PRACTICAL USES OF EXISTENTIAL RULES IN KNOWLEDGE REPRESENTATION

Part 1: Basics / Rules for Ontology Reasoning

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

Special thanks to Irina Dragoste,[1] Ceriel Jacobs,[2] and Maximilian Marx[1]
for their invaluable contributions to the software used in this tutorial

KR, 13 September 2020

Goals of this tutorial

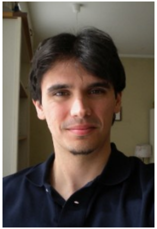
Topic: Existential rules as an approach to declarative computation, some of its application areas in AI, and practical tools to implement them in practice.

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

David Carral

Markus Krötzsch



Jacopo Urbani

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:

$$\forall x, y, z.\quad \text{contains}(x, y) \wedge \text{contains}(y, z) \rightarrow \text{contains}(x, z)$$

$$\forall x.\quad \text{drink}(x) \wedge \text{contains}(x, \text{carbonDioxide}) \rightarrow \text{fizzyDrink}(x)$$

Simple rules: Datalog

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

Wanted: A way to define derived relations, possibly recursively

Example:

$$\forall x, y, z. \quad \mathsf{contains}(x, y) \land \mathsf{contains}(y, z) \to \mathsf{contains}(x, z)$$

$$\forall x. \quad \mathsf{drink}(x) \land \mathsf{contains}(x, \mathsf{carbonDioxide}) \to \mathsf{fizzyDrink}(x)$$

universal quantifier (usually not written)

variable

constant symbol

predicate

General form:

$$\underbrace{\text{conjunction of relational atoms}}_{\text{rule body}} \to \underbrace{\text{relational atom}}_{\text{rule head}}$$

Evaluating Datalog

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

Example: We use rules as before

$$(R1) \qquad \mathsf{contains}(x, y) \wedge \mathsf{contains}(y, z) \to \mathsf{contains}(x, z)$$

$$(R2) \qquad \mathsf{drink}(x) \wedge \mathsf{contains}(x, \mathsf{carbonDioxide}) \to \mathsf{fizzyDrink}(x)$$

on a database with the following facts:

drink(limeAndSoda)

contains(limeAndSoda, limeSyrup) contains(limeAndSoda, sodaWater)

contains(sodaWater, water) contains(sodaWater, carbonDioxide)

Applying rules yields:

from R1: contains(limeAndSoda, water)

from R1: contains(limeAndSoda, carbonDioxide)

from R2: 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.

Getting practical: VLog + Rulewerk

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
  (see `https://iccl.inf.tu-dresden.de/web/Rules_KR_Tutorial_2020/en`)
- Open a command line in the tutorial directory and type:
  `java -jar rulewerk-client.jar`

Rules in Rulewerk

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:

```
fizzyDrink(?x) :- drink(?x), contains(?x,"carbon dioxide") .
```

universal variables marked by ?

`:-` as implication written head-first

comma is “and”

"strings" also work

everything ends with a dot

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.

Hands-On #1: Using Rulewerk client (2)

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
$\rightsquigarrow$ `@assert` is not the way to get there

Supported sources for larger datasets:

- RLS files with Rulewerk knowledge bases[1]
- CSV files (one predicate per file)[2]
- RDF graphs in NTriples format[2] or any other standard format[1] (one ternary triple-predicate per file)
- OWL ontologies (converted to rules and facts)[1]
- Graal knowledge bases[1]
- Trident database files (large-scale, disk-based RDF graph index; open source)[2]
- SPARQL query results[2]
- Other ODBC database connectors[3]

[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:**

Title	Painter	Shown in picture
"Mona Lisa"	"Leonardo da Vinci"	"Lisa del Giocondo"
"Mona Lisa"	"Leonardo da Vinci"	"landscape"
"Self-Portrait with Monkey"	"Frida Kahlo"	"Frida Kahlo"
"Self-Portrait with Monkey"	"Frida Kahlo"	"monkey"
	... (205,236 more rows)	

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

Instance	Class
"Dresden"	"city"
"Rhodes"	"island"
"Frida Kahlo"	"human"
... (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") .
```

Hands-On #2: Handling larger knowledge bases (3)

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:
```
@clear ALL .
```
(2) Load the knowledge base and view the loaded knowledge base:
```
@load "art/rules.rls" .
@showkb .
```
(3) Invoke VLog: `@reason .`

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

Hint: Note how `COUNT` and `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:

$$\langle\text{number of predicate names}\rangle \times \langle\text{number of constants}\rangle^{\langle\text{max. predicate arity}\rangle}$$

- 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:

$$\langle\text{number of predicate names}\rangle \times \langle\text{number of constants}\rangle^{\langle\text{max. predicate arity}\rangle}$$

- 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 can not 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:

$$\text{painting}(x, y, z) \wedge \text{type}(u, z) \to \exists v.\text{painting}(x, y, v) \wedge \text{type}(v, z)$$

(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

Hands-On #3: Adding existential rules

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:

```
@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

$\leadsto$ 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

$\leadsto$ 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.

$\leadsto$ 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 Chase

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 exists.

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:

$$\mathsf{painting}(x, y, \mathsf{fork}) \land \neg\mathsf{painting}(x, y, \mathsf{knife}) \to \mathsf{query}(x, y).$$

Negation

Negation is another extremely useful extension of rule languages.

Example:

$$\text{painting}(x, y, \text{fork}) \wedge \neg\text{painting}(x, y, \text{knife}) \rightarrow \text{query}(x, y).$$

Mixing negation with recursion can be complicated:

Example:

$$\neg p(x) \rightarrow q(x) \qquad\qquad \neg q(x) \rightarrow p(x)$$

- 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:

$$\text{painting}(x, y, \text{fork}) \land \neg\text{painting}(x, y, \text{knife}) \to \text{query}(x, y).$$

Mixing negation with recursion can be complicated:

Example:

$$\neg p(x) \to q(x) \qquad\qquad \neg q(x) \to p(x)$$

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

$\leadsto$ 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").

Hands-On #4: Adding 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
  $\forall x.\text{Portrait}(x) \to \exists y.\text{Painting}(x) \wedge \text{depicts}(x, y) \wedge \text{Person}(y)$
- New subsumptions might be inferred, e.g., Portrait $\sqsubseteq$ Painting (classification)

The DL $\mathcal{EL}^+_\perp$ in a nutshell

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

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

$\perp$	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"
$C \sqcap D$	intersection	"set of all elements that are in class C and in class D"

The DL $\mathcal{EL}^{+}_{\perp}$ in a nutshell

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

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

$\perp$	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"
$C \sqcap D$	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"

How to reason in $\mathcal{EL}^{+}_{\perp}$

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?

Prior research . . .

Published: 17 November 2013

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 …

$$\mathsf{R}_0\ \frac{\mathsf{init}(C)}{C \sqsubseteq C} \qquad \mathsf{R}_\top\ \frac{\mathsf{init}(C)}{C \sqsubseteq \top} : \top \text{ occurs negatively in } \mathcal{O} \qquad \mathsf{R}_\bot\ \frac{E \xrightarrow{R} C \quad C \sqsubseteq \bot}{E \sqsubseteq \bot}$$

$$\mathsf{R}_\sqcap^-\ \frac{C \sqsubseteq D_1 \sqcap D_2}{C \sqsubseteq D_1 \quad C \sqsubseteq D_2} \qquad \mathsf{R}_\sqcap^+\ \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occur negatively in } \mathcal{O}$$

$$\mathsf{R}_\exists^-\ \frac{E \sqsubseteq \exists R.C}{E \xrightarrow{R} C} \qquad \mathsf{R}_\exists^+\ \frac{E \xrightarrow{R} C \quad C \sqsubseteq D}{E \sqsubseteq \exists S.D} : \begin{array}{l} R \sqsubseteq^*_\mathcal{O} S \\ \exists S.D \text{ occurs negatively in } \mathcal{O} \end{array}$$

$$\mathsf{R}_\sqsubseteq\ \frac{C \sqsubseteq D}{C \sqsubseteq E} : D \sqsubseteq E \in \mathcal{O} \qquad \mathsf{R}_\circ\ \frac{E \xrightarrow{R_1} C \quad C \xrightarrow{R_2} D}{E \xrightarrow{S} D} : \begin{array}{l} R_1 \sqsubseteq^*_\mathcal{O} S_1 \\ R_2 \sqsubseteq^*_\mathcal{O} S_2 \\ S_1 \circ S_2 \sqsubseteq S \in \mathcal{O} \end{array} \qquad \mathsf{R}_\rightsquigarrow\ \frac{E \xrightarrow{R} C}{\mathsf{init}(C)}$$

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

How to read such rules

General form of the rules:

$$\text{rule name}\ \frac{\text{pre-condition}}{\text{conclusion}} : \text{side condition}$$

For example:

$$\mathsf{R}^+_\sqcap\ \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occur negatively in } \mathcal{O}$$

where the parts have the following meaning:

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

Encoding a calculus in rules

$$\mathsf{R}_0\ \frac{\mathsf{init}(C)}{C \sqsubseteq C} \qquad \mathsf{R}_\top\ \frac{\mathsf{init}(C)}{C \sqsubseteq \top} : \top \text{ occurs negatively in } \mathcal{O} \qquad \mathsf{R}_\bot\ \frac{E \xrightarrow{R} C \quad C \sqsubseteq \bot}{E \sqsubseteq \bot}$$

$$\mathsf{R}_\sqcap^-\ \frac{C \sqsubseteq D_1 \sqcap D_2}{C \sqsubseteq D_1 \quad C \sqsubseteq D_2} \qquad \mathsf{R}_\sqcap^+\ \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occur negatively in } \mathcal{O}$$

$$\mathsf{R}_\exists^-\ \frac{E \sqsubseteq \exists R.C}{E \xrightarrow{R} C} \qquad \mathsf{R}_\exists^+\ \frac{E \xrightarrow{R} C \quad C \sqsubseteq D}{E \sqsubseteq \exists S.D} : \begin{array}{l} R \sqsubseteq^*_\mathcal{O} S \\ \exists S.D \text{ occurs negatively in } \mathcal{O} \end{array}$$

$$\mathsf{R}_\sqsubseteq\ \frac{C \sqsubseteq D}{C \sqsubseteq E} : D \sqsubseteq E \in \mathcal{O} \qquad \mathsf{R}_\circ\ \frac{E \xrightarrow{R_1} C \quad C \xrightarrow{R_2} D}{E \xrightarrow{S} D} : \begin{array}{l} R_1 \sqsubseteq^*_\mathcal{O} S_1 \\ R_2 \sqsubseteq^*_\mathcal{O} S_2 \\ S_1 \circ S_2 \sqsubseteq S \in \mathcal{O} \end{array} \qquad \mathsf{R}_\rightsquigarrow\ \frac{E \xrightarrow{R} C}{\mathsf{init}(C)}$$

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

Encoding a calculus in rules

Three different types of inferences

$$\mathsf{R}_0\ \frac{\mathsf{init}(C)}{C \sqsubseteq C} \qquad \mathsf{R}_\top\ \frac{\mathsf{init}(C)}{C \sqsubseteq \top} : \top \text{ occurs negatively in } \mathcal{O} \qquad \mathsf{R}_\bot\ \frac{E \xrightarrow{R} C \quad C \sqsubseteq \bot}{E \sqsubseteq \bot}$$

$$\mathsf{R}_\sqcap^-\ \frac{C \sqsubseteq D_1 \sqcap D_2}{C \sqsubseteq D_1 \quad C \sqsubseteq D_2} \qquad \mathsf{R}_\sqcap^+\ \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occur negatively in } \mathcal{O}$$

$$\mathsf{R}_\exists^-\ \frac{E \sqsubseteq \exists R.C}{E \xrightarrow{R} C} \qquad \mathsf{R}_\exists^+\ \frac{E \xrightarrow{R} C \quad C \sqsubseteq D}{E \sqsubseteq \exists S.D} : \begin{array}{l} R \sqsubseteq^*_\mathcal{O} S \\ \exists S.D \text{ occurs negatively in } \mathcal{O} \end{array}$$

$$\mathsf{R}_\sqsubseteq\ \frac{C \sqsubseteq D}{C \sqsubseteq E} : D \sqsubseteq E \in \mathcal{O} \qquad \mathsf{R}_\circ\ \frac{E \xrightarrow{R_1} C \quad C \xrightarrow{R_2} D}{E \xrightarrow{S} D} : \begin{array}{l} R_1 \sqsubseteq^*_\mathcal{O} S_1 \\ R_2 \sqsubseteq^*_\mathcal{O} S_2 \\ S_1 \circ S_2 \sqsubseteq S \in \mathcal{O} \end{array} \qquad \mathsf{R}_\rightsquigarrow\ \frac{E \xrightarrow{R} C}{\mathsf{init}(C)}$$

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

Encoding a calculus in rules

Three different types of inferences

$$\mathsf{R}_0\ \frac{\mathsf{init}(C)}{C \sqsubseteq C} \qquad \mathsf{R}_\top\ \frac{\mathsf{init}(C)}{C \sqsubseteq \top} : \top \text{ occurs negatively in } \mathcal{O} \qquad \mathsf{R}_\bot\ \frac{E \xrightarrow{R} C \quad C \sqsubseteq \bot}{E \sqsubseteq \bot}$$

$$\mathsf{R}^-_\sqcap\ \frac{C \sqsubseteq D_1 \sqcap D_2}{C \sqsubseteq D_1 \quad C \sqsubseteq D_2} \qquad \mathsf{R}^+_\sqcap\ \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occur negatively in } \mathcal{O}$$

Four kinds of side conditions

$$\mathsf{R}^-_\exists\ \frac{E \sqsubseteq \exists R.C}{E \xrightarrow{R} C} \qquad \mathsf{R}^+_\exists\ \frac{E \xrightarrow{R} C \quad C \sqsubseteq D}{E \sqsubseteq \exists S.D} : \begin{array}{l} R \sqsubseteq^*_{\mathcal{O}} S \\ \exists S.D \text{ occurs negatively in } \mathcal{O} \end{array}$$

$$\mathsf{R}_\sqsubseteq\ \frac{C \sqsubseteq D}{C \sqsubseteq E} : D \sqsubseteq E \in \mathcal{O} \qquad \mathsf{R}_\circ\ \frac{E \xrightarrow{R_1} C \quad C \xrightarrow{R_2} D}{E \xrightarrow{S} D} : \begin{array}{l} R_1 \sqsubseteq^*_{\mathcal{O}} S_1 \\ R_2 \sqsubseteq^*_{\mathcal{O}} S_2 \\ S_1 \circ S_2 \sqsubseteq S \in \mathcal{O} \end{array} \qquad \mathsf{R}_\rightsquigarrow\ \frac{E \xrightarrow{R} C}{\mathsf{init}(C)}$$

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

Encoding expressions in predicates

We simply turn every expression in the calculus into a fact:

Expression in calculus	Encoding in Datalog facts
C occurs negatively in $\mathcal{O}$	$\texttt{nf:isSubClass(}C\texttt{)}$
$C \sqsubseteq D \in \mathcal{O}$	$\texttt{nf:subClassOf(}C\texttt{,}D\texttt{)}$
$R \sqsubseteq^*_{\mathcal{O}} S$	$\texttt{nf:subProOf(}R\texttt{,}S\texttt{)}$
$S_1 \circ S_2 \sqsubseteq S$	$\texttt{nf:subPropChain(}S_1\texttt{,}S_2\texttt{,}S\texttt{)}$
$C \sqsubseteq D$	$\texttt{inf:subClassOf(}C\texttt{,}D\texttt{)}$
$E \xrightarrow{R} C$	$\texttt{inf:ex(}E\texttt{,}R\texttt{,}C\texttt{)}$
init(C)	$\texttt{inf:init(}C\texttt{)}$

Encoding class expressions

We also need to encode the structure of class expressions

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

$$\texttt{nf:conj}(\texttt{"}A \sqcap \exists R.(B \sqcap C)\texttt{"}, A, \texttt{"}\exists R.(B \sqcap C)\texttt{"})$$

$$\texttt{nf:exists}(\texttt{"}\exists R.(B \sqcap C)\texttt{"}, R, \texttt{"}B \sqcap C\texttt{"})$$

$$\texttt{nf:conj}(\texttt{"}B \sqcap C\texttt{"}, B, C)$$

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

Expression in calculus	Encoding in Datalog facts
$\top$	`owl:Thing`
$\bot$	`owl:Nothing`
$X = \exists R.C$	`nf:exists(`X`,`R`,`C`)`
$X = C \sqcap D$	`nf:conj(`X`,`C`,`D`)`
C occurs negatively in $\mathcal{O}$	`nf:isSubClass(`C`)`
$C \sqsubseteq D \in \mathcal{O}$	`nf:subClassOf(`C`,`D`)`
$R \sqsubseteq^*_{\mathcal{O}} S$	`nf:subProOf(`R`,`S`)`
$S_1 \circ S_2 \sqsubseteq S$	`nf:subPropChain(`S_1`,`S_2`,`S`)`
$C \sqsubseteq D$	`inf:subClassOf(`C`,`D`)`
$E \xrightarrow{R} C$	`inf:ex(`E`,`R`,`C`)`
init(C)	`inf:init(`C`)`

Encoding calculus rules in Datalog

Now all rules from the paper can simply be transcoded

Example:

$$\mathsf{R}^{+}_{\sqcap}\ \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occur negatively in } \mathcal{O}$$

becomes

```
inf:subClassOf(?C,?D1andD2) :-
    inf:subClassOf(?C,?D1), inf:subClassOf(?C,?D2),
    nf:conj(?D1andD2,?D1,?D2), nf:isSubClass(?D1andD2) .
```

Bringing it all together

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 `nf:` 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,galen: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 `inf:subClassOf`, but restricted to named classes.

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

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

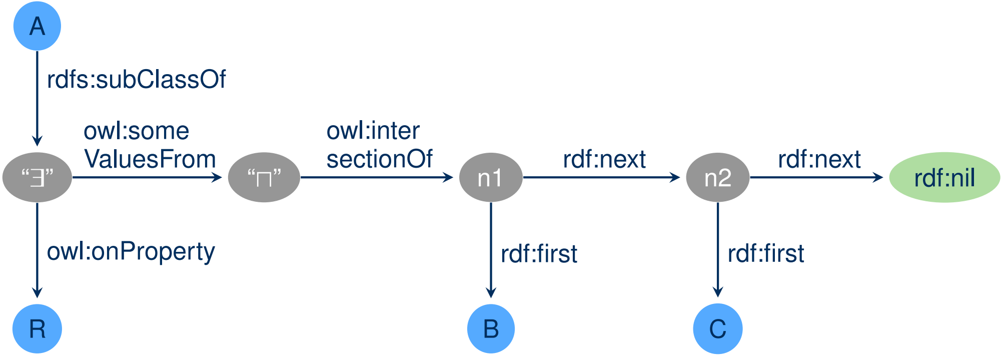
Can't we do this in rules, too?

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 $A \sqsubseteq \exists R.(B \sqcap C)$:



Extracting $\mathcal{EL}$ from RDF

Observation: OWL/RDF contains enough auxiliary nodes to use to represent subexpressions!

Extracting $\mathcal{EL}$ from RDF

Observation: OWL/RDF contains enough auxiliary nodes to use to represent subexpressions!

Making suitable rules is not hard:

- Extracting $C \sqsubseteq D$:

```
nf:subClassOf(?C,?D) :- TRIPLE(?C, rdfs:subClassOf, ?D) .
```

- Extracting $\exists R.X$:

```
nf:exists(?X,?R,?C) :- TRIPLE(?X, owl:someValuesFrom, ?C),
                       TRIPLE(?X, owl:onProperty, ?R) .
```

- Extracting binary $B \sqcap C$:

```
ex:conj(?X,?B,?C) :-
    TRIPLE(?X, owl:intersectionOf, ?L1),
    TRIPLE(?L1,rdf:next,?L2), TRIPLE(?L2,rdf:next,rdf:nil),
    TRIPLE(?L1,rdf:first,?B), TRIPLE(?L2,rdf:first,?C) .
```

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

Reusing sub-expressions

Problem: The same class expression can occur thousands of times in one ontology
$\leadsto$ 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
$\leadsto$ 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
$\leadsto$ 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.:

```
synEx(?X,?R,?C) :- TRIPLE(?X, owl:someValuesFrom, ?C),
                   TRIPLE(?X, owl:onProperty, ?R) .
```

- 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) .
nf:exists(!New,?R,?Rep) :- synExRep(?X,?R,?Rep) .
          repOf(?X,?N) :- synExRep(?X,?R,?Rep), nf:exists(?N,?R,?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-subClassOf.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!

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:

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

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

Summary

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, ...

References (1)

Further reading about VLog and Rulewerk:

[1] David Carral, Irina Dragoste, Larry González, Ceriel J. H. Jacobs, Markus Krötzsch, Jacopo Urbani: **VLog: A Rule Engine for Knowledge Graphs.** ISWC (2) 2019: 19-35
Current main reference for Rulewerk (formerly: VLog4j)

[2] Jacopo Urbani, Markus Krötzsch, Ceriel J. H. Jacobs, Irina Dragoste, David Carral: **Efficient Model Construction for Horn Logic with VLog: System Description.** IJCAR 2018: 680-688 Introduction of existential rules in VLog; performance benchmarks

[3] Jacopo Urbani, Ceriel J. H. Jacobs, Markus Krötzsch: **Column-Oriented Datalog Materialization for Large Knowledge Graphs.** AAAI 2016: 258-264 Original publication about VLog's design and optimisations

References (2)

Further Datalog Applications and Systems:

[4] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, Christopher Ré: **EmptyHeaded: A Relational Engine for Graph Processing.** ACM Trans. Database Syst. 42(4): 20:1-20:44 (2017)

[5] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, Geoffrey Washburn: **Design and Implementation of the LogicBlox System.** SIGMOD Conference 2015: 1371-1382

[6] Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, Swan Rocher, Clément Sipieter: **Graal: A Toolkit for Query Answering with Existential Rules.** RuleML 2015: 328-344

[7] Luigi Bellomarini, Emanuel Sallinger, Georg Gottlob: **The Vadalog System: Datalog-based Reasoning for Knowledge Graphs.** Proc. VLDB Endow. 11(9): 975-987 (2018)

References (3)

[8] David Carral, Irina Dragoste, Markus Krötzsch: **Reasoner = Logical Calculus + Rule Engine.** KI - Künstliche Intelligenz, 2020. Related approaches are presented in the next parts of this tutorial

[9] Cognitect, Inc.: **Datomic.** Product website, accessed Sept 2020, `https://www.datomic.com/`.

[10] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, Donatello Santoro: **That's All Folks! LLUNATIC Goes Open Source.** Proc. VLDB Endow. 7(13): 1565-1568 (2014)

[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/`

[12] Nikolaos Konstantinou, Edward Abel, Luigi Bellomarini, Alex Bogatu, Cristina Civili, Endri Irfanie, Martin Koehler, Lacramioara Mazilu, Emanuel Sallinger, Alvaro A. A. Fernandes, Georg Gottlob, John A. Keane, Norman W. Paton: **VADA: an architecture for end user informed data preparation.** J. Big Data 6: 74 (2019)

References (4)

[13] Benno Kruit, Hongu He, and Jacopo Urbani: **Tab2Know: Building a Knowledge Base from Tables in Scientific Papers.** International Semantic Web Conference 2020, to appear. Also presented in the final part of this tutorial

[14] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, Jay Banerjee: **RDFox: A Highly-Scalable RDF Store.** International Semantic Web Conference (2) 2015: 3-20

[15] Robert Piro, Yavor Nenov, Boris Motik, Ian Horrocks, Peter Hendler, Scott Kimberly, Michael Rossman: **Semantic Technologies for Data Analysis in Health Care.** International Semantic Web Conference (2) 2016: 400-417

References (5)

Rules for reasoning in $\mathcal{EL}$

[16] David Carral, Irina Dragoste, Markus Krötzsch: **Reasoner = Logical Calculus + Rule Engine.** KI - Künstliche Intelligenz, 2020. Further discussion of this use case (rules for reasoning)

[17] Yevgeny Kazakov, Markus Krötzsch, Frantisek Simancik: **The Incredible ELK – From Polynomial Procedures to Efficient Reasoning with $\mathcal{EL}$ Ontologies.** J. Autom. Reason. 53(1): 1-61 (2014) Source of the DL reasoning calculus used herein

[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:

$$\text{painting}(\text{Marine}, \text{Renoir}, \text{Guernsey}) \qquad \text{type}(\text{Guernsey}, \text{island})$$

$$\text{painting}(\text{Marine}, \text{Renoir}, \text{island})$$

$$\text{painting}(x, y, z) \wedge \text{type}(u, z) \rightarrow \exists v.\text{painting}(x, y, v) \wedge \text{type}(v, z)$$

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!

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
- 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.

Performance tuning 1: Decompose rules

Some rules are hard to process:

```
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) .
```

Performance tuning 1: Decompose rules

Some rules are hard to process:

```
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):

$$(\texttt{nf:exists(?Y,?S,?D)} \bowtie \texttt{nf:subProp(?R,?S)}) \bowtie \texttt{inf:ex(?E,?R,?C)}$$

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

Performance tuning 1: Decompose rules

Some rules are hard to process:

```
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):

$$(\texttt{nf:exists(?Y,?S,?D)} \bowtie \texttt{nf:subProp(?R,?S)}) \bowtie \texttt{inf:ex(?E,?R,?C)}$$

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

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 2: Argument order

Argument order in derived predicates can be changed:

```
inf:subClassOf(?E,?Y) :- inf:ex(?E,?R,?C), aux(?C,?R,?Y) .
```

Performance tuning 2: Argument order

Argument order in derived predicates can be changed:

```
inf:subClassOf(?E,?Y) :- inf:ex(?E,?R,?C), aux(?C,?R,?Y) .
```

For this rule, it would work better if we flipped the order of `inf:ex`:

```
inf:subClassOf(?E,?Y) :- inf:xe(?C,?R,?E), aux(?C,?R,?Y) .
```

Of course, this must be done across all rules!

Performance tuning 2: Argument order

Argument order in derived predicates can be changed:

```
inf:subClassOf(?E,?Y) :- inf:ex(?E,?R,?C), aux(?C,?R,?Y) .
```

For this rule, it would work better if we flipped the order of `inf:ex`:

```
inf:subClassOf(?E,?Y) :- inf:xe(?C,?R,?E), aux(?C,?R,?Y) .
```

Of course, this must be done across all rules!

An optimised version of the calculus is in file `el/elk-caclulus-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!)