PRACTICAL USES OF EXISTENTIAL RULES IN KNOWLEDGE REPRESENTATION

Part 1: Introduction to Existential Rules

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

ECAI, 4 September 2020

Goals of this tutorial

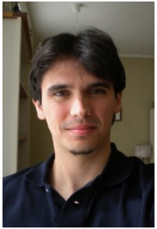
Topic: Existential rules as an approach to declarative computation, some of its application areas in AI, and practical tools to implement them in practice.

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
- **Part 2: Existential Rules in Knowledge Representation**
  - Implementing a lightweight description logic reasoner
  - Guidelines for problem encoding and optimisation
- **Part 3: Reasoning Beyond Polynomial Time**
  - Augmenting Datalog with sets for reasoning in expressive description logics
  - Using existential rules to simulate sets
- **Part 4: Practical Applications of Rules**
  - Probabilistic inference with Datalog
  - Data integration
  - Stream reasoning

Part I: 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:

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

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

Simple rules: Datalog

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

Wanted: A way to define derived relations, possibly recursively

Example:

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

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

universal quantifier (usually not written)

variable

constant symbol

predicate

General form:

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

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
- 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

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

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
$\leadsto$ `@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

We will use data files found in the tutorial folder

(1) Open the file `drinks/rules.rls` in a text editor
note how `@source` statements at the top are used to load data from CSV

(2) Switch to the Rulewerk client and (if still running) delete the data used in the previous hands-on:
```
@clear ALL .
```

(3) Load the knowledge base and view the loaded knowledge base:
```
@load "drinks/rules.rls" .
@showkb .
```

(4) Invoke VLog: `@reason .`

(5) Try some queries to explore the data and inferences:
```
@query COUNT alcoholicBeverage(?X) .
@query ingredient(?drink,?ingredient,?quantity) LIMIT 10 .
@query contains(?X,"chili pepper") .
```

Hint: Note how `COUNT` and `LIMIT` help us to deal with larger query results.

Hands-On #2: Content of knowledge base (1)

```
%%% Declare external data sources:
% Set of known drinks, loaded in unary predicate "drink":
@source drink[1] : load-csv("drinks.csv") .

% Data for ingredients and garnishes (recipe, ingredient, amount):
@source ingredient[3] : load-csv("ingredients.csv") .
@source garnish[3] : load-csv("garnishes.csv") .

% Data about what contains what (container, containee)
@source contains[2] : load-csv("containments.csv") .

% General subclass relationships (subclass, superclass):
@source subClassOf[2] : load-csv("subclasses.csv") .
```

Hands-On #2: Content of knowledge base (2)

```
%%% We can add some more facts here:
drink("lime & soda") .
ingredient("lime & soda", "lime syrup", "2cl") .
ingredient("lime & soda", "carbonated water", "60cl") .
garnish("lime & soda", "lime slice", "1") .
contains("lime syrup", "lime") .
subClassOf("lime syrup", "fruit syrup") .
```

Hands-On #2: Content of knowledge base (3)

```
%%% Rules:
% Preparations contain their ingredients and garnishes:
contains(?X,?Y) :- ingredient(?X,?Y,?amount) .
contains(?X,?Y) :- garnish(?X,?Y,?amount) .
% Containment is transitive:
contains(?X,?Z) :- contains(?X,?Y), contains(?Y,?Z) .
% Contained things are inherited from superclasses:
contains(?X,?Z) :- subClassOf(?X,?Y), contains(?Y,?Z) .
% Class hierarchy is reflexive and transitive:
subClassOf(?X,?X) :- subClassOf(?X,?Y) .
subClassOf(?X,?Y) :- subClassOf(?X,?Y), subClassOf(?Y,?Z) .

% Define some derived classes to query for:
alcoholicBeverage(?X) :- drink(?X), contains(?X,"ethanol") .
spicedDrink(?X) :- drink(?X), contains(?X,?Y), subClassOf(?Y,"spice").
```

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:

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

- 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).

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.

Moreover, not 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 if "lime & soda" does not contain alcohol
- Datalog cannot decide if the database contains an even number of drinks

Beyond Datalog: Existential rules

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

Example:

$$\text{alcoholicBeverage}(x) \to \exists v, w.\text{ingredient}(x, v, w), \text{contains}(v, \text{ethanol})$$

(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

Assume we found another database that relates drinks to their ingredients, but now without specific quantity. This data is stored in files `ingredients2.csv` and `garnishes2.csv`.

We add the new data to our drinks knowledge base:

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

(2) Load the previous knowledge base: `@load "drinks/rules.rls" .`

(3) Add the new data sources:

```
@addsource madeWith[2]:  load-csv("drinks/ingredients2.csv")
@addsource garnishedWith[2]:  load-csv("drinks/garnishes2.csv")
```

(4) Use existential rules to incorporate the data into our existing relations:

```
@assert ingredient(?X,?Y,!Z) :- madeWith(?X,?Y).
@assert garnish(?X,?Y,!Z) :- garnishedWith(?X,?Y).
```

The `!` marks existentially quantified variables.

(5) Reason and use queries to see the changed results:

```
@reason .
@query COUNT alcoholicBeverage(?X) .
@query contains(?X,"chili pepper") .
```

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.

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.

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 used the following statements in our hands-on:

$$\text{madeWith}(\text{Mojito}, \text{sugar}) \qquad \text{ingredient}(\text{Mojito}, \text{sugar}, \text{2tsp})$$

$$\text{madeWith}(x, y) \to \exists v.\text{ingredient}(x, y, v)$$

If we replace v with $f(x, y)$, then $\text{ingredient}(\text{Mojito}, \text{sugar}, f(\text{Mojito}, \text{sugar}))$ would be derived by a reasoner. The restricted chase would not derive this.

Negation

Negation is another extremely useful extension of rule languages.

Example:

$$\text{drink}(x) \wedge \neg\text{alcoholicBeverage}(x) \rightarrow \text{nonAlcoholicBeverage}(x)$$

Negation

Negation is another extremely useful extension of rule languages.

Example:

$$\text{drink}(x) \land \neg\text{alcoholicBeverage}(x) \to \text{nonAlcoholicBeverage}(x)$$

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

Negation

Negation is another extremely useful extension of rule languages.

Example:

$$\text{drink}(x) \wedge \neg\text{alcoholicBeverage}(x) \rightarrow \text{nonAlcoholicBeverage}(x)$$

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

$\leadsto$ 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").

Summary

What we learned

- Datalog is a versatile language that appears in many formats and uses
- Two extensions increase its expressivity:
  - Existential quantifiers in heads
  - Negation in bodies (here: stratified only)
- VLog and Rulewerk fast, free tools for this language

Up next: our first concrete use case

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