

Existential Notation3 Logic

DÖRTHE ARNDT

*Computational Logic Group, TU Dresden, Dresden, Germany and
ScaDS.AI, Dresden/Leipzig, Germany
(e-mail: doerthe.arndt@tu-dresden.de)*

STEPHAN MENNICKE

*Knowledge-Based Systems Group, TU Dresden, Dresden, Germany
(e-mail: stephan.mennicke@tu-dresden.de)*

submitted 1 April 2024; revised 17 March 2025; accepted 25 March 2025

Abstract

In this paper, we delve into Notation3 Logic (N3), an extension of Resource Description Framework (RDF), which empowers users to craft rules introducing fresh blank nodes to RDF graphs. This capability is pivotal in various applications such as ontology mapping, given the ubiquitous presence of blank nodes directly or in auxiliary constructs across the Web. However, the availability of fast N3 reasoners fully supporting blank node introduction remains limited. Conversely, engines like VLog or Nemo, though not explicitly designed for Semantic Web rule formats, cater to analogous constructs, namely existential rules.

We investigate the correlation between N3 rules featuring blank nodes in their heads and existential rules. We pinpoint a subset of N3 that seamlessly translates to existential rules and establish a mapping preserving the equivalence of N3 formulae. To showcase the potential benefits of this translation in N3 reasoning, we implement this mapping and compare the performance of N3 reasoners like EYE and cwm against VLog and Nemo, both on native N3 rules and their translated counterparts. Our findings reveal that existential rule reasoners excel in scenarios with abundant facts, while the EYE reasoner demonstrates exceptional speed in managing a high volume of dependent rules.

Additionally to the original conference version of this paper, we include all proofs of the theorems and introduce a new section dedicated to N3 lists featuring built-in functions and how they are implemented in existential rules. Adding lists to our translation/framework gives interesting insights on related design decisions influencing the standardization of N3.

KEYWORDS: Notation3, Resource Description Framework, blank nodes, existential rules, rule reasoning

1 Introduction

Notation3 Logic (N_3) is an extension of the Resource Description Framework (RDF) which allows the user to quote graphs, to express rules, and to apply built-in functions on the components of RDF triples (Woensel *et al.* 2023; Berners-Lee *et al.* 2008). Facilitated by reasoners like cwm (Berners-Lee 2009), Data-Fu (Harth and Käfer 2018), or EYE

(Verborgh and De Roo 2015), N_3 rules directly consume and produce RDF graphs. This makes N_3 well-suited for rule exchange on the Web. N_3 supports the introduction of new blank nodes through rules, that is, if a blank node appears in the head¹ of a rule, each new match for the rule body produces a new instance of the rule's head containing *fresh* blank nodes. This feature is interesting for many use cases – mappings between different vocabularies include blank nodes, workflow composition deals with unknown existing instances (Verborgh *et al.* 2017) – but it also impedes reasoning tasks: from a logical point of view these rules contain existentially quantified variables in their heads. Reasoning with such rules is known to be undecidable in general and very complex on decidable cases (Baget *et al.* 2011; Krötzsch *et al.* 2019).

Even though recent projects like jen3² or RoXi (Bonte and Ongenae 2023) aim at improving the situation, the number of fast N_3 reasoners fully supporting blank node introduction is low. This is different for reasoners acting on existential rules, a concept very similar to blank-node-producing rules in N_3 , but developed for databases. Sometimes it is necessary to uniquely identify data by a value that is not already part of the target database. One tool to achieve that are *labeled nulls* which – just as blank nodes – indicate *the existence* of a value. This problem from databases and the observation that rules may provide a powerful, yet declarative, means of computing has led to more extensive studies of existential rules (Baget *et al.* 2011; Cali *et al.* 2010). Many reasoners like for example VLog (Carral *et al.* 2019) or Nemo (Ivliev *et al.* 2023) apply dedicated strategies to optimize reasoning with existential rules.

This paper aims to make existing and future optimizations on existential rules usable in the Semantic Web. We introduce a subset of N_3 supporting existential quantification but ignoring features of the language not covered in existential rules, like for example built-in functions or lists. We provide a mapping between this logic and existential rules: The mapping and its inverse both preserve equivalences of formulae, enabling N_3 reasoning via existential rule technologies. We discuss how the framework can be extended to also support lists – a feature of N_3 used in many practical applications, for example to support n-ary predicates. We implement the defined mapping in python and compare the reasoning performance of the existential rule reasoners Vlog and Nemo, and the N_3 reasoners EYE and cwm for two benchmarks: one applying a fixed set of rules on a varying size of facts, and one applying a varying set of highly dependent rules to a fixed set of facts. In our tests VLog and Nemo together with our mapping outperform the traditional N_3 reasoners EYE and cwm when dealing with a high number of facts while EYE is the fastest on large dependent rule sets. This is a strong indication that our implementation will be of practical use when extended by further features.

We motivate our approach by providing examples of N_3 and existential rule formulae, and discuss how these are connected, in Section 2. In Section 3 we provide a more formal definition of Existential N_3 (N_3^\exists), introduce its semantics and discuss its properties. We then formally introduce existential rules, provide the mapping from N_3^\exists into this logic, and prove its truth-preserving properties in Section 4. N_3 lists and the built-ins

¹ To stay consistent across frameworks, we use the terms *head* and *body* throughout the whole paper. The head is the part of the rule occurring at the end of the implication arrow, the body the part at its beginning (backward rules: “head \leftarrow body;” forward rules: “body \rightarrow head”).

² <https://github.com/william-vw/jen3>

associated with them are introduced as N_3 primitives as well as their existential rule translations are subject to Section 5. In Section 6 we discuss our implementation and provide an evaluation of the different reasoners. Related work is presented in Section 7. We conclude our discussion in Section 8. Furthermore, the code needed for reproducing our experiments is available on [GitHub](https://github.com/smennicke/n32rules) (<https://github.com/smennicke/n32rules>).

This article is an extended and revised version of our work (Arndt and Mennicke 2023a) presented at Rules and Reasoning – 7th International Joint Conference (RuleML+RR) 2023. Compared to the conference paper, we include full proofs to all theorems and lemmas. Furthermore, we strengthen the statements of correctness of our translation (Theorem 4.3 in Section 4), imposing stronger guarantees with effectively the same proofs as we had for the conference version, back then included in the technical appendix (Arndt and Mennicke 2023b) only. A discussion about the particular difference is appended to Theorem 4.3. Finally, we extend our considerations by N_3 lists and respective built-ins (cf. Section 5).

2 Motivation

N_3 has been introduced as a rule-based extension of RDF. As in RDF, N_3 knowledge is stated in triples consisting of *subject*, *predicate*, and *object*. In ground triples these can either be Internationalized Resource Identifiers (IRIs) or literals. The expression

$$:lucy :knows :tom. \quad (1)$$

means³ that “*lucy knows tom.*” Sets of triples are interpreted as their conjunction. Like RDF, N_3 supports blank nodes, usually starting with $_:$, which stand for (implicitly) existentially quantified variables. The statement

$$:lucy :knows _ :x. \quad (2)$$

means “*there exists someone who is known by lucy.*” N_3 furthermore supports implicitly universally quantified variables, indicated by a leading question mark (?), and implications which are stated using graphs, i.e., sets of triples, surrounded by curly braces ({}), as body and head connected via an arrow ($= >$). The formula

$$\{ :lucy :knows ?x \} => \{ ?x :knows :lucy \}. \quad (3)$$

means that “*everyone known by Lucy also knows her.*” Furthermore, N_3 allows the use of blank nodes in rules. These blank nodes are not quantified outside the rule like the universal variables, but in the rule part they occur in, that is either in its body or its head.

$$\{ ?x :knows :tom \} => \{ ?x :knows _ :y. _ :y :name "Tom" \}. \quad (4)$$

means “*everyone knowing Tom knows someone whose name is Tom.*”

This last example shows, that N_3 supports rules concluding the *existence* of certain terms which makes it easy to express them as *existential rules*. An existential rule is a

³ We omit name spaces for brevity.

first-order sentence of the form

$$\forall \mathbf{x}, \mathbf{y}. \varphi[\mathbf{x}, \mathbf{y}] \rightarrow \exists \mathbf{z}. \psi[\mathbf{y}, \mathbf{z}] \quad (5)$$

where $\mathbf{x}, \mathbf{y}, \mathbf{z}$ are mutually disjoint lists of variables, φ and ψ are conjunctions of atoms using only variables from the given lists, and φ is referred to as the *body* of the rule while ψ is called the *head*. Using the basic syntactic shape of (5) we go through all the example N_3 formulae (1)–(4) again and represent them as existential rules. To allow for the full flexibility of N_3 and RDF triples, we translate each RDF triple, just like the one in (1) into a first-order atom $tr(:lucy, :knows, :tom)$. Here, tr is a ternary predicate holding subject, predicate, and object of a given RDF triple. This standard translation makes triple predicates (e.g., $:knows$) accessible as terms. First-order atoms are also known as *facts*, finite sets of facts are called *databases*, and (possibly infinite) sets of facts are called *instances*. Existential rules are evaluated over instances (cf. Section 4).

Compared to other rule languages, the distinguishing feature of existential rules is the use of existentially quantified variables in the head of rules (cf. \mathbf{z} in (5)). The N_3 formula in (2) contains an existentially quantified variable and can, thus, be encoded as

$$\rightarrow \exists x. tr(:lucy, :knows, x) \quad (6)$$

Rule (6) has an empty body, which means the head is unconditionally true. Rule (6) is satisfied on instances containing any fact $tr(:lucy, :knows, _)$ (e.g., $tr(:lucy, :knows, :tim)$ so that variable x can be bound to $:tim$).

The implication of (3) has

$$\forall x. tr(:lucy, :knows, x) \rightarrow tr(x, :knows, :lucy) \quad (7)$$

as its (existential) rule counterpart, which does not contain any existentially quantified variables. Rule (7) is satisfied in the instance

$$\mathcal{I}_1 = \{tr(:lucy, :knows, :tom), tr(:tom, :knows, :lucy)\}$$

but not in

$$\mathcal{K}_1 = \{tr(:lucy, :knows, :tom)\}$$

since the only fact in \mathcal{K}_1 matches the body of the rule, but there is no fact reflecting on its (instantiated) head (i.e., the required fact $tr(:tom, :knows, :lucy)$ is missing). Ultimately, the implication (4) with blank nodes in its head may be transferred to a rule with an existential quantifier in the head:

$$\forall x. tr(x, :knows, :tom) \rightarrow \exists y. (tr(x, :knows, y) \wedge tr(y, :name, "Tom")). \quad (8)$$

It is clear that rule (8) is satisfied in instance

$$\mathcal{I}_2 = \{tr(:lucy, :knows, :tom), tr(:tom, :name, "Tom")\}.$$

However, instance \mathcal{K}_1 does not satisfy rule (8) because although the only fact satisfies the rule's body, there are no facts jointly satisfying the rule's head.

Note, for query answering over databases and rules, it is usually not required to decide for a concrete value of y (in rule (8)). Many implementations, therefore, use some form of abstraction: for instance, Skolem terms. VLog and Nemo implement the *standard chase* which uses another set of terms, so-called *labeled nulls*. Instead of injecting arbitrary

constants for existentially quantified variables, (globally) fresh nulls are inserted in the positions existentially quantified variables occur. Such a labeled null embodies the existence of a constant on the level of instances (just like blank nodes in RDF graphs). Let n be such a labeled null. Then \mathcal{I}_2 can be generalized to

$$\mathcal{I}_3 = \{tr(:lucy, :knows, :tom), tr(:lucy, :knows, n), tr(n, :name, "Tom")\},$$

on which rule (8) is satisfied, binding null n to variable y . \mathcal{I}_3 is, in fact, more general than \mathcal{I}_2 by the following observation: There is a mapping from \mathcal{I}_3 to \mathcal{I}_2 that is a homomorphism (see Subsection 4.1 for a formal introduction) but not vice versa. The homomorphism here maps the null n (from \mathcal{I}_3) to the constant `:tom` (in \mathcal{I}_2). Intuitively, the existence of a query answer (for a conjunctive query) on \mathcal{I}_3 implies the existence of a query answer on \mathcal{I}_2 . Existential rule reasoners implementing some form of *the chase* aim at finding the most general instances (*universal models*) in this respect (Deutsch *et al.* 2008).

In the remainder of this paper, we further analyze the relation between N_3 and existential rules. First, we give a brief formal account of the two languages and then provide a correct translation function from N_3 to existential rules.

3 Existential N_3

In the previous section we introduced essential elements of N_3 , namely triples and rules. N_3 also supports more complex constructs like lists, nesting of rules, and quotation. As these features are not covered by existential rules, we define a subset of N_3 excluding them, called *existential N_3* (N_3^\exists). This fragment of N_3 is still very powerful as it covers ontology mapping, one of N_3 's main use cases. Many ontologies rely on patterns including auxiliary blank nodes. N_3^\exists supports the production of these.⁴ In practice, these mappings are often connected with build-in functions like calculations or string operations,⁵ these are not covered yet, but could be added. A more difficult feature to add would be the support of so-called rule-producing rules: In N_3 it is possible to nest rules into the head of other rules. While this technique does not yield more expressivity, it is commonly used to translate from RDF datasets to N_3 rules (see e.g., Arndt *et al.* 2016). Such rule-producing rules can not be covered by existential rules as these only allow the derivation of facts.

We base our definitions on so-called *simple N_3 formulae* (Arndt 2019, Chapter 7), these are N_3 formulae which do not allow for nesting.

3.1 Syntax

N_3^\exists relies on the RDF alphabet. As the distinction is not relevant in our context, we consider IRIs and literals together as constants. Let C be a set of such constants, U

⁴ A very good example of such an ontology is the RDF version of HL7 FHIR (<https://www.hl7.org/fhir/rdf.html>). In HL7 FHIR literal values are always used in combination with the predicate `fhir:v`. The connection to concepts is always done through blank nodes representing so-called *primitive elements*. If we want to map from ontologies containing various datatype properties like for example FOAF (<http://xmlns.com/foaf/spec/>) to FHIR, we need to create new blank nodes.

⁵ Splitting the first name from the last name in a string if the target ontology requires to have these separated would be a practical example.

f ::=	formulae:	t ::=	terms:
t t t.	atomic formula	ex	existential variables
{e}=>{e}.	implication	c	constants
f f	conjunction		
n ::=	N3 terms:	e ::=	expressions:
uv	universal variables	n n n.	triple expression
t	terms	e e	conjunction expression

Fig. 1. Syntax of N_3^\exists .

a set of universal variables (starting with $?$), and E a set of existential variables (i.e., blank nodes). If the sets C , U , E , and $\{\{, \}, =, >, .\}$ are mutually disjoint, we call $\mathfrak{A} := C \cup U \cup E \cup \{\{, \}, =, >, .\}$ an N_3 *alphabet*. Figure 1 provides the syntax of N_3^\exists over \mathfrak{A} .

N_3^\exists fully covers RDF. RDF formulae are conjunctions of atomic formulae. Just as generalized RDF (Cyganiak *et al.* 2014), N_3^\exists allows for literals and blank nodes to occur in subject, predicate, and object position. The same holds for universal variables which are not present in RDF. This syntactical freedom is inherited from full N3 and makes it possible to – among other things – express the rules for RDF/S (Hayes and Patel-Schneider 2014, Appendix A) and OWL-RL (Motik *et al.* 2009, Section 4.3) entailment via N3. As an example for that, consider the following rule⁶ for inverse properties:

$$\{?p1 \text{ owl:inverseOf } ?p2 . ?x ?p1 ?y .\} \Rightarrow \{?y ?p2 ?x\}. \quad (9)$$

If we apply this rule on triple (1) in combination with

$$:\text{knows owl:inverseOf } : \text{isKnownBy}. \quad (10)$$

we derive

$$:\text{tom} : \text{isKnownBy} : \text{lucy}. \quad (11)$$

Similar statements and rules can be made for triples including literals. We can for example declare that the `:name` from rule (4) is the `owl:inverseOf` of `:isNameOf`.⁷ With rule (10) we then derive from

$$_:x : \text{name} \text{ "Tom"}. \quad (12)$$

that

$$\text{ "Tom" } : \text{isNameOf } _:x. \quad (13)$$

In that sense the use of generalized RDF ensures that all logical consequences we are able to produce via rules can also be stated in the language. This *principle of syntactical completeness* is also the reason to allow literals and blank nodes in predicate position. As universals may occur in predicate position, this also needs to be the case for all other kinds of symbols.

⁶ This rule corresponds to `prp-inv1` in OWL profiles (Motik *et al.* 2009).

⁷ Note that this is not possible in OWL itself as datatype properties cannot be inverted.

Currently, there is one exception to our principle: The syntax above allows rules having new universal variables in their head like for example

$$\{:\text{lucy} : \text{knows} : \text{tom}\} \Rightarrow \{?x : \text{is} : \text{happy}\}. \quad (14)$$

which results in a rule expressing “if *lucy* knows *tom*, everyone is happy.” This implication is problematic: Applied on triple (1), it yields $?x : \text{is} : \text{happy}$. which is a triple containing a universal variable. Such triples are not covered by our syntax, the rule thus introduces a fact we cannot express. Therefore, we restrict N_3^\exists rules to *well-formed implications* which rely on *components*. *Components* can be seen as direct parts⁸ an N_3 formula consists of. Let f be a formula or an expression over an alphabet \mathfrak{A} . The set $\text{comp}(f)$ of *components* of f is defined as:

- If f is an atomic formula or a triple expression of the form $t_1 t_2 t_3$., $\text{comp}(f) = \{t_1, t_2, t_3\}$.
- If f is an implication of the form $e_1 \Rightarrow e_2$., then $\text{comp}(f) = \{e_1, e_2\}$.
- If f is a conjunction of the form $f_1 f_2$, then $\text{comp}(f) = \text{comp}(f_1) \cup \text{comp}(f_2)$.

A rule $\{e_1\} \Rightarrow \{e_2\}$. is called *well-formed* if $(\text{comp}(e_2) \setminus \text{comp}(e_1)) \cap U = \emptyset$. For the remainder of this paper we assume all implications to be well-formed. Note that this definition of well-formed formulae is closely related to the idea of safety in logic programing. Well-formed rules are safe.

3.2 Semantics

In order to define the semantics of N_3^\exists we first note, that in our fragment of N_3 all quantification of variables is only defined implicitly. The blank node in triple (2) is understood as an existentially quantified variable, the universal in formula (3) as universally quantified. Universal quantification spans over the whole formula – variable $?x$ occurring in body and head of rule (3) is universally quantified for the whole implication – while existential quantification is local – the conjunction in the head of rule (4) is existentially quantified there. Adding new triples as conjuncts to formula (4) like

$$:\text{lucy} : \text{knows} _ : y. _ : y : \text{likes} : \text{cake}. \quad (15)$$

leads to the new statement that “*lucy* knows someone who likes cake” but even though we are using the same blank node identifier $_ : y$ in both formulae, the quantification of the variables in this formula is totally separated and the person named “Tom” is not necessarily related to the cake-liker. With the goal to deal with this locality of blank node scoping, we define substitutions which are only applied on components of formulae and leave nested elements like for example the body and head of rule (3) untouched.

A *substitution* σ is a mapping from a set of variables $X \subset U \cup E$ to the set of N_3 terms. We *apply* σ to a term, formula or expression x as follows:

- $x\sigma = \sigma(x)$ if $x \in X$,
- $(s p o)\sigma = (s\sigma)(p\sigma)(o\sigma)$ if $x = s p o$ is an atomic formula or a triple expression,
- $(f_1 f_2)\sigma = (f_1\sigma)(f_2\sigma)$ if $x = f_1 f_2$ is a conjunction,
- $x\sigma = x$ else.

⁸ As full N_3 supports graph terms, it could also be seen as a subset of terms as defined for full N_3 .

For formula $f = _ : x : p : o. \{ _ : x : b : c \} = > \{ _ : x : d : e \}$, substitution σ and $_ : x \in \text{dom}(\sigma)$, we get: $f\sigma = \sigma(_ : x) : p : o. \{ _ : x : b : c \} = > \{ _ : x : d : e \}$.⁹ We use the substitution to define the semantics of N_3^\exists which additionally makes use of *N3 interpretations* $\mathcal{I} = (\mathcal{D}, \mathbf{a}, \mathbf{p})$ consisting of (1) a set \mathcal{D} , called the domain of \mathcal{I} ; (2) a mapping $\mathbf{a} : C \rightarrow \mathcal{D}$, called the object function; (3) a mapping $\mathbf{p} : \mathcal{D} \rightarrow 2^{\mathcal{D} \times \mathcal{D}}$, called the predicate function.

Just as the function IEXT in RDF's simple interpretations (Hayes 2004), N_3 's predicate function maps elements from the domain of discourse to a set of pairs of domain elements and is not applied on relation symbols directly. This makes quantification over predicates possible while not exceeding first-order logic in terms of complexity. To introduce the semantics of N_3^\exists , let $\mathcal{I} = (\mathcal{D}, \mathbf{a}, \mathbf{p})$ be an N_3 interpretation. For an N_3^\exists formula f :

1. If $W = \text{comp}(f) \cap E \neq \emptyset$, then $\mathcal{I} \models f$ iff $\mathcal{I} \models f\mu$ for some substitution $\mu : W \rightarrow C$.
2. If $\text{comp}(f) \cap E = \emptyset$:
 - (a) If f is an atomic formula $t_1 t_2 t_3$, then $\mathcal{I} \models t_1 t_2 t_3$ iff $(\mathbf{a}(t_1), \mathbf{a}(t_3)) \in \mathbf{p}(\mathbf{a}(t_2))$.
 - (b) If f is a conjunction $f_1 f_2$, then $\mathcal{I} \models f_1 f_2$ iff $\mathcal{I} \models f_1$ and $\mathcal{I} \models f_2$.
 - (c) If f is an implication, then $\mathcal{I} \models \mathbf{e}_1 = > \mathbf{e}_2$ iff $\mathcal{I} \models e_2 \sigma$ if $\mathcal{I} \models e_1 \sigma$ for all substitutions σ on the universal variables $\text{comp}(\mathbf{e}_1) \cap U$ by constants.

The semantics as defined above uses a substitution into the set of constants instead of a direct assignment to the domain of discourse to interpret quantified variables. This design choice inherited from N_3 ensures referential opacity of quoted graphs and means, in essence, that quantification always refers to named domain elements.

With that semantics, we call an interpretation \mathfrak{M} *model* of a dataset Φ , written as $\mathfrak{M} \models \Phi$, if $\mathfrak{M} \models f$ for each formula $f \in \Phi$. We say that two sets of N_3^\exists formulae Φ and Ψ are *equivalent*, written as $\Phi \equiv \Psi$, if for all interpretations \mathfrak{M} : $\mathfrak{M} \models \Phi$ iff $\mathfrak{M} \models \Psi$. If $\Phi = \{\phi\}$ and $\Psi = \{\psi\}$ are singleton sets, we write $\phi \equiv \psi$ omitting the brackets.

3.2.1 Piece normal form

N_3^\exists formulae consist of conjunctions of triples and implications. For our goal of translating such formulae to existential rules, it is convenient to consider sub-formulae separately.

Below, we therefore define the so-called *Piece Normal Form* (PNF) for N_3^\exists formulae and show that each such formula f is equivalent to a set of sub-formulae Φ (i.e., $\Phi \equiv \phi$) in PNF. We proceed in two steps. First, we separate formulae based on their blank node components. If two parts of a conjunction share a blank node component, as in formula (15), we cannot split the formula into two since the information about the co-reference would get lost. However, if conjuncts either do not contain blank nodes or only contain disjoint sets of these, we can split them into so-called *pieces*: Two formulae f_1 and f_2 are called *pieces* of a formula f if $f = f_1 f_2$ and $\text{comp}(f_1) \cap \text{comp}(f_2) \cap E = \emptyset$. For such formulae we know:

⁹ Note that the semantics of *simple formulae* on which N_3^\exists 's semantics is based, relies on two ways to apply a substitution which is necessary to handle nested rules, since such constructs are excluded in N_3^\exists , we simplified here.

Lemma 3.1 (Pieces).

Let $f = f_1 f_2$ be an N_3^\exists conjunction and let $\text{comp}(f_1) \cap \text{comp}(f_2) \cap E = \emptyset$, then for each interpretation \mathfrak{I} , $\mathfrak{I} \models f$ iff $\mathfrak{I} \models f_1$ and $\mathfrak{I} \models f_2$.

Proof.

1. If $\text{comp}(f) \cap E = \emptyset$ the claim follows immediately by point 2b in the semantics definition.
2. If $W = \text{comp}(f) \cap E \neq \emptyset$: (\Rightarrow) If $\mathfrak{I} \models f$ then there exists a substitution $\mu : \text{comp}(f) \cap E \rightarrow C$ such that $\mathfrak{I} \models f\mu$, that is $\mathfrak{I} \models (f_1\mu) (f_2\mu)$. According to the previous point that implies $\mathfrak{I} \models f_1\mu$ and $\mathfrak{I} \models f_2\mu$ and thus $\mathfrak{I} \models f_1$ and $\mathfrak{I} \models f_2$. (\Leftarrow) If $\mathfrak{I} \models f_1$ and $\mathfrak{I} \models f_2$, then there exist two substitutions $\mu_1 : \text{comp}(f_1) \cap E \rightarrow C$ and $\mu_2 : \text{comp}(f_2) \cap E \rightarrow C$ such that $\mathfrak{I} \models f_1\mu_1$ and $\mathfrak{I} \models f_2\mu_2$. As the domains of the two substitutions are disjoint (by assumption), we can define the substitution $\mu : \text{comp}(f) \cap E \rightarrow C$ as follows:

$$\mu(v) = \begin{cases} \mu_1(v) & \text{if } v \in \text{comp}(f_1) \\ \mu_2(v) & \text{else} \end{cases}$$

Then $\mathfrak{I} \models f\mu$ and therefore $\mathfrak{I} \models f$.

□

If we recursively divide all pieces into sub-pieces, we get a maximal set $F = \{f_1, f_2, \dots, f_n\}$ for each formula f such that $F \equiv \{f\}$ and for all $1 \leq i, j \leq n$, $\text{comp}(f_i) \cap \text{comp}(f_j) \cap E \neq \emptyset$ implies $i = j$.

Second, we replace all blank nodes occurring in rule bodies by *fresh* universals. The rule

$$\{ _ : x : \text{likes} : \text{cake} \} \Rightarrow \{ : \text{cake} : \text{is} : \text{good} \}.$$

becomes

$$\{ ?y : \text{likes} : \text{cake} \} \Rightarrow \{ : \text{cake} : \text{is} : \text{good} \}.$$

Note that both rules have the same meaning, namely “if someone likes cake, then cake is good.” We generalize that:

Lemma 3.2 (Eliminating Existentials).

Let $f = \mathbf{e}_1 \Rightarrow \mathbf{e}_2$ and $g = \mathbf{e}'_1 \Rightarrow \mathbf{e}_2$ be N_3^\exists implications such that $\mathbf{e}'_1 = \mathbf{e}_1\sigma$ for some injective substitution $\sigma : \text{comp}(e_1) \cap E \rightarrow U \setminus \text{comp}(e_1)$ of the existential variables of e_1 by universals. Then $f \equiv g$.

Proof.

We first note that $\text{comp}(f) \cap E = \emptyset$ and $\text{comp}(g) \cap E = \emptyset$ since both formulae are implications.

(\Rightarrow) We assume that $\mathfrak{M} \not\models g$ for some model \mathfrak{M} . That is, there exists a substitution $\nu : (\text{comp}(e'_1) \cup \text{comp}(e_2)) \cap U \rightarrow C$ such that $\mathfrak{M} \models e'_1\nu$ and $\mathfrak{M} \not\models e_2\nu$. We show that $\mathfrak{M} \models e_1\nu$: As $((\text{comp}(e_1) \cup \text{comp}(e_2)) \cap U) \subset ((\text{comp}(e'_1) \cup \text{comp}(e_2)) \cap U)$, we know that

$\text{comp}(e_1\nu) \cap U = \emptyset$. With the substitution $\mu := \nu \circ \sigma$ for the existential variables in $e_1\nu$ we get $\mathfrak{M} \models (e_1\nu)\sigma$ and thus $\mathfrak{M} \models (e_1\nu)$, but as $\mathfrak{M} \not\models (e_2\nu)$ we can conclude that $\mathfrak{M} \not\models f$.

(\Leftarrow) We assume that $\mathfrak{M} \not\models f$. That is, there exists a substitution $\nu : (\text{comp}(e_1) \cup \text{comp}(e_2)) \cap U \rightarrow C$ such that $\mathfrak{M} \models e_1\nu$ and $\mathfrak{M} \not\models e_2\nu$. As $\mathfrak{M} \models e_1\nu$, there exists a substitution $\mu : \text{comp}(e_1\nu) \cap E \rightarrow C$ such that $\mathfrak{M} \models (e_1\nu)\mu$. With that we define a substitution $\nu' : (\text{comp}(e_1) \cup \text{comp}(e_2)) \cap U \rightarrow C$ as follows: $\nu' : U \rightarrow C$ as follows:

$$\nu'(v) = \begin{cases} \mu(\sigma^{-1}(v)) & \text{if } v \in \text{range}(\sigma) \\ \nu(v) & \text{else} \end{cases}$$

With that substitution we get $\mathfrak{M} \models e'_1\nu'$ but $\mathfrak{M} \not\models e_2\nu'$ and thus $\mathfrak{M} \not\models g$. \square

For a rule f we call the formula f' in which all existentials occurring in its body are replaced by universals following Lemma 3.2 the *normalized* version of the rule. We call an N_3^\exists formula f *normalized*, if all rules occurring in it as conjuncts are normalized. Combining the findings of the two previous lemmas, we introduce the *Piece Normal Form*:

Definition 3.3 (Piece Normal Form).

A finite set $\Phi = f_1, f_2, \dots, f_n$ of N_3^\exists formulae is in *piece normal form (PNF)* if all $f_i \in \Phi$ ($1 \leq i \leq n$) are normalized and $n \in \mathbb{N}$ is the maximal number such that for $1 \leq i, j \leq n$, $\text{comp}(f_i) \cap \text{comp}(f_j) \cap E \neq \emptyset$ implies $i = j$. If $f_i \in \Phi$ is a conjunction of atomic formulae, we call f_i an *atomic piece*.

We get the following result for N_3^\exists formulae:

Theorem 3.4.

For every well-formed N_3^\exists formula f , there exists a set $F = \{f_1, f_2, \dots, f_k\}$ of N_3^\exists formulae such that $F \equiv \{f\}$ and F is in *piece normal form*.

Proof.

The claim follows immediately from Lemma 3.1 and Lemma 3.2. \square

Since the piece normal form F of N_3^\exists formula f is obtained by only replacing variables and separating conjuncts of f into the set form, the overall size of F is linear in f .

4 From N3 to existential rules

Due to Theorem 3.4, we translate sets F of N_3^\exists formulae in PNF (cf. Definition 3.3) to sets of existential rules $\mathcal{T}(F)$ without loss of generality. As a preliminary step, we introduce the language of existential rules formally. Later on, we explain and formally define the translation function already sketched in Section 2. We close this section with a correctness argument, paving the way for existential rule reasoning for N_3^\exists formulae.

4.1 Foundations of existential rule reasoning

As for N_3 , we consider a first-order vocabulary, consisting of countably infinite mutually disjoint sets of constants (**C**), variables (**V**), and additionally so-called (labeled) nulls

(\mathbf{N}).¹⁰ As already mentioned in Section 2, we use the same set of constants as N_3 formulae, meaning $\mathbf{C} = C$. Furthermore, let \mathbf{P} be a (countably infinite) set of *relation names*, where each $p \in \mathbf{P}$ comes with an arity $ar(p) \in \mathbb{N}$. \mathbf{P} is disjoint from the term sets \mathbf{C} , \mathbf{V} , and \mathbf{N} . We reserve the ternary relation name $tr \in \mathbf{P}$ for our encoding of N_3 triples. If $p \in \mathbf{P}$ and $t_1, t_2, \dots, t_{ar(p)}$ is a list of terms (i.e., each $t_i \in \mathbf{C} \cup \mathbf{N} \cup \mathbf{V}$), $p(t_1, t_2, \dots, t_{ar(p)})$ is called an *atom*. We often use \mathbf{t} to summarize a term list like t_1, \dots, t_n ($n \in \mathbb{N}$), and treat it as a set whenever order is irrelevant. An atom $p(\mathbf{t})$ is *ground* if $\mathbf{t} \subseteq \mathbf{C}$. An *instance* is a (possibly infinite) set \mathcal{I} of variable-free atoms and a finite set of ground atoms \mathcal{D} is called a *database*.

For a set of atoms $\mathcal{I}[A]$ and an instance \mathcal{I} , we call a function h from the terms occurring in $\mathcal{I}[A]$ to the terms in \mathcal{I} a *homomorphism from $\mathcal{I}[A]$ to \mathcal{I}* , denoted by $h : \mathcal{I}[A] \rightarrow \mathcal{I}$, if (1) $h(c) = c$ for all $c \in \mathbf{C}$ (occurring in $\mathcal{I}[A]$), and (2) $p(\mathbf{t}) \in \mathcal{I}[A]$ implies $p(h(\mathbf{t})) \in \mathcal{I}$. If any homomorphism from $\mathcal{I}[A]$ to \mathcal{I} exists, write $\mathcal{I}[A] \rightarrow \mathcal{I}$. Please note that if n is a null occurring in $\mathcal{I}[A]$, then $h(n)$ may be a constant or null.

For an (*existential*) rule $r : \forall \mathbf{x}, \mathbf{y}. \varphi[\mathbf{x}, \mathbf{y}] \rightarrow \exists \mathbf{z}. \psi[\mathbf{y}, \mathbf{z}]$ (cf. (5)), rule body ($\text{body}(r) := \varphi$) and head ($\text{head}(r) := \psi$) will also be considered as sets of atoms for a more compact representation of the semantics. The notation $\varphi[\mathbf{x}, \mathbf{y}]$ ($\psi[\mathbf{y}, \mathbf{z}]$, resp.) indicates that the only variables occurring in φ (ψ , resp.) are $\mathbf{x} \cup \mathbf{y}$ ($\mathbf{y} \cup \mathbf{z}$, resp.). A finite set of existential rules Σ is called an (*existential*) *rule program*.

Let r be a rule and \mathcal{I} an instance. We call a homomorphism $h : \text{body}(r) \rightarrow \mathcal{I}$ a *match for r in \mathcal{I}* . Match h is *satisfied for r in \mathcal{I}* if there is an extension h^* of h (i.e., $h \subseteq h^*$) such that $h^*(\text{head}(r)) \subseteq \mathcal{I}$. If all matches of r are satisfied in \mathcal{I} , we say that r is satisfied in \mathcal{I} , denoted by $\mathcal{I} \models r$. For a rule program Σ and database \mathcal{D} , instance \mathcal{I} is a *model of Σ and \mathcal{D}* , denoted by $\mathcal{I} \models \Sigma, \mathcal{D}$, if $\mathcal{D} \subseteq \mathcal{I}$ and $\mathcal{I} \models r$ for each $r \in \Sigma$.

Labeled nulls play the role of fresh constants without further specification, just like blank nodes in RDF or N_3 . The chase is a family of algorithms that soundly produces models of rule programs by continuously applying rules for unsatisfied matches. Rule heads are then instantiated and added to the instance. Existentially quantified variables are thereby replaced by (globally) fresh nulls in order to facilitate arbitrary constant injections. More formally, we call a sequence $\mathcal{D}^0 \mathcal{D}^1 \mathcal{D}^2 \dots$ a *chase sequence of rule program Σ and database \mathcal{D}* if (1) $\mathcal{D}^0 = \mathcal{D}$ and (2) for $i > 0$, \mathcal{D}^i is obtained from \mathcal{D}^{i-1} by applying a rule $r \in \Sigma$ for match h in \mathcal{D}^{i-1} (i.e., $h : \text{body}(r) \rightarrow \mathcal{D}^{i-1}$ is an unsatisfied match and $\mathcal{D}^i = \mathcal{D}^{i-1} \cup \{h^*(\text{head}(r))\}$ for an extension h^* of h). The *chase of Σ and \mathcal{D}* is the limit of a chase sequence $\mathcal{D}^0 \mathcal{D}^1 \mathcal{D}^2 \dots$, that is $\bigcup_{i \geq 0} \mathcal{D}^i$. Although chase sequences are not necessarily finite,¹¹ the chase always is a (possibly infinite) model¹² (Deutsch *et al.* 2008). The described version of the chase is called *standard chase* or *restricted chase*.

We say that two rule programs Σ_1 and Σ_2 are *equivalent*, denoted $\Sigma_1 \triangleq \Sigma_2$, if for all instances \mathcal{I} , $\mathcal{I} \models \Sigma_1$ if and only if $\mathcal{I} \models \Sigma_2$. Equivalences of existential rules have been

¹⁰ We choose here different symbols to disambiguate between existential rules and N_3 , although vocabularies partially overlap.

¹¹ This also means there is no guarantee of termination.

¹² Not just any model, but a universal model, which is a model that has a homomorphism to any other model of the database and rule program. Up to homomorphisms, universal models are unique, justifying the use of the article *the* for *the chase*.

extensively studied in the framework of data exchange (Fagin *et al.* 2008; Pichler *et al.* 2011). Our equivalence is very strong and is called *logical equivalence* in the data exchange literature. For an alternative equivalence relation between rule programs, we could have equally considered equality of ground models (i.e., those models that are null-free). Let us define this equivalence as follows: $\Sigma_1 \equiv_g \Sigma_2$ if for each ground instance \mathcal{I} , $\mathcal{I} \models \Sigma_1$ if and only if $\mathcal{I} \models \Sigma_2$. The following lemma helps simplifying the proofs concerning the correctness of our transformation¹³ function later on.

Lemma 4.1.

\equiv and \equiv_g coincide.

Proof.

Of course, $\equiv \subseteq \equiv_g$ holds since the set of all ground models of a rule program is a subset of all models of that program.

Towards showing $\equiv_g \subseteq \equiv$, assume rule programs Σ_1 and Σ_2 such that $\Sigma_1 \equiv_g \Sigma_2$, but $\Sigma_1 \not\equiv \Sigma_2$. Then there is a model $\mathcal{I}[M]$ of Σ_1 , such that $\mathcal{I}[M] \not\models \Sigma_2$ (or vice versa), implying that for some rule $r \in \Sigma_2$ there is a match h in $\mathcal{I}[M]$ but for no extension h^* , we get $h^*(\text{head}(r)) \subseteq \mathcal{I}[M]$. As $\Sigma_1 \equiv_g \Sigma_2$, $\mathcal{I}[M]$ cannot be a ground instance and, thus, contains at least one null.

Claim: Because of $\mathcal{I}[M]$, there is a ground instance $\mathcal{I}[M]_g$, such that $\mathcal{I}[M]_g \models \Sigma_1$ and $\mathcal{I}[M]_g \not\models \Sigma_2$. But then $\mathcal{I}[M]_g$ constitutes a counterexample to the assumption that $\Sigma_1 \equiv_g \Sigma_2$. Thus, the assumption $\Sigma_1 \not\equiv \Sigma_2$ would be disproven.

In order to show the claim, we construct $\mathcal{I}[M]_g$ from $\mathcal{I}[M]$ by replacing every null n in $\mathcal{I}[M]$ by a (globally) fresh constant c_n . Since there might not be enough constants – $\mathcal{I}[M]$ may already use all countably infinite constants $c \in \mathbf{C}$ – we take a little detour: although the set of constants in use might be infinite in $\mathcal{I}[M]$, the number of constants used inside the rule programs Σ_1 and Σ_2 is finite.¹⁴ Create instance $\mathcal{I}[M]''$ from $\mathcal{I}[M]$ by replacing all constants c not part of Σ_1 or Σ_2 by fresh nulls n_c . Once again, $\mathcal{I}[M]$ may already use up all nulls $n \in \mathbf{N}$. So we have to take yet another detour from $\mathcal{I}[M]$ to $\mathcal{I}[M]'$ as follows:

Let $\gamma : \mathbf{N} \rightarrow \mathbf{N}$ be a (necessarily injective) enumeration of \mathbf{N} . Define $\eta : \mathbf{C} \cup \mathbf{N} \rightarrow \mathbf{C} \cup \mathbf{N}$ by (1) $\eta(c) := c$ for all $c \in \mathbf{C}$ and (2) $\eta(n) := \eta^{-1}(2 \cdot \eta(n))$. Then $\mathcal{I}[M]'$ is produced by applying η to $\mathcal{I}[M]$. Note, for each number $k \in \mathbf{N}$, $\eta^{-1}(2k+1)$ is not a null in $\mathcal{I}[M]'$. Since η is an isomorphism between $\mathcal{I}[M]$ and $\mathcal{I}[M]'$, we get that $\mathcal{I}[M] \models \Sigma$ if and only if $\mathcal{I}[M]' \models \Sigma$ for all rule programs Σ . Recall that isomorphic models preserve all first-order sentences (see, e.g., Ebbinghaus *et al.* 1994). Hence, $\mathcal{I}[M]' \models \Sigma_1$ and $\mathcal{I}[M]' \not\models \Sigma_2$.

Next we construct $\mathcal{I}[M]''$ from $\mathcal{I}[M]'$ by function ω mapping the terms occurring in $\mathcal{I}[M]'$ to $\mathbf{C} \cup \mathbf{N}$, such that (1) $\omega(c) = c$ if c is a constant occurring in $\Sigma_1 \cup \Sigma_2$, (2) $\omega(d)$ is a fresh null n_d if d is a constant not occurring in $\Sigma_1 \cup \Sigma_2$, and (3) $\omega(n) = n$ otherwise. ω exists because the number of nulls in use by $\mathcal{I}[M]'$ is countably infinite. Note that ω is injective and $\omega(\mathcal{I}[M]') = \mathcal{I}[M]''$ uses only finitely many constants. Once again we show that $\mathcal{I}[M]' \models \Sigma$ if and only if $\mathcal{I}[M]'' \models \Sigma$ for arbitrary rule programs Σ , implying that $\mathcal{I}[M]'' \models \Sigma_1$ and $\mathcal{I}[M]'' \not\models \Sigma_2$:

¹³ We are going to use the terms *translation* and *transformation* synonymously throughout the rest of this section.

¹⁴ Recall, a rule program is defined as a finite set of existential rules.

Let $r \in \Sigma$ with match h in $\mathcal{I}[M]'$. If h is satisfied in $\mathcal{I}[M]'$, then there is an extension h^* , such that $h^*(\text{head}(r)) \subseteq \mathcal{I}[M]'$. By definition of ω and, thus, the construction of $\mathcal{I}[M]''$, $\omega \circ h$ is a match for r in $\mathcal{I}[M]''$ and $\omega \circ h^*$ its extension with $\omega \circ h^*(\text{head}(r)) \subseteq \mathcal{I}[M]''$. The converse direction uses the the same argumentation, now from $\mathcal{I}[M]''$ to $\mathcal{I}[M]'$, using the fact that ω is injective.

From $\mathcal{I}[M]''$ we can finally construct ground instance $\mathcal{I}[M]_g$ by ν mapping all (finitely many) constants c in $\mathcal{I}[M]''$ to themselves and every null n in $\mathcal{I}[M]''$ to a fresh constant c_n . It holds that $\mathcal{I}[M]'' \models \Sigma$ if and only if $\nu(\mathcal{I}[M]') = \mathcal{I}[M]_g \models \Sigma$ (for all rule programs Σ) by a similar argumentation as given in the step from $\mathcal{I}[M]'$ to $\mathcal{I}[M]''$ above. Thus, $\mathcal{I}[M]_g \models \Sigma_1$ and $\mathcal{I}[M]_g \not\models \Sigma_2$, which completes proof. \square

4.2 The translation function from N_3 to existential rules

The translation function \mathcal{T} maps sets $F = \{f_1, \dots, f_k\}$ of N_3^\exists formulae in PNF to existential rule programs Σ . Before going into the intricates of \mathcal{T} for every type of piece, consider the auxiliary function $\mathbb{T} : C \cup E \cup U \rightarrow \mathbf{C} \cup \mathbf{V}$ mapping N_3 terms to terms in our rule language (cf. previous subsection):

$$\mathbb{T}(t) := \begin{cases} v_{\mathbf{x}}^\forall & \text{ift} = ?\mathbf{x} \in U \\ v_{\mathbf{y}}^\exists & \text{ift} = - : \mathbf{y} \in E \\ t & \text{ift} \in C, \end{cases}$$

where $v_{\mathbf{x}}^\forall, v_{\mathbf{y}}^\exists \in \mathbf{V}$ and $t \in \mathbf{C}$ (recall that $C = \mathbf{C}$). While variables in N_3 belong to either E or U , this separation is lost under function \mathbb{T} . For enhancing readability of subsequent examples, the identity of variables preserves this information by using superscripts \exists and \forall . Function \mathbb{T} naturally extends to triples $g = \mathbf{t}^1 \mathbf{t}^2 \mathbf{t}^3$: $\mathbb{T}(g) := tr(\mathbb{T}(t^1), \mathbb{T}(t^2), \mathbb{T}(t^3))$. We provide the translation for every piece $f_i \in F$ ($1 \leq i \leq k$) and later collect the full translation of F as the union of its translated pieces.

4.2.1 Translating atomic pieces

If f_i is an atomic piece, $f_i = \mathbf{g}_1 \mathbf{g}_2 \dots \mathbf{g}_l$ for some $l \geq 1$ and each g_j ($1 \leq j \leq l$) is an atomic formula. The translation of f_i is the singleton set $\mathcal{T}(f_i) = \{\rightarrow \exists \mathbf{z}. \mathbb{T}(g_1) \wedge \mathbb{T}(g_2) \wedge \dots \wedge \mathbb{T}(g_l)\}$, where \mathbf{z} is the list of translated existential variables (via \mathbb{T}) from existentials occurring in f_i . For example, the formula in (15) constitutes a single piece $f_{(15)}$ which translates to

$$\mathcal{T}(f_{(15)}) = \{\rightarrow \exists v_{\mathbf{y}}^\exists. tr(: \text{lucy}, : \text{knows}, v_{\mathbf{y}}^\exists) \wedge tr(v_{\mathbf{y}}^\exists, : \text{likes}, : \text{cake})\}.$$

4.2.2 Translating rules

For rule-shaped pieces $f_i = \{\mathbf{e}_1\} \Rightarrow \{\mathbf{e}_2\}$, we also obtain a single existential rule. Recall that our PNF ensures all variables in e_1 to be universals and all universal variables in e_2 to also occur in e_1 . If $e_1 = \mathbf{g}_1^1 \mathbf{g}_1^2 \dots \mathbf{g}_1^m$ and $e_2 = \mathbf{g}_2^1 \mathbf{g}_2^2 \dots \mathbf{g}_2^n$, $\mathcal{T}(f_i) := \{\forall \mathbf{x}. \bigwedge_{j=1}^m \mathbb{T}(g_1^j) \rightarrow \exists \mathbf{z}. \bigwedge_{j=1}^n \mathbb{T}(g_2^j)\}$ where \mathbf{x} and \mathbf{z} are the lists of translated universals and existentials, respectively. Applying \mathcal{T} to the N_3 formula in (4), which is a piece according to Definition

3.3, we obtain

$$\mathcal{T}(f(4)) = \{ \forall v_x^\forall. \text{tr}(v_x^\forall, : \mathbf{knows}, : \mathbf{tom}) \rightarrow \exists v_y^\exists. \text{tr}(v_x^\forall, v_y^\exists) \wedge \text{tr}(v_y^\exists, : \mathbf{name}, \mathbf{"Tom"}) \},$$

which is the same rule as given in (8) up to a renaming of (bound) variables.

4.2.3 Translating the PNF

For a set $F = \{f_1, f_2, \dots, f_k\}$ of N_3^\exists formulae in PNF, $\mathcal{T}(F)$ is the union of all translated constituents (i.e., $\mathcal{T}(F) := \bigcup_{i=1}^k \mathcal{T}(f_i)$). Please note that \mathcal{T} does not exceed a polynomial overhead in its input size.

4.3 Correctness of the translation

Let F be a set of N_3^\exists formulae in PNF. Its translation $\mathcal{T}(F)$ provides the following soundness guarantee: For every ground model $\mathcal{I}[M]$ of $\mathcal{T}(F)$, there is an interpretation of F that is itself a model.

Theorem 4.2.

Let F be a set of N_3^\exists formulae in PNF and $\mathcal{I}[M]$ a ground instance. Define the canonical interpretation of $\mathcal{I}[M]$ by $\mathfrak{J}(\mathcal{I}[M]) = (C, \mathbf{a}, \mathbf{p})$ such that

- $\mathbf{a}(t) := t$ for all $t \in C$ and
- $\mathbf{p}(p) := \{(s, o) \mid \text{tr}(s, p, o) \in \mathcal{I}[M]\}$ for all $p \in C$.

$\mathcal{I}[M]$ is a model of $\mathcal{T}(F)$ if and only if $\mathfrak{J}(\mathcal{I}[M])$ is a model of F .

Proof.

By induction on the number k of pieces in $F = \{f_1, f_2, \dots, f_k\}$:

Base: For $k = 1$, $F = \{f\}$ and f is either (a) an atomic piece or (b) a rule, being the cases we need to distinguish.

- (a) $\mathcal{T}(F) = \mathcal{T}(f) = \{\rightarrow \exists \mathbf{z}. \bigwedge_{i=1}^n \text{tr}(s_i, p_i, o_i)\}$. Every model of $\mathcal{T}(F)$ satisfies its single rule, meaning that if $\mathcal{I}[M]$ is a model, there is a homomorphism h^* from $\mathcal{A} = \{\text{tr}(s_i, p_i, o_i) \mid 1 \leq i \leq n\}$ to $\mathcal{I}[M]$. From $\mathcal{I}[M]$ we get $\mathfrak{J}(\mathcal{I}[M]) = (C, \mathbf{a}, \mathbf{p})$ with $(s_i, o_i) \in \mathbf{p}(p_i)$ for all $i \in \{1, \dots, n\}$. We need to show that $\mathfrak{J}(\mathcal{I}[M])$ is a model for F (i.e., for f).

If f contains existentials (i.e., $\text{comp}(f) \cap E = W$ is nonempty), we need to find a substitution $\mu : W \rightarrow C$ such that $\mathfrak{J}(\mathcal{I}[M]) \models f\mu$. Define $\mu : W \rightarrow C$ alongside h^* : $\mu(_ : \mathbf{y}) = h^*(v_y^\exists)$ for each $_ : \mathbf{y} \in W$. For each atomic formula $g_j = \mathbf{s}_j \mathbf{p}_j \mathbf{o}_j$ of f , we get $\mathfrak{J}(\mathcal{I}[M]) \models g_j \mu$ since $\text{tr}(h^*(s_j), h^*(p_j), h^*(o_j)) \in \mathcal{I}[M]$ implies $(h^*(s_j), h^*(o_j)) \in \mathbf{p}(h^*(p_j))$ and, thus, $(\mathbf{a}(s_j \mu), \mathbf{a}(o_j \mu)) \in \mathbf{p}(\mathbf{a}(p_j \mu))$. This argument holds for every atomic formula g_j of f , implying $\mathfrak{J}(\mathcal{I}[M]) \models F$. The converse direction uses the same argumentation backwards, constructing h^* from μ . If f does not contain any existentials, then f is itself an atomic formula and the result follows as the special case $n = 1$.

- (b) If $F = \{f\}$ and $f = \{\mathbf{e}_1\} \Rightarrow \{\mathbf{e}_2\}$, then $\mathcal{T}(F) = \{\forall \mathbf{x}. \varphi \rightarrow \exists \mathbf{z}. \psi\}$ where φ and ψ are translated conjunctions from e_1 and e_2 .

Let $\mathfrak{J}(\mathcal{I}[M])$ be a model of F . To show that $\mathcal{I}[M]$ is a model of $\mathcal{T}(F)$, it suffices to prove, for each match h of the rule, the existence of an extension h^* (of h), such

that $h^*(\psi) \subseteq \mathcal{I}[M]$. Let h be a match for the body of the rule and the body of the rule is a conjunction of atoms. Define σ with $\sigma(?x) := h(v_x^\forall)$ for each universal variable in e_1 . σ is a substitution such that $\mathcal{I}[\mathcal{I}[M]] \models e_1\sigma$. In order to prove this claim, let $\mathbf{s}p\mathbf{o}$ be a triple in e_1 . Hence, $tr(s, p, o) \in \varphi$ and, by the choice of h , $tr(h(s), h(p), h(o)) \in \mathcal{I}[M]$. This implies that $(h(s), h(o)) \in \mathbf{p}(h(p))$, which also implies $(s\sigma, o\sigma) \in \mathbf{p}(p\sigma)$. As this argument holds for all triples in e_1 , the claim follows. Please note that, as in case (a), this reasoning can be converted to construct a match h from a substitution σ .

Since $\mathcal{I}[\mathcal{I}[M]]$ is a model of f , there is a substitution $\mu : \text{comp}(e_2) \cap E \rightarrow C$, such that $\mathcal{I}[\mathcal{I}[M]] \models e_2\sigma\mu$. Define $h^* := h \cup \{w \mapsto \mu(w) \mid w \in \text{comp}(e_2) \cap E\}$. It holds that h^* satisfies match h since for each atomic formula $\mathbf{s}_i\mathbf{p}_i\mathbf{o}_i$ of e_2 , we get $\mathbf{a}(\mu(\sigma(s_i)), \mu(\sigma(o_i))) \in \mathbf{p}(\mathbf{a}(\mu(\sigma(p_i))))$ implying $tr(\mu(\sigma(s_i)), \mu(\sigma(p_i)), \mu(\sigma(o_i))) \in \mathcal{I}[M]$ and $h^*(\mathbb{T}(x)) = \mu(\sigma(x))$ ($x \in \{s_i, p_i, o_i\}$) providing a match for $tr(\mathbb{T}(s_i), \mathbb{T}(p_i), \mathbb{T}(o_i))$ (part of the head ψ). As this argument holds for all atomic formulae of e_2 , match h is satisfied via h^* . As before, the construction can be inverted, obtaining μ from h^* and σ from h , which completes the proof for this case.

Step: Let $F = \{f_1, f_2, \dots, f_k, f_{k+1}\}$ be a set of N_3^\exists formulae in PNF. By induction hypothesis, $\mathcal{I}[M]$ is a model of $\mathcal{T}(\{f_1, f_2, \dots, f_k\})$ if and only if $\mathcal{I}[\mathcal{I}[M]]$ is a model of $\{f_1, f_2, \dots, f_k\}$. Also by induction hypothesis, $\mathcal{I}[M]$ is a model of $\mathcal{T}(\{f_{k+1}\})$ iff $\mathcal{I}[\mathcal{I}[M]]$ is a model of $\{f_{k+1}\}$. Thus, $\mathcal{I}[M]$ is a model of $\mathcal{T}(F)$ if and only if it is a model of $\mathcal{T}(\{f_1 f_2 \dots f_k\})$ and of $\mathcal{T}(\{f_{k+1}\})$ if and only if $\mathcal{I}[\mathcal{I}[M]]$ is a model of $\{f_1 f_2 \dots f_k\}$ and of $\{f_{k+1}\}$ if and only if $\mathcal{I}[\mathcal{I}[M]]$ is a model of F . □

Consequently, the only (ground) models $\mathcal{T}(F)$ has are models of the original set of N_3^\exists formulae F . To complete the correctness argument, $\mathcal{T}(F)$ can have only those models relating to the ones of F , which is also true by the following theorem.

Theorem 4.3.

Let \mathfrak{M} be an N_3 interpretation, then there exists an instance $\mathcal{I}[M]$ such that for each set F of N_3^\exists formulae in PNF

$$\mathfrak{M} \models F \text{ if and only if } \mathcal{I}[\mathcal{I}[M]] \models F$$

(cf. [Theorem 4.2](#)).

Proof.

For $\mathfrak{M} = (\mathfrak{D}, \mathbf{a}, \mathbf{p})$, define $\mathfrak{M}_g = (C, \mathbf{b}, \mathbf{q})$ such that (a) \mathbf{b} is the identity on C (i.e., $\mathbf{b}(c) = c$ for all $c \in C$) and (b) $\mathbf{q}(p) := \{(s, o) \mid (\mathbf{a}(s), \mathbf{a}(o)) \in \mathbf{p}(\mathbf{a}(p))\}$ for all $p \in C$. Based on \mathfrak{M}_g , we can define instance $\mathcal{I}[M] := \{tr(s, p, o) \mid (s, o) \in \mathbf{q}(p)\}$.

Since $\mathcal{I}[\mathcal{I}[M]] = \mathfrak{M}_g$, it remains to be shown that $\mathfrak{M} \models F$ if and only if $\mathfrak{M}_g \models F$. We proceed by induction on (the number of pieces) $|F| = k$.

Base: If $k = 1$, then $F = \{f\}$ and two cases arise: (a) f is an atomic piece $\mathbf{g}_1 \dots \mathbf{g}_1$ (for some $l \geq 1$) and (b) f is an N_3 rule $\{\mathbf{e}_1\} \Rightarrow \{\mathbf{e}_2\}$.

- (a) $\mathfrak{M} \models f$ if and only if $\mathfrak{M} \models f\mu$ for some $\mu : \text{comp}(f) \cap E \rightarrow C$ if and only if for each atomic formula $\mathbf{g}_i = \mathbf{s}_i\mathbf{p}_i\mathbf{o}_i$ in f , $(\mathbf{a}(\mu(s_i)), \mathbf{a}(\mu(o_i))) \in \mathbf{p}(\mathbf{a}(\mu(p_i)))$ (by the semantics

of N_3^{\exists} if and only if $(\mathbf{b}(\mu(s_i)), \mathbf{b}(\mu(o_i))) \in \mathbf{q}(\mathbf{b}(\mu(p_i)))$ (by construction of \mathfrak{M}_g) if and only if $\mathfrak{M}_g \models f\mu$ if and only if $\mathfrak{M}_g \models f$ (by the semantics of N_3^{\exists}).

- (b) $\mathfrak{M} \models f$ if and only if, for each substitution $\sigma : U \rightarrow C$ with $\mathfrak{M} \models e_1\sigma$, there is a substitution $\mu : \text{comp}(e_2) \cap E \rightarrow C$ such that $\mathfrak{M} \models e_2\sigma\mu$.

For respective substitutions $\sigma : U \rightarrow C$ and $\mu : \text{comp}(e_2) \cap E \rightarrow C$, $\mathfrak{M} \models e_1\sigma$ if and only if $(\mathbf{a}(\sigma(s)), \mathbf{a}(\sigma(o))) \in \mathbf{p}(\mathbf{a}(\sigma(p)))$ for each atomic formula \mathbf{spo} in e_1 (by the semantics of N_3^{\exists}) if and only if $(\mathbf{b}(\sigma(s)), \mathbf{b}(\sigma(o))) \in \mathbf{q}(\mathbf{b}(\sigma(p)))$ for each atomic formula $\mathbf{s p o}$ of e_1 if and only if $\mathfrak{M}_g \models e_1\sigma$.

The same argument can be used to argue for $\mathfrak{M} \models e_2\sigma\mu$ if and only if $\mathfrak{M}_g \models e_2\sigma\mu$. Thus, for each $\sigma : U \rightarrow C$ for which $\mathfrak{M} \models e_1\sigma$ there is a substitution $\mu : \text{comp}(e_2) \cap E \rightarrow C$ such that $\mathfrak{M} \models e_2\sigma\mu$ and we obtain $\mathfrak{M}_g \models e_1\sigma$ and $\mathfrak{M}_g \models e_2\sigma\mu$, and vice versa.

Step: For $F = \{f_1, \dots, f_k, f_{k+1}\}$, the induction hypothesis applies to $F' = \{f_1, \dots, f_k\}$ and $F'' = \{f_{k+1}\}$:

$$\begin{aligned} \mathfrak{M} \models F & \quad \text{iff} \quad \mathfrak{M} \models F' \text{ and } \mathfrak{M} \models F'' & & \text{(by theorem 3.1)} \\ & \quad \text{iff} \quad \mathfrak{M}_g \models F' \text{ and } \mathfrak{M}_g \models F'' & & \text{(by induction hypothesis)} \\ & \quad \text{iff} \quad \mathfrak{M}_g \models F & & \text{(by theorem 3.1)} \end{aligned}$$

□

In the conference version of this paper, we have shown the following result to justify correctness of \mathcal{T} , relating equivalent N_3 formulae to their translations.

Corollary 4.4 (Theorem 2 (Arndt and Mennicke 2023a)).

For sets of N_3^{\exists} formulae F and G in PNF, $F \equiv G$ if and only if $\mathcal{T}(F) \trianglelefteq \mathcal{T}(G)$.

This kind of correctness argument has been coined to the term of *full abstraction* (Gorla and Nestmann 2016).

Proof.

Let F and G be sets of N_3^{\exists} formulae in PNF. The proof dissects the statement in its two logical parts:

Soundness (\Rightarrow) Assume $F \equiv G$. We need to show that $\mathcal{T}(F) \trianglelefteq \mathcal{T}(G)$. Let $\mathcal{I}[M]$ be a ground model of $\mathcal{T}(F)$. Then by Theorem 4.2, $\mathcal{J}(\mathcal{I}[M])$ is a model of F . By assumption ($F \equiv G$), $\mathcal{J}(\mathcal{I}[M])$ is a model of G and, again by an application of Theorem 4.2, we get that $\mathcal{I}[M]$ must be a model of $\mathcal{T}(G)$. Hence, $\mathcal{T}(F) \trianglelefteq_g \mathcal{T}(G)$ which implies $\mathcal{T}(F) \trianglelefteq \mathcal{T}(G)$ by Lemma 4.1. The converse direction, starting from ground models of $\mathcal{T}(G)$, uses the inverse argumentation.

Completeness (\Leftarrow) We prove the contrapositive: $F \not\equiv G$ implies $\mathcal{T}(F) \not\trianglelefteq \mathcal{T}(G)$. Assume $F \not\equiv G$. Then there is a model \mathfrak{M} such that $\mathfrak{M} \models F$ and $\mathfrak{M} \not\models G$, or vice versa. Since the cases are symmetric, without loss of generality, assume $\mathfrak{M} \models F$ and $\mathfrak{M} \not\models G$ and suppose, towards a contradiction, that $(\star) \mathcal{T}(F) \trianglelefteq \mathcal{T}(G)$. By Theorem 4.3, there is an instance $\mathcal{I}[M]$ such that $\mathfrak{M} \models H$ if and only if $\mathcal{J}(\mathcal{I}[M]) \models H$ for arbitrary sets H of N_3^{\exists} formulae in PNF. Thus, $\mathcal{J}(\mathcal{I}[M]) \models F$ and $(\star\star) \mathcal{J}(\mathcal{I}[M]) \not\models G$. By Theorem 4.2, we get that $\mathcal{I}[M] \models \mathcal{T}(F)$ and, by assumption (\star) , $\mathcal{I}[M] \models \mathcal{T}(G)$.

But then Theorem 4.2 allows for the conclusion that $\mathfrak{I}(\mathcal{I}[M]) \models G$, contradicting $(\star\star)$. Therefore, the assumption (\star) is false, meaning $\mathcal{T}(F) \not\equiv \mathcal{T}(G)$.

□

In the conference version of this paper, Corollary 4.4 originated from the attempt to evade trivial and/or *unexpectedly simple yet undesirable* transformations, which our transformation function \mathcal{T} does not belong to because it possesses even stronger guarantees as proven in Theorems 4.2 and 4.3. For instance, consider a translation \mathcal{T}_0 mapping all sets of N_3^\exists formulae in PNF to some fixed set of rules Σ_0 (e.g., $\Sigma_0 = \emptyset$). This translation is surely *sound* since the image of \mathcal{T}_0 consists of the single set of rules Σ_0 , being trivially equivalent to itself. On the other hand, requiring *completeness* rules out \mathcal{T}_0 as a *good* transformation, since also non-equivalent N_3 formulae map to the same (i.e., equivalent) rule program.

While \mathcal{T}_0 can be ruled out as trivially incomplete by Corollary 4.4, undesirable transformations are still in reach: Consider an arbitrary enumeration of equivalence classes of sets of N_3^\exists formulae in PNF $\mathfrak{F}_1, \mathfrak{F}_2, \mathfrak{F}_3, \dots$ and for each of these classes a first (e.g., the lexicographically smallest) element $1^{\text{st}}(\mathfrak{F}_i)$. Define $\mathcal{T}_1(F) := \mathcal{T}(1^{\text{st}}(\mathfrak{F}_{i+1}))$ if $F \in \mathfrak{F}_i$. \mathcal{T}_1 is *sound* and *complete* in the sense of Corollary 4.4, but fails in having stronger guarantees on the relationship between the different semantic worlds. $\mathcal{T}_1(F)$ may end up with a rule program that speaks about a completely different vocabulary – in consequence, completely different subjects – than F does. Even worse, the *correctness* of \mathcal{T}_1 does not even require the intermediate transformation \mathcal{T} to be correct in any way: different equivalence classes must just be associated with (semantically) different rule programs.

\mathcal{T}_0 is *incomplete* with respect to Corollary 4.4. \mathcal{T}_1 on the other hand satisfies Corollary 4.4 but it does not share the same strong characteristics of \mathcal{T} proven in Theorems 4.2 and 4.3: Models of translated formulas cannot easily be converted into models of the original N_3 formula, making such transformations hard to use in contexts in which we want to employ existential rule reasoning as alternatives to existing N_3 reasoners. In retrospect, the connection between N_3 models and models of existential rules through \mathcal{T} turns out to be much deeper than captured by Corollary 4.4 alone. This depth has been exploited within the proof of Corollary 4.4 that used to be available in our technical appendix (Arndt and Mennicke 2023b) only. Nevertheless, most parts of the original proofs are kept and allow for the stronger statements we now describe by Theorems 4.2 and 4.3. The new and stronger formulation is meaningful to *reasoning* since all reasoning results can be translated back-and-forth through \mathcal{T} . It is the combination of Theorems 4.2 and 4.3 and Corollary 4.4 that makes our function \mathcal{T} a useful contribution. Note that similar (and further) issues regarding *full abstraction* have been uncovered in other communities before (Gorla and Nestmann 2016; Parrow 2016).

Apart from correctness of \mathcal{T} and the close connection between N_3 models and models of transformed N_3^\exists formulae, we have no further guarantees. As N_3^\exists reasoning does not necessarily stop, there is no need for termination of the chase over translated rule programs. We expect that the similarity between the identified N_3 fragment of N_3^\exists and existential rules allows for the adoption of sufficient conditions for finite models, for instance, by means of acyclicity (see Cuenca Grau *et al.* 2013 for a survey).

5 Reasoning with lists

So far, we discussed N_3^\exists as a fragment of N_3 which can directly be mapped to existential rules. In this section, we detail how N_3^\exists and our translation to existential rules can be extended towards supporting lists. Lists is a very important concept in N_3 . We first explain them in more detail and provide their semantics. Then we explain how lists and list functions can be covered by existential rules. We finish the section by discussing different ways to implement list functions in N_3 .

5.1 N_3 lists

Before introducing them formally, we explain the role of lists in Notation3 Logic by examples. N_3 is based on RDF, but, in contrast to RDF, N_3 treats lists as first-class citizens. To illustrate this, we take a closer look at the following triple containing a list:

$$:lucy :likes (:cake :chocolate :tea). \quad (16)$$

Stating that lucy likes cake, chocolate and tea. If we understand the above as an example of RDF-turtle (Beckett *et al.* 2014), the list-notation () is syntactic sugar for:

$$\begin{aligned} &:lucy :likes _ :l1. \\ &_ :l1rdf:first :cake; rdf:rest _ :l2. \\ &_ :l2rdf:first :chocolate; rdf:rest _ :l3. \\ &_ :l3rdf:first :tea; rdf:restrdf:nil. \end{aligned} \quad (17)$$

According to RDF semantics the predicates `rdf:first` and `rdf:rest` are properties whose domain is the class of lists, for `rdf:rest` the range is the class of lists and `rdf:nil` is itself a list. Their meaning is not specified any further.

In N_3 , the list in (16) itself is understood as a resource and not just as syntactic sugar for (17). The predicates `rdf:first` and `rdf:rest` have a more specific meaning: they stand for the relation between a list and its first element, a list and its rest list, that is the list, we obtain if we remove the first element, respectively. The rule

$$\{ (:a :b :c) \text{ rdf:first } ?x; \text{ rdf:rest } ?y \} \Rightarrow \{ ?x : \text{and } ?y \}. \quad (18)$$

for example, yields

$$:a : \text{and} (:b :c). \quad (19)$$

The constant `rdf:nil` stands for the empty list and can also be written as ().

If we define the semantics in a naive way, N_3 's view of lists is not fully compatible with the (syntactic sugar) view of RDF. Suppose, we have a new triple stating the food preferences of Tom (which coincide with Lucy's preferences):

$$:tom :likes (:cake :chocolate :tea). \quad (20)$$

If we apply the N_3 rule

$$\{ ?x :likes ?z. ?y :likes ?z \} \Rightarrow \{ ?x : \text{sharesPreferencesWith } ?y \}. \quad (21)$$

on triple (20) and (16), we retrieve¹⁵ that

:lucy :sharesPreferencesWith :tom. (22)

Now, we replace (20) by the first-rest combination it stands for, namely

:ben :likes _:k1. (23)
_:k1 rdf:first :cake; rdf:rest _:k2.
_:k2rdf:first :chocolate; rdf:rest _:k3.
_:k3 rdf:first :tea; rdf:rest rdf:nil.

If we again apply rule (21), but this time on the list representations (17) and (23), it is not evident that we get triple (22) as a result. The lists are represented by blank nodes `_:l1` and `_:k1`, and it is not immediately clear that these refer to the same list. The original informal N_3 specification overcomes the problems caused by the different representations by providing the following three axioms (Berners-Lee *et al.* 2008; Berners-Lee and Connolly 2011):

Existence of Lists All lists exist. That is, the triple `[rdf:first :a; rdf:rest rdf:nil]` does not carry any new information.

Uniqueness of Lists Two lists having the same `rdf:first`-element and also the same `rdf:rest`-element are equal. If we add the notion of equality¹⁶ $(=)$: $\{?L1 \text{ rdf:first } ?X; \text{ rdf:rest } ?R. \text{ ?L12 rdf:first } ?X; \text{ rdf:rest } ?R.\} = > \{?L1 = ?L2\}$.

Functionality The predicates `rdf:first` and `rdf:rest` are functional properties. If we again add equality $(=)$: $\{?S \text{ rdf:first } ?O1, ?O2.\} = > \{?O1 = ?O2\}$. $\{?S \text{ rdf:rest } ?O1, ?O2.\} = > \{?O1 = ?O2\}$.

The first axiom guarantees that there is no new information added when translating from the native list notation (16) to the first-rest notation (17). The second and the third are important for the other direction, and, in a modified version, also for the purposes of our research which is to express N_3 lists and list predicates by means of existential rules. We will come back to that in Subsection 5.2.

Before introducing the non-basic list predicates, we provide the syntax and semantics of the extension of N_3 ³ with basic lists. We start with the syntax and extend the grammar provided in Figure 1 as follows:

- the set \mathbf{t} of term additionally contains the empty list `()` and the concept (1) of list terms, with

$$\begin{aligned} \mathbf{l} ::= & \\ & \mathbf{t} \\ & \mathbf{l} \mathbf{t} \end{aligned}$$

¹⁵ Of course, we retrieve more, namely, that Tom shares preferences with Lucy and that both share preferences with themselves.

¹⁶ Note that this equality is not that same kind of equality that the N_3 predicate `log:equalTo` provides. The latter is on syntax and not on the semantic level.

- the set \mathbf{n} of N_3 terms additionally contains the concept (\mathbf{k}) of N_3 list terms, with

$$\begin{aligned} \mathbf{k} ::= & \\ & \mathbf{n} \\ & \mathbf{k} \mathbf{n} \end{aligned}$$

We further need to extend the *application* of a substitution introduced in Subsection 3.2 by $(\mathbf{t}_1 \dots \mathbf{t}_n)\sigma = (\mathbf{t}_1\sigma \dots \mathbf{t}_n\sigma)$ if $x = (\mathbf{t}_1 \dots \mathbf{t}_n)$ is a list, and the object function \mathbf{a} of N_3 interpretations $\mathcal{I} = (\mathcal{D}, \mathbf{a}, \mathbf{p})$ as follows: If $\mathbf{t} = (\mathbf{t}_1 \dots \mathbf{t}_n)$ then $\mathbf{a}(\mathbf{t}) = (\mathbf{a}(\mathbf{t}_1) \dots \mathbf{a}(\mathbf{t}_n))$. If $\mathbf{t} = ()$ then $\mathbf{a}(\mathbf{t}) = ()$.

Note, that with our extension the domain \mathcal{D} of a model for a graph containing a list term also needs to contain a list of domain elements. However, the number of lists necessarily contained in \mathcal{D} is determined by the number of lists which can be produced using the alphabet. It is countable and does not depend on \mathcal{D} itself. If \mathcal{D} contains all lists which can be constructed using the interpretations of the N_3 terms, then axiom 1 (existence of lists) is fulfilled.

We finish the definition of the semantics of N_3^\exists with basic lists as follows:

Given an N_3 alphabet which contains the list constants `rdf:first` and `rdf:rest`, and an N_3 Interpretation $\mathcal{I} = (\mathcal{D}, \mathbf{a}, \mathbf{p})$. We say that \mathcal{I} is a model according to the simple list semantics of a formula ϕ , written as $\mathcal{I} \models_{sl} \phi$ iff $\mathcal{I} \models \phi$ and for triples containing `rdf:first` or `rdf:rest` in predicate position:

- $\mathcal{I} \models_{sl} s \text{ rdf:first } o$. iff $\mathbf{a}(s) = (s_1 \dots s_n)$ and $\mathbf{a}(o) = s_1$
- $\mathcal{I} \models_{sl} s \text{ rdf:rest } o$. iff $\mathbf{a}(s) = (s_1 s_2 \dots s_n)$ and $\mathbf{a}(o) = (s_2 \dots s_n)$

Note that with this definition, we also fulfill the two missing axioms stated above. The syntactic list structure maps to a list structure in the domain of discourse. This domain list can only have one first element and only one rest list, and it is fully determined by these two parts.

In addition to `rdf:first` and `rdf:rest`, N_3 contains a few more special predicates which make it easier to handle lists. In our list-extension of N_3^\exists we include¹⁷ `list:last`, `list:in`, `list:member`, `list:append`, and `list:remove`: `list:last` is used to relate a list to its last argument¹⁸ $((:a :b :c) \text{ list:last } :c.)$, `list:member` defines the relation between a list and its member $((:a :b :c) \text{ list:member } :a, :b, :c.)$, `list:in` is the inverse of `list:member` $((:b \text{ list:in } (:a :b :c).)$, `list:append` expresses that the list in object position is the combination of the two lists in subject position $(((:a :b) (:c :d)) \text{ list:append } (:a :b :c :d).)$, and by `list:remove` we express that the object list is the list we get by removing all occurrences of the second argument of the subject list of the first argument of the subject list $(((:a :b :a :c) :a) \text{ list:remove } (:b :c).)$.

Note, that N_3 built-ins are not defined as functions but as relations. As a consequence of that, they can be used in different ways. We illustrate this on the predicate `list:append`. If we write the following rule

$$\{((:a :b) (:c :d)) \text{ list:append } ?x\} \Rightarrow \{:\text{result} : \text{is } ?x\}. \quad (24)$$

¹⁷ The list predicates are specified at <https://w3c.github.io/N3/reports/20230703/builtins.html\#list>.

We exclude the rather complex predicates `list:iterate` and `list:memberAt`.

¹⁸ We give an example of one or more triples (in brackets) which need to be true after each explanation.

a reasoner will retrieve

$$\text{:result :is (:a :b :c :d)}. \quad (25)$$

But we can also write a rule like

$$\{(?x ?y) \text{list:append (:a :b :c)}\} \Rightarrow \{?x : \text{and } ?y\}. \quad (26)$$

which yields

$$\begin{aligned} &() : \text{and} (:a :b :c). \\ &(:a) : \text{and} (:b :c). \\ &(:a :b) : \text{and} (:c). \\ &(:a :b :c) : \text{and} (). \end{aligned}$$

Additionally, it is possible that only one of the two variables in the subject list is instantiated, with

$$\{((:a :b) ?y) \text{list:append (:a :b :c)}\} \Rightarrow \{:\text{we :get } ?y\}. \quad (27)$$

for example, we get

$$:\text{we :get } (:c). \quad (28)$$

On a practical level, however, this understanding of built-ins as relations comes with some limitations. If the presence of a built-in predicate causes a rule to produce infinitely many results, like it is the case with

$$\{?x \text{list:last :c}\} \Rightarrow \{:\text{we :get } ?x\}. \quad (29)$$

where all possible lists having `:c` as last element need to be produced, reasoning engines normally ignore the rule.¹⁹ We will define the full meaning of built-in predicates in our semantics, but our translation to existential rules provided in the next section will only focus on built-in predicates producing a limited number of solutions.

We now come to the semantics of list predicates. Given an N_3 alphabet which contains the list constants `rdf:first`, `rdf:rest`, `list:in`, `list:member`, `list:append`, `list:last` and `list:remove`, and an N_3 Interpretation $\mathcal{I} = (\mathfrak{D}, \mathbf{a}, \mathbf{p})$. We say that \mathcal{I} is a model according to list semantics of a formula ϕ , written as $\mathcal{I} \models_l \phi$ iff $\mathcal{I} \models_{sl} \phi$ and the following conditions hold:

- $\mathcal{I} \models_l s \text{list:in } o$. iff $\mathbf{a}(o) = (o_1 \dots o_n)$ and $\mathbf{a}(s) = o_i$ for some i with $1 \leq i \leq n$,
- $\mathcal{I} \models_l s \text{list:member } o$. if $\mathbf{a}(s) = (s_1 \dots s_n)$ and $\mathbf{a}(o) = s_i$ for some i with $1 \leq i \leq n$,
- $\mathcal{I} \models_l s \text{list:append } o$. iff $\mathbf{a}(s) = ((a_1 \dots a_n)(b_1 \dots b_m))$, $0 \leq n$, $0 \leq m$, and $\mathbf{a}(o) = (a_1 \dots a_n b_1 \dots b_m)$,
- $\mathcal{I} \models_l s \text{list:last } o$ iff $\mathbf{a}(s) = (s_1 \dots s_n)$ and $\mathbf{a}(o) = s_n$,
- $\mathcal{I} \models_l s \text{list:remove } o$ iff $\mathbf{a}(s) = ((a_1 \dots a_n) b)$ and $\mathbf{a}(o) = (a_i)_{a_i \neq b}$

In the next section we discuss how lists and list predicates can be modeled with existential rules.

¹⁹ To be more precise, the N_3 specification comes with so-called argument-modes specifying which arguments need to be instantiated for the predicate to be called, see also Woensel & Hochstenbach (2023).

5.2 Implementing N3 lists in existential rules

We model lists alongside the RDF representation of the previous subsection, sticking to the criteria imposed by N3, predominantly *uniqueness of lists* and *functionality*. For readability purposes we subsequently diverge from using our triple predicate *tr* for predicates concerning lists. Instead of $tr(x, \text{rdf} : \text{first}, y)$ we use an auxiliary binary predicate *first* and write $first(x, y)$. Similarly we use $rest(x, y)$ to denote $tr(x, \text{rdf} : \text{rest}, y)$. For technical reasons, we use a unary predicate *list* to identify all those objects that are lists. Before modeling lists and their functions, let us formulate the criteria based on the three predicates: A model $\mathcal{I}[M]$ of rule set Σ and database \mathcal{D} satisfies

Uniqueness of Lists if for all lists l_1 and l_2 (i.e., $list(l_1), list(l_2) \in \mathcal{I}[M]$), $first(l_1, x), first(l_2, x) \in \mathcal{I}[M]$ and $rest(l_1, r), rest(l_2, r) \in \mathcal{I}[M]$ implies $l_1 = l_2$;

Functionality if for all lists l (i.e., $list(l) \in \mathcal{I}[M]$), $first(l, x), first(l, y) \in \mathcal{I}[M]$ implies $x = y$, and $rest(l, x), rest(l, y) \in \mathcal{I}[M]$ implies $x = y$.

Towards **Existence of Lists**, we ensure existence of the empty list:

$$\rightarrow list(\text{rdf} : \text{nil}) \quad (30)$$

Given that many rule reasoners operate via materialization of derived facts, we should not fully implement the **Existence of Lists** criterion since materializing all lists certainly entails an infinite process. Instead, we create lists on-demand. The binary *getList* predicate expects a list element x (to be added) and a list l , and creates a new list with *first* element x and *rest* l :

$$getList(x, l) \wedge list(l) \rightarrow \exists l'. list(l') \wedge first(l', x) \wedge rest(l', l) \quad (31)$$

With this interface in place, we replicate example (16) as follows:

$$\begin{aligned} &\rightarrow getList(:\text{tea}, \text{rdf} : \text{nil}) \\ &first(l, : \text{tea}) \wedge rest(l, \text{rdf} : \text{nil}) \rightarrow getList(: \text{chocolate}, l) \\ &first(l, : \text{chocolate}) \wedge rest(l, l') \wedge \\ &first(l', : \text{tea}) \wedge rest(l', \text{rdf} : \text{nil}) \rightarrow getList(: \text{tea}, l) \\ &first(l, : \text{cake}) \wedge rest(l, l') \wedge \\ &first(l', : \text{chocolate}) \wedge rest(l', l'') \wedge \\ &first(l'', : \text{tea}) \wedge rest(l'', \text{rdf} : \text{nil}) \rightarrow tr(: \text{lucy}, : \text{likes}, l) \end{aligned}$$

This rather cumbersome encoding implements **Uniqueness of Lists**. Towards a much simpler encoding, suppose we only take the following rule obtaining the same list as above:

$$\begin{aligned} &\rightarrow \exists l_1, l_2, l_3. list(l_1) \wedge list(l_2) \wedge list(l_3) \wedge \\ &\quad first(l_1, : \text{cake}) \wedge rest(l_1, l_2) \wedge \\ &\quad first(l_2, : \text{chocolate}) \wedge rest(l_2, l_3) \wedge \\ &\quad first(l_3, : \text{tea}) \wedge rest(l_3, \text{rdf} : \text{nil}) \end{aligned} \quad (32)$$

The rule itself can now be combined with other rules as well as the previous one. However, uniqueness can be violated when the restricted chase is used for reasoning. Recall from

Subsection 4.1 that the restricted chase creates new facts (by instantiating rule heads) only if the rule matches are not yet satisfied. Suppose we create an alternative list that is the same as before but replaces `:cake` for `:cookies`:

$$\begin{aligned} \rightarrow \exists l_1, l_2, l_3. & \text{list}(l_1) \wedge \text{list}(l_2) \wedge \text{list}(l_3) \wedge \\ & \text{first}(l_1, \text{:cookies}) \wedge \text{rest}(l_1, l_2) \wedge \\ & \text{first}(l_2, \text{:chocolate}) \wedge \text{rest}(l_2, l_3) \wedge \\ & \text{first}(l_3, \text{:tea}) \wedge \text{rest}(l_3, \text{rdf:nil}) \end{aligned} \quad (33)$$

While the list created by rule (33) is surely distinct from the one created through rule application of (32), they also obtain different sublists. After a restricted chase over rule set $\{(32), (33)\}$ and the empty database, we get two distinct lists l and l' such that $\text{first}(l, \text{:tea})$, $\text{first}(l', \text{:tea})$, $\text{rest}(l, \text{rdf:nil})$, $\text{rest}(l', \text{rdf:nil})$, contradicting **Uniqueness of Lists**. The reason for this is that the application condition of the restricted chase checks whether the head of the rule is already satisfied. If not, the full head is instantiated with (globally) fresh nulls in place of the existentially quantified variables. Our encoding via rule (31) overcomes this issue by step-wise introducing new list elements. If a sublist already exists, rule creation is not triggered redundantly.

Theorem 5.1.

Let \mathcal{D} be a database, Σ a rule set, and \mathcal{I} the restricted chase of Σ and \mathcal{D} . If the only rules in Σ using predicates *list*, *first*, or *rest* in their heads are those of (30) and (31), then \mathcal{I} satisfies (a) **Uniqueness of Lists** and (b) **Functionality**.

Proof.

Functionality follows from the fact that the only rule introducing *first*- and *rest*-atoms is (31) and, thereby, uniquely determines first and rest elements for a list term. Thus, predicates *first* and *rest* are functional.

Regarding **Uniqueness of Lists**, we observe that only rule (31) introduces lists together with their (functional) *first* and *rest* atoms. Hence, if there were two lists l_1 and l_2 with the same first and rest elements, then the respective chase sequence $\mathcal{D}^0 \mathcal{D}^1 \mathcal{D}^2 \dots$ contains a member \mathcal{D}^i in which (without loss of generality) l_1 is contained. Furthermore, there is a later instance \mathcal{D}^j ($j > i$) in which l_2 is not yet contained but is about to be added to \mathcal{D}^{j+1} . But rule (31) is already satisfied in \mathcal{D}^j for the respective first/rest elements. Thus, l_2 will never be instantiated by the restricted chase and can, thus, not be part of the chase. \square

Before we get into the intricates of appending two or more lists, let us briefly show the rules for implementing `list:last` and `list:in` (and `list:member` as the inverse of `list:in`), represented by binary predicate symbols *last* and *isIn*.

$$\text{first}(l, x) \wedge \text{rest}(l, \text{rdf:nil}) \rightarrow \text{last}(x, l) \quad (34)$$

$$\text{rest}(l, l') \wedge \text{last}(y, l') \rightarrow \text{last}(y, l) \quad (35)$$

$$\text{first}(l, x) \rightarrow \text{isIn}(l, x) \quad (36)$$

$$\text{rest}(l, l') \wedge \text{isIn}(l', y) \rightarrow \text{isIn}(l, y) \quad (37)$$

Note, these rules are sufficient for creating all necessary facts to obtain the required results. Regarding list concatenation via `list:append`, we introduce the ternary predicate *append* with the appended list in the first position and the two constituent lists in second and last. First, every list *l* prepended by the empty list yields itself:

$$\text{list}(l) \rightarrow \text{append}(l, \text{rdf:nil}, l) \quad (38)$$

Second, if we append lists l_1 and l_2 to get l_3 (i.e., $\text{append}(l_3, l_1, l_2)$), and x is the first element of l_2 , then l_3 can also be obtained by appending x to l_1 , and the result to the rest of l_2 . Therefore, we need an auxiliary set of rules that appends a single element x to a list l :

$$\text{append}(l_3, l_1, l_2) \wedge \text{first}(l_2, x) \rightarrow \text{getAppendS}(l_1, x) \quad (39)$$

$$\text{getAppendS}(l, x) \wedge \text{rest}(l, l') \rightarrow \text{getAppendS}(l', x) \quad (40)$$

Rule (39) requests a new list that starts with the same elements as l_1 and appends the additional element x . Rule (40) recursively pushes the request through the list. Once, the empty list (`rdf:nil`) is reached, appending the element x is the same as prepending it to `rdf:nil`:

$$\begin{aligned} \text{getAppendS}(\text{rdf:nil}, x) &\rightarrow \text{getList}(x, \text{rdf:nil}) \\ \text{getAppendS}(\text{rdf:nil}, x) \wedge \text{list}(l) &\wedge \\ \text{first}(l, x) \wedge \text{rest}(l, \text{rdf:nil}) &\rightarrow \text{appendS}(l, \text{rdf:nil}, x) \end{aligned} \quad (41)$$

These rules create a fresh list with first element x and rest `rdf:nil` if necessary. Predicate *appendS* stands for *append singleton* and, therefore, $\text{appendS}(l, l', x)$ tells that list l is the result of appending x to list l' . The recursive step is implemented as follows:

$$\begin{aligned} \text{getAppendS}(l, x) \wedge \text{first}(l, y) \wedge \text{rest}(l, l') \wedge \text{appendS}(l'', l', x) &\rightarrow \text{getList}(y, l'') \\ \text{getAppendS}(l, x) \wedge \text{first}(l, y) \wedge \text{rest}(l, l') \wedge \\ \text{appendS}(l'', l', x) \wedge \text{list}(l_\nu) \wedge \text{first}(l_\nu, y) \wedge \text{rest}(l_\nu, l'') &\rightarrow \text{appendS}(l_\nu, l, x) \end{aligned} \quad (42)$$

So if a list l shall be appended by singleton x and we already know that for the rest of l (i.e., l') there is a version with appended x (i.e., l''), then l appended by x is the new list formed by the first element of l (i.e., y) and l'' as rest.

Last, appending two lists can also be requested via rules. Once more, we use a predicate for this request, namely *getAppend*. This predicate is an interface for users (i.e., other rules) to create lists beyond predicate *getList*. Such requests are served by the following rules:

$$\begin{aligned} \text{getAppend}(\text{rdf:nil}, l_2) &\rightarrow \text{append}(l_2, \text{rdf:nil}, l_2) \\ \text{getAppend}(l_1, l_2) \wedge \text{first}(l_1, x) \wedge \text{rest}(l_1, l'_1) &\rightarrow \text{getAppend}(l'_1, l_2) \\ \text{getAppend}(l_1, l_2) \wedge \text{first}(l_1, x) \wedge \text{rest}(l_1, l'_1) \wedge \\ \wedge \text{append}(l_3, l'_1, l_2) &\rightarrow \text{getList}(x, l_3) \\ \text{getAppend}(l_1, l_2) \wedge \text{first}(l_1, x) \wedge \text{rest}(l_1, l'_1) \wedge \\ \text{append}(l_3, l'_1, l_2) \wedge \text{first}(l'_3, x) \wedge \text{rest}(l'_3, l_3) &\rightarrow \text{append}(l'_3, l_1, l_2) \end{aligned}$$

The remove functionality can be implemented in a similar fashion. Note that none of the additionally instantiated rules for list built-ins use predicates *list*, *first*, or *rest* in their heads. Thus, Theorem 5.1 still holds in rule sets using built-in functions. Throughout the rest of this subsection we aim at showing how the framework implements the examples given throughout Subsection 5.1 as well as an example of list usage inside N_3 rules.

5.2.1 Appending lists

First, recall the following N_3 rule (cf. (24)):

$$\{((:a :b)(:c :d))list:append?x\} \Rightarrow \{result :is ?x\}.$$

For the implementation of this rule, we need to make sure the constant lists (the operands of *list:append*) exist:

$$\begin{aligned} & \rightarrow getList(:b, rdf:nil) \\ list(l) \wedge first(l, :b) \wedge rest(l, rdf:nil) & \rightarrow getList(:a, l) \\ & \rightarrow getList(:d, rdf:nil) \\ list(l) \wedge first(l, :d) \wedge rest(l, rdf:nil) & \rightarrow getList(:c, l) \end{aligned}$$

After these rules have been used, the lists in example (24) are guaranteed to exist. Next, we can request to append the two lists matched within the rule:

$$\begin{aligned} & list(l_1) \wedge first(l_1, :a) \wedge rest(l_1, l'_1) \wedge \\ & first(l'_1, :b) \wedge rest(l'_1, rdf:nil) \wedge \\ & list(l_2) \wedge first(l_2, :c) \wedge rest(l_2, l'_2) \wedge \\ & first(l'_2, :d) \wedge rest(l'_2, rdf:nil) \rightarrow getAppend(l_1, l_2) \end{aligned}$$

After this rule we are guaranteed to have all lists in place for implementing our rule.

$$\begin{aligned} & list(l_1) \wedge first(l_1, :a) \wedge rest(l_1, l'_1) \wedge \\ & first(l'_1, :b) \wedge rest(l'_1, rdf:nil) \wedge \\ & list(l_2) \wedge first(l_2, :c) \wedge rest(l_2, l'_2) \wedge \\ & first(l'_2, :d) \wedge rest(l'_2, rdf:nil) \wedge \\ & append(x, l_1, l_2) \rightarrow tr(:result, :is, x) \end{aligned}$$

Second, we reconsider rule (26):

$$\{(?x ?y)list:append(:a :b :c)\} \Rightarrow \{?x :and ?y\}.$$

In this example we need to ensure the resulting list exists. Our rule framework (especially rules (38)–(42)) takes care of dissecting the list into its fragment. Thus, the example rule can be implemented, once the list $(:a :b :c)$ has been created as before, by

$$\begin{aligned} & list(l) \wedge first(l, :a) \wedge rest(l, l') \wedge \\ & first(l', :b) \wedge rest(l', l'') \wedge \\ & first(l'', :c) \wedge rest(l'', rdf:nil) \wedge \\ & append(l, x, y) \rightarrow tr(x, :and, y) \end{aligned}$$

5.2.2 List creation in rules

Last, we consider an N_3 rule that identifies two lists in its body and creates a new list based on some elements identified within the list. The following rule identifies two lists, one with three elements ($?x$, $?y$, and $?z$) and one with two elements ($?a$ and $?b$), and then creates a new list with first element $?y$ and a rest list with the singleton element $?b$:

$$\{ :s :p (?x ?y ?z). :k :l (?a ?b) \} \Rightarrow \{ :h :i (?y ?b) \}. \quad (43)$$

This rule needs splitting into creating the list for the result and then creating the output triple:

$$\begin{aligned} & list(l_1) \wedge first(l_1, x) \wedge rest(l_1, x_l) \wedge \\ & \quad first(x_l, y) \wedge rest(x_l, y_l) \wedge \\ & \quad first(y_l, z) \wedge rest(y_l, \mathbf{rdf} : \mathbf{nil}) \wedge \\ & list(l_2) \wedge first(l_2, a) \wedge rest(l_2, a_l) \wedge \\ & \quad first(a_l, b) \wedge rest(a_l, \mathbf{rdf} : \mathbf{nil}) \wedge \\ & \quad tr(:s, :p, l_1) \wedge tr(:k, :l, l_2) \rightarrow getList(b, \mathbf{rdf} : \mathbf{nil}) \end{aligned}$$

$$\begin{aligned} & list(l_1) \wedge first(l_1, x) \wedge rest(l_1, x_l) \wedge \\ & \quad first(x_l, y) \wedge rest(x_l, y_l) \wedge \\ & \quad first(y_l, z) \wedge rest(y_l, \mathbf{rdf} : \mathbf{nil}) \wedge \\ & list(l_2) \wedge first(l_2, a) \wedge rest(l_2, a_l) \wedge \\ & \quad first(a_l, b) \wedge rest(a_l, \mathbf{rdf} : \mathbf{nil}) \wedge \\ & \quad tr(:s, :p, l_1) \wedge tr(:k, :l, l_2) \wedge \\ & list(l) \wedge first(l, b) \wedge rest(l, \mathbf{rdf} : \mathbf{nil}) \rightarrow getList(y, l) \end{aligned}$$

$$\begin{aligned} & list(l_1) \wedge first(l_1, x) \wedge rest(l_1, x_l) \wedge \\ & \quad first(x_l, y) \wedge rest(x_l, y_l) \wedge \\ & \quad first(y_l, z) \wedge rest(y_l, \mathbf{rdf} : \mathbf{nil}) \wedge \\ & list(l_2) \wedge first(l_2, a) \wedge rest(l_2, a_l) \wedge \\ & \quad first(a_l, b) \wedge rest(a_l, \mathbf{rdf} : \mathbf{nil}) \wedge \\ & \quad tr(:s, :p, l_1) \wedge tr(:k, :l, l_2) \wedge \\ & list(l') \wedge first(l', b) \wedge rest(l', \mathbf{rdf} : \mathbf{nil}) \wedge \\ & \quad list(l) \wedge first(l, y) \wedge rest(l, l') \rightarrow tr(:h, :i, l) \end{aligned}$$

This construction may become complicated if several list built-ins are co-dependent.

5.3 N_3 list predicates as syntactic sugar

As detailed in the previous section, N_3 list predicates can be expressed by means of existential rules if the reasoning is performed under similar premises as the restricted chase. This is particularly interesting in the context of Notation3 Logic: it is well-known that list predicates `list:in`, `list:member`, `list:append`, `list:last`, and `list:remove` introduced in Subsection 5.1 are syntactic sugar, and, therefore, can be expressed using rules in combination with the predicates `rdf:first` and `rdf:rest`. Typically these rules are only written for reasoners supporting backward-chaining, that is, with algorithms performing reasoning starting from the goal and following rules from head to body until some factual evidence is found.²⁰

For better illustration, consider the following N_3 rules implementing `list:append`.²¹

$$\{((\text{?x}) \text{list:append ?x}) \leq \{\} \}. \quad (44)$$

$$\{(\text{?x ?y}) \text{list:append ?z} \leq \{\text{?x rdf:first ?a. ?x rdf:rest ?r.} \quad (45)$$

$$\text{?z rdf:first ?a. ?z rdf:rest ?q.}$$

$$(\text{?r ?y}) \text{list:append ?q} \} \}.$$

If these rules are used in backward-chaining, they get triggered by each execution of a rule containing a triple with the predicate `list:append`. If we, for example, would like to get all instances of the triple `:result :is ?x`, which can be derived by rule (24), the triple in the body of the rule triggers rule (45), to test whether there is evidence for the triple `((:a :b) (:c :d)) list:append ?x`. The rule is again followed in a backwards direction yielding:

$$(:a :b) \text{rdf:first :a; rdf:rest (:b)}. \quad (46)$$

$$\text{?x rdf:first :a; rdf:rest ?q.}$$

$$((:b) (:c :d)) \text{list:append ?q..}$$

The triples in the first line of this example got instantiated according to the semantics of `rdf:first` and `rdf:rest`. This instantiation also caused the triples in the following two lines to partially instantiated. Since there is not enough information to instantiate the triples from the second line, a (backward) reasoner would continue with the last triple which again has `list:append` in predicate position. Rule (45) is called again. This time we retrieve:

$$(:b) \text{rdf:first :b; rdf:rest ()}. \quad (47)$$

$$\text{?q rdf:first :b; rdf:rest ?q2.}$$

$$((:c :d)) \text{list:append ?q2..}$$

²⁰ This kind of reasoning is very similar to Prolog's SLD resolution (Nilsson and Maluszynski 1990).

²¹ N_3 allows rules to be written in a backwards, that is instead of $A = > B$, we write $B <= A$. The backward notation is usually used to indicate that this rule is expected to be reasoned with via backward-chaining. We use this notation here, the model-theoretic semantics keeps being the same as before.

Again following the rules backwards, we can apply rule (44) to get a value for ?q2:

((() (:c :d)) list:append(:c :d).

With this information, we get a binding for ?q in (47):

(:b) rdf:first :b; rdf:rest ().
 (:b :c :d) rdf:first :b; rdf:rest (:c :d).
 ((() (:c :d)) list:append(:c :d)..

Subsequently, we obtain a new binding ?x in (46):

(:a :b) rdf:first :a; rdf:rest (:b).
 (:a :b :c :d) rdf:first :a; rdf:rest (:b :c :d).
 ((:b) (:c :d)) list:append (:b :c :d)..

This produces `:we :get (:a :b :c :d).` as a solution. The backward-chaining process produces triples on-demand: only if a rule premise depends on the information, a backward rule is called to retrieve it, and this allows us to have infinitely large models which we do not materialize during reasoning.

In the N_3 community, this and other examples are normally used to argue that N_3 reasoners should support backwards-reasoning as a way to only produce triples when these are needed to find instances for a goal. Following the findings of the previous subsection, it is not true that we necessarily need backward rules to support triple production on-demand. Instead of writing rule (44) and (45), we can also add the triple `(:a :b) :getAppend (:c :d).` to our initial rule (24). With the following rules, we retrieve the same result as above:

{() :getAppend ?y}}=>{((() ?y) list:append ?y}.
 {?x :getAppend ?y; rdf:rest ?b}}=>{?b :getAppend ?y}.
 {?x :getAppend ?y; rdf:first ?a; rdf:rest ?b}.
 (?b ?y) list:append ?z. ?z2 rdf:first ?a ; rdf:rest ?z }
 =>{(?x ?y) list:append ?z2 }.}

These rules follow the structure of the rules in the previous subsection with the exception that we do not need list constructors in N_3 . If we apply our rules to the fact above, we successively construct the triples `((() (:c :d)) list:append (:c :d).`, `((:b) (:c :d)) list:append (:b :c :d).`, and `((:a :b) (:c :d)) list:append (:a :b :c :d).` These can then directly be used in rules. In more complicated cases, where the arguments of the predicate `list:append` do not appear partially instantiated in rule bodies, the relevant instances of the fact `?x :getAppend ?y.` need to be constructed via rules just as it is the case for existential rules. As N_3 follows the axioms introduced in Subsection 5.1, the first-rest interpretation of RDF lists is equivalent to N_3 's representation of lists as first-class citizens. As a consequence, the rules actually work for all examples introduced above. Similarly, the other list predicates can be written by means of `rdf:first` and `rdf:rest`, and handled via backward-chaining or, alternatively, with some version of the chase.

Note, the backward rules handling `list:append` can be mimicked by splitting them in several forward rules acting on a *getter triple*, that is, a triple causing the production of the required instance of the predicate. We additionally need rules producing the required instances of that getter triple, at which point we need to be careful considering dependencies between triples. However, the mechanism introduced in the previous subsection provides us with a possibility to do reasoning on-demand in a purely forward manner.

6 Evaluation

The considerations provided above allow us to use existential rule reasoners to perform N_3^{\exists} reasoning. We would like to find out whether our finding is of practical relevance, that is whether we can identify datasets on which existential rule reasoners, running on the rule translations, outperform classical N_3 reasoners provided with the original data.

In order to do this we have implemented \mathcal{T} as a python script that takes an arbitrary N_3^{\exists} formula f , constructs its set representation F in PNF, and produces the set of rules $\mathcal{T}(F)$. This script and some additional scripts to translate existential rules (with at most binary predicates) to N_3^{\exists} formulae are available on [GitHub](#). Our implementation allows us to compare N_3 reasoners with existential rule reasoners, performance-wise. As existential rule reasoners we chose VLog (Carral *et al.* 2019), a state-of-the-art reasoning engine designed for working with large piles of input data, and Nemo (Ivliev *et al.* 2023), a recently released rust-based reasoning engine. As N_3 reasoners we chose cwm (Berners-Lee 2009) and EYE (Verborgh and De Roo 2015) which – due to their good coverage of N_3 features – are most commonly used. All experiments have been performed on a laptop with 11th Gen Intel Core i7-1165G7 CPU, 32 GB of RAM, and 1TB disk capacity, running a Ubuntu 22.04 LTS.

6.1 Datasets

We performed our experiments on two datasets: Lubm from the *Chasebench* (Benedikt *et al.* 2017) provides a fixed set of 136 rules and varies in the number of facts these rules are applied; the Deep Taxonomy (DT) benchmark developed for the *WellnessRules* project (Boley *et al.* 2009) consists of one single fact and a varying number of mutually dependent rules.

The *Chasebench* is a benchmarking suite for existential rule reasoning. Among the different scenaria in Chasebench we picked Lubm for its direct compatibility with N_3 : all predicates in Lubm have at most arity 2. Furthermore, Lubm allows for a glimpse on scalability since Lubm comes in different database sizes. We have worked with Lubm 001, 010, and 100, roughly referring to dataset sizes of a hundred thousand, one million and ten million facts. We translated Lubm data and rules into a canonical N_3 format. Predicate names and constants within the dataset become IRIs using the example prefix. An atom like `src_advisor(Student441, Professor8)` becomes the triple `:Student441 :src_advisor :Professor8..` For atoms using unary predicates, like `TeachingAssistant(Student498)`, we treat `:TeachingAssistant` as a class and relate `:Student498` via `rdf:type` to the class. For any atom A , we denote its canonical translation into triple format by $t(A)$. Note this canonical translation only applies to atoms

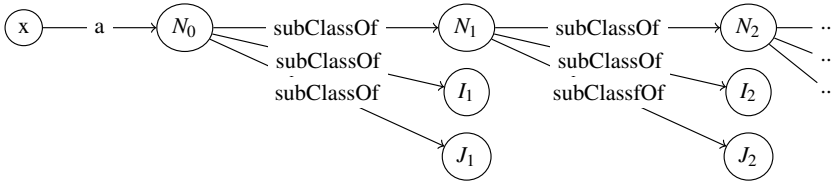


Fig. 2. Structure of the deep taxonomy benchmark.

of unary and binary predicates. For the existential rule

$$\forall \mathbf{x}. B_1 \wedge \dots \wedge B_m \rightarrow \exists \mathbf{z}. H_1 \wedge \dots \wedge H_n$$

we obtain the canonical translation by applying t to all atoms, respecting universally and existentially quantified variables (i.e., universally quantified variables are translated to universal N_3 variables and existentially quantified variables become blank nodes):

$$\{t(B_1) \dots t(B_m)\} \Rightarrow \{t(H_1) \dots t(H_n)\}.$$

All N_3 reasoners have reasoned over the canonical translation of data and rules which was necessary because of the lack of an N_3 version of Lubm. Since we are evaluating VLog's and Nemo's performance on our translation \mathcal{T} , we converted the translated Lubm by \mathcal{T} back to existential rules before reasoning. Thereby, former unary and binary atoms were turned into triples and then uniformly translated to tr -atoms via \mathcal{T} .

The *Deep Taxonomy benchmark* simulates deeply nested RDFS-subclass reasoning.²² It contains one individual which is member of a class. This class is subclass of three other classes of which one again is subclass of three more classes and so on. Figure 2 illustrates this idea. The benchmark provides different depths for the subclass chain and we tested with the depths of 1,000 and 100,000. The reasoning tests for the membership of the individual in the last class of the chain. For our tests, the subclass declarations were translated to rules, the triple `:N0 rdfs:subClassOf :N1` became

$$\{ ?x \ a \ :N0. \} \Rightarrow \{ ?x \ a \ :N1. \}.$$

This translation also illustrates why this rather simple reasoning case is interesting: we have a use case in which we depend on long chains of rules executed after each other. The reasoner EYE allows the user to decide per rule whether it is applied using forward- or backward-reasoning, at least if the head of the rule does not contain blank nodes. For this dataset, we evaluated full backward- and full forward-reasoning, separately.

6.2 Results

Table 1 presents the running times of the four reasoners and additionally gives statistics about the sizes of the given knowledge base (# facts) and the rule set (# rules). For DT we display two reasoning times for EYE, one produced by only forward reasoning (EYE-fw), one for only backward-reasoning (EYE-bw). Note, that for the latter, the reasoner does not produce the full deductive closure of the dataset, but answers a query instead. As Lubm contains rules with blank nodes in their heads, full backward reasoning was not

²² N_3 available at: <http://eulersharp.sourceforge.net/2009/12dtb/>.

Table 1. *Experimental results*

Dataset	# facts	# rules	cwm	EYE-fw	EYE-bw	VLog	Nemo
DT 1000	1	3,001	180 _s	0.1 _s	0.001 _s	1.6 _s	1.7 _s
DT 100000	1	30,001	—	0.3 _s	0.003 _s	—	—
Lubm 001	100,543	136	117.4 _s	3.4 _s	—	0.2 _s	2.4 _s
Lubm 010	1,272,575	136	—	44.8 _s	—	4.3 _s	31.2 _s
Lubm 100	13,405,381	136	—	—	—	47.3 _s	362 _s

possible in that case, the table is left blank. EYE performs much better than VLog and Nemo for the experiments with DT. Its reasoning time is off by one order of magnitude. Conversely, VLog and Nemo could reason over all the Lubm datasets while EYE has thrown an exception after having read the input facts. The reasoning times of VLog are additionally significantly lower than the times for EYE. While Nemo shows a similar runtime on DT as VLog, it is slower on Lubm. However, we may be quite optimistic regarding its progress in runtime behavior, as Nemo already shows better running times on the original Lubm datasets. The reasoner cwm is consistently slower than the other three and from Lubm 010 on. All reasoners tried to find the query answers/deductive closures for at least ten minutes (i.e., — in Table 1 indicates a time-out).

6.3 Discussion

In all our tests we observe a very poor performance of cwm which is not surprising, given that this reasoner has not been updated for some time. The results for EYE, VLog and Nemo are more interesting as they illustrate the different strengths of the reasoners.

For very high numbers of rules compared to the amount of data, EYE performs much better than VLog and Nemo. The good results of 0.1 and 0.3 s can even be improved by using backward reasoning. This makes EYE very well-suited for use cases where we need to apply complex rules on datasets of low or medium size. This could be interesting in decentralized set-ups such as policy-based access control for the Solidproject.²³ On the other hand we see that VLog and Nemo perform best when provided with large datasets and lower numbers of rules. This could be useful use cases involving bigger datasets in the Web like Wikidata or DBpedia.²⁴

From the perspective of this paper, these two findings together show the relevance of our work: we observed big differences between the tools' reasoning times and these differences depended on the use cases. In other words, there are use cases which could benefit from our translation and we thus do not only make the first steps towards having more N_3 reasoners available but also broaden the scope of possible N_3 applications.

²³ <https://solidproject.org/>.

²⁴ <https://www.wikidata.org/> and <https://www.dbpedia.org/>

7 Related work

When originally proposed as a W3C member submission (Berners-Lee and Connolly 2011), the formal semantics of N_3 was only introduced informally. As a consequence, different systems, using N_3 , interpreted concepts like nested formulae differently (Arndt *et al.* 2019). Since then, the relation of N_3 to other Web standards has been studied from a use-case perspective (Arndt 2019) and a W3C Community group has been formed (Woensel *et al.* 2023), which recently published the semantics of N_3 without functions (Arndt and Champin 2023). Even with these definitions, the semantic relation of the logic to other standards, especially outside the Semantics Web, has not been studied thoroughly.

For N_3 's subset RDF, (De bruijn and Heymans 2007) provide a translation to first-order logic and F-Logic using similar embeddings (e.g., a ternary predicate to represent triples) to the ones in this paper, but do not cover rules. Boley (2016) supports N_3 in his RuleML Knowledge-Interoperation Hub providing a translation of N_3 to PSOA RuleML. This can be translated to other logics. But the focus is more on syntax than on semantics.

In Description Logics (DL), rewritings in rule-based languages have their own tradition (see, e.g., Carral and Krötzsch (2020) for a good overview of existing rewritings and their complexity, as well as more references). The goal there is to (1) make state-of-the-art rule reasoners available for DLs and, thereby, (2) use a fragment of a rule language that reflects on the data complexity of the given DL fragment. Also practical tools have been designed to capture certain profiles of the Web Ontology Language (OWL), like the Orel system (Krötzsch *et al.* 2010) and, more recently, DaRLing (Fiorentino *et al.* 2020). To the best of our knowledge, a rewriting for N_3 as presented in this paper did not exist before. Also, existential rule reasoning engines have not been compared to the existing N_3 reasoners.

8 Conclusion

In this paper we studied the close relationship between N_3 rules supporting blank node production and existential rules. N_3 without special features like built-in functions, nesting of rules, or quotation can be directly mapped to existential rules with ternary predicates. In order to show that, we defined a mapping between N_3^{\exists} – N_3 without the aforementioned features – and existential rules. We argued that this mapping and its inverse preserve the equivalence and non-equivalence between datasets. This result allows us to trust the reasoning results when applying the mapping in practice, that is, when (1) translating N_3^{\exists} to existential rules, (2) reasoning within that framework, and (3) using the inverse mapping to transfer the result back into N_3 .

We applied that strategy and compared the reasoning times of the N_3 reasoners cwm and EYE with the existential rule reasoners VLog and Nemo. The goal of that comparison was to find out whether there are use cases for which N_3 reasoning can benefit from the findings on existential rules. We tested the reasoners on two datasets: DT consisting of one single fact and a varying number of mutually dependent rules and Lubm consisting of a fixed number of rules and a varying number of facts. EYE performs better on DT while VLog and Nemo showed their strength on Lubm. We see that as an indication that for use cases of similar nature, that is, reasoning on large numbers of facts, our approach

could be used to improve reasoning times. More generally, we see that reasoners differ in their strengths and that, by providing the reversible translation between N_3^\exists and existential rules, we increase the number of reasoners (partly) supporting N_3 and the range of use cases the logic can support in practice. We see our work as an important step towards fully establishing rule-based reasoning in the Semantic Web.

Of course, N_3 also contains constructs and built-in predicates which are not supported (yet) by our translation. In order to test how extensible our framework is, we provided strategies to also cover lists and their built-in predicates in the translation. Lists were constructed using nulls, which made reasoning with them dependent on the chase applicable. We provided rules to mimic the list-append function of N_3 under the standard chase, which is also implemented in some N_3 reasoners. The existential rules version of the append function came with rules that allow for list construction *on-demand*. This *on-demandness* is very interesting in many situations and, maybe even more important, believed by the N_3 community to only be possible employing backward reasoning. In that sense we also contribute to the ongoing discussion in that community whether the intended reasoning direction should be part of the semantics, which we would clearly argue against.

As many N_3 use cases rely on more powerful N_3 predicates and logical features such as support for graph terms and nested rules, future work should include the extension of our translation towards full coverage of N_3 . As a direct candidate, we would like to investigate the intricate consequences of non-monotonic reasoning in the presence of existentially quantified variables (Ellmauthaler *et al.* 2022). Another direction of future work is to investigate the differences and similarities we found in our evaluation in more detail: while showing differences in their performance, the reasoners produced the exact same result sets (modulo isomorphism) when acting on rules introducing blank nodes. That is, the different reasoning times do not stem from handling of existentially quantified rule heads but from other optimization techniques. Fully understanding these differences will help the N_3 and the existential rules communities to further improve their tools. In that context, it would also be interesting to learn if EYE's capability to combine forward and backward reasoning could improve the reasoning times for data sets including existentially quantified rule heads.

We thus hope that our research on existential N_3 will spawn further investigations of powerful data-centric features in data-intensive rule reasoning as well as significant progress in tool support towards these features. Ultimately, we envision a Web of data and rule exchange, fully supported by the best tools available as converging efforts of the N_3 community, the existential rule reasoning community, and possibly many others.

Acknowledgements

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) in project number 13GW0552B ([KIMEDS](#)), 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).

This is an extended version of a paper presented at the 2023 RuleML+RR Conference. The authors are grateful to Anna Fensel, Ana Ozaki and Ahmet Soylu, the conference program co-chairs, for encouraging us to submit the paper to the journal *Theory and Practice of Logic Programming* and for their help in overseeing the review process.

References

- HAYES, P. 2004. RDF Semantics. W3C Recommendation. Available at <http://www.w3.org/TR/rdf-mt/>
- CYGANIAK, R., WOOD, D. AND LANTHALER, M. 2014. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. Available at <http://www.w3.org/TR/rdf11-concepts/>
- HAYES, P. AND PATEL-SCHNEIDER, P. F. 2014. RDF 1.1 Semantics. W3C Recommendation. Available at <http://www.w3.org/TR/rdf11-mt/>
- ARNDT, D. AND CHAMPIN, P.-A. 2023. Notation3 Semantics. W3C Community Group Report. Available at <https://w3c.github.io/N3/spec/semantics>
- ARNDT, D., DE MEESTER, B., BONTE, P., SCHABALLIE, J., BHATTI, J., DEREUDDRE, W., VERBORGH, R., ONGENAE, F., DE TURCK, F., VAN DE WALLE, R. AND MANNENS, E. 2016. Improving OWL RL reasoning in N3 by using specialized rules. In *Ontology Engineering (OWLED 2015)*. LNCS, Springer, Springer, Cham 93–104
- ARNDT, D. AND MENNICKE, S. 2023a. Notation3 as an existential rule language. In *Proc. Rules and Reasoning - 7th Int. Joint Conf. (RuleML+RR'23)*, A. FENSEL, A. OZAKI, D. ROMAN and SOYLU A., Eds. LNCS, vol. 14244, Springer, Springer, Cham 70–85
- ARNDT, D. AND MENNICKE, S. 2023b. Notation3 as an existential rule language. CoRR abs/2308.07332v1. <https://arxiv.org/abs/2308.07332v1>
- ARNDT, D., SCHRIJVERS, T., DE ROO, J. AND VERBORGH, R. 2019. Implicit quantification made explicit: how to interpret blank nodes and universal variables in Notation3 logic. *Journal of Web Semantics* 58
- ARNDT, D. 2019. *Notation3 as the unifying logic for the semantic web*. Ph.D. thesis, Ghent University.
- BAGET, J.-F., LECLÈRE, M., MUGNIER, M.-L. AND SALVAT, E. 2011. On rules with existential variables: walking the decidability line. *Artificial Intelligence* 175, 9–10, 1620–1654.
- BECKETT, D., BERNERS-LEE, T., PRUD'HOMMEAUX, E. AND CAROTHERS, G. 2014. Turtle - Terse RDF Triple Language. w3c Recommendation
- BENEDIKT, M., KONSTANTINIDIS, G., MECCA, G., MOTIK, B., PAPOTTI, P., SANTORO, D. AND TSAMOURA, E. 2017. Benchmarking the chase. In *Proc. 36th Symposium on Principles of Database Systems (PODS'17)*, E. SALLINGER, J. V. DEN BUSSCHE and F. GEERTS, Eds. ACM, New York, NY, USA 37–52.
- BERNERS-LEE, T. (2000–2009). cwm. W3C. <http://www.w3.org/2000/10/swap/doc/cwm.html>
- BERNERS-LEE, T. AND CONNOLLY, D. (2011). Notation3 (n3): a readable rdf syntax. w3c team submission. <http://www.w3.org/TeamSubmission/n3/>
- BERNERS-LEE, T., CONNOLLY, D., KAGAL, L., SCHARF, Y. AND HENDLER, J. 2008. N3Logic: a logical framework for the world wide web. *Theory and Practice of Logic Programming* 8, 3, 249–269.

- BOLEY, H. (2016) The ruleml knowledge-interoperation hub. In *Rule Technologies. Research, Tools, and Applications*, J. J. ALFERES, L. BERTOSSI, G. GOVERNATORI, P. FODOR and D. ROMAN, Eds. Springer, Cham 19–33
- BOLEY, H., OSMUN, T. M. AND CRAIG, B. L. (2009) Wellnessrules: a web 3.0 case study in ruleml-based prolog-n3 profile interoperation. In *Rule Interchange and Applications*, G. GOVERNATORI, J. HALL and A. PASCHKE, Eds. Springer, Berlin, Heidelberg, pp. 43–52.
- BONTE, P. AND ONGENAE, F. (2023) RoXi: a framework for reactive reasoning, *The Semantic Web: ESWC. 2023 Satellite Events. LNCS*, 13998, Springer, Cham pp. 159–163.
- CALÌ, A., GOTTLOB, G. AND PIERIS, A. (2010) Query answering under non-guarded rules in datalog+/. In Proc. 4th Int. Conf. on Web Reasoning and Rule Systems (RR 2010), P. HITZLER and T. LUKASIEWICZ, Eds. LNCS, vol. 6333, Springer, Berlin, Heidelberg 1–17
- CARRAL, D., DRAGOSTE, I., GONZÁLEZ, L., JACOBS, C., KRÖTZSCH, M. AND URBANI, J. (2019) VLog: a rule engine for knowledge graphs. In Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II), C. GHIDINI et al., Eds. LNCS, vol. 11779, Springer, Berlin, Heidelberg 19–35
- CARRAL, D. AND KRÖTZSCH, M. 2020. Rewriting the description logic ALCHIQ to disjunctive existential rules, Proceedings of the 29th International Joint Conference on Artificial Intelligence. C. BESSIERE, Ed. IJCAI, 1777–1783
- CUENCA GRAU, B., HORROCKS, I., KRÖTZSCH, M., KUPKE, C., MAGKA, D., MOTIK, B. AND WANG, Z. 2013. Acyclicity notions for existential rules and their application to query answering in ontologies. *Journal of Artificial Intelligence Research* 47, 741–808.
- DE BRUIJN, J. AND HEYMANS, S. (2007) Logical foundations of (e)rdf(s): complexity and reasoning, *The Semantic Web*, K. ABERER, K.-S. CHOI, N. NOY, D. ALLEMANG, K.-I. LEE, L. NIXON, J. GOLBECK, P. MIKA, D. MAYNARD, R. MIZOGUCHI, G. SCHREIBER and P. CUDRÉ-MAUROUX, Eds. Springer, Berlin, Heidelberg 86–99,
- DEUTSCH, A., NASH, A. AND REMMEL, J. B. (2008) The chase revisited. In Proc. 27th Symposium on Principles of Database Systems (PODS'08), M. LENZERINI and D. LEMBO, Eds. ACM, New York, NY, USA 149–158
- EBBINGHAUS, H.-D., FLUM, J. AND THOMAS, W. 2021. *Semantics of First-Order Languages*. Springer, Cham 27–57.
- ELLMATHALER, S., KRÖTZSCH, M. AND MENNICKE, S. (2022) Answering queries with negation over existential rules. In Proc. 36th AAAI Conf. on Artif. Intell. (AAAI'22), C. E. BRODLEY and P. STONE, Eds. AAAI Press, 5626–5633
- FAGIN, R., KOLAITIS, P. G., NASH, A. AND POPA, L. 2008. Towards a theory of schema-mapping optimization. In Proc. 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'08), ACM, New York, NY, USA 33–42
- FIorentino, A., ZANGARI, J. AND MANNA, M. 2020. DaRLing: a datalog rewriter for OWL 2 RL ontological reasoning under SPARQL queries. *Theory and Practice of Logic Programming* 20,6, 958–973.
- GORLA, D. AND NESTMANN, U. (2016). Full abstraction for expressiveness: history, myths and facts. *Math. Struct. Comput. Sci.* 26, 4, 639–654
- HARTH, A. AND KÄFER, T. 2018. Rule-based programming of user agents for linked data. In Proc. 11th Int. Workshop on Linked Data on the Web at the Web Conference (WWW'27), CEUR-WS, T. Berners-Lee, S. Capadisli, S. Dietze, A. Hogan, K. Janowicz, and J. Lehmann, Eds. Aachen.
- 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 Proc. 39th Int. Conf. on Logic Programming (ICLP'23), E. PONTELLI, S. COSTANTINI, C. DODARO, S. A. GAGGL, R. CALEGARI, A. S. D'AVILA GARCEZ, F. FABIANO, A. MILEO, A. RUSSO and F. TONI, Eds. EPTCS, vol. 385, 333–335

- KRÖTZSCH, M., MARX, M. AND RUDOLPH, S. (2019) The power of the terminating chase. In Proc. 22nd Int. Conf. on Database Theory (ICDT'19), P. BARCELÓ and M. CALAUTTI, Eds. LIPIcs, vol. 127, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:17
- KRÖTZSCH, M., MEHDI, A. AND RUDOLPH, S. 2010. Orel: Database-driven reasoning for OWL 2 profiles. In Proc. 23rd Int. Workshop on Description Logics (DL'10), V. HAARSLEV, D. TOMAN and G. WEDDELL, Eds. vol. 573. CEUR-WS.org, CEUR Workshop Proceedings and Aachen 114–124
- MOTIK, B., CUENCA GRAU, B., HORROCKS, I., WU, Z.FOKOUE, A. AND LUTZ, C. 2009. OWL 2 Web Ontology Language: Profiles. W3C Recommendation. Available at <http://www.w3.org/TR/owl2-profiles/>
- NILSSON, U. AND MALUSZYSKI, J. 1990. *Logic, Programming and PROLOG*. John Wiley & Sons, Inc., USA.
- PARROW, J. (2016). General conditions for full abstraction. *Math. Struct. Comput. Sci.* 26, 4, 655–657.
- PICHLER, R., SALLINGER, E. and SAVENKOV, V. 2011. Relaxed notions of schema mapping equivalence revisited. In Proc. 14th International Conference on Database Theory (ICDT'11), ACM, New York, NY, USA, 90–101
- VERBORGH, R., ARNDT, D., VAN HOECKE, S., DE ROO, J., MELS, G., STEINER, T. AND GABARRÓ, J. 2017. The pragmatic proof: hypermedia API composition and execution. *Theory and Practice of Logic Programming* 17, 1, 1–48.
- VERBORGH, R. AND DE ROO, J. 2015. Drawing conclusions from linked data on the web: the EYE reasoner. *IEEE Software* 32,5, 23–27.
- WOENSEL, W. V., ARNDT, D., CHAMPIN, P.-A., TOMASZUK, D. AND KELLOGG, G. 2023. Notation3 Language. W3C Community Group Report. Available at <https://w3c.github.io/N3/reports/20230703/>
- WOENSEL, W. V. AND HOCHSTENBACH, P. 2023. Notation3 builtin functions. W3C Community Group Report. Available at <https://w3c.github.io/N3/reports/20230703/builtins.html>