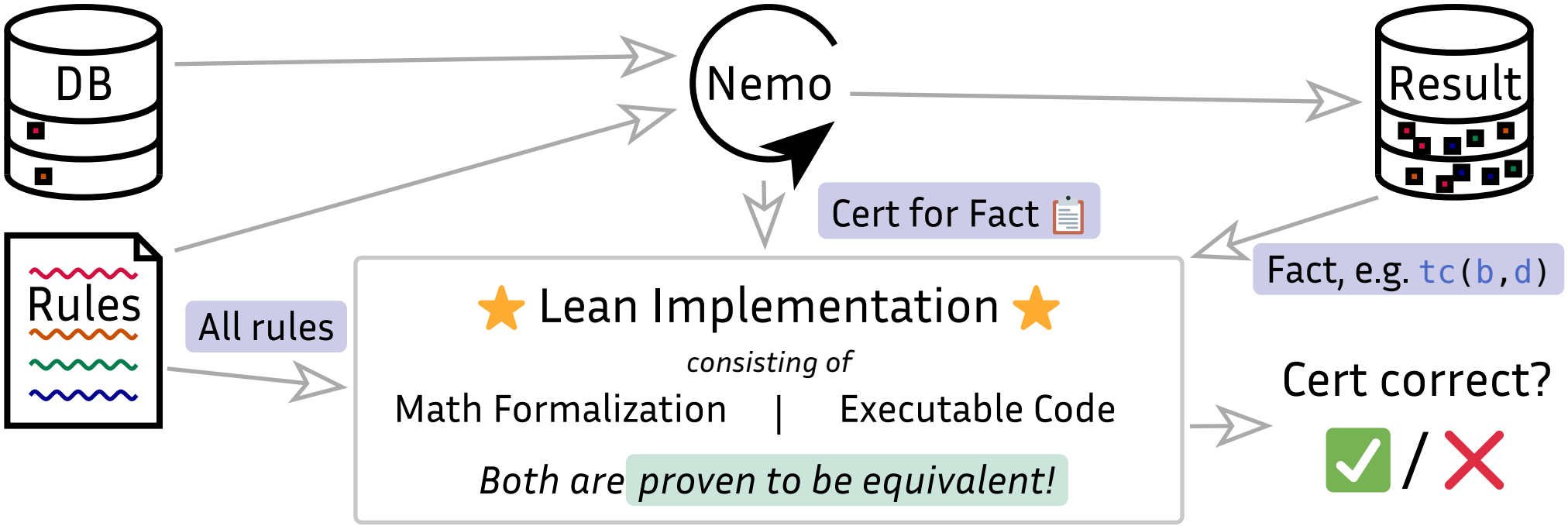
The pipeline



The pipeline

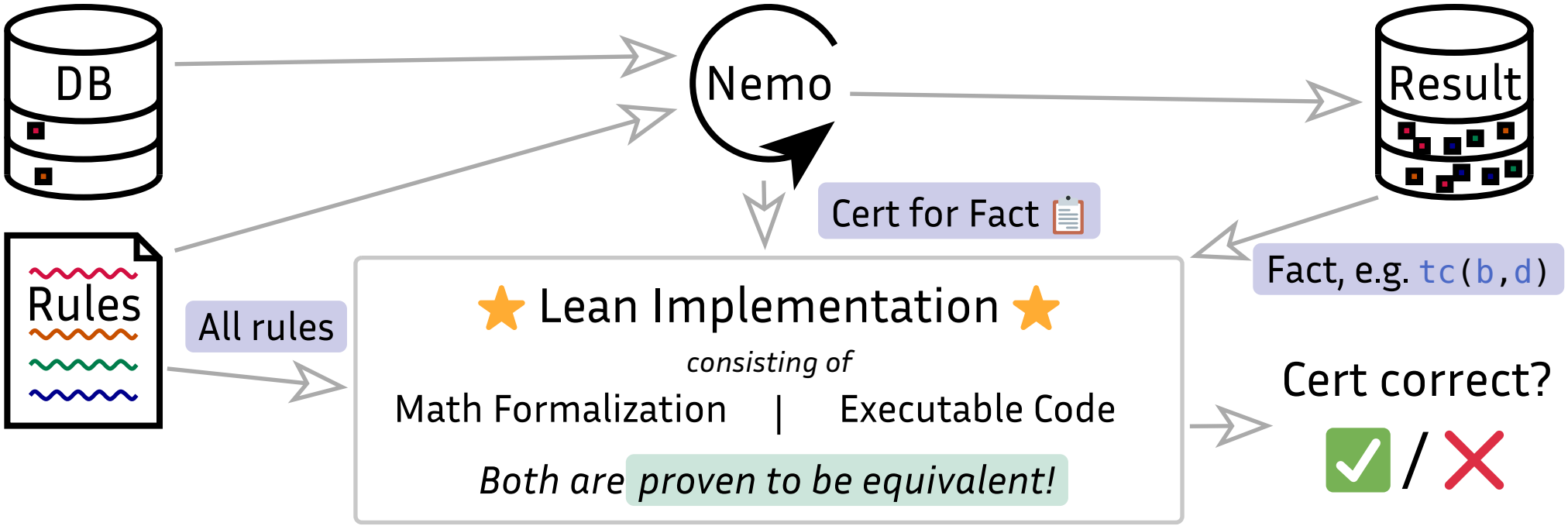


The pipeline



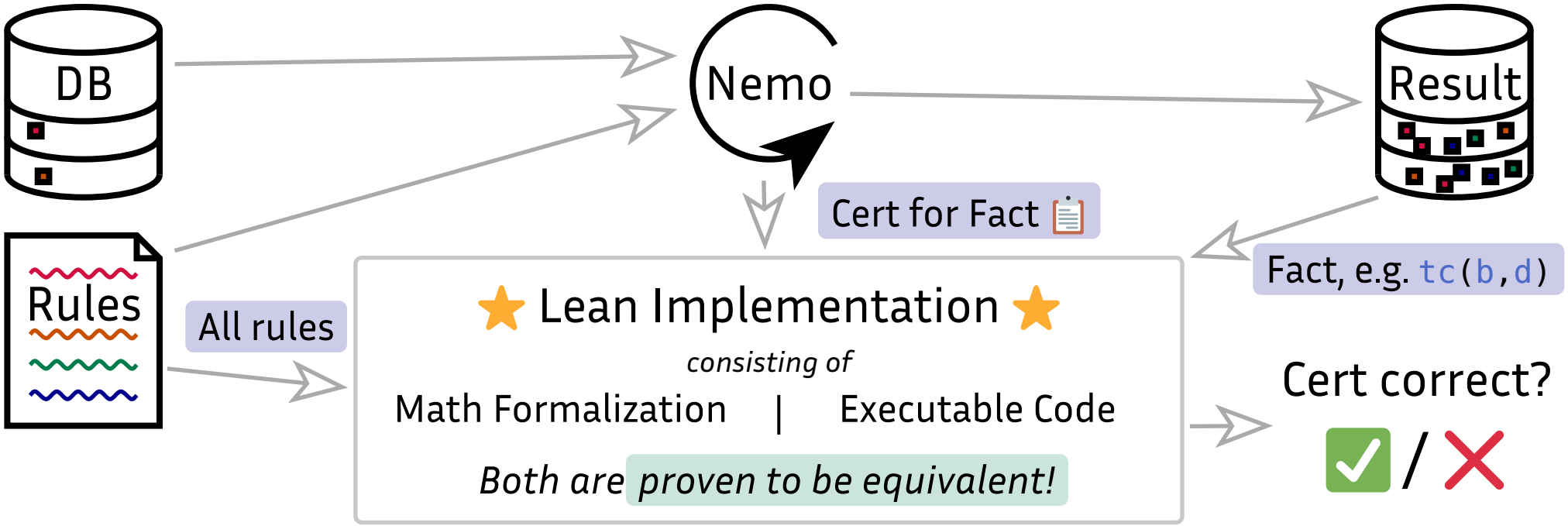
Who do we need to trust?

The pipeline



Who do we need to trust? - *We only trust the Lean kernel!*

The pipeline 🏗️



🤓 Who do we need to trust? - *We only trust the Lean kernel!* *

*Technically, we also trust that our mathematical formalization of Datalog semantics is correct but this is simple enough.

And currently we also trust the reasoning engine to faithfully load initial facts from the database and not to make them up.

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2.
3.

Datalog: model-theoretic semantics

```
def models (i: Interpretation) (kb: KB) : Prop :=
  (∀ r ∈ kb.prog.groundProgram, i.satisfiesRule r) ∧
  (∀ (a: GroundAtom), kb.db.contains a → a ∈ i)
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2.
3.

Datalog: model-theoretic semantics

```
def models (i: Interpretation) (kb: KB) : Prop :=
  (∀ r ∈ kb.prog.groundProgram, i.satisfiesRule r) ∧
  (∀ (a: GroundAtom), kb.db.contains a → a ∈ i)

def modelTheoSem (kb: KB) : Interpretation :=
  { a: GroundAtom | ∀ (i: Interpretation), i.models kb → a ∈ i }
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2.
3.

Datalog: model-theoretic semantics

```
def models (i: Interpretation) (kb: KB) : Prop :=
  (∀ r ∈ kb.prog.groundProgram, i.satisfiesRule r) ∧
  (∀ (a: GroundAtom), kb.db.contains a → a ∈ i)

def modelTheoSem (kb: KB) : Interpretation :=
  { a: GroundAtom | ∀ (i: Interpretation), i.models kb → a ∈ i }
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2.
3.

Datalog: proof-theoretic semantics

```
structure ProofTree (kb: KB) where
  tree : ProofTreeSkeleton -- This is a simple inductive tree
  isValid : tree.isValid kb -- intuitive but does not fit here
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2.
3.

Datalog: proof-theoretic semantics

```
structure ProofTree (kb: KB) where
  tree : ProofTreeSkeleton -- This is a simple inductive tree
  isValid : tree.isValid kb -- intuitive but does not fit here
def proofTheoSem (kb: KB) : Interpretation :=
  { a: GroundAtom | ∃ (t: ProofTree kb), t.root = a }
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2.
3.

Datalog: proof-theoretic semantics

```
structure ProofTree (kb: KB) where
  tree : ProofTreeSkeleton -- This is a simple inductive tree
  isValid : tree.isValid kb -- intuitive but does not fit here
def proofTheoSem (kb: KB) : Interpretation :=
  { a: GroundAtom | ∃ (t: ProofTree kb), t.root = a }
theorem semEq (kb: KB) : kb.proofTheoSem = kb.modelTheoSem
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2.
3.

Datalog: proof-theoretic semantics

```
structure ProofTree (kb: KB) where
  tree : ProofTreeSkeleton -- This is a simple inductive tree
  isValid : tree.isValid kb -- intuitive but does not fit here
def proofTheoSem (kb: KB) : Interpretation :=
  { a: GroundAtom | ∃ (t: ProofTree kb), t.root = a }
theorem semEq (kb: KB) : kb.proofTheoSem = kb.modelTheoSem
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2. Implement executable procedure checking validity of a given proof tree.
3.

Executable Proof Tree Validation Procedure

```
def checkValidity (t : ProofTreeSkeleton) (m : SymbolSequenceMap)
  (d : Database) : Except String Unit := ...
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2. Implement executable procedure checking validity of a given proof tree.
3. Show that 2 is true if and only if the tree is indeed valid according to 1.

Executable Proof Tree Validation Procedure

```
def checkValidity (t : ProofTreeSkeleton) (m : SymbolSequenceMap)
  (d : Database) : Except String Unit := ...

theorem checkValidityOkIffValid {t: ProofTreeSkeleton} {kb: KB} :
  t.checkValidity kb.prog.toSymSeqMap kb.db = Except.ok () ↔
    t.isValid kb
```

The main ingredients of our tool 🧙

1. Formalize Datalog (model-theoretic and proof-theoretic semantics)
2. Implement executable procedure checking validity of a given proof tree.
3. Show that 2 is true if and only if the tree is indeed valid according to 1.

Executable Proof Tree Validation Procedure

```
def checkValidity (t : ProofTreeSkeleton) (m : SymbolSequenceMap)
  (d : Database) : Except String Unit := ...

theorem checkValidityOkIffValid {t: ProofTreeSkeleton} {kb: KB} :
  t.checkValidity kb.prog.toSymSeqMap kb.db = Except.ok () ↔
    t.isValid kb
```

More efficient proof structures

- Redundancies in proof trees potentially cause exponential blow-up.

More efficient proof structures

- Redundancies in proof trees potentially cause exponential blow-up.

Inefficient Transitive Closure

```
tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :- e(?X, ?Y),
  u(?Y, ?Z), v(?Y, ?Z).
```

```
u(?X, ?Y) :- tr(?X, ?Y).
v(?X, ?Y) :- tr(?X, ?Y).
```

More efficient proof structures

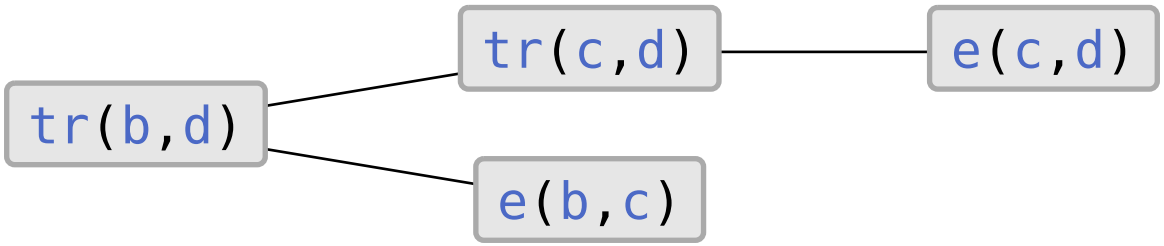
- Redundancies in proof trees potentially cause exponential blow-up.

Inefficient Transitive Closure

```
tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :- e(?X, ?Y),
  u(?Y, ?Z), v(?Y, ?Z).

u(?X, ?Y) :- tr(?X, ?Y).
v(?X, ?Y) :- tr(?X, ?Y).
```

The proof tree for `tr(b,d)` from before:



More efficient proof structures

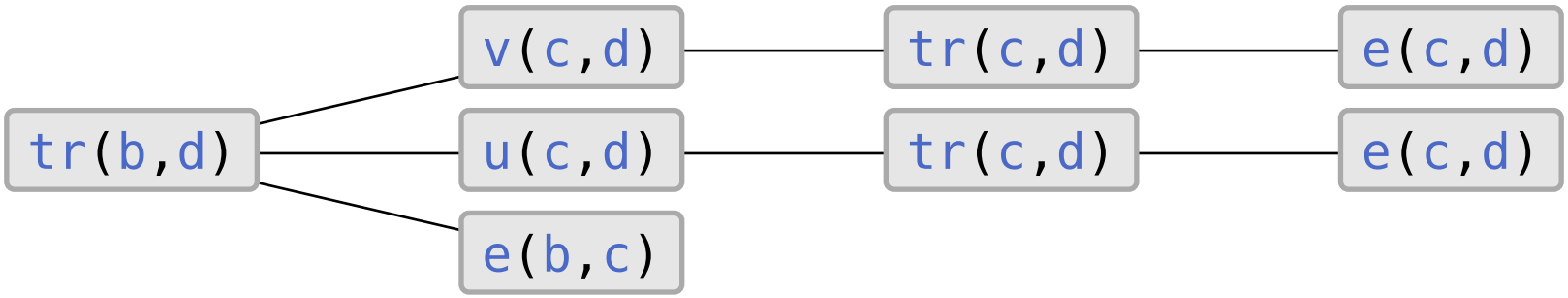
- Redundancies in proof trees potentially cause exponential blow-up.

Inefficient Transitive Closure

```
tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :- e(?X, ?Y),
  u(?Y, ?Z), v(?Y, ?Z).
```

```
u(?X, ?Y) :- tr(?X, ?Y).
v(?X, ?Y) :- tr(?X, ?Y).
```

The proof tree for `tr(b,d)` with the inefficient program:



More efficient proof structures

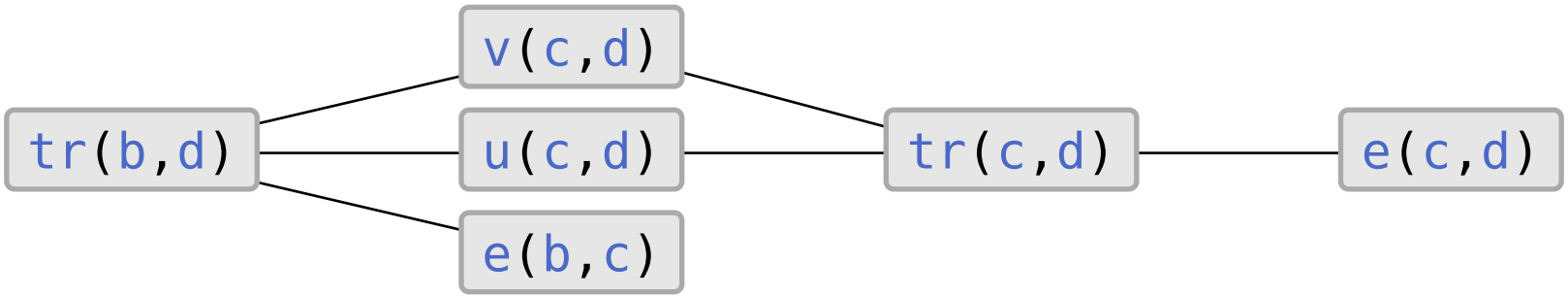
- Redundancies in proof trees potentially cause exponential blow-up.

Inefficient Transitive Closure

```
tr(?X, ?Y) :- e(?X, ?Y).
tr(?X, ?Z) :- e(?X, ?Y),
  u(?Y, ?Z), v(?Y, ?Z).

u(?X, ?Y) :- tr(?X, ?Y).
v(?X, ?Y) :- tr(?X, ?Y).
```

A proof DAG for `tr(b,d)` with the inefficient program:



Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Unordered DAGs

```
def PreGraph := Std.HashMap A (List A)
def Graph := { pg : PreGraph A // pg.complete }
```

We ended up contributing a few HashMap results to the standard library.

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Unordered DAGs

```
def PreGraph := Std.HashMap A (List A)
def Graph := { pg : PreGraph A // pg.complete }
def toProofTree (G: Graph GroundAtom) (kb: KB) ... : ProofTree kb
```

We ended up contributing a few HashMap results to the standard library.

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Unordered DAGs

```
def PreGraph := Std.HashMap A (List A)
def Graph := { pg : PreGraph A // pg.complete }
def toProofTree (G: Graph GroundAtom) (kb: KB) ... : ProofTree kb
def checkValidity (G : Graph GroundAtom) (m : SymSeqMap)
    (d : Database) : Except String Unit :=
  G.verify_via_dfs (fun n => G.check_local_validity m d n)
```

We ended up contributing a few HashMap results to the standard library.

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Unordered DAGs

```
def PreGraph := Std.HashMap A (List A)
def Graph := { pg : PreGraph A // pg.complete }
def toProofTree (G: Graph GroundAtom) (kb: KB) ... : ProofTree kb
def checkValidity (G : Graph GroundAtom) (m : SymSeqMap)
    (d : Database) : Except String Unit :=
  G.verify_via_dfs (fun n => G.check_local_validity m d n)
theorem checkValidityIsOkIffAcyclicAndAllValid ...
```

We ended up contributing a few HashMap results to the standard library.

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Unordered DAGs

```
def PreGraph := Std.HashMap A (List A)
def Graph := { pg : PreGraph A // pg.complete }
def toProofTree (G: Graph GroundAtom) (kb: KB) ... : ProofTree kb
def checkValidity (G : Graph GroundAtom) (m : SymSeqMap)
    (d : Database) : Except String Unit :=
  G.verify_via_dfs (fun n => G.check_local_validity m d n)
theorem checkValidityIsOkIffAcyclicAndAllValid ...
```

We ended up contributing a few HashMap results to the standard library.

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Ordered DAGs

```
def OrderedProofGraph :=
  { arr : Array (GroundAtom × List ℕ) //
    ∀ i : Fin arr.size, ∀ j ∈ arr[i].snd, j < i }
```

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Ordered DAGs

```
def OrderedProofGraph :=
  { arr : Array (GroundAtom × List ℕ) //
    ∀ i : Fin arr.size, ∀ j ∈ arr[i].snd, j < i }
def toProofTree (G: OrderedProofGraph) (kb: KB) ...: ProofTree kb
```

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Ordered DAGs

```
def OrderedProofGraph :=
  { arr : Array (GroundAtom × List ℕ) //
    ∀ i : Fin arr.size, ∀ j ∈ arr[i].snd, j < i }
def toProofTree (G: OrderedProofGraph) (kb: KB) ...: ProofTree kb
def checkValidity (G: OrderedProofGraph) ... : Except String Unit
```

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Ordered DAGs

```
def OrderedProofGraph :=
  { arr : Array (GroundAtom × List ℕ) //
    ∀ i : Fin arr.size, ∀ j ∈ arr[i].snd, j < i }
def toProofTree (G: OrderedProofGraph) (kb: KB) ...: ProofTree kb
def checkValidity (G: OrderedProofGraph) ... : Except String Unit
theorem validIff : G.checkValidity ... = Except.ok ↔ G.isValid kb
```

Formalizing Proof DAGs ⌨

Unordered DAGs

Fewer demands on format.

Ordered DAGs

More efficient.

Ordered DAGs

```
def OrderedProofGraph :=
  { arr : Array (GroundAtom × List ℕ) //
    ∀ i : Fin arr.size, ∀ j ∈ arr[i].snd, j < i }
def toProofTree (G: OrderedProofGraph) (kb: KB) ...: ProofTree kb
def checkValidity (G: OrderedProofGraph) ... : Except String Unit
theorem validIff : G.checkValidity ... = Except.ok ↔ G.isValid kb
```

Performance is not half bad! 💪

Examples

(1a) - Check one fact for TC of chain of length 1000 **(1b)** - All facts for TC of chain of length 100

(2) - `tc(0,20)` for inefficient TC on chain of length 20 **(3)** - Random facts for EL example

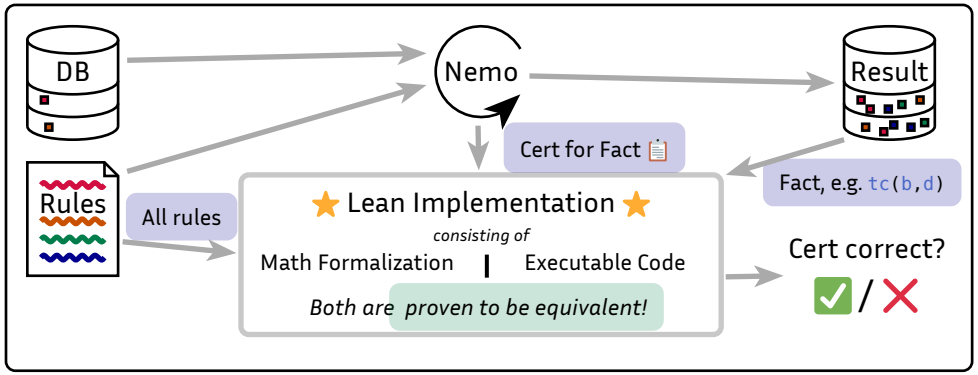
	Nemo 🕰️	Tree 🗃️	Unord. 🗃️	Ord. 🗃️	Tree ⏱️	Unord. ⏱️	Ord. ⏱️
(1a)	59s	320KB	515KB	320KB	0.1s	0.1s	0.1s
(1b)	<0.1s	53MB	1.7MB	564KB	2.7s	0.2s	0.1s
(2)	<0.1s	313MB	16KB	12KB	16s	<0.1s	<0.1s
(3)	7.8s	8.7MB	9.9MB	4.4MB	0.3s	0.4s	0.2s

🕰️ - (Nemo) Reasoning Time 🗃️ - File Size of Certificate (JSON) ⏱️ - Validation Time of Lean Tool

Summing up ⏪

We implemented a fully verified tool for verifying Datalog proof DAGs.

We also have a naive check for result completeness (not part of this talk).



What's next?

- Support for additional Datalog features: stratified negation, existentials, ...
- Closer integration with my existential rule library? https://dmfa.dev/lean

📢 **Check out our tool at: https://github.com/knowsys/CertifyingDatalog**

🤓 Also check out Nemo at: https://github.com/knowsys/nemo

References

Barendregt H, Wiedijk F (2005) The challenge of computer mathematics. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 363:2351–2375. https://doi.org/10.1098/rsta.2005.1650

Benzaken V, Contejean E, Dumbravă Ş-G Certifying Standard and Stratified Datalog Inference Engines in SSReflect. pp 171–188

Gebser M, Kaminski R, Kaufmann B, Schaub T (2019) Multi-shot ASP solving with clingo. Theory Pract Log Program 19:27–82. https://doi.org/10.1017/S1471068418000054

References

Ivliev A, Gerlach L, Meusel S, et al (2024) Nemo: Your Friendly and Versatile Rule Reasoning Toolkit. In: Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning. pp 743–754

Jordan H, Scholz B, Subotic P (2016) Soufflé: On Synthesis of Program Analyzers. In: Chaudhuri S, Farzan A (eds) Computer Aided Verification - 28th Int. Conf. (CAV'16). Springer, pp 422–430

Nenov Y, Piro R, Motik B, et al RDFox: A Highly-Scalable RDF Store. pp 3–20

References

Schlichtkrull A, Rydhof Hansen R, Nielson F (2024) Isabelle-verified correctness of Datalog programs for program analysis. In: Proc. of the 39th ACM/SIGAPP Symposium on Applied Computing (SAC 2024). ACM, pp 1731–1734

Urbani J, Jacobs C, Krötzsch M (2016) Column-Oriented Datalog Materialization for Large Knowledge Graphs. In: Schuurmans D, Wellman MP (eds) Proc. of the 30th AAAI Conf. on Artificial Intelligence (AAAI'16). AAAI Press, pp 258–264