Nemo: Your Friendly and Versatile Rule Reasoning Toolkit

Alex Ivliev , Lukas Gerlach , Simon Meusel , Jakob Steinberg , Markus Krötzsch
Knowledge-Based Systems Group, TU Dresden, Dresden, Germany
{alex.ivliev, lukas.gerlach, markus.kroetzsch}@tu-dresden.de,
{simon.meusel, jakob_maximilian.steinberg}@mailbox.tu-dresden.de

Abstract

We present Nemo, a toolkit for rule-based reasoning and data processing that emphasises robustness and ease of use. Nemo's core is a scalable and efficient main-memory reasoner that supports an expressive extension of Datalog with support for datatypes, existential rules, aggregates, and (stratified) negation. Built around this core is a versatile system of libraries and applications for interfacing with several data formats and programming languages, use as a web application, and IDE integration. In this system description, we present this toolkit and discuss relevant application areas in rule-based knowledge representation, knowledge graph processing, and reasoner prototyping. Our evaluation on a range of tasks from these areas demonstrates Nemo's robust performance in comparison to state-of-the-art rule engines.

1 Introduction

Declarative rule languages are central to knowledge representation and reasoning (KRR), be it as the foundation of logic programming (Körner et al. 2022; Calimeri et al. 2020), as a design principle for ontology languages (ter Horst 2005; Motik et al. 2009; Krötzsch, Rudolph, and Hitzler 2013), or merely as a computational framework for reasoning (Gómez Álvarez, Rudolph, and Strass 2023; Simančík, Kazakov, and Horrocks 2011; Krötzsch 2011). Indeed, few logical paradigms embody such harmony of intuitive meaning, formal semantics, and practical execution. In recent years, rules have therefore become an important approach to declarative computation, not only in KRR but also in many fields concerned with the management, integration, and analysis of data (Aberger et al. 2016; Seo, Guo, and Lam 2015; Aref et al. 2015).

One of the most basic rule languages is Datalog (Abiteboul, Hull, and Vianu 1994), which is also the core of many more complex formalisms. Recent years saw much increased interest in Datalog (Alviano and Pieris 2024) and closely related languages such as existential rules (Benedikt et al. 2017; Mugnier and Thomazo 2014). Accordingly, many rule reasoning systems have been presented, which we can roughly classify in the following types (Ivliev et al. 2023):

1. Answer set programming solvers (Gebser, Kaufmann, and Schaub 2012; Alviano et al. 2017) and logic programming systems such as Prolog (Körner et al. 2022)
2. Knowledge graph and deductive database engines such as RDFox (Nenov et al. 2015), VLog (Urbani, Jacobs, and Krötzsch 2016), Vadalog (Bellomarini, Sallinger, and Gottlob 2018), and Graal (Baget et al. 2015)
3. Specialised data-analytics systems such as Soufflé (Jordan, Scholz, and Subotic 2016), LogicBlox (Aref et al. 2015), SocialLite (Seo, Guo, and Lam 2015), or EmptyHeaded (Aberger et al. 2016)
4. Data management frameworks such as Datomic, Google Logica, and CozoDB

In spite of this apparent abundance, researchers and practitioners may struggle to find a rule engine that suits their needs. First of all, the design space for such systems is very large, and the breadth of applications depends on a wide range of relevant features. For example, tools of type (4) are more similar to rule-based database APIs than to reasoners for knowledge representation. Moreover, many systems described in the literature may not be a viable choice in practice, due to discontinuation of the project, or due to cost and access restrictions of closed-source commercial systems. In the field of logic programming, these problems have been overcome and advanced open-source systems such as Clingo (Gebser et al. 2019) are available to researchers, whereas the situation in other rule system categories is more precarious.[1]

We therefore present *Nemo*, a new rule reasoning toolkit for applications of type (2) and (3), where the computation of logical entailments (or query results) from a variety of inputs is the main reasoning task. Nemo's rule language is an extension of Datalog with various datatypes, negation, aggregates (both stratified), and many datatype-specific functions and operators known from query languages like SPARQL. At the same time, Nemo is an existential rule reasoner with a fast implementation of the *restricted* (or *standard*) *chase* algorithm. Supported formats and types for data values conform to the RDF standard (Cyganiak, Wood, and Lanthaler 2014).

Nemo aims to combine robust performance with flexible and convenient use. Thanks to its modular architecture, we can provide command-line clients for all major platforms, a public Web application with built-in rule editor, an IDE plugin for rule editing in VSCode, and APIs for integration

[1] Among the mentioned systems of types (2) and (3), the only open-source tool with a release in the past twelve months is Soufflé.

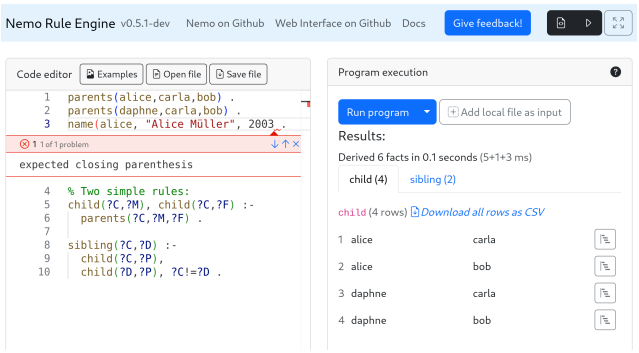

Figure 1: Rule reasoning in the browser with Nemo Web

in several programming languages. Nemo's optimised main-memory data storage can handle hundreds of millions of facts on modern laptops, and several billions on mid-range servers. All components of Nemo are free and open source, and their development repositories public. Most of Nemo is written in Rust, a language that emphasizes type and memory safety, and code quality is an explicit concern.

In this system description, we present Nemo's overall architecture (Section 2) and rule language (Section 3). We then highlight three important application areas in KRR, and show how Nemo can be used in each of them (Section 4). Our performance evaluation and feature analyses position Nemo in the wider space of modern rule engines (Section 5). We close with an outlook on Nemo's next steps.

2 System Overview

Nemo comprises several applications and programming libraries. This section gives an overview of the main components, related functionality, and implementation aspects.[2]

Nemo's main functionality is the materialisation of logical consequences from input data using a given set of rules, called *program* in this context. Data in Nemo is relational, based on predicate names of fixed arities, over a domain that includes both abstract identifiers ("logical constants", named nulls) and concrete data values (numbers, strings, dates, etc.). Graph data, e.g., in RDF format (Cyganiak, Wood, and Lanthaler 2014), is an important special case. As its rule language, Nemo supports an extension of Datalog with many features, discussed in detail in Section 3. In the simplest case, the program can be a set of Datalog rules, with the data given as logic facts directly within the program. Elaborate cases may realise complex data transformations or reasoning tasks, as shown in Section 4.

Nemo also supports *tracing*, i.e., it can provide a justification for its inferences by producing a proof tree that shows the concrete rule applications and input facts used to obtain a conclusion. In contrast to exhaustive *provenance* computation (Elhalawati, Krötzsch, and Mennicke 2022), tracing is very efficient and adds no cost to reasoning.

Two distinct applications support the evaluation of programs: the Nemo command-line client *nmo*[3] and a browser-based application *Nemo Web*.[4] Both applications are run entirely on the user's computer: the command-line client as a native binary, and Nemo Web as a *WebAssembly* executable that runs in browsers without prior installation.

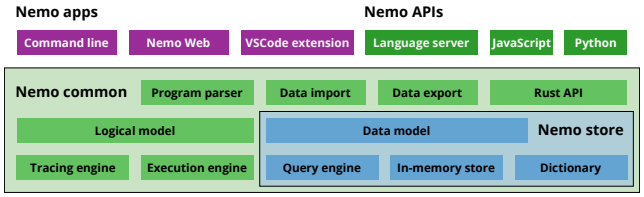

Figure 2: Main components of Nemo

Ease of use is a major design goal for Nemo, setting it apart from typical research prototypes. A key concern is to support programmers in using our custom Datalog syntax and in modelling computation with logical rules. Nemo Web therefore includes a powerful rule editor component with syntax highlighting and auto-completion (see Figure 1). Advanced editing support is provided by the *Nemo VSCode extension*,[5] which augments VSCode (and compatible IDEs) with advanced features for rule development, such as highlighting, completion, error reporting, syntax-aware searching and renaming, and refactoring. All Nemo applications provide convenient parameters and meaningful error messages.

Architecture The main components of Nemo are illustrated in Figure 2. Most parts are written in Rust, with the exception of Nemo Web, the VSCode extension (both TypeScript), and the JavaScript and Python APIs. Nemo consists of several libraries (Rust *crates*), most important of which is *Nemo common* which bundles all major functions, and *Nemo store*, the main memory storage engine. For the Web application and JavaScript API, Rust is compiled into system-independent WebAssembly, which can be run in any modern browser. The VSCode extension builds on an API that uses the open *language server protocol* (LSP 2024), which is also a basis for extensions in many other IDEs.

Rule Parsing and Representation Nemo's *program parser* creates an abstract syntax tree (AST), which is the core of Nemo's advanced rule language support. We use *Parser Expression Grammars* and follow Medeiros and Mascarenhas (2018) for syntax error recovery. In a second step, the AST is converted into a *logical model*, which is a more traditional structural model that abstracts from most syntactic details. The logical reasoning community has often neglected the first stage, and established tools such as the OWL API (Horridge, Bechhofer, and Noppens 2007) only offer a structural view that is not usable for advanced editing features. Nemo's logical model is closely connected to the *data model* in *Nemo store*, which defines the structure and supported datatypes of domain elements.

[2] A detailed documentation is available online: https://knowsys.github.io/nemo-doc

[3] We avoid a naming conflict with the Linux file manager *nemo*. Installation instructions and pre-compiled binaries are in our main repository https://github.com/knowsys/nemo.

[4] Accessible at https://tools.iccl.inf.tu-dresden.de/nemo/.

[5] See https://github.com/knowsys/nemo-vscode-extension

Data Import and Export In addition to logical facts given in rules files, Nemo can also import (possibly large) datasets that are encoded in CSV (and other forms of *delimiter-separated values*) or RDF. For RDF, triples (NTriples, Turtle, RDF/XML) and quads (TRiG, NQuads) are supported. Data can be loaded from local files or online URLs. All formats are also available for export, and (on the command line) results can also be printed to the console. Import and export can be controlled by various parameters, and support optional GZip (de)compression.

Main-memory Data Store Data in *Nemo store* is organised in tables, using data structures that are designed to be both fast and compact. Inspired by VLog, we use hierarchically sorted, column-based tables and compress columns using run-length encoding with increments (Urbani, Jacobs, and Krötzsch 2016). Unlike VLog, however, our columns contain domain elements of several types: fixed-size values (e.g., 32bit floats or 64bit signed ints) are stored directly, while variable-sized data (e.g., strings and IRIs) is first mapped to integer ids through a *dictionary*. Columns are first sorted by type (e.g., all integers before all floats), so that access operations can process values of the same type in fast loops without checking the type of each value.

Nemo store does not support rule reasoning, but can be used to achieve the effect of a single rule application by materialising the results of a query through its *query engine*. For this task, we mostly access tables as *trie structures* (Fredkin 1960), and we evaluate conjunctions using *leapfrog trie-join*, a popular worst-case optimal multiway join algorithm with asymptotic advantages over binary join plans (Veldhuizen 2014). Operations like union or difference use analogous trie-based algorithms. For operations like projection and reordering, which cannot be implemented efficiently with trie iterators, Nemo uses row-based temporary tables.

Program Execution The *execution engine* materialises inferences by semi-naive bottom-up evaluation (Abiteboul, Hull, and Vianu 1994), where existential quantifiers in conclusions are handled with a 1-parallel restricted chase algorithm (Benedikt et al. 2017). Like VLog, we avoid costly table updates by storing the fresh results of each rule application in separate *delta tables* (Urbani, Jacobs, and Krötzsch 2016). This is useful for semi-naive evaluation and helps the *tracing engine* to re-construct traces. The cost of combining many delta tables is mitigated by caching such unions.

Rule dependencies are analysed by overestimating dataflow during materialisation (González et al. 2022), and this information is used to stratify non-monotonic features (negation, aggregates), and to heuristically select rule application orders with fewer rule applications overall.

The execution engine also prepares query plans for Nemo store to execute. In our setting with its streaming multi-way operations, the order of variables (columns) largely determines the query plan. We heuristically aim for variable orders which avoid inefficiencies (such as cross products) and reorderings (e.g., for aggregation operations), and which produce results in an order that is useful for subsequent rule applications.

3 Language Features

Nemo builds upon the declarative rule language Datalog (Abiteboul, Hull, and Vianu 1994), and extends it with many features of modern query and rule languages. This section gives an overview of the most important such extensions: datatypes and related functions, existential rules, negation, and aggregation.

A Datalog rule is a logical formula of the form $H \leftarrow B$, where the *head* H and *body* B are conjunctions of first-order atoms, and all variables are implicitly universally quantified. Nemo combines traditional logic programming syntax with notation from the query language SPARQL (Harris and Seaborne 21 March 2013), which encodes identifiers in more robust ways (supporting special characters and data types), as illustrated next:

```
1 parents(alice,carla,bob) .
2 parents(<https://example.org/daphne>,carla,bob) .
3 nameAndYearOfBirth(alice,"Alice Müller",2003) .
4 % Two simple rules:
5 child(?C,?M), child(?C,?F) :- parents(?C,?M,?F) .
6 sibling(?C,?D) :- child(?C,?P),
7                   child(?D,?P), ?C != ?D .
```

Lines 1–3 provide three facts, illustrating some basic syntax for various kinds of data. In general, all syntactic forms available in RDF and SPARQL can be used, e.g., `"論理学"@ja` and `"3.1"^^<http://www.w3.org/2001/XMLSchema#float>`. To load larger datasets from files, one may use statements such as `@import parents :- csv{resource = "my-data.csv"} .` Lines 5–7 show two simple rules, where `?` marks variables (as in SPARQL) and `!=` encodes inequality. Nemo supports non-monotonic negation using the following syntax:

```
8 onlyChild(?C) :- child(?C,_), ~sibling(?C,_) .
```

As usual in logic programming, `_` denotes body variables that occur only once, i.e., both `_` in Line 8 denote distinct variables. Moreover, Nemo considers variables that occur only in negated body atoms to be existentially quantified beneath the negation. Hence, `~sibling(?C,_)` requires that *all* `sibling` facts do not match this pattern, whereas `child(?C,_)` requires *some* matching fact. A similar convention is used in the answer set programming, e.g., in current versions of Clingo (Gebser, Kaufmann, and Schaub 2012).

Aggregate functions, which compute results from sets or multi-sets of tuples, have a long history in logic programming (Ross and Sagiv 1997; Alviano et al. 2011; Zaniolo et al. 2017). Nemo supports common aggregates such as `#count`, `#sum`, or `#max`, as shown next:

```
9 childCount(?P,#count(?C)) :- child(?C,?P) .
```

Rule 9 counts, for each value of `?P`, the number of distinct values of `?C` that satisfy the rule body. In general, we distinguish three kinds of variables: *aggregate variables* occur in aggregate functions (in the head), *group-by variables* are the head variables that are not aggregate variables; and *body-only variables* that do not appear in the head. Aggregation in steps: (1) find all rule matches, (2) project away bindings of

body-only variables and remove duplicates that agree on all remaining variables, (3) group the set of projected matches by distinct values of group-by variables, and (4) apply the aggregation function on each group. Duplicate elimination in (2) means that we always aggregate over sets, corresponding to the keyword `DISTINCT` in many query languages. Users control the semantics through the use of variables in the head:

```
10 sum1(?A,?B,#sum(?N)) :- p(?A,?B,?N) .
11 sum2(?A,#sum(?N,?B)) :- p(?A,?B,?N) .
12 sum3(?A,   #sum(?N)) :- p(?A,?B,?N) .
```

Rule 10 sums up the numbers N for each pair of As and Bs; rule 11 sums up the Ns from all pairs of Ns and Bs for each A (where some pairs may have the same N value); and rule 12 sums up the distinct Ns for each A.

Nemo supports *existential rules*, which allow existentially quantified variables in rule heads. We discuss this feature in more detail in Section 4.1. Syntactically, existentially quantified variables are denoted by `!`:

```
13 child(?C,!P), child(!P,?G) :- grandchild(?C,?G) .
```

For each match of the body of rule 13, Nemo creates a fresh value for `!P`, provided that the head is not satisfied for any existing value yet (*restricted chase*). Fresh values are represented in Nemo by *named nulls*, which we treat as a separate type of domain element distinct from other elements. Named nulls are identified with RDF *blank nodes* in data import and export, and denoted accordingly (using notation like `_:42`).

Finally, Nemo also supports many different datatypes – including integer, string, language-tagged string, single and double precision float, and boolean – and related functions. The latter include many functions known from SPARQL `FILTER` expressions, such as `SQRT` (square root of a number) or `STRLEN` (length of a string), as well as standard arithmetic operators. Functions can be nested arbitrarily and may occur anywhere in a rule. Moreover, comparison operators like `<` or `!=`, can be used as atoms (see line 7), provided that variable bindings are sufficiently determined by other parts of the body (in particular, one cannot define body matches based on systems of inequalities).

Nemo is dynamically typed and allows any type of data to occur in any position, without requiring a fixed schema.[6] Functions such as `STRLEN` are not defined for all types of inputs, and rules carry the implicit condition that they only apply to variable bindings for which all functions are defined.

The semantics of Nemo follows from the declarative conditions that define how single rules are applied. Doing so until a (possibly infinite) least fixed-point is reached defines an (operational) semantics. For many fragments of our language, this semantics agrees with a well-understood model theory: least Herbrand models for Datalog, perfect models for Datalog with stratified negation, universal models for existential rules. However, existential rules are not always compatible with other Datalog extensions: even stratified negation can lead to situations where entailments depend on seemingly arbitrary choices of the order of rule applications (Krötzsch 2020). Only few cases have yet been found that to prevent such problems (Ellmauthaler, Krötzsch, and Mennicke 2022), so Nemo does not restrict to these limited cases, and rather warns users about the subtleties of existential rules in the documentation.

[6] This is the standard choice in knowledge representation, also seen in OWL, RDF, SPARQL, Prolog, and ASP.

4 Application Areas

With its many language features, Nemo can support use cases in various areas of knowledge representation and reasoning. In this section, we highlight existential rule reasoning, processing of (large) knowledge graphs, and the use of rules for implementing prototype reasoners for other logics. Our discussion also illustrates how various features of Nemo are relevant in these cases.

4.1 Existential Rule Reasoning

As introduced in Section 3, Nemo can represent and reason with rules that feature existential variables.

```
14 p(?Y,!Z), p(!Z,?Y) :- p(?X,?Y) .
```

Running the restricted (a.k.a. standard) chase on such *existential rules* (a.k.a. *tuple-generating dependencies (tdgs)*), yields a *universal model* that allows for conjunctive query answering (Deutsch, Nash, and Remmel 2008). Reasoners like Nemo typically implement the chase, but reasoning over (fragments of) existential rules can also be achieved by query rewriting and combined methods, as supported in Vadalog (Bellomarini, Sallinger, and Gottlob 2018) and Graal (Baget et al. 2015).

In general, query answering over existential rules is undecidable (Beeri and Vardi 1981), so no approach covers all cases. For the chase, undecidability is tied to the problem of non-termination, which may occur due to the creation of new values. Detecting this situation is also undecidable (Gogacz and Marcinkowski 2014; Grahne and Onet 2018), and much research has focused on identifying language fragments and methods to decide (non)termination (Gogacz, Marcinkowski, and Pieris 2020; Fagin et al. 2005; Cuenca Grau et al. 2012; Carral, Dragoste, and Krötzsch 2017; Gerlach and Carral 2023b; Gerlach and Carral 2023a; Hanisch and Krötzsch 2024). Nemo does not currently implement any such check, and relies on users for ensuring termination. New, generalised criteria would be necessary to do so, since Nemo has further ways of creating new values using datatype functions. Even without such functions, recent results showed that the fragment of existential rules for which the restricted chase terminates is not even semi-decidable, but is powerful enough to express all homomorphism-closed decidable queries (Bourgaux et al. 2021). These results apply to Nemo as well.

Different variants of the chase algorithm terminate on different rule sets. The *skolem chase*, for example, terminates on strictly less rule sets compared to the restricted chase. In rule reasoners that do not allow for existential variables, but function symbols, such as Soufflé (Jordan, Scholz, and Subotic 2016) or Gringo (Gebser et al. 2019), one can replace existential variables by skolemisation, effectively executing

a skolem chase. For example, rule 14 can be skolemised as follows.

```
15  p(?Y,f(?Y)), p(f(?Y),?Y) :- p(?X,?Y).
```

However, the skolem chase does not always terminate on rule 14. Consider for example the fact `p(a,b)`. The skolem chase introduces infinitely many facts `p(b,f(b))`, `p(f(b),b)`, `p(f(b),f(f(b)))`, `p(f(f(b)),f(b))`, ... For the restricted chase, this is impossible as the facts `p(a,b)`, `p(b,n)`, `p(n,b)` already satisfy rule 14 (where n is a fresh null). Therefore, Nemo has a termination advantage over reasoners that only allow running (or simulating) the skolem chase. This is not exclusive for Nemo though as other reasoners, e.g., VLog (Carral et al. 2019a), also implement the restricted chase.

While the chase is conceptually simple, large datasets demand efficient and sophisticated implementations. Practically motivated research in this direction optimises, e.g., rule application orders or join algorithms (González et al. 2022; Veldhuizen 2014). For empirical evaluation, there exists a set of common benchmarks (Benedikt et al. 2017)[7]. As Nemo natively supports existential rules, it is easy to implement the respective programs and compare to existing reasoners. We give a detailed evaluation in Section 5 together with more details on existing reasoning systems.

4.2 Knowledge Graph Processing

Reasoning with (large) knowledge graphs is an important task in KRR, semantic web, and data management, which also drives development of rule reasoning systems (Nenov et al. 2015; Aberger et al. 2017; Seo, Guo, and Lam 2015). Relevant use cases include free and open knowledge graphs such as Wikidata (Vrandečić and Krötzsch 2014), DBpedia (Bizer et al. 2009), YAGO (Suchanek et al. 2024), or DBLP,[8] but also graphs-based data management and integration projects in companies. Main challenges for reasoners are full support for data exchange standards such as RDF (Cyganiak, Wood, and Lanthaler 2014), and handling very large amounts of data, e.g., billions of facts in the case of Wikidata.

Since RDF entities are first-class citizens in Nemo, it is well-suited to interact with modern knowledge graphs, for which RDF is the most common exchange format today. Thanks to the efficient underlying implementations, Nemo can indeed handle knowledge bases with hundreds of millions or billions of facts, provided that sufficient main memory is available (see Section 5).

As a concrete illustration of a practically relevant processing task on knowledge graphs, we consider the creation process of the recent YAGO 4.5 knowledge base (Suchanek et al. 2024). YAGO is constructed as a curated subset of Wikidata with improved ontological models and additional links to external ontological vocabularies. The ontology modelling follows several design principles, including that of avoiding redundancy in data. A particular case are inverse properties, which may store the same information in two facts. For example, persons may have a nationality whereas countries may have citizens. To select which of two inverse properties to keep, YAGO proposes to prefer the property that, on average, has fewer values (in our example, nationality rather than citizen). While YAGO 4.5 was created by performing this analysis in ad hoc scripts, it is also an excellent use case for aggregates and numeric data in Nemo, as shown in the following set of rules, where `predToKeep` denotes preferred properties.

```
16  @prefix wikibase:
17    <http://wikiba.se/ontology#>.
18  @prefix wdt:
19    <http://www.wikidata.org/prop/direct/>.
20
21  @import wikidata :- {resource="wikidata.nt.gz"}.
22
23  inversePropEntities(?s,?o) :-
24    wikidata(?s,wdt:P1696,?o),
25    COMPARE(STR(?s),STR(?o)) = -1.
26  inverse(?p1,?p2) :-
27    inversePropEntities(?e1,?e2),
28    wikidata(?e1,wikibase:directClaim,?p1),
29    wikidata(?e2,wikibase:directClaim,?p2).
30  pred(?s) :- inverse(?s,_).
31  pred(?o) :- inverse(_,?o).
32
33  objCountPerPredAndSource(?p,?s,#count(?o)) :-
34    pred(?p), wikidata(?s,?p,?o).
35  totalSubjCountPerPred(?p,#count(?s)) :-
36    objCountPerPredAndSource(?p,?s,_).
37
38  avgObjCountPerPredicate(?p,
39    DOUBLE(#sum(?count,?s)) / ?totalSubj) :-
40      objCountPerPredAndSource(?p,?s,?count),
41      totalSubjCountPerPred(?p,?totalSubj).
42
43  rightSmaller(?right) :-
44    inverse(?left,?right),
45    avgObjCountPerPredicate(?left,?countLeft),
46    avgObjCountPerPredicate(?right,?countRight),
47    ?countRight < ?countLeft.
48
49  predToKeep(?right) :- rightSmaller(?right).
50  predToKeep(?left) :- inverse(?left,?right),
51    ~rightSmaller(?right).
```

In line 21, we load all Wikidata triples into the ternary `wikidata` predicate. Line 23 extracts property names that are inverses of each other (encoded in Wikidata through the property `wdt:P1696`). Here, we ignore properties that are the inverses of themselves and we only pick the pairs of properties where the left hand side is lexicographically smaller to avoid introducing both directions of symmetries. This, so far, only marks *property entities* in Wikidata and not the corresponding *RDF properties*, which we in turn look up in rule 26. For a convenient way of accessing all relevant predicates, we introduce `pred` in lines 30 and 31. Line 33 counts the number of objects per subject and relevant predicate to be able to compute the average number of objects per subject in rule 38. In line 43, for every pair of inverse properties, we mark those on the right side of the relation where the average object per

[7] https://github.com/knowsys/nemo-examples/tree/main/chasebench

[8] https://dblp.org/rdf/

subject count is smaller. If this condition is fulfilled, we want to keep the respective property, which we indicate through rule 49. Otherwise, we keep the left property using rule 50. Then, **predToKeep** contains exactly the properties desired by the YAGO 4.5 analysis.[9]

4.3 Reasoner Prototyping

Thanks to the various language features, particularly support for existentials, in Nemo, we can have (prototypical) implementations of reasoning algorithms for many KR formalisms, e.g. for OWL (OWL Working Group 2009) (description logic) ontologies, DatalogMTL (Brandt et al. 2018), standpoint logic (Gómez Álvarez, Rudolph, and Strass 2023), Datalog(S) (Carral et al. 2019b), and stream reasoning (Urbani, Krötzsch, and Eiter 2022).

$\mathcal{EL}$-Reasoning $\mathcal{EL}$ (Baader, Brandt, and Lutz 2005) is a rather simple class of description logic ontologies that allows for efficient reasoning implementations (Kazakov, Krötzsch, and Simančík 2013) based on only a few "rules". To be precise, we consider rules for $\mathcal{EL}^+_\bot$. In fact, these rules can be represented in Datalog. Given an OWL ontology in RDF (NTriples) format, the necessary preprocessing can also be done directly within Nemo.[10] The preprocessing yields the following tables: **isMainClass**, **isSubClass**, **conj**, **exists**, **subClassOf**, **subPropChain**, **subProp**. We can then exhaustively compute all (implicit) subclass relations within the ontology and identify when two main classes are in a subclass relation by utilising the following Nemo rules.

```
init(?C) :- isMainClass(?C) .
init(?C) :- ex(?E,?R,?C) .

subClassOf(?C,?C) :- init(?C) .
subClassOf(?C,owl:Thing) :- isMainClass(?C) .
subClassOf(?C,?D1), subClassOf(?C,?D2) :-
    subClassOf(?C,?Y), conj(?Y,?D1,?D2) .
subClassOf(?C,?I) :- subClassOf(?C,?D1),
    subClassOf(?C,?D2),
    conj(?I,?D1,?D2), isSubClass(?I) .

ex(?E,?R,?C) :- subClassOf(?E,?Y),
    exists(?Y,?R,?C) .
subClassOf(?E,?Y) :- ex(?E,?R,?C),
    subClassOf(?C,?D), subProp(?R,?S),
    exists(?Y, ?S,?D), isSubClass(?Y) .
ex(?E,?S,?D) :- ex(?E,?R1,?C), ex(?C,?R2,?D),
    subProp(?R1,?S1), subProp(?R2,?S2),
    subPropChain(?S1,?S2,?S) .

subClassOf(?C,?E) :- subClassOf(?C,?D),
    subClassOf(?D,?E) .
subClassOf(?E, owl:Nothing) :-
    ex(?E,?R,?C), subClassOf(?C,owl:Nothing) .
mainSubClassOf(?A,?B) :- subClassOf(?A,?B),
    isMainClass(?A), isMainClass(?B) .
```

[9]The Wikidata/Yago 4.5 example and the corresponding outputs are available at https://github.com/knowsys/nemo-examples/tree/main/examples/wikidata-yago-like-inverse-property-cleanup

[10]https://github.com/knowsys/nemo-examples/blob/main/examples/owl-el/from-owl-rdf/owl-rdf-preprocessing.rls

The intuition of the predicates from the program is as follows: **init**(?C) contains all relevant OWL classes ?C; **subClassOf**(?X,?Y) expresses that ?X is a subclass of ?Y; **ex**(?X,?R,?Y) expresses that every element of class ?X is connected to an element of ?Y via an ?R property; and **mainSubClassOf**(?X,?Y) expresses a subclass relation between two main classes ?X and ?Y.

DatalogMTL DatalogMTL (Brandt et al. 2018) extends Datalog with metric temporal logic operators that indicate if a fact holds sometimes/always in the future/past, potentially restricted to the scope of a given interval. For DatalogMTL programs with closed intervals, we give a translation into a Nemo program by explicitely annotating time points at predicates. We illustrate this idea with an example from the MeTeoR DatalogMTL-reasoner (Wang et al. 2022).[11]

Example 1. *This example checks if two sensors indicate high temperatures for a long enough interval or with a high enough frequency.* [12]

$$Overheat(x) \leftarrow \boxminus_{[0,10]} HighTemp(x) \quad (1)$$

$$Overheat(x) \leftarrow \boxminus_{[0,20]} \diamondminus_{[0,5.5]} HighTemp(x) \quad (2)$$

$$Alert \leftarrow Overheat(x) \wedge Overheat(y) \wedge NextTo(x,y) \quad (3)$$

Intuitively, a sensor reports Overheat if it reports HighTemp constantly for an interval of 10 minutes ($\boxminus_{[0,10]}$) or at least once every 5.5 minutes ($\diamondminus_{[0,5.5]}$) within the last 20 minutes ($\boxminus_{[0,20]}$). An Alert is raised if two sensors NextTo each other report Overheat.

The following Nemo program is a rule-by-rule translation. Note that we split the second rule into two to disentangle the nested box and diamond operator.

```
Overheat(?Sensor,?Start + 10.0,?End) :-
    HighTemp(?Sensor,?Start,?End),
    ?End - ?Start >= 10.0.

HighTempSometimes(?Sensor,?Start,?End + 5.5) :-
    HighTemp(?Sensor,?Start,?End).
Overheat(?Sensor,?Start + 20.0,?End) :-
    HighTempSometimes(?Sensor,?Start,?End),
    ?End - ?Start >= 20.0.

Alert(?SensorA,MAX(?sA,?sB),MIN(?eA,?eB)),
Alert(?SensorB,MAX(?sA,?sB),MIN(?eA,?eB)) :-
    Overheat(?SensorA,?sA,?eA),
    Overheat(?SensorB,?sB,?eB),
    SameLocation(?sA,?sB),
    ?sA <= ?eB, ?sB <= ?eA.
```

Depending on the input facts, it can be vital to have auxiliary rules for merging intervals, e.g.

```
HighTempSometimes(?Sensor,?sA,?eB) :-
  HighTempSometimes(?Sensor,?sA,?eA),
  HighTempSometimes(?Sensor,?sB,?eB),
?sA <= ?eB, ?sB <= ?eA,
?sA <= ?sB, ?eA <= ?eB.
```

[11]https://github.com/wdimmy/MeTeoR

[12]https://github.com/knowsys/nemo-examples/blob/main/examples/datalogMtlSensor/datalogMtlSensor.rls

Otherwise, we may not detect Overheat for a sensor like:

```
99  HighTemp("sensor2",3.5,3.5).
100 HighTemp("sensor2",5.1,5.1).
101 HighTemp("sensor2",10.0,10.0).
102 HighTemp("sensor2",14.7,14.7).
103 HighTemp("sensor2",20.0,20.0).
```

The translation has some limitations. Currently open intervals are not supported but could potentially be modelled with additional rules. Also, for programs with periodic solutions, Nemo might run into termination issues. Non-recursive programs with closed intervals are fully supported.

Other Translations into Existential Rules For many other KR formalisms, translations into existential rules have already been studied. We can directly use these to run the respective formalisms within Nemo. No (further) restrictions to fragments of the respective formalisms are necessary. This subsection lists a few such ideas from different KR areas.

Stream reasoning is a temporal reasoning setting that receives its name from supporting streams of input data. For a common stream reasoning framework, called LARS (Beck, Dao-Tran, and Eiter 2018), recent work shows a translation into existential rules (Urbani, Krötzsch, and Eiter 2022) that preserves the semantics and can directly be used in Nemo.

Standpoint logic is a multi-modal framework that allows reasoning with different interprations of the same underlying ontology. It is in itself applicable to many other KR formalisms. For example, it allows the creation of the standpoint description logic *standpoint* $\mathcal{EL}$ (Gómez Álvarez, Rudolph, and Strass 2023). Similar to the $\mathcal{EL}$ reasoning idea discussed earlier in this section, we can leverage a translation into existential rules to support standpoint $\mathcal{EL}$ reasoning within Nemo.

Datalog(S) extends plain Datalog with a datatype for sets (Carral et al. 2019b). The introducing paper also describes a translation into existential rules. Thereby, a set datatype could be augmented in Nemo as well. In the future, we still plan to have a native set datatype for efficiency.

5 Evaluation

In this section, we compare Nemo to several state-of-the-art systems, and investigate the feasibility of using Nemo on very large knowledge graphs. The goal of our comparative evaluation is to position Nemo in relation to other highly advanced systems, with a particular focus on features that Nemo is aimed at. In particular, the evaluation should not be misunderstood as an absolute ranking of systems: each system has additional strengths beyond those considered here, which make it superior for corresponding use cases.

To select a representative set of systems to compare to, we focus on actively developed, mature systems from a range of application areas. We have arrived at the following list:

Soufflé Originally developed for rule-based program analysis (Jordan, Scholz, and Subotic 2016), Soufflé is one of most active and advanced open-source Datalog reasoners today, with many additional features. It is the only tool in our list that supports a special *compiled* mode, where rule sets are first transformed into C programs, which that are then compiled to obtain an optimised reasoner.

Gringo Developed as the grounder for the ASP tool *clingo* (Gebser et al. 2019), this system is in fact a high-performing Datalog engine in its own right. It also supports a variety of language features from ASP.

RDFox Evolved from an academic open-source system (Nenov et al. 2015), RDFox has been developed into a mature commercial product in recent years. It has a strong focus on the RDF data model and knowledge graph processing. RDFox is not open-source software.

VLog Designed as a plain Datalog engine and existential rule reasoner, VLog also provides basic RDF compatibility and can process large knowledge graphs (Carral et al. 2019a). Some of the basic implementation approaches in Nemo were based on our earlier experience with VLog, making it especially interesting for comparing performance.

5.1 Feature Comparison

We qualitatively compare the chosen systems with respect to their overall features, and have compiled the results in Table 1. The information in this table is taken from the online documentation of the individual tools. Some common aspects were not mentioned, e.g., all systems are main-memory based and rely on materialisation (bottom-up reasoning) as their main approach.

Most features listed in Table 1 should be self-explaining. All systems except VLog have had recent releases, and all provide a range of APIs and user interfaces. Soufflé mentions a Web interface on its web site, but the server was unreachable at the time of this writing. Besides Nemo, also Clingo and Soufflé offer VS Code extensions. Nemo, RDFox, and Soufflé all have tracing capabilities that return single proof trees.

We distinguish basic types for simple data literals from complex types, which represent structured data objects. We omit some details, e.g., the fact that some tools have several integer types. The row *RDF types* states if a tool can handle the rich set of types in the RDF standard, syntactically and semantically, VLog is special here in that it can process RDF syntax faithfully, but internally treats all values as strings. The *type model* states if a tool accepts arbitrary data in all contexts (dynamic) or requires a fixed schema to be given with the rules (static).

All tools support negation, and all except VLog support some form of aggregation. Both features do typically require stratification, unless clingo is used for full ASP reasoning. RDFox has by far the largest list of supported aggregates, not fully listed in the table. Gringo supports a special `sum+` aggregate that computes a monotonic sum by ignoring negative values, which is useful with recursive, non-stratified aggregates (which the other tools do not allow). All tools can simulate existential rules, sometimes with effort on the user side, but only Nemo and VLog seem to support the standard (restricted) chase. Nemo and RDFox feature the largest sets of built-in functions, whereas Gringo and Soufflé provide extension points for defining own functions by custom implementations. All tools except Gringo can import data from

	Nemo	Gringo/Clingo	RDFox	Soufflé	VLog
Latest version	0.5.1 (Jun 2024)	5.7.1 (Feb 2024)	7.1b (Jul 2024)	2.4.1 (Nov 2023)	1.3.6 (Dec 2022)
License	Apache2.0/MIT	MIT	proprietary	UPL 1.0	Apache 2.0
Language	Rust	C++	C++	C++	C++
Further APIs	Python, JavaScript	Rust, Python, Java, Haskell, Prolog, JavaScript	Java, HTTP, SPARQL	Java, Python	Java
UIs	command-line, Web (WASM)	command-line, Web (WASM)	command-line, shell, REST	command-line, Web (unavailable)	command-line
IDEs	VS Code	VS Code	—	VS Code	—
Rule processing	interpreted	interpreted	interpreted	interpreted, compiled	interpreted
Explanation	proof tree	—	proof tree	proof tree	—
Basic types	int, str, float, bool	int, str	int, str, float, bool	int, str, float, bool	str
Complex types	—	function terms	dates/times	records, function terms[a]	—
RDF types	✓	—	✓	—	(✓) only in syntax
Data model	relational	relational	triple/quad	relational	relational
Type model	dynamic	dynamic	dynamic	static	dynamic
Negation	stratified	arbitrary	stratified	stratified	stratified
Aggregation	stratified	arbitrary	stratified	stratified	—
Aggregates	sum, count, min, max	sum(+), count, min, max	sum, count, min, max, avg, sample, . . .	sum, count, min, max	—
Existentials	✓ (standard)	— (skolem[b])	— (skolem[c])	— (skolem[b])	✓ (standard, skolem)
Functions	arithmetic, string, SPARQL	arithmetic, user-defined	arithmetic, string, SPARQL	arithmetic, string, user-defined	—
Imports	DSV, RDF (all)	—	DSV, RDF (all)	DSV, sqlite3	CSV, N-triples
Exports	DSV, RDF (all)	facts, smodels	facts, RDF (all), SPARQL	DSV, sqlite3	CSV
IO features	gzip, URL import	—	DBMS sources	gzip	gzip, SPARQL import

[a]Function terms can be emulated in Soufflé with user-declared *Algebraic Data Types*.
[b]Users can simulate a skolem chase by manually creating skolem function terms.
[c]The current RDFox manual does not mention existentials, but the function `SKOLEM` could be used to mint skolem terms manually.

Table 1: Feature Comparison of Several Related Systems

files in other formats, and RDFox further provides connectors to database backends. In these rows, *DSV* denotes delimiter-separated values (including CSV and TSV), whereas *RDF (all)* stands for all RDF quad and triple syntaxes.

In summary, we find that all analysed systems include many application-oriented features, that support their claimed maturity and illustrate their different application focus. VLog is the most limited, but we note that its Java API *Rulewerk* (formerly *Vlog4j*, (Carral et al. 2019a)) adds further features not mentioned here, e.g., reading all RDF formats.

5.2 Performance Evaluation

Next, we evaluate the overall reasoning performance of Nemo on materialisation tasks for Datalog and existential rules. Our measurements are performed on a regular notebook (Dell XPS 13; Ubuntu Linux 22.04; Intel i7-1165G7@2.80GHz; 16GB RAM; 512 GB SSD). For this evaluation, we consider the command-line clients of all open-source systems discussed in Section 5.1, and additionally Nemo Web (Firefox v125.0). For Soufflé, we evaluate the rule interpreter (i) and the compiled mode (c). We do not evaluate RDFox here, since our request for an academic license has not been granted at the time when this research was conducted.[13]

For benchmarking plain Datalog, we consider the OWL reasoning implementation of Section 4.3 for medical ontologies Galen (EL version by Kazakov, Krötzsch, and Simančík (2013)) and a version of SNOMED CT (a large proprietary ontology). The OWL EL reasoner is a versatile and challenging benchmark for plain Datalog, as it includes strong recursive dependencies and longer rule bodies.

For existential rules, we consider several tasks from the chase benchmark of Benedikt et al. (2017). Both Doctors-

[13]RDFox is a product of Oxford Semantic Technologies, which has been acquired by Samsung during the time of this writing (https://www.oxfordsemantic.tech/blog/samsung-electronics-announces-acquisition-of-oxford-semantic-technologies-uk-based-knowledge-graph-startup). This may explain the possibly temporary disruption in the usual licensing process.

Benchmark	Rules	Input	Derivations	Nemo	Nemo Web	Gringo	Soufflé (i)	Soufflé (c)	VLog
GALEN	12	143,480	1,881,946	**5.4**	8.8	22.0	13.7	10.4	42.7
SNOMED CT	12	1,554,340	24,428,006	**76.4**	129.6	95.4	157.5	114.9	*oom*
DOCTORS-1M	5	9,515,900	792,500	3.3	5.8	7.1	2.4	**2.0**	2.9
ONT-256	529	2,146,490	5,673,985	**14.2**	29.2	23.9	19.2	14.9	22.2
LUBM-01K	136	133,573,854	186,742,694	167.6	*oom*	*oom*	170.4	**145.7**	188.4
DEEP-100	1100	1,000	20,262	3.1	5.4	**0.3**	27.6	*timeout*	23.2
DEEP-200	1200	1,000	982,501	**8.0**	13.9	*timeout*	*timeout*	*timeout*	*timeout*

Table 2: Number of rules, input facts, derivations, and materialisation times in seconds (*timeout*: 30min, *oom*: out of memory)

1M and ONT-256 are benchmarks for data integration; LUBM-01K is a widely used synthetic benchmark for semantic web reasoning; and DEEP-100 and DEEP-200 are two versions of a synthetic benchmark for existential rules designed to exhibit worst-case complexity. All of the benchmarks terminate under skolem and restricted chase.

Rules and inputs have been adopted to the capabilities of the various systems. Data was provided in gzipped CSV files, except for Gringo, which received a list of facts. Existential rules used the native restricted chase implementation for Nemo and VLog, and were manually skolemised for the other systems. In Soufflé, this required a new algebraic type with a case for each skolem function. In general, the skolem chase is easier to compute than the restricted chase, but can lead to more redundant results (Benedikt et al. 2017). Programs, datasets (except SNOMED), and measurements for each system are online at https://github.com/knowsys/nemo-examples/tree/main/evaluations/kr2024.

Table 2 shows statistics about the benchmarks, and total runtimes (averaged across three runs), including both loading and reasoning. Compilation time for Soufflé (c) is not included. Fastest times are highlighted in bold for each benchmark. With existential rules, the number of derivations may vary depending on the used chase algorithms and internal choices regarding the application order of the rules, but were still very similar across systems (shown numbers are for Nemo). Command-line clients could use all system memory, whereas Nemo Web is restricted to less than 4GB (WebAssembly being 32bit).

All tested systems show very good performance on this limited hardware. While some benchmarks seem particularly difficult (SNOMED, LUBM, DEEP), each system shows specific strengths and weaknesses, which illustrates the great variance of underlying implementation methods. Surprising cases include DEEP-100, where Gringo is extremely fast (and correct), and interpreted Soufflé is superior to the compiled version. Gringo struggles with very large input data (LUBM-01K), but can still show excellent performance on big ontologies (SNOMED CT). Moreover, we observe that the performance advantage of Soufflé's compiled code is not decisive.

Nemo's generally fares well in the comparison, with significant performance advantages on plain Datalog and complex existential rule sets, and very good overall consistency across all tests. Nemo Web also performs very well for a system-independent, browser-based application, and is just $1.6 - 2.0$ times slower than the command-line client.

5.3 Processing Large Knowledge Graphs

To demonstrate Nemo's ability to handle very large datasets, we ran the program for selecting inverse properties described in Section 4.2 on an RDF dump of Wikidata. We used the Wikidata "truthy" dump of 11 May 2024, which contains 8.1 billion RDF triples (61 GB in gzipped N-Triples format). A laptop does not suffice here, so we run on a mid-range server (Linux NixOS 23.11; 2×QuadCore Intel Xeon E5-2637@3.5GHz; 768GiB RAM; 2x3.5TB SSD).

Nemo could complete the reasoning task in about 8 hours. Of this time, 475 min were spent on parsing the input RDF file and creating sorted index structures, and 9.6 min were used for the actual execution of rules. Nemo's static query optimiser chose to index all data in two orders, and 93min of the 475min were used to build this (possibly avoidable) second ordering. The two indices required a total of 220GB of memory (not counting the string dictionary).

The overall performance makes the use of Nemo feasible in practice, also given the complexity of the rules (with joins, string comparisons, arithmetic operations, and aggregates). For context, processing of Wikidata for Yago 4.5 took about 12 hours (Suchanek et al. 2024) (note that this is not the same computational task, but it shows the typical time scale of such analyses). We see a lot of potential in optimising further for such very large data sets, e.g., by parallelising IO.

6 Conclusions

Nemo is not just a single application, but a comprehensive framework for rule reasoning, which includes a scalable rule engine, a feature-rich rule language, and advanced user interfaces with state-of-the-art developer support. We demonstrate how Nemo can be of use in a wide variety of applications from the KR community. In spite of its many features, Nemo exhibits competitive and reliable performance on diverse benchmarks.

Next on Nemo's roadmap are improvements on the a wide variety of aspects of the overall toolkit. Regarding expressive power, we plan to enhance Nemo with native support for complex types, such as function terms and sets. User-defined functions are considered as a way to add further flexibility and unlock additional use cases. Moreover, we consider the support for non-stratified uses of aggregates, while still maintaining declarativity.

Regarding performance, Nemo is currently running in a single thread, and multi-threaded reasoning is a natural step to boost performance further. Especially in developer tools and the Nemo Web application, highlighting static analysis

results of programs, such as rule dependencies, and improved support for tracing will not only simplify debugging but help to achieve a more robust and explainable reasoning system. In this respect, we also see the incorporation of techniques for (non-)termination detection within the scope of the Nemo toolkit. We hope that our efforts in making Nemo as robust and convenient as possible will inspire further uses in the KR community. We welcome feedback, feature requests, and code contributions in our public repositories.

Acknowledgments

This work is partly supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) in project number 389792660 (TRR 248, Center for Perspicuous Systems), by the Bundesministerium für Bildung und Forschung (BMBF, Federal Ministry of Education and Research) under European ITEA project 01IS21084 (InnoSale, Innovating Sales and Planning of Complex Industrial Products Exploiting Artifical Intelligence) and in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI), and by BMBF and DAAD (German Academic Exchange Service) in project 57616814 (SECAI, School of Embedded and Composite AI).

References

Aberger, C. R.; Tu, S.; Olukotun, K.; and Ré, C. 2016. EmptyHeaded: A relational engine for graph processing. In Özcan, F.; Koutrika, G.; and Madden, S., eds., *Proc. 2016 ACM SIGMOD Int. Conf. on Management of Data*, 431–446. ACM.

Aberger, C. R.; Lamb, A.; Tu, S.; Nötzli, A.; Olukotun, K.; and Ré, C. 2017. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.* 42(4):20:1–20:44.

Abiteboul, S.; Hull, R.; and Vianu, V. 1994. *Foundations of Databases*. Addison Wesley.

Alviano, M., and Pieris, A., eds. 2024. *Selected Papers from Datalog 2.0 2022*, volume 24(2) of *Theory Pract. Log. Program.* Cambridge University Press.

Alviano, M.; Calimeri, F.; Faber, W.; Leone, N.; and Perri, S. 2011. Unfounded sets and well-founded semantics of answer set programs with aggregates. *J. Artif. Intell. Res.* 42:487–527.

Alviano, M.; Calimeri, F.; Dodaro, C.; Fuscà, D.; Leone, N.; Perri, S.; Ricca, F.; Veltri, P.; and Zangari, J. 2017. The ASP system DLV2. In Balduccini, M., and Janhunen, T., eds., *Proc. 14th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, volume 10377 of *LNCS*, 215–221. Springer.

Aref, M.; ten Cate, B.; Green, T. J.; Kimelfeld, B.; Olteanu, D.; Pasalic, E.; Veldhuizen, T. L.; and Washburn, G. 2015. Design and implementation of the LogicBlox system. In Sellis, T.; Davidson, S.; and Ives, Z., eds., *Proc. 2015 ACM SIGMOD Int. Conf. on Mngmt of Data*, 1371–1382.

Baader, F.; Brandt, S.; and Lutz, C. 2005. Pushing the $\mathcal{EL}$ envelope. In Kaelbling, L., and Saffiotti, A., eds., *Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI'05)*, 364–369. Professional Book Center.

Baget, J.; Leclère, M.; Mugnier, M.; Rocher, S.; and Sipieter, C. 2015. Graal: A toolkit for query answering with existential rules. In Bassiliades, N.; Gottlob, G.; Sadri, F.; Paschke, A.; and Roman, D., eds., *Proc. 9th Int. Web Rule Symposium (RuleML'15)*, volume 9202 of *LNCS*, 328–344. Springer.

Beck, H.; Dao-Tran, M.; and Eiter, T. 2018. LARS: A logic-based framework for analytic reasoning over streams. *Artif. Intell.* 261:16–70.

Beeri, C., and Vardi, M. Y. 1981. The implication problem for data dependencies. In Even, S., and Kariv, O., eds., *Proc. 8th Colloquium on Automata, Languages and Programming (ICALP'81)*, volume 115 of *LNCS*, 73–85. Springer.

Bellomarini, L.; Sallinger, E.; and Gottlob, G. 2018. The Vadalog system: Datalog-based reasoning for knowledge graphs. *Proc. VLDB Endowment* 11(9):975–987.

Benedikt, M.; Konstantinidis, G.; Mecca, G.; Motik, B.; Papotti, P.; Santoro, D.; and Tsamoura, E. 2017. Benchmarking the chase. In *Proc. 36th Symp. on Principles of Database Systems (PODS'17)*, 37–52. ACM.

Bizer, C.; Lehmann, J.; Kobilarov, G.; Auer, S.; Becker, C.; Cyganiak, R.; and Hellmann, S. 2009. DBpedia – A crystallization point for the Web of Data. *J. of Web Semantics* 7(3):154–165.

Bourgaux, C.; Carral, D.; Krötzsch, M.; Rudolph, S.; and Thomazo, M. 2021. Capturing homomorphism-closed decidable queries with existential rules. In Bienvenu, M.; Lakemeyer, G.; and Erdem, E., eds., *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning, KR 2021, Online event, November 3-12, 2021*, 141–150.

Brandt, S.; Kalayci, E. G.; Ryzhikov, V.; Xiao, G.; and Zakharyaschev, M. 2018. Querying log data with metric temporal logic. *J. Artif. Intell. Res.* 62:829–877.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 input language format. *Theory Pract. Log. Program.* 20(2):294–309.

Carral, D.; Dragoste, I.; González, L.; Jacobs, C.; Krötzsch, M.; and Urbani, J. 2019a. VLog: A rule engine for knowledge graphs. In Ghidini et al., C., ed., *Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II)*, volume 11779 of *LNCS*, 19–35. Springer.

Carral, D.; Dragoste, I.; Krötzsch, M.; and Lewe, C. 2019b. Chasing sets: How to use existential rules for expressive reasoning. In Kraus, S., ed., *Proc. 28th Int. Joint Conf. on Artificial Intelligence (IJCAI'19)*, 1624–1631. ijcai.org.

Carral, D.; Dragoste, I.; and Krötzsch, M. 2017. Restricted chase (non)termination for existential rules with disjunctions. In Sierra, C., ed., *Proc. 26th Int. Joint Conf. on Artificial Intelligence (IJCAI'17)*, 922–928. ijcai.org.

Cuenca Grau, B.; Horrocks, I.; Krötzsch, M.; Kupke, C.; Magka, D.; Motik, B.; and Wang, Z. 2012. Acyclicity conditions and their application to query answering in description logics. In Brewka, G.; Eiter, T.; and McIlraith, S. A., eds., *Proc. 13th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'12)*, 243–253. AAAI Press.

Cyganiak, R.; Wood, D.; and Lanthaler, M., eds. 2014. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. Available at http://www.w3.org/TR/rdf11-concepts/.

Deutsch, A.; Nash, A.; and Remmel, J. B. 2008. The chase revisited. In Lenzerini, M., and Lembo, D., eds., *Proc. 27th Symp. on Principles of Database Systems (PODS'08)*, 149–158. ACM.

Elhalawati, A.; Krötzsch, M.; and Mennicke, S. 2022. An existential rule framework for computing why-provenance on-demand for datalog. In Governatori, G., and Turhan, A., eds., *Proc. 2nd Int. Joint Conf. on Rules and Reasoning (RuleML+RR'22)*, volume 13752 of *LNCS*, 146–163. Springer.

Ellmauthaler, S.; Krötzsch, M.; and Mennicke, S. 2022. Answering queries with negation over existential rules. In *Proc. 36th AAAI Conf. on Artificial Intelligence (AAAI'22)*, 5626–5633.

Fagin, R.; Kolaitis, P. G.; Miller, R. J.; and Popa, L. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* 336(1):89–124.

Fredkin, E. 1960. Trie memory. *Commun. ACM* 3(9):490–499.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.* 19(1):27–82.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.* 187:52–89.

Gerlach, L., and Carral, D. 2023a. Do Repeat Yourself: Understanding Sufficient Conditions for Restricted Chase Non-Termination. In *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning*, 301–310.

Gerlach, L., and Carral, D. 2023b. General acyclicity and cyclicity notions for the disjunctive skolem chase. In Williams, B.; Chen, Y.; and Neville, J., eds., *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, 6372–6379. AAAI Press.

Gogacz, T., and Marcinkowski, J. 2014. All-instances termination of chase is undecidable. In Esparza, J.; Fraigniaud, P.; Husfeldt, T.; and Koutsoupias, E., eds., *Proc. 41st Int. Colloquium on Automata, Languages, and Programming (ICALP'14); Part II*, volume 8573 of *LNCS*, 293–304. Springer.

Gogacz, T.; Marcinkowski, J.; and Pieris, A. 2020. All-instances restricted chase termination. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'20, 245–258. New York, NY, USA: Association for Computing Machinery.

González, L.; Ivliev, A.; Krötzsch, M.; and Mennicke, S. 2022. Efficient dependency analysis for rule-based ontologies. In Sattler, U.; Hogan, A.; Keet, M.; Presutti, V.; Almeida, J. P. A.; Takeda, H.; Monnin, P.; Pirrò, G.; and d'Amato, C., eds., *Proc. 21st International Semantic Web Conference (ISWC 2022)*, volume 13489 of *LNCS*, 267–283. Springer.

Grahne, G., and Onet, A. 2018. Anatomy of the chase. *Fundam. Inform.* 157(3):221–270.

Gómez Álvarez, L.; Rudolph, S.; and Strass, H. 2023. Pushing the boundaries of tractable multiperspective reasoning: A deduction calculus for standpoint EL+. In Marquis, P.; Son, T. C.; and Kern-Isberner, G., eds., *Proc. 20th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'23)*, 333–343. IJCAI.

Hanisch, P., and Krötzsch, M. 2024. Chase termination beyond polynomial time. *Proc. ACM Manag. Data* 2(2):93.

Harris, S., and Seaborne, A., eds. 21 March 2013. *SPARQL 1.1 Query Language*. W3C Recommendation. Available at http://www.w3.org/TR/sparql11-query/.

Horridge, M.; Bechhofer, S.; and Noppens, O. 2007. Igniting the OWL 1.1 touch paper: The OWL API. In Golbreich, C.; Kalyanpur, A.; and Parsia, B., eds., *Proc. OWLED 2007 Workshop on OWL: Experiences and Directions*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Ivliev, A.; Ellmauthaler, S.; Gerlach, L.; Marx, M.; Meißner, M.; Meusel, S.; and Krötzsch, M. 2023. Nemo: First glimpse of a new rule engine. In Pontelli, E.; Costantini, S.; Dodaro, C.; Gaggl, S. A.; Calegari, R.; d'Avila Garcez, A. S.; Fabiano, F.; Mileo, A.; Russo, A.; and Toni, F., eds., *Proc. 39th Int. Conf. on Logic Programming (ICLP'23)*, volume 385 of *EPTCS*, 333–335.

Jordan, H.; Scholz, B.; and Subotic, P. 2016. Soufflé: On synthesis of program analyzers. In Chaudhuri, S., and Farzan, A., eds., *Proc. 28th Int. Conf. on Computer Aided Verification (CAV'16), Part II*, volume 9780 of *LNCS*, 422–430. Springer.

Kazakov, Y.; Krötzsch, M.; and Simančík, F. 2013. The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. *J. of Automated Reasoning* 53:1–61.

Körner, P.; Leuschel, M.; Barbosa, J.; Costa, V. S.; Dahl, V.; Hermenegildo, M. V.; Morales, J. F.; Wielemaker, J.; Diaz, D.; and Abreu, S. 2022. Fifty years of Prolog and beyond. *Theory Pract. Log. Program.* 22(6):776–858.

Krötzsch, M.; Rudolph, S.; and Hitzler, P. 2013. Complexities of Horn description logics. *ACM Trans. Comput. Logic* 14(1):2:1–2:36.

Krötzsch, M. 2011. Efficient rule-based inferencing for OWL EL. In Walsh (2011), 2668–2673.

Krötzsch, M. 2020. Computing cores for existential rules with the standard chase and ASP. In Calvanese, D.; Erdem, E.; and Thielscher, M., eds., *Proc. 17th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'20)*, 603–613. IJCAI.

Microsoft. 2024. *Official page for Language Server Protocol*. https://microsoft.github.io/language-server-protocol/, accessed May 2024.

Medeiros, S., and Mascarenhas, F. 2018. Syntax error recovery in parsing expression grammars. In Haddad, H. M.; Wainwright, R. L.; and Chbeir, R., eds., *Proc. 33rd Annual*

ACM Symposium on Applied Computing (SAC'18), 1195–1202. ACM.

Motik, B.; Cuenca Grau, B.; Horrocks, I.; Wu, Z.; Fokoue, A.; and Lutz, C., eds. 2009. *OWL 2 Web Ontology Language: Profiles*. W3C Recommendation. Available at http://www.w3.org/TR/owl2-profiles/.

Mugnier, M., and Thomazo, M. 2014. An introduction to ontology-based query answering with existential rules. In Koubarakis, M.; Stamou, G. B.; Stoilos, G.; Horrocks, I.; Kolaitis, P. G.; Lausen, G.; and Weikum, G., eds., *Reasoning Web: Reasoning on the Web in the Big Data Era – 10th International Summer School*, volume 8714 of *LNCS*, 245–278. Springer.

Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; and Banerjee, J. 2015. RDFox: A highly-scalable RDF store. In et al., M. A., ed., *Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II*, volume 9367 of *LNCS*, 3–20. Springer.

OWL Working Group, W. 2009. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation. Available at http://www.w3.org/TR/owl2-overview/.

Ross, K. A., and Sagiv, Y. 1997. Monotonic aggregation in deductive database. *J. Comput. Syst. Sci.* 54(1):79–97.

Seo, J.; Guo, S.; and Lam, M. S. 2015. SociaLite: An efficient graph query language based on datalog. *IEEE Trans. Knowl. Data Eng.* 27(7):1824–1837.

Simančík, F.; Kazakov, Y.; and Horrocks, I. 2011. Consequence-based reasoning beyond Horn ontologies. In Walsh (2011), 1093–1098.

Suchanek, F. M.; Alam, M.; Bonald, T.; Chen, L.; Paris, P.; and Soria, J. 2024. YAGO 4.5: A large and clean knowledge base with a rich taxonomy. In Yang, G. H.; Wang, H.; Han, S.; Hauff, C.; Zuccon, G.; and Zhang, Y., eds., *Proc. 47th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR'24)*, 131–140. ACM.

ter Horst, H. J. 2005. Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary. *J. of Web Semantics* 3(2–3):79–115.

Urbani, J.; Jacobs, C.; and Krötzsch, M. 2016. Column-oriented Datalog materialization for large knowledge graphs. In Schuurmans, D., and Wellman, M. P., eds., *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16)*, 258–264. AAAI Press.

Urbani, J.; Krötzsch, M.; and Eiter, T. 2022. Chasing Streams with Existential Rules. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, 415–419.

Veldhuizen, T. L. 2014. Triejoin: A simple, worst-case optimal join algorithm. In Schweikardt, N.; Christophides, V.; and Leroy, V., eds., *Proc. 17th Int. Conf. on Database Theory (ICDT'14)*, 96–106.

Vrandečić, D., and Krötzsch, M. 2014. Wikidata: A free collaborative knowledgebase. *Commun. ACM* 57(10).

Walsh, T., ed. 2011. *Proc. 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI'11)*. AAAI Press/IJCAI.

Wang, D.; Hu, P.; Wałęga, P. A.; and Cuenca Grau, B. 2022. Meteor: Practical reasoning in datalog with metric temporal operators. In *Proc. 36th AAAI Conf. on Artificial Intelligence (AAAI'22)*, 5906–5913.

Zaniolo, C.; Yang, M.; Das, A.; Shkapsky, A.; Condie, T.; and Interlandi, M. 2017. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory Pract. Log. Program.* 17(5-6):1048–1065.