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, September 4, 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:**

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

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

\[ \text{conjunction of relational atoms} \rightarrow \text{relational atom} \]
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:

\[\text{drink(limeAndSoda)}\]

\[\text{contains(limeAndSoda, limeSyrup)} \quad \text{contains(limeAndSoda, sodaWater)}\]

\[\text{contains(sodaWater, water)} \quad \text{contains(sodaWater, carbonDioxide)}\]

Applying rules yields:

\[\text{from R1: contains(limeAndSoda, water)}\]
\[\text{from R1: contains(limeAndSoda, carbonDioxide)}\]
\[\text{from R2: fizzyDrink(limeAndSoda)}\]
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, Graal, RD-Fox, Llunatic, Vadalog, VLog/Rulewerk, Idots

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
- Decision support
- Data access and management
- Health care data analysis
- Knowledge graph management
- Ontology reasoning
- Data integration
- Integrated AI systems

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.
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

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

could be written as:

\[
fizzyDrink(?x) :- \text{drink}(?x), \text{contains}(?x,"\text{carbon dioxide"}) .
\]
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
\[ \neg \text{@assert} \text{ is not the way to get there} \]

**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.
%% 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 garnish[3] : load-csv("garnishes.csv") .

% Data about what contains what (container, containee)

% General subclass relationships (subclass, superclass):
@source subClassOf[2] : load-csv("subclasses.csv") .
% 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") .
%%% 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:
  \[
  \text{<number of predicate names>} \times \text{<number of constants>} <\text{max. predicate arity>}
  \]

- 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.

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:

\[ \text{alcoholicBeverage}(x) \rightarrow \exists v, w. \text{ingredient}(x, v, w), \text{contains}(v, \text{ethanol}) \]

(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
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

→ 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.

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

Markus Krötzsch, September 4, 2020 Practical Uses of Existential Rules in Knowledge Representation slide 22 of 25
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

$\sim$ 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.

$\sim$ 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 exist.
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.

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:

- madeWith(Mojito, sugar)
- ingredient(Mojito, sugar, 2tsp)
- madeWith($x$, $y$) $\rightarrow \exists v. ingredient(x, y, v)$

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

Negation is another extremely useful extension of rule languages.

Example:

drink(x) \land \neg\text{alcoholicBeverage}(x) \rightarrow \text{nonAlcoholicBeverage}(x)
Negation

Negation is another extremely useful extension of rule languages.

Example:

\[\text{drink}(x) \land \neg\text{alcoholicBeverage}(x) \rightarrow \text{nonAlcoholicBeverage}(x)\]

Mixing negation with recursion can be complicated:

Example:

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

- 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:
\[\text{drink}(x) \land \neg \text{alcoholicBeverage}(x) \rightarrow \text{nonAlcoholicBeverage}(x)\]

Mixing negation with recursion can be complicated:

Example:
\[\neg p(x) \rightarrow q(x)\]
\[\neg q(x) \rightarrow p(x)\]

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

\[\neg\text{VLog}\] 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”).
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