PRACTICAL USES OF EXISTENTIAL RULES IN KNOWLEDGE REPRESENTATION

Part 1: Basics / Rules for Ontology Reasoning

David Carral, Markus Krötzsch, and Jacopo Urbani
1. TU Dresden
2. Vrije Universiteit Amsterdam

Special thanks to Irina Dragoste, Ceriel Jacobs, and Maximilian Marx for their invaluable contributions to the software used in this tutorial

KR, 13 September 2020
Goals of this tutorial

**Topic:** Existential rules as an approach to *declarative computation*, some of its application areas in AI, and *practical tools* to implement them in practice.

**Learning objectives:**

- Understand what existential rules are and how they are used
- Get concrete insights into diverse use cases
- Learn about useful modelling and optimisation methods
- Get to know software tools to build your own applications
Tutorial structure

- **Part 1: Introduction to Existential Rules**
  - Basic concepts
  - Getting acquainted with the tools
  - Implementing a lightweight description logic reasoner

- **Part 2: Application scenarios in KR and beyond**
  - Reasoning in Datalog(S) and expressive DLs
  - Probabilistic inference with Datalog
  - Data integration
  - Stream reasoning
Introduction to Existential Rules
What is a rule?

In symbolic AI, a **rule** is some form of logical implication.

**Different areas consider different kinds of rules:**

- Logic programming: **PROLOG**
- Optimisation and problem solving: **Answer set programming**
- Recursive database queries: **Datalog**
- Data management: **database dependencies**
- Ontological modelling: **existential rules**
- ...
What is a rule?

In symbolic AI, a rule is some form of logical implication.

Different areas consider different kinds of rules:

- Logic programming: PROLOG
- Optimisation and problem solving: Answer set programming
- Recursive database queries: Datalog
- Data management: database dependencies
- Ontological modelling: existential rules
- ...

In this tutorial: Declarative, deterministic rule languages
Including: Datalog, existential rules, database dependencies, + some negation
But excluding: PROLOG, ASP, other non-logical rules
Simple rules: Datalog

**Given:** A relational structure (a.k.a. database)

**Wanted:** A way to define derived relations, possibly recursively

**Example:**

\[
\forall x, y, z. \quad \text{contains}(x, y) \land \text{contains}(y, z) \rightarrow \text{contains}(x, z)
\]

\[
\forall x. \quad \text{drink}(x) \land \text{contains}(x, \text{carbonDioxide}) \rightarrow \text{fizzyDrink}(x)
\]
Simple rules: Datalog

**Given:** A relational structure (a.k.a. database)

**Wanted:** A way to define derived relations, possibly recursively

**Example:**

\[ \forall x, y, z. \; \text{contains}(x, y) \land \text{contains}(y, z) \rightarrow \text{contains}(x, z) \]

\[ \forall x. \; \text{drink}(x) \land \text{contains}(x, \text{carbonDioxide}) \rightarrow \text{fizzyDrink}(x) \]

**General form:**

[conjunction of relational atoms] \rightarrow [relational atom]

- **universal quantifier** (usually not written)
- **variable**
- **constant symbol**
- **predicate**
Evaluating Datalog

Datalog rules iteratively are “applied” to the given relations until saturation.

**Example:** We use rules as before

(R1) \( \text{contains}(x, y) \land \text{contains}(y, z) \rightarrow \text{contains}(x, z) \)

(R2) \( \text{drink}(x) \land \text{contains}(x, \text{carbonDioxide}) \rightarrow \text{fizzyDrink}(x) \)

on a database with the following facts:

\[
\begin{align*}
\text{drink(limeAndSoda)} \\
\text{contains(limeAndSoda, limeSyrup)} & \quad \text{contains(limeAndSoda, sodaWater)} \\
\text{contains(sodaWater, water)} & \quad \text{contains(sodaWater, carbonDioxide)}
\end{align*}
\]

Applying rules yields:

from R1: \( \text{contains(limeAndSoda, water)} \)
from R1: \( \text{contains(limeAndSoda, carbonDioxide)} \)
from R2: \( \text{fizzyDrink(limeAndSoda)} \)
A brief history of Datalog

1970s and 1980s: The Good Old Days
Datalog is invented and studied as recursive database query language

1990s: The Datalog Winter
Logic Programming semantic wars
Datalog given up and forgotten in data management

Since the 2000s: Renaissance
Rise of graph-based data
Old values of elegance and declarativity return
Explosion in Datalog research, tools, and applications
Datalog today

Many implementations
Emptyheaded\(^4\), Graal\(^6\), RDFox\(^{14}\), Llunatic\(^{10}\), Vadalog\(^7\), VLog/Rulewerk\(^1\), and various others

Commercial exploitation
Successful companies (e.g., Semmle, LogicBlox, DIADEM, cognitect) and recent start-ups (e.g., Oxford Semantic Technologies, DeepReason.ai)

Applications in many areas
- Source code analysis\(^{11}\)
- Decision support\(^5\)
- Data access and management\(^9\)
- Health care data analysis\(^{15}\)
- Knowledge graph management\(^7\)
- Ontology reasoning\(^8\)
- Data integration\(^{13}\)
- Integrated AI systems\(^{12}\)

Research connections
Datalog is relevant in many areas: Answer Set Programming, database dependencies, existential rules, constraint satisfaction problems

The highlights show topics that appear in this tutorial. The [references] link to further details.
In this tutorial, we use two free & open source software tools:

- **VLog**: A rule reasoner (memory-based, scalable, fast)
- **Rulewerk**: A rule toolkit (convenient, interactive client, Java API)

Both come integrated in the interactive **Rulewerk client**

**Getting ready:**
- Requirements: Windows/MacOS/Linux; Java 8 or above
- Download and decompress tutorial resource package
- Open a command line in the tutorial directory and type: 
  `java -jar rulewerk-client.jar`
Rulewerk uses a Prolog-like syntax for rules, with “semantic web”-style identifiers.

**Example:** The rule

\[ \text{drink}(x) \land \text{contains}(x, \text{carbonDioxide}) \rightarrow \text{fizzyDrink}(x) \]

could be written as:

\[ \text{fizzyDrink}(\text{?x}) :\right \text{- drink}(\text{?x}), \text{contains}(\text{?x}, \text{"carbon dioxide"}) . \]
Hands-On #1: Using Rulewerk client (1)

The client is controlled using @commands (including the command @help)

Start Rulewerk client, and follow these steps:

(1) Add some facts to your knowledge base:
   @assert drink("lime & soda") .
   @assert contains("lime & soda","lime syrup") .
   @assert contains("lime & soda","soda water") .
   @assert contains("soda water","carbon dioxide") .
   @assert contains("soda water","water") .

(2) Add some rules, too:
   @assert fizzyDrink(?x) :- drink(?x), contains(?x,"carbon dioxide") .
   @assert contains(?x,?z) :- contains(?x,?y), contains(?y,?z) .

(3) Check what you have now:
   @showkb .

Hint: You can use TAB to auto-complete commands and up/down to access the history.
Hint: Omitting the initial @ or final . is tolerated.
Now let’s see what VLog can infer here:

(4) Call VLog to process our knowledge base:
   @reason .

(5) Ask some queries:
   @query contains(\(?x,\?y\) .
   @query fizzyDrink(\(?x\) .

(6) Export all inferences to a file:
   @export INFERENCES "limeAndSoda.rls" .

Hint: @export uses Rulewerk’s native syntax for facts and rules. @load can import this again.
Beyond toy examples

VLog is designed for knowledge bases of hundreds of millions of facts

\[\sim \texttt{@assert} \text{ is not the way to get there}\]

**Supported sources for larger datasets:**

- RLS files with Rulewerk knowledge bases
- CSV files (one predicate per file)
- RDF graphs in NTriples format or any other standard format (one ternary triple-predicate per file)
- OWL ontologies (converted to rules and facts)
- Graal knowledge bases
- Trident database files (large-scale, disk-based RDF graph index; open source)
- SPARQL query results
- Other ODBC database connectors

1 loaded by Rulewerk
2 configured in Rulewerk, natively loaded by VLog (most scalable)
3 only with direct low-level VLog usage; not available through Rulewerk
Hands-On #2: Handling larger knowledge bases (1)

We will use data files found in the tutorial folder.

**File** art/paintings.csv **is about artworks in the following format:**

<table>
<thead>
<tr>
<th>Title</th>
<th>Painter</th>
<th>Shown in picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Mona Lisa&quot;</td>
<td>&quot;Leonardo da Vinci&quot;</td>
<td>&quot;Lisa del Giocondo&quot;</td>
</tr>
<tr>
<td>&quot;Mona Lisa&quot;</td>
<td>&quot;Leonardo da Vinci&quot;</td>
<td>&quot;landscape&quot;</td>
</tr>
<tr>
<td>&quot;Self-Portrait with Monkey&quot;</td>
<td>&quot;Frida Kahlo&quot;</td>
<td>&quot;Frida Kahlo&quot;</td>
</tr>
<tr>
<td>&quot;Self-Portrait with Monkey&quot;</td>
<td>&quot;Frida Kahlo&quot;</td>
<td>&quot;monkey&quot;</td>
</tr>
</tbody>
</table>

... (205,236 more rows)

**File** art/types.csv **assigns types to some things:**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Dresden&quot;</td>
<td>&quot;city&quot;</td>
</tr>
<tr>
<td>&quot;Rhodes&quot;</td>
<td>&quot;island&quot;</td>
</tr>
<tr>
<td>&quot;Frida Kahlo&quot;</td>
<td>&quot;human&quot;</td>
</tr>
</tbody>
</table>

... (52,310 more rows)

We use plain strings for readability, which leads to some ambiguities and errors that shall not concern us in the example. The data was extracted from Wikidata.
Hands-On #2: Handling larger knowledge bases (2)

File art/rules.rls uses these sources:

% First declare the data sources:
@source painting[3] : load-csv("paintings.csv") .
@source type[2] : load-csv("types.csv") .

% Find self-portraits:
selfPortrait(?Art,?Creator) :- painting(?Art,?Creator,?Creator) .

% Find paintings of islands:
islandArt(?Art,?Creator,?Motive)
    :- painting(?Art,?Creator,?Motive), type(?Motive,"island") .
We use this knowledge base in Rulewerk:

(1) Switch to the Rulewerk client and (if still running) delete the data used in the previous hands-on:
   \texttt{@clear ALL .}

(2) Load the knowledge base and view the loaded knowledge base:
   \texttt{@load "art/rules.rls" .}
   \texttt{@showkb .}

(3) Invoke VLog: \texttt{@reason .}

(4) Try some queries to explore the data and inferences:
   \texttt{@query selfPortrait(?Art,?By) LIMIT 10 .}
   \texttt{@query COUNT islandArt(?Art,?By,?Island) .}
   \texttt{@query islandArt(?Art,?By,"Rhodes") .}

Hint: Note how \texttt{COUNT} and \texttt{LIMIT} help us to deal with larger query results.
The Limits of Datalog

What kind of problems can we solve in Datalog?

• The number of rule applications is bound by the number of possible facts:

  \(<\text{number of predicate names}> \times \text{number of constants}\)^{\text{max. predicate arity}}

• In the worst case, fact query entailment can be decided in this time
The Limits of Datalog

What kind of problems can we solve in Datalog?

- The number of rule applications is bound by the number of possible facts:

  \(<\text{number of predicate names}> \times \text{number of constants}\)^{<\text{max. predicate arity}>}

- In the worst case, fact query entailment can be decided in this time

**Theorem:** Deciding fact entailment for Datalog is ExpTime-complete, and P-complete with respect to the size of the database (data complexity).
The Limits of Datalog

What kind of problems can we solve in Datalog?

- The number of rule applications is bound by the number of possible facts:

  \(<\text{number of predicate names}> \times \text{<number of constants}>\)^{<\text{max. predicate arity}>}

- In the worst case, fact query entailment can be decided in this time:

**Theorem:** Deciding fact entailment for Datalog is ExpTime-complete, and P-complete with respect to the size of the database (data complexity).

**Corollary:** If a problem can be solved by a fixed Datalog rule set, then it can be solved in polynomial time.

**Corollary:** Problems with worst-case complexity above P cannot be solved in this way.
The Limits of Datalog

Not even all polynomially solvable problems can be solved in Datalog.

**Example:** Datalog is monotone, i.e., it can only solve problems where “more input” leads to “more output”. For example:

- We cannot check which paintings do not show Rhodes
- Datalog cannot decide if the database contains an even number of paintings

Remark: The second type of “parity” query cannot even be solved when adding negation to Datalog.
Beyond Datalog: Existential rules

We can extend the expressivity of Datalog using existential quantifiers in rule heads:

**Example:** When a painting is said to show a class, it is actually meant that it shows some instance of that class:

\[
painting(x, y, z) \land type(u, z) \rightarrow \exists v. painting(x, y, v) \land type(v, z)
\]

(as before, the universal quantifier is omitted)

**Practical applications:**

- Express unknown information (related: NULLs in databases, blank nodes in RDF)
- Creating auxiliary (graph) structure
- Expanding the computational universe
In Rulewerk, existential variables are written with `!` instead of `?`.

1. Continue from Hands-On #2 (or do `@import "art/rules.rls"`).
2. Add the example rule:
   ```prolog
   @assert painting(?Art,?By,!New), type(!New,?Class) :-
       painting(?Art,?By,?Class), type(?X,?Class).
   ```
3. `@reason`.
4. Do you find more island paintings now?
The Chase

How can we apply rules with existential variables in the head?

Make sure that the required element exists!
... and create new elements if deemed necessary to satisfy a rule
¬ different concrete implementations possible
The Chase

How can we apply rules with existential variables in the head?

Make sure that the required element exists!

... and create new elements if deemed necessary to satisfy a rule

\[ \rightarrow \text{different concrete implementations possible} \]

**Danger!** If rule applications can add new elements, then recursive rules can produce infinitely many distinct facts. The computation might never terminate, since we are forever “chasing after” a state where all rules are satisfied for all elements.

\[ \rightarrow \text{many variants of this chase algorithm exist} \]

**Some well-known truths:**

- Termination (for all practical chase algorithms) is undecidable for a given rule set and database
- Corollary: even when the chase terminates, it can run very long
- Fact entailment over existential rules is undecidable
The specific chase procedure used in VLog is as follows:

- **restricted**: check if suitable elements exist before making new ones (a.k.a. “standard chase”)
- **Datalog-first**: apply Datalog rules before considering rules with $\exists$
- **1-parallel**: apply each rule in parallel in all possible ways

Other chase types exist.

Skolemisation is another method to handle existentials, which can also be applied in the chase. See remarks at the end of this slide set (link).
Negation

Negation is another extremely useful extension of rule languages.

Example:

\[
painting(x, y, \text{fork}) \land \neg painting(x, y, \text{knife}) \rightarrow \text{query}(x, y).
\]
Negation

Negation is another extremely useful extension of rule languages.

Example:

\[
painting(x, y, \text{fork}) \land \neg painting(x, y, \text{knife}) \rightarrow \text{query}(x, y).
\]

Mixing negation with recursion can be complicated:

Example:

\[
\neg p(x) \rightarrow q(x) \quad \neg q(x) \rightarrow p(x)
\]

- different meaning in different logic programming paradigms
- simple bottom-up chase will fail – reasoning by cases required
Negation

Negation is another extremely useful extension of rule languages.

Example:

\[
painting(x, y, \text{fork}) \land \neg \text{painting}(x, y, \text{knife}) \rightarrow \text{query}(x, y).
\]

Mixing negation with recursion can be complicated:

Example:

\[
\neg p(x) \rightarrow q(x) \quad \neg q(x) \rightarrow p(x)
\]

- different meaning in different logic programming paradigms
- simple bottom-up chase will fail – reasoning by cases required

\text{\sim} \text{ VLog forbids recursive dependencies through negation (stratified negation)}

Note: This is still not enough to guarantee declarative behaviour, since existential quantifiers and negation can interact in strange ways. This is an ongoing research topic. There are many safe cases, e.g., using negation only on atoms that cannot be inferred by rules (“input negation”).
In Rulewerk, negation in rules is written as ∼

(1) Continue from Hands-On #3 (or do `@import "art/rules.rls"`.)

(2) Add the example rule:

```
@assert query(?Art,?By) :- painting(?Art,?By,"fork"),
    ∼painting(?Art,?By,"knife") .
```

(3) `@reason`.

(4) `@query query(?Art,?By)`.

Exercise: Find paintings that show an artist who did not create the painting.
Using Existential Rules in KR
Description Logics

Description logics (DLs) are influential and widely used ontology languages

- basis of the W3C Web Ontology Language standard OWL
- specific DLs achieve good trade-offs between expressivity and complexity

**Schema modelling in DLs:**

- Predicate logic based on classes (unary predicates, e.g., “Painting”) and properties (binary predicates, e.g., “depicts”)
- **Class subsumptions** specify relations of complex class expressions, e.g.:

  \[
  \text{Portrait} \sqsubseteq \text{Painting} \sqcap \exists \text{depicts}.\text{Person}
  \]

  “every portraits is a painting that depicts a person”, equivalent to

  \[
  \forall x. \text{Portrait}(x) \rightarrow \exists y. \text{Painting}(x) \land \text{depicts}(x, y) \land \text{Person}(y)
  \]

- New subsumptions might be inferred, e.g., Portrait \(\sqsubseteq\) Painting (classification)
The DL $\mathcal{EL}^+_{\bot}$ in a nutshell

The $\mathcal{EL}$ family of DLs is simple and supports polynomial time standard reasoning.

The DL $\mathcal{EL}^+_{\bot}$ supports the following class expressions to describe derived classes:

- $\bot$ empty class (bottom) “the empty set”
- $\top$ universal class (top) “set of all elements”
- $\exists R.C$ existential restriction “set of all elements that have an $R$-relation to some element in class $C$”
- $C \cap D$ intersection “set of all elements that are in class $C$ and in class $D$”
The DL $\mathcal{EL}^+_\bot$ in a nutshell

The $\mathcal{EL}$ family of DLs is simple and supports polynomial time standard reasoning

The DL $\mathcal{EL}^+_\bot$ supports the following class expressions to describe derived classes:

$\bot$ empty class (bottom) “the empty set”

$\top$ universal class (top) “set of all elements”

$\exists R.C$ existential restriction “set of all elements that have an $R$-relation to some element in class $C$”

$C \cap D$ intersection “set of all elements that are in class $C$ and in class $D$”

Class expressions and properties can be used in axioms:

$C \sqsubseteq D$ class subsumption “Every $C$ is also a $D$”

$R \sqsubseteq S$ property subsumption “Every relation of type $R$ is also one of type $S$”

$R \circ S \sqsubseteq T$ property chain “Elements connected by a chain of relations $R$ followed by $S$ are also directly connected by $T$”
Fact: Every $\mathcal{EL}_\perp^+$ axiom is equivalent to an existential rule.

Note: This property generalises to all “Horn Description Logics”.

Problem: DLs are based on different reasoning methods. The rules they yield do often not lead to a terminating chase.

How can we classify $\mathcal{EL}$ ontologies in rules?
The Incredible ELK

From Polynomial Procedures to Efficient Reasoning with $\mathcal{EL}$ Ontologies

Yevgeny Kazakov, Markus Krötzsch & František Simančík

*Journal of Automated Reasoning* 53, 1–61(2014) | [Cite this article]

518 Accesses | 93 Citations | [Metrics]
Prior research . . .

Fig. 3 Optimized inference rules for classification of $\mathcal{E}L_\bot^+$ ontologies
How to read such rules

General form of the rules:

\[
\text{rule name} \quad \frac{\text{pre-condition}}{\text{conclusion}} : \text{side condition}
\]

For example:

\[
R_{\mathcal{O}}^+ \quad \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occur negatively in } \mathcal{O}
\]

where the parts have the following meaning:

- \( \mathcal{O} \): the given \( \mathcal{EL}^+ \) ontology
- \( C, D_1, D_2 \): arbitrary (possibly nested) \( \mathcal{EL}^+ \) class expressions
- “to occur negatively”: to appear in a subclass position
Encoding a calculus in rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_0$</td>
<td>$\frac{\text{init}(C)}{C \sqsubseteq C}$</td>
</tr>
<tr>
<td>$R_T$</td>
<td>$\frac{\text{init}(C)}{C \sqsubseteq \top}$ : $\top$ occurs negatively in $\mathcal{O}$</td>
</tr>
<tr>
<td>$R_\cap$</td>
<td>$\frac{C \sqsubseteq D_1 \cap D_2}{C \sqsubseteq D_1 \cap D_2}$</td>
</tr>
<tr>
<td>$R_\triangledown$</td>
<td>$\frac{E \sqsubseteq \exists R.C}{E \sqsubseteq R.C}$</td>
</tr>
<tr>
<td>$R_\supseteq$</td>
<td>$\frac{E \sqsubseteq D}{E \sqsubseteq D}$ : $D_1 \cap D_2$ occur negatively in $\mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{E \sqsubseteq \exists S.D}{E \sqsubseteq \exists S.D}$ : $\exists S.D$ occurs negatively in $\mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{C \sqsubseteq D \land E \sqsubseteq D \in \mathcal{O}}{C \sqsubseteq E}$ : $D \sqsubseteq E \in \mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{E \sqsubseteq S}{E \sqsubseteq S}$ : $R \sqsubseteq S \in \mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{C \sqsubseteq D_1 \cap D_2}{C \sqsubseteq D_1 \cap D_2}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{E \sqsubseteq \exists R.C}{E \sqsubseteq R.C}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{E \sqsubseteq \exists S.D}{E \sqsubseteq \exists S.D}$ : $\exists S.D$ occurs negatively in $\mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{C \sqsubseteq D \land E \sqsubseteq D \in \mathcal{O}}{C \sqsubseteq E}$ : $D \sqsubseteq E \in \mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{E \sqsubseteq S}{E \sqsubseteq S}$ : $R \sqsubseteq S \in \mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{C \sqsubseteq D_1 \cap D_2}{C \sqsubseteq D_1 \cap D_2}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{E \sqsubseteq \exists R.C}{E \sqsubseteq R.C}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{E \sqsubseteq \exists S.D}{E \sqsubseteq \exists S.D}$ : $\exists S.D$ occurs negatively in $\mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{C \sqsubseteq D \land E \sqsubseteq D \in \mathcal{O}}{C \sqsubseteq E}$ : $D \sqsubseteq E \in \mathcal{O}$</td>
</tr>
<tr>
<td>$R_\subseteq$</td>
<td>$\frac{E \sqsubseteq S}{E \sqsubseteq S}$ : $R \sqsubseteq S \in \mathcal{O}$</td>
</tr>
</tbody>
</table>

**Fig. 3** Optimized inference rules for classification of $\mathcal{EL}^{+}_{\bot}$ ontologies
Encoding a calculus in rules

Three different types of inferences

- **$R_0$:** init($C$)
  \[ \frac{\text{ init } (C) }{ C \sqsubseteq C } \]

- **$R_T$:** init($C$)
  \[ \frac{\text{ init } (C) }{ C \sqsubseteq T } \] : $T$ occurs negatively in $\mathcal{O}$

- **$R_\sqcap$:** $C \sqsubseteq D_1 \sqcap D_2$
  \[ \frac{ C \sqsubseteq D_1 \quad C \sqsubseteq D_2 }{ C \sqsubseteq D_1 \sqcap D_2 } \]

- **$R_\exists$:** $C \sqsubseteq \exists R.C$
  \[ \frac{ E \sqsubseteq \exists R.C }{ E \sqsubseteq C \sqsubseteq R } \]

- **$R_\equiv$:** $C \equiv D$
  \[ \frac{ E \equiv D }{ C \equiv E } \]

- **$R_\Rightarrow$:** $E \Rightarrow C$
  \[ \frac{ E \Rightarrow C \quad C \equiv D }{ E \equiv \exists S.D } \] : $\exists S.D$ occurs negatively in $\mathcal{O}$

- **$R_\Leftrightarrow$:** $R \Leftrightarrow S$
  \[ \frac{ E \Leftrightarrow C \quad C \equiv D \quad R \sqsubseteq \exists S.D }{ E \equiv \exists S.D } \]

Fig. 3 Optimized inference rules for classification of $\mathcal{EL}^+_\sqcap$ ontologies
Encoding a calculus in rules

Three different types of inferences

Four kinds of side conditions

Fig. 3 Optimized inference rules for classification of $\mathcal{EL}_\bot^+$ ontologies
We simply turn every expression in the calculus into a fact:

<table>
<thead>
<tr>
<th>Expression in calculus</th>
<th>Encoding in Datalog facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$ occurs negatively in $O$</td>
<td>nf:isSubClass($C$)</td>
</tr>
<tr>
<td>$C \sqsubseteq D \in O$</td>
<td>nf:subClassOf($C,D$)</td>
</tr>
<tr>
<td>$R \sqsubseteq^*_O S$</td>
<td>nf:subPro0f($R,S$)</td>
</tr>
<tr>
<td>$S_1 \circ S_2 \sqsubseteq S$</td>
<td>nf:subPropChain($S_1,S_2,S$)</td>
</tr>
<tr>
<td>$C \sqsubseteq D$</td>
<td>inf:subClass0f($C,D$)</td>
</tr>
<tr>
<td>$E \overset{R}{\rightarrow} C$</td>
<td>inf:ex($E,R,C$)</td>
</tr>
<tr>
<td>init($C$)</td>
<td>inf:init($C$)</td>
</tr>
</tbody>
</table>
Encoding class expressions

We also need to encode the structure of class expressions

Example:
The class $A \sqcap \exists R. (B \sqcap C)$ is encoded by facts

$n:conj("A \sqcap \exists R. (B \sqcap C)", A, "\exists R. (B \sqcap C)")$

$n:exists("\exists R. (B \sqcap C)", R, "B \sqcap C")$

$n:conj("B \sqcap C", B, C)$

where every sub-expression is represented by a constant.

Expressions $\top$ and $\bot$ are encoded by their special OWL names $\text{owl:Thing}$ and $\text{owl:Nothing}$. 
Encoding class expressions

We also need to encode the structure of class expressions

We use an obvious encoding where every sub-expression becomes a fact.

**Example:** The class $A \sqcap \exists R.(B \sqcap C)$ is encoded by facts

```
nf:conj("A \sqcap \exists R.(B \sqcap C)" , A, "\exists R.(B \sqcap C)")
nf:exists("\exists R.(B \sqcap C)" , R, "B \sqcap C")
nf:conj("B \sqcap C" , B, C)
```

where every sub-expression is represented by a constant.

Expressions $\top$ and $\bot$ are encoded by their special OWL names `owl:Thing` and `owl:Nothing`. 
## Encoding expressions in predicates

<table>
<thead>
<tr>
<th>Expression in calculus</th>
<th>Encoding in Datalog facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\top$</td>
<td><code>owl:Thing</code></td>
</tr>
<tr>
<td>$\bot$</td>
<td><code>owl:Nothing</code></td>
</tr>
<tr>
<td>$X = \exists R.C$</td>
<td><code>nf:exists(X,R,C)</code></td>
</tr>
<tr>
<td>$X = C \cap D$</td>
<td><code>nf:conj(X,C,D)</code></td>
</tr>
<tr>
<td>$C$ occurs negatively in $O$</td>
<td><code>nf:isSubClass(C)</code></td>
</tr>
<tr>
<td>$C \sqsubseteq D \in O$</td>
<td><code>nf:subClassOf(C,D)</code></td>
</tr>
<tr>
<td>$R \sqsubseteq^* S$</td>
<td><code>nf:subProOf(R,S)</code></td>
</tr>
<tr>
<td>$S_1 \circ S_2 \sqsubseteq S$</td>
<td><code>nf:subPropChain(S_1,S_2,S)</code></td>
</tr>
<tr>
<td>$C \sqsubseteq D$</td>
<td><code>inf:subClassOf(C,D)</code></td>
</tr>
<tr>
<td>$E \overset{R}{\rightarrow} C$</td>
<td><code>inf:ex(E,R,C)</code></td>
</tr>
<tr>
<td><code>init(C)</code></td>
<td><code>inf:init(C)</code></td>
</tr>
</tbody>
</table>
Now all rules from the paper can simply be transcoded

Example:

\[
R^+_\cap \quad \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \cap D_2} : D_1 \cap D_2 \text{ occur negatively in } \mathcal{O}
\]

becomes

\[
\text{inf:subClassOf}(\text{?C}, \text{?D1andD2}) : - \\
\text{inf:subClassOf}(\text{?C}, \text{?D1}), \text{inf:subClassOf}(\text{?C}, \text{?D2}), \text{nf:conj} (\text{?D1andD2}, \text{?D1}, \text{?D2}), \text{nf:isSubClass} (\text{?D1andD2})
\]
Steps to produce the Datalog rules:

1. Read the paper carefully and understand the rule structure
2. Define predicates to encode the relevant expressions
3. Rewrite the rules in the new language

Steps to classify an ontology:

1. Encode the ontology using facts for the nfo: predicates
2. Store the facts in an rls file, or in csv files
3. Evaluate this data with the calculus rules
4. Computed subclass relations are in predicate inf:subClassOf

Remark: Performance can often be improved by tweaking rules.
See performance hints at the end of this slide set (link).
Hands-On #5: Classifying Galen-EL

Let’s classify the Galen ontology (EL version)

(1) @clear ALL . (if still running)

(2) Register normalised Galen sources and load calculus:
   @load "el/galen-sources.rls" .
   @load "el/elk-calculus.rls" .

(3) @reason .

(4) Try some queries:\(^1\)
   @query COUNT mainSubClassOf(\(?A,\?B\)) .
   @query mainSubClassOf(\(?A,\text{galen:}\text{Pulse}\)) .

(5) Export classification to file:
   @query mainSubClassOf(\(?A,\?B\) EXPORTCSV "galen-inf-subclass.csv" .

\(^1\) Our output predicate mainSubClassOf is inferred to be the same as \text{inf:subClassOf},
but restricted to named classes.
The calculus requires us to pre-compute facts for the ontology encoding

- Standard libraries like the OWL API for Java can help
- But it still requires another software tool
Normalisation

The calculus requires us to pre-compute facts for the ontology encoding

- Standard libraries like the OWL API for Java can help
- But it still requires another software tool

Can’t we do this in rules, too?

**Rationale:**

- OWL (DL) ontologies are typically stored in an RDF encoding
- Rulewerk and VLog can read RDF data natively
- Rules can perform structural transformations
The RDF format describes labelled graphs, and DL axioms are encoded in graphs as well.

The following graph encodes $A \sqsubseteq \exists R. (B \cap C)$:
Observation: OWL/RDF contains enough auxiliary nodes to use to represent subexpressions!
Observation: OWL/RDF contains enough auxiliary nodes to use to represent subexpressions!

Making suitable rules is not hard:

- Extracting $C \subseteq D$:
  \[
  \text{nf:subClassOf}(?C, ?D) :\text{-} \text{TRIPLE}(?C, \text{rdfs:subClassOf}, ?D) .
  \]

- Extracting $\exists R. X$:
  \[
  \]

- Extracting binary $B \cap C$:
  \[
  \text{ex:conj}(?X, ?B, ?C) :\text{-} \\
  \text{TRIPLE}(?X, \text{owl:intersectionOf}, ?L1), \\
  \text{TRIPLE}(?L1, \text{rdf:next}, ?L2), \text{TRIPLE}(?L2, \text{rdf:next}, \text{rdf:nil}), \\
  \text{TRIPLE}(?L1, \text{rdf:first}, ?B), \text{TRIPLE}(?L2, \text{rdf:first}, ?C) .
  \]

The general case requires some more rules, since OWL encodes n-ary conjunctions as linked lists.

Markus Krötzsch, 13 September 2020
Reusing sub-expressions

**Problem**: The same class expression can occur thousands of times in one ontology.

$\Rightarrow$ duplicated structures, which will all be inferred to be equivalent!
Reusing sub-expressions

**Problem:** The same class expression can occur thousands of times in one ontology → duplicated structures, which will all be inferred to be equivalent!

**Solution:** Replace auxiliary nodes by new elements, unique for each expression
Reusing sub-expressions

**Problem:** The same class expression can occur thousands of times in one ontology
~ duplicated structures, which will all be inferred to be equivalent!

**Solution:** Replace auxiliary nodes by new elements, unique for each expression

**Approach:**
- Mark the “main classes” that are not used in auxiliary positions (using negation)
- Use auxiliary predicates for syntactic extraction, e.g.:

  ```
  ```
- Create and define representatives for every expression, recursively:

  ```
  repOf(?X,?X) :- nf:isMainClass(?X).
  synExRep(?X,?R,?Rep) :- synEx(?X,?R,?Y), repOf(?Y,?Rep).
  ```
Hands-On #6: Normalising Galen

Rules for OWL $\mathcal{EL}$ normalisation are given in `el/elk-normalisation.rls`

Steps to normalise Galen EL from OWL/RDF

1. `@clear ALL . (if still running)
2. Load Galen from RDF:
   - `@load RDF "el/galen-el.rdf" .`
3. Load the normalisation rules:
   - `@load "el/elk-normalisation.rls" .`
4. `@reason .`
5. Check result, e.g.,
   - `@query nf:exists(?X,?R,?C) LIMIT 10 .`
6. Export normalised facts to CSV, e.g.,
   - `@query nf:subClassOf(?C,?D) EXPORTCSV "my-galen-subClass0f.csv" .`
Putting it all together

We have just implemented a complete $\mathcal{EL}$ reasoner in 46 existential rules: just load elk-normalisation.rls and elk-calculus-optimised.rls together with the triples of a OWL/RDF file!

How about performance?
- Running normalisation and reasoning separately is faster than doing everything in one step (more rules – harder to optimise for VLog)
- Performance is below dedicated OWL EL reasoners, but practical:

<table>
<thead>
<tr>
<th></th>
<th>Laptop, Intel i7 2.70GHz, 4G Java heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalisation only</td>
<td>Reasoning only</td>
</tr>
<tr>
<td>GALEN EL (250K triples)</td>
<td>2.5sec</td>
</tr>
<tr>
<td>SNOMED CT (2.9M triples)</td>
<td>30sec</td>
</tr>
</tbody>
</table>

But then again, this only took <50 lines of code!
We have just implemented a complete $\mathcal{EL}$ reasoner in 46 existential rules: just load elk-normalisation.rls and elk-calculus-optimised.rls together with the triples of a OWL/RDF file!

How about performance?

- Running normalisation and reasoning separately is faster than doing everything in one step (more rules – harder to optimise for VLog)
- Performance is below dedicated OWL EL reasoners, but practical:

<table>
<thead>
<tr>
<th></th>
<th>Normalisation only</th>
<th>Reasoning only</th>
<th>All in one</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Laptop, Intel i7 2.70GHz, 4G Java heap]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GALEN EL (250K triples)</td>
<td>2.5sec</td>
<td>25sec</td>
<td>4min</td>
</tr>
<tr>
<td>SNOMED CT (2.9M triples)</td>
<td>30sec</td>
<td>2min</td>
<td>9min</td>
</tr>
</tbody>
</table>

But then again, this only took <50 lines of code!
What we learned

- Datalog and its extensions are simple rule languages
- Rulewerk/VLog are fast, free tools for existential rules with stratified negation
- Many rules-based reasoning calculi can be implemented in rules:
  1. Develop suitable encoding
  2. Translate and debug rules
  3. Optimise performance
- Rules also help with related tasks (normalisation, reduction, result comparison, ...)

Up next: reasoning beyond P, probabilistic reasoning, stream reasoning, ...
Further reading about VLog and Rulewerk:

Current main reference for Rulewerk (formerly: VLog4j)

Introduction of existential rules in VLog; performance benchmarks

Original publication about VLog’s design and optimisations
References (2)

Further Datalog Applications and Systems:


References (3)


[11] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor: **codeQuest: Scalable Source Code Queries with Datalog.** ECOOP 2006: 2-27 This approach has been further developed into commercial services of Semmle, https://semmle.com/

References (4)


References (5)

Rules for reasoning in $\mathcal{EL}$


[18] Markus Krötzsch: Efficient Rule-Based Inferencing for OWL EL. IJCAI 2011: 2668-2673 An earlier, less efficient Datalog calculus for $\mathcal{EL}$
Appendix
Skolemisation

One can also handle existential quantifiers by applying them with skolem terms

- Done in many existential rule reasoners internally
- Skolem terms also work in other logic programming tools, e.g., ASP solvers

**Example:** We encountered the following statements in our hands-on:

\[
\begin{align*}
\text{painting}(\text{Marine}, \text{Renoir}, \text{Guernsey}) & \quad \text{type}(\text{Guernsey}, \text{island}) \\
\text{painting}(\text{Marine}, \text{Renoir}, \text{island}) & \\
\text{painting}(x, y, z) \land \text{type}(u, z) \rightarrow \exists v. \text{painting}(x, y, v) \land \text{type}(v, z)
\end{align*}
\]

If we replace \(v\) with \(f(x, y)\), then \(\text{painting}(\text{Marine}, \text{Renoir}, f(\text{Marine}, \text{Renoir}))\) would be derived by a reasoner. The restricted chase would not derive any such fact, since the constant Guernsey can already take the place of the existential \(v\).
Performance tuning for VLog

Performance can often be improved by adjusting rules . . .

. . . but effective tuning requires knowledge of the reasoner!

Special aspects of VLog:

• Predicate tuples are indexed in their given order
  
  Fast:
  
  p(?X,?Y,?Z), q(?X,?Y,?V)
  
  Slow:
  
  p(?Z,?Y,?X), q(?V,?X,?Y)
  
• Body conjunctions are evaluated using binary joins
  
  Fast: short bodies; selective binary joins
  
  Slow: long bodies; possibly very un-selective joins

Running in VLog in debug-mode can yield insights on slow rule executions.

Markus Krötzsch, 13 September 2020  Practical Uses of Existential Rules in Knowledge Representation
Performance tuning for VLog

Performance can often be improved by adjusting rules . . .
. . . but effective tuning requires knowledge of the reasoner!

Special aspects of VLog:

- Predicate tuples are indexed in their given order

- Body conjunctions are evaluated using binary joins

- Join order is determined by heuristics (esp. predicate size)
  - Fast: short bodies; selective binary joins
  - Slow: long bodies; possibly very un-selective joins

Running in VLog in debug-mode can yield insights on slow rule executions.
Some rules are hard to process:

```prolog
inf:subClassOf(?E,?Y) :- inf:ex(?E,?R,?C), inf:subClassOf(?C,?D),
                    nf:subProp(?R,?S), nf:exists(?Y,?S,?D), nf:isSubClass(?Y) .
```

Likely bad join order (starting from small predicates):

```
(nf:exists(?Y,?S,?D) ⊞ ◁ nf:subProp(?R,?S)) ⊞ ◁ inf:ex(?E,?R,?C)
```

But most ontologies have very few properties (\(?R, ?S\) each used in a large part of the existential restrictions {essentially a product \(nf:exists(?Y,?S,?D) \times inf:ex(?E,?R,?C)\)).

Solution: Replace problematic rule by several rules:

```
subExt(?D,?R,?Y) :- nf:subProp(?R,?S), nf:exists(?Y,?S,?D), nf:isSubClass(?Y) .
aux(?C,?R,?Y) :- inf:subClassOf(?C,?D), subExt(?D,?R,?Y) .
inf:subClassOf(?E,?Y) :- inf:ex(?E,?R,?C), aux(?C,?R,?Y) .
```
Performance tuning 1: Decompose rules

Some rules are hard to process:

\[
\]

Likely bad join order (starting from small predicates):

\[
\]

But most ontologies have very few properties (?R, ?S), each used in a large part of the existential restrictions \(\sim\) essentially a product \(\text{nf:exists}(?Y, ?S, ?D) \times \text{inf:ex}(?E, ?R, ?C)\)
Performance tuning 1: Decompose rules

Some rules are hard to process:

\[
\text{inf:subClassOf}(\text{?E,?Y}) :\text{ inf:ex}(\text{?E,?R,?C}), \text{ inf:subClassOf}(\text{?C,?D}),
\text{ nf:subProp}(\text{?R,?S}), \text{ nf:exists}(\text{?Y,?S,?D}), \text{ nf:isSubClass}(\text{?Y}) .
\]

Likely bad join order (starting from small predicates):

\[
(\text{nf:exists}(\text{?Y,?S,?D}) \Join \text{nf:subProp}(\text{?R,?S})) \Join \text{inf:ex}(\text{?E,?R,?C})
\]

But most ontologies have very few properties \((?R, ?S)\), each used in a large part of the existential restrictions \(\leadsto\) essentially a product \(\text{nf:exists}(\text{?Y,?S,?D}) \times \text{inf:ex}(\text{?E,?R,?C})\)

Solution: Replace problematic rule by several rules:

\[
\text{subExt}(\text{?D,?R,?Y}) :\text{ nf:subProp}(\text{?R,?S}), \text{ nf:exists}(\text{?Y,?S,?D}),
\text{ nf:isSubClass}(\text{?Y}) .
\text{aux}(\text{?C,?R,?Y}) :\text{ inf:subClassOf}(\text{?C,?D}), \text{ subExt}(\text{?D,?R,?Y}) .
\text{inf:subClassOf}(\text{?E,?Y}) :\text{ inf:ex}(\text{?E,?R,?C}), \text{ aux}(\text{?C,?R,?Y}) .
\]
Performance tuning 2: Argument order

**Argument order in derived predicates can be changed:**

\[
\text{inf:subClassOf}(\text{?E}, \text{?Y}) \leftarrow \text{inf:ex}(\text{?E}, \text{?R}, \text{?C}), \text{aux}(\text{?C}, \text{?R}, \text{?Y}).
\]
Argument order in derived predicates can be changed:

\[
\text{inf:subClassOf}(\text{?E}, \text{?Y}) : \text{inf:ex}(\text{?E}, \text{?R}, \text{?C}), \text{aux}(\text{?C}, \text{?R}, \text{?Y}) .
\]

For this rule, it would work better if we flipped the order of \text{inf:ex}:

\[
\text{inf:subClassOf}(\text{?E}, \text{?Y}) : \text{inf:xe}(\text{?C}, \text{?R}, \text{?E}), \text{aux}(\text{?C}, \text{?R}, \text{?Y}) .
\]

Of course, this must be done across all rules!
Performance tuning 2: Argument order

Argument order in derived predicates can be changed:

\[
\]

For this rule, it would work better if we flipped the order of \text{inf:ex}:

\[
\]

Of course, this must be done across all rules!

An optimised version of the calculus is in file \text{el/elk-calculus-optimised.rls}. Try it with Galen.

**General guideline:** There is no simple rule for how to improve performance, since many optimisations interact. Try what works best.

(The fastest results come from making typos: be sure to check correctness, too!)