TPLP: Page 1–8. © The Author(s), 2021. Published by Cambridge University Press 2021
doi:10.1017/xxxxx

Existential Notation3 Logic

DÖRTHE ARNDT
Computational Logic Group, TU Dresden, Germany

STEPHAN MENNICKE
Knowledge-Based Systems Group, TU Dresden, Germany

submitted xx xx xxxx; revised xx xx xxxx; accepted xx xx xxxx

Abstract

In this paper, we delve into Notation3 Logic (N3), an extension of 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.

1 Introduction

Notation3 Logic (N3) (cf. Woensel et al. (2023); Berners-Lee et al. (2008)) 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. Facilitated by reasoners like cwm (Berners-Lee (2009)), Data-Fu (Harth and Käfer (2018)), or EYE (Verborgh and De Roo (2015)), N3 rules directly consume and produce RDF graphs. This makes N3 well-suited for rule exchange on the Web. N3 supports the introduction of new blank nodes through rules, that is, if a blank node appears in the head[1] 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

[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 ← body", forward rules: "body → head").

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 jen3[2] or RoXi (Bonte and Ongenae (2023)) aim at improving the situation, the number of fast N3 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 N3, 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); Calì 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 N3 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 N3 reasoning via existential rule technologies. We discuss how the framework can be extended to also support lists – a feature of N3 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 N3 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 N3 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 N3 and existential rule formulae, and discuss how these are connected, in Sect. 2. In Sect. 3 we provide a more formal definition of Existential N3 ($N3^{\exists}$), introduce its semantics and discuss its properties. We then formally introduce existential rules, provide the mapping from $N3^{\exists}$ into this logic, and prove its truth-preserving properties in Sect. 4. N3 lists and the built-ins associated with them are introduced as N3 primitives as well as their existential rule translations are subject to Sect. 5. In Sect. 6 we discuss our implementation and provide an evaluation of the different reasoners. Related work is presented in Sect. 7. We conclude our discussion in Sect. 8. Furthermore, the code needed for reproducing our experiments is available on GitHub (`https://github.com/smennicke/n32rules`).

This article is an extended and revised version of our work (cf. Arndt and Mennicke (2023)) presented at Rules and Reasoning – 7th International Joint Conference

[2] `https://github.com/william-vw/jen3`

(RuleML+RR) 2023. Compared to the original paper, we include full proofs to all theorems and lemmas. Furthermore, we extend our considerations by N3 lists and respective built-ins (cf. Sect. 5).

2 Motivation

N3 has been inroduced as a rule-based extension of RDF. As in RDF, N3 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.` (1)

means[3] that *"lucy knows tom"*. Sets of triples are interpreted as their conjunction. Like RDF, N3 supports blank nodes, usually starting with `_:`, which stand for (implicitly) existentially quantified variables. The statement

`:lucy :knows _:x.` (2)

means *"there exists someone who is known by lucy"*. N3 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}.` (3)

means that *"everyone known by Lucy also knows her"*. Furthermore, N3 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"}.` (4)

means *"everyone knowing Tom knows* someone *whose name is* Tom*"*.

This last example shows, that N3 supports rules concluding the *existence* of certain terms which makes it easy to express them as *existential rules*. An existential rule is a first-order sentence of the form

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

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 N3 formulae (1)–(4) again and represent them as existential rules. To allow for the full flexibility of N3 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. Sect. 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 N3 formula

[3] We omit name spaces for brevity.

in (2) contains an existentially quantified variable and can, thus, be encoded as

$$\rightarrow \exists x.\ tr(\texttt{:lucy}, \texttt{:knows}, x) \tag{6}$$

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

The implication of (3) has

$$\forall x.\ tr(\texttt{:lucy}, \texttt{:knows}, x) \rightarrow tr(x, \texttt{:knows}, \texttt{:lucy}) \tag{7}$$

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

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

but not in $$\mathcal{K}_1 = \{tr(\texttt{:lucy}, \texttt{:knows}, \texttt{:tom})\}$$

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 $tr(\texttt{:tom}, \texttt{:knows}, \texttt{:lucy})$ 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:

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

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

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

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

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

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 Sect. 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 N3 and existential rules. First, we give a brief formal account of the two languages and then provide a correct translation function from N3 to existential rules.

`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 N3$^\exists$

3 Existential N3

In the previous section we introduced essential elements of N3, namely triples and rules. N3 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 N3 excluding them, called *existential N3* ($N3^\exists$).[4] We base our definitions on so-called *simple N3 formulae* (Arndt, Dörthe 2019, Chapter 7), these are N3 formulae which do not allow for nesting.

3.1 Syntax

$N3^\exists$ 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 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 $\{\{,\},\texttt{=>},\texttt{.}\}$ are mutually disjoint, we call $\mathfrak{A} := C \cup U \cup E \cup \{\{,\},\texttt{=>},\texttt{.}\}$ an *N3 alphabet*. Fig. 1 provides the syntax of $N3^\exists$ over $\mathfrak{A}$.

$N3^\exists$ fully covers RDF – RDF formulae are conjunctions of atomic formulae – but allows literals and blank nodes to occur in subject, predicate, and object position. On top of the triples, it supports rules containing existential and universal variables. Note, that the syntax allows rules having new universal variables in their head like for example

```
{:lucy :knows :tom}=>{?x :is :happy}.                    (9)
```

which results in a rule expressing *"if lucy knows tom, everyone is happy"*. This implication is problematic: Applied on triple (1), it yields `?x :is :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 $N3^\exists$ rules to *well-formed implications* which rely on *components*. A *component* of a formula or an expression is an N3 term which does not occur nested in a rule. More formally, let f be a formula or an expression over an alphabet $\mathfrak{A}$. The set *comp*(f) of *components* of f is defined as:

- If f is an atomic formula or a triple expression of the form $t_1\ t_2\ t_3.$, $comp(f) = \{t_1, t_2, t_3\}$.

[4] This fragment is expressive enough to support basic use cases like user-defined ontology mapping. Here it is important to note that RDF lists can be expressed using first-rest pairs.

- If f is an implication of the form `{`e_1`}=>{`e_2`}.`, then $comp(f) = \{$`{`e_1`}`$,$`{`e_2`}`$\}$.
- If f is a conjunction of the form $f_1 f_2$, then $comp(f) = comp(f_1) \cup comp(f_2)$.

A rule `{e`$_1$`}=>{e`$_2$`}.` is called *well-formed* if $(comp(\mathtt{e}_2) \setminus comp(\mathtt{e}_1)) \cap U = \emptyset$. For the remainder of this paper we assume all implications to be well-formed.

3.2 Semantics

In order to define the semantics of $N3^{\exists}$ we first note, that in our fragment of N3 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

```
:lucy :knows _:y. _:y :likes :cake.
```
(10)

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 seperated 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 N3 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.

For formula $f =$ `_:x :p :o. {_:x :b :c}=>{_:x :d :e}.`, substitution σ and `_:x` $\in \mathrm{dom}(\sigma)$, we get: $f\sigma = \sigma($`_:x`$)$`:p :o. {_:x :b :c}=>{_:x :d :e}.`[5] We use the substitution to define the semantics of $N3^{\exists}$ which additionally makes use of *N3 interpretations* $\mathfrak{I} = (\mathfrak{D}, \mathfrak{a}, \mathfrak{p})$ consisting of (1) the domain of $\mathfrak{I}$, $\mathfrak{D}$; (2) $\mathfrak{a} : C \to \mathfrak{D}$ called the object function; (3) $\mathfrak{p} : \mathfrak{D} \to 2^{\mathfrak{D}\times\mathfrak{D}}$ called the predicate function.

Just as the function IEXT in RDF's simple interpretations, see Hayes (2004), N3'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* $N3^{\exists}$, let $\mathfrak{I} = (\mathfrak{D}, \mathfrak{a}, \mathfrak{p})$ be an N3 interpretation. For an $N3^{\exists}$ formula f:

1. If $W = \mathrm{comp}(f) \cap E \neq \emptyset$, then $\mathfrak{I} \models f$ iff $\mathfrak{I} \models f\mu$ for some substitution $\mu : W \to C$.
2. If $\mathrm{comp}(f) \cap E = \emptyset$:

[5] Note that the semantics of *simple formulae* on which $N3^{\exists}$'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 $N3^{\exists}$, we simplified here.

(a) If f is an atomic formula $t_1\, t_2\, t_3$, then $\mathfrak{I} \models t_1\, t_2\, t_3.$ iff $(\mathfrak{a}(t_1), \mathfrak{a}(t_3)) \in \mathfrak{p}(\mathfrak{a}(t_2))$.
(b) If f is a conjunction $f_1 f_2$, then $\mathfrak{I} \models f_1 f_2$ iff $\mathfrak{I} \models f_1$ and $\mathfrak{I} \models f_2$.
(c) If f is an implication, then $\mathfrak{I} \models$ `{`e_1`}=>{`e_2`}` iff $\mathfrak{I} \models e_2\sigma$ if $\mathfrak{I} \models e_1\sigma$ for all substitutions σ on the universal variables $\text{comp}(\mathsf{e}_1) \cap U$ 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 N3 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 $N3^{\exists}$ 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.

Piece Normal Form $N3^{\exists}$ 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 seperately. Below, we therefore define the so-called *Piece Normal Form* (PNF) for $N3^{\exists}$ formulae and show that each such formula f is equivalent to a set of sub-formulae Φ (i.e., $\Phi \equiv f$) 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 (10), 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:

Lemma 1 (Pieces) *Let $f = f_1 f_2$ be an $N3^{\exists}$ 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 $W = \text{comp}(f) \cap E \neq \emptyset$:
(⇒) If $\mathfrak{I} \models f$ then there exists a substitution $\mu : \text{comp}(f) \cap E \to C$ 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 $\mathfrak{I} \models f_2$.
(⇐) If $\mathfrak{I} \models f_1$ and $\mathfrak{I} \models f_2$, then there exist two substitutions $\mu_1 : \text{comp}(f_1) \cap E \to C$ and $\mu_2 : \text{comp}(f_2) \cap E \to C$ 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 $\mu : \text{comp}(f) \cap E \to C$ 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 $F = \{f_1, f_2, \ldots, f_n\}$ 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 :likes :cake}=>{:cake :is :good}.` becomes `{?y :likes :cake}=>{:cake :is :good}.` Note that both rules have the same meaning, namely *"if someone likes cake, then cake is good."*. We generalize that:

Lemma 2 (Eliminating Existentials) *Let* $f = \texttt{\{}e_1\texttt{\}=>\{}e_2\texttt{\}}$ *and* $g = \texttt{\{}e_1'\texttt{\}=>\{}e_2\texttt{\}}$ *be* $N3^\exists$ *implications such that* $e_1' = e_1\sigma$ *for some injective substitution* $\sigma : \mathrm{comp}(e_1) \cap E \to U \setminus \mathrm{comp}(e_1)$ *of the existential variables of* e_1 *by universals. Then:* $f \equiv g$

Proof: We first note that $comp(f) \cap E = \emptyset$ and $\mathrm{comp}(g) \cap E = \emptyset$ 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 $\nu : (\mathrm{comp}(e_1') \cup \mathrm{comp}(e_2)) \cap U \to C$ 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 $((\mathrm{comp}(e_1) \cup \mathrm{comp}(e_2)) \cap U) \subset ((\mathrm{comp}(e_1') \cup \mathrm{comp}(e_2)) \cap U)$, we know that $\mathrm{comp}(e_1\nu) \cap U = \emptyset$. 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 $\nu : (\mathrm{comp}(e_1) \cup \mathrm{comp}(e_2)) \cap U \to C$ 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 $\mu : \mathrm{comp}(e_1\nu) \cap E \to C$ such that $\mathfrak{M} \models (e_1\nu)\mu$. With that we define a substitution $\nu' : (\mathrm{comp}(e_1) \cup \mathrm{comp}(e_2)) \cap U \to C$ as follows: $\nu' : U \to C$ as follows:

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

With that substitution we get $\mathfrak{M} \models e_1'\nu'$ but $\mathfrak{M} \not\models e_2\nu'$ and thus $\mathfrak{M} \not\models g$. □

For a rule f we call the formula f' in which all existentials occurring in its body are replaced by universals following Lemma 2 the *normalized* version of the rule. We call an $N3^\exists$ formula f *normalized*, if all rules occurring in it as conjuncts are normalized. This allows us to introduce the *Piece Normal Form*:

Theorem 3 (Piece Normal Form) *For every well-formed* $N3^\exists$ *formula* f*, there exists a set* $F = \{f_1, f_2, \ldots, f_k\}$ *of* $N3^\exists$ *formulae such that* $F \equiv \{f\}$ *and* F *is in* piece normal form *(PNF). That is, all* $f_i \in F$ *are normalized formulae and* $k \in \mathbb{N}$ *is the maximal number such that for* $1 \leq i, j \leq k$*,* $comp(f_i) \cap comp(f_j) \cap E \neq \emptyset$ *implies* $i = j$*. If* f_i *(*$1 \leq i \leq k$*) is a conjunction of atomic formulae, we call* f_i *an* atomic piece*.*

Proof: The claim follows immediately from Lemma 1 and Lemma 2. □

Since the piece normal form F of $N3^\exists$ 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

Without loss of generality, we translate sets F of $N3^\exists$ formulae in PNF (cf. Theorem 3) to sets of existential rules $\mathcal{T}(F)$. As a preliminary step, we introduce the language of existential rules formally. Later on, we explain and define the translation function that has already been sketched in Sect. 2. The section closes with a correctness argument, establishing a strong relationship between existential rules and $N3^\exists$.

4.1 Foundations of Existential Rule Reasoning

For existential rules, we also consider a first-order vocabulary, consisting of constants ($\mathbf{C}$) and variables ($\mathbf{V}$), and additionally so-called (labeled) nulls ($\mathbf{N}$)[6]. As already mentioned in Sect. 2, we use the same set of constants as N3 formulae, meaning $\mathbf{C} = C$. Furthermore, let $\mathbf{P}$ be a set of *relation names*, where each $p \in \mathbf{P}$ comes with an arity $ar(p) \in \mathbb{N}$. $\mathbf{C}$, $\mathbf{V}$, $\mathbf{N}$, and $\mathbf{P}$ are countably infinite and pair-wise disjoint. We use the ternary relation name $tr \in \mathbf{P}$ to encode N3 triples in Sect. 2. If $p \in \mathbf{P}$ and $t_1, t_2, \ldots, t_{ar(p)}$ is a list of terms (i.e., $t_i \in \mathbf{C} \cup \mathbf{N} \cup \mathbf{V}$), $p(t_1, t_2, \ldots, t_{ar(p)})$ is called an *atom*. We often use $\mathbf{t}$ to summarize a term list like $t_1, \ldots, t_n$ ($n \in \mathbb{N}$), 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 $\mathcal{A}$ and an instance $\mathcal{I}$, we call a function h from the terms occurring in $\mathcal{A}$ to the terms in $\mathcal{I}$ a *homomorphism from* $\mathcal{A}$ *to* $\mathcal{I}$, denoted by $h : \mathcal{A} \to \mathcal{I}$, if (1) $h(c) = c$ for all $c \in \mathbf{C}$ (occurring in $\mathcal{A}$), and (2) $p(\mathbf{t}) \in \mathcal{A}$ implies $p(h(\mathbf{t})) \in \mathcal{I}$. If any homomorphism from $\mathcal{A}$ to $\mathcal{I}$ exists, write $\mathcal{A} \to \mathcal{I}$. Please note that if n is a null occurring in $\mathcal{A}$, then $h(n)$ may be a constant or null.

For an *(existential) rule* $r\colon \forall \mathbf{x}, \mathbf{y}.\ \varphi[\mathbf{x}, \mathbf{y}] \to \exists \mathbf{z}.\ \psi[\mathbf{y}, \mathbf{z}]$ (cf. (5)), rule bodies ($\mathsf{body}(r)$) and heads ($\mathsf{head}(r)$) will also be considered as sets of atoms for a more compact representation of the semantics. Let r be a rule and $\mathcal{I}$ an instance. We call a homomorphism $h : \mathsf{body}(r) \to \mathcal{I}$ a *match for* r *in* $\mathcal{I}$. A match h is *satisfied for* r *in* $\mathcal{I}$ if there is an extension $h^\star$ of h (i.e., $h \subseteq h^\star$) such that $h^\star(\mathsf{head}(r)) \subseteq \mathcal{I}$. 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 set Σ and database $\mathcal{D}$, we call an instance $\mathcal{I}$ 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 $r \in \Sigma$. We say that two rule sets Σ_1 and Σ_2 are *equivalent*, denoted $\Sigma_1 \leftrightarrows \Sigma_2$, iff for all instances $\mathcal{I}$, $\mathcal{I} \models \Sigma_1$ iff $\mathcal{I} \models \Sigma_2$.

Labeled nulls play the role of fresh constants without further specification, just like blank nodes in RDF or N3. The chase is a family of algorithms that soundly produces models of rule sets by continuously applying rules for unsatisfied matches. Rule heads are then instantiated and added to the instance. Existentially quantified variables are replaced by (globally) fresh nulls in order to facilitate for arbitrary constants. More formally, we call a sequence $\mathcal{D}^0\mathcal{D}^1\mathcal{D}^2\ldots$ a *chase sequence of* Σ *and* $\mathcal{D}$ 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., $h : \mathsf{body}(r) \to \mathcal{D}^{i-1}$ is an unsatisfied match and $\mathcal{D}^i = \mathcal{D}^{i-1} \cup \{h^\star(\mathsf{head}(r))\}$ for an extension $h^\star$ of h). The *chase of* Σ *and* $\mathcal{D}$ is the limit of a chase sequence $\mathcal{D}^0\mathcal{D}^1\mathcal{D}^2\ldots$, i.e., $\bigcup_{i\geq 0} \mathcal{D}^0$. Although the chase is not guaranteed to terminate, it always produces a (possibly infinite) model[7] (cf. Deutsch et al.).

For an alternative equivalence relation between rule sets, we could have equally considered equality of ground models (i.e., null-free ones). Let us define $\leftrightarrows_g$ as follows: $\Sigma_1 \leftrightarrows_g \Sigma_2$ if, and only if, for each ground instance $\mathcal{I}$, $\mathcal{I} \models \Sigma_1$ iff $\mathcal{I} \models \Sigma_2$. The following

[6] We choose here different symbols to disambiguate between existential rules and N3, although vocabularies partially overlap.

[7] 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 set. Up to homomorphisms, universal models are the smallest among all models.

lemma, showing that $\leftrightarrows = \leftrightarrows_g$, helps simplifying the proofs concerning the correctness of our translation function later on.

Lemma 4 *$\leftrightarrows$ and $\leftrightarrows_g$ coincide.*

PROOF: Of course, $\leftrightarrows \subseteq \leftrightarrows_g$ holds since since the set of all ground models of a rule set is a subset of all models of a rule set. For the converse direction, let Σ_1 and Σ_2 be rule sets, such that $\Sigma_1 \leftrightarrows_g \Sigma_2$. Towards a contradiction, assume $\Sigma_1 \not\leftrightarrows \Sigma_2$. Then there is a model $\mathcal{M}$ of Σ_1, such that $\mathcal{M} \not\models \Sigma_2$, implying that for some rule $r \in \Sigma_2$ there is a match h in $\mathcal{M}$ but for no extension $h^\star$, we get $h^\star(\mathsf{head}(r)) \subseteq \mathcal{M}$. As $\Sigma_1 \leftrightarrows_g \Sigma_2$, $\mathcal{M}$ cannot be a ground instance and, thus, contains at least one null. **Claim:** Then there is a ground instance $\mathcal{M}_g$, such that $\mathcal{M}_g \models \Sigma_1$ and $\mathcal{M}_2 \not\models \Sigma_2$. But then $\mathcal{M}_g$ constitutes a counterexample to the assumption that $\Sigma_1 \leftrightarrows_g \Sigma_2$.

It remains to be shown that the claim actually holds. Our plan is to construct $\mathcal{M}_g$ from $\mathcal{M}$ by replacing every null n in $\mathcal{M}$ by a fresh constant c_n. Unfortunately, there might be not enough constants since $\mathcal{M}$ may already use all countably infinite constant $c \in \mathbf{C}$. Therefore, we take a little detour: the set of used constants might be infinite in $\mathcal{M}$, but the constants used in the rule sets Σ_1 and Σ_2 is finite. Therefore, we will create an instance $\mathcal{M}''$ from $\mathcal{M}$ by replacing all constants c not part of Σ_1 or Σ_2 by fresh nulls n_c. Unfortunately, once again, $\mathcal{M}$ may already use all nulls $n \in \mathbf{N}$. So we have to take another detour from $\mathcal{M}$ to $\mathcal{M}'$ as follows: Let $\gamma : \mathbf{N} \to \mathbb{N}$ be a (necessarily injective) enumeration of $\mathbf{N}$. Define $\eta : \mathbf{C} \cup \mathbf{N} \to \mathbf{C} \cup \mathbf{N}$ by (1) $\eta(c) := c$ for all $c \in \mathbf{C}$ and (2) $\eta(n) := \eta^{-1}(2 \cdot \eta(n))$. Then apply η to $\mathcal{M}$ to obtain $\mathcal{M}'$. Note, for each number $k \in \mathbb{N}$, $\eta^{-1}(2k+1)$ is not a null in $\mathcal{M}'$. It holds that $\mathcal{M} \models \Sigma$ iff $\mathcal{M}' \models \Sigma$ since η is an isomorphism between $\mathcal{M}$ and $\mathcal{M}'$. Recall that isomorphic models preserve all first-order sentences (see, e.g., Ebbinghaus et al.). Hence, $\mathcal{M}' \models \Sigma_1$ and $\mathcal{M}' \not\models \Sigma_2$.

Now we construct $\mathcal{M}''$ from $\mathcal{M}'$ by function ω mapping the terms occurring in $\mathcal{M}'$ 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 $\omega(n) = n$ if n otherwise. ω exists because there are countably infinitely many nulls not used by $\mathcal{M}'$. Note that ω is injective and $\omega(\mathcal{M}') = \mathcal{M}''$ uses only finitely many constants. Once again we show that $\mathcal{M}' \models \Sigma$ iff $\mathcal{M}'' \models \Sigma$ for arbitrary rule sets Σ, implying that $\mathcal{M}'' \models \Sigma_1$ and $\mathcal{M}'' \not\models \Sigma_2$. Let $r \in \Sigma$ with match h in $\mathcal{M}'$. If h is satisfied in $\mathcal{M}'$, then there is an extension $h^\star$, such that $h^\star(\mathsf{head}(r)) \subseteq \mathcal{M}'$. By definition of ω and, thus, the construction of $\mathcal{M}''$, $\omega \circ h$ is a match for r in $\mathcal{M}''$ and $\omega \circ h^\star$ its extension with $\omega \circ h^\star(\mathsf{head}(r)) \subseteq \mathcal{M}''$. The converse direction uses the the same argumentation, now from $\mathcal{M}''$ to $\mathcal{M}'$, using the fact that ω is injective.

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

We are going to use the auxiliary equivalence $\leftrightarrows_g$ in later proofs.

4.2 The Translation Function from N3 to Existential Rules

The translation function $\mathcal{T}$ maps sets $F = \{f_1, \ldots, f_k\}$ of $N3^{\exists}$ formulae in PNF to sets of rules Σ. Before we go into the details of the translation for every type of piece, we consider an auxiliary function $\mathbb{T} : C \cup E \cup U \to \mathbf{C} \cup \mathbf{V}$ mapping N3 terms to terms in our rule language (cf. previous subsection):

$$\mathbb{T}(t) := \begin{cases} v^{\forall}_{\mathtt{x}} & \text{if } t = \mathtt{?x} \in U \\ v^{\exists}_{\mathtt{y}} & \text{if } t = \mathtt{\_:y} \in E \\ t & \text{if } t \in C, \end{cases}$$

where $v^{\forall}_{\mathtt{x}}, v^{\exists}_{\mathtt{y}} \in \mathbf{V}$ and $t \in \mathbf{C}$ (i.e., we assume $C \subseteq \mathbf{C}$). While variables in N3 belong to either E or U, this separation is lost under function $\mathbb{T}$. For enhancing readability of subsequent examples, the identity of the variable preserves this information by using superscripts $\exists$ and $\forall$. 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.

Translating Atomic Pieces. If f_i is an atomic piece, it has the form $f_i = g_1\ g_2\ \ldots\ g_l$ 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 $\mathcal{T}(f_i) = \{\to \exists \mathbf{z}.\ tr(\mathbb{T}(g_1)) \wedge tr(\mathbb{T}(g_2)) \wedge \ldots \wedge tr(\mathbb{T}(g_l))\}$, where $\mathbb{T}(g_j) = \mathbb{T}(t^1_j), \mathbb{T}(t^2_j), \mathbb{T}(t^3_j)$ if $g_j = t^1_j\ t^2_j\ t^3_j$ and $\mathbf{z}$ is the list of translated existential variables (via $\mathbb{T}$) from existentials occurring in f. For example, the formula in (10) constitutes a single piece $f_{(10)}$ which translates to a set containing the rule

$$\to \exists v^{\exists}_{\mathtt{y}}.\ tr(\mathtt{:lucy}, \mathtt{:knows}, v^{\exists}_{\mathtt{y}}) \wedge tr(v^{\exists}_{\mathtt{y}}, \mathtt{:likes}, \mathtt{:cake}).$$

Translating Rules. For f_i being a rule $\{e_1\}\texttt{=>}\{e_2\}$ we also obtain a single rule. Recall that the PNF ensures all variables of e_1 to be universals and all universal variables of e_2 to also occur in e_1. If $e_1 = g^1_1\ g^2_1 \cdots g^m_1$ and $e_2 = g^1_2\ g^2_2 \cdots g^n_2$, $\mathcal{T}(f_i) = \{\forall \mathbf{x}.\ \bigwedge_{j=1}^{m} tr(\mathbb{T}(g^j_1)) \to \exists \mathbf{z}.\ \bigwedge_{j=1}^{n} tr(\mathbb{T}(g^j_2))\}$ where $\mathbf{x}$ and $\mathbf{z}$ are the lists of translated universals and existentials, respectively. Applying the translation to the N3 formula in (4), which is a piece according to our definitions, we obtain again a singleton set, now containing the rule

$$\forall v^{\forall}_{\mathtt{x}}.\ tr(v^{\forall}_{\mathtt{x}}, \mathtt{:knows}, \mathtt{:tom}) \to \exists v^{\exists}_{\mathtt{y}}.\ tr(v^{\forall}_{\mathtt{x}}, \mathtt{:knows}, v^{\exists}_{\mathtt{y}}) \wedge tr(v^{\exists}_{\mathtt{y}}, \mathtt{:name}, \mathtt{"Tom"}),$$

which is the same rule as (8) up to a renaming of (bound) variables, called α-conversion (cf. Ebbinghaus et al.).

Translating Sets. For the set $F = \{f_1, f_2, \ldots, f_k\}$ of $N3^{\exists}$ 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 of its input.

The correctness argument for $\mathcal{T}$ splits into *soundness* – whenever we translate two equivalent $N3^{\exists}$ formulae, their translated rules turn out to be equivalent as well – and *completeness* – formulae that are not equivalent are translated to rule sets that are not equivalent. Although the different formalisms have quite different notions of models, models of a translated rule sets $\mathcal{M}$ can be converted into models of the original N3 formula by using a Herbrand argument.

Lemma 5 *Let F be a set of $N3^{\exists}$ formulae in PNF and $\mathcal{M}$ be a ground instance. Define* the canonical interpretation of $\mathcal{M}$ *by* $\mathfrak{I}(\mathcal{M}) = (C, \mathfrak{a}, \mathfrak{p})$ *such that*

- $\mathfrak{a}(t) := t$ *for all* $t \in C$ *and*
- $\mathfrak{p}(p) := \{(s, o) \mid tr(s, p, o) \in \mathcal{M}\}$ *for all* $p \in C$.

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

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

Base: For $k = 1$, $F = \{f\}$ and f is either (a) an atomic piece or (b) a rule. In case (a), $\mathcal{T}(F) = \mathcal{T}(f) = \{\to \exists\mathbf{z}.\ \bigwedge_{i=1}^{n} tr(s_i, p_i, o_i)\}$. Every model of $\mathcal{T}(F)$ satisfies its single rule, meaning that if $\mathcal{M}$ is a model, there is a homomorphism $h^\star$ from $\mathcal{A} = \{tr(s_i, p_i, o_i) \mid 1 \le i \le n\}$ to $\mathcal{M}$. Then $\mathfrak{I}(\mathcal{M}) = (C, \mathfrak{a}, \mathfrak{p})$ with $(s_i, o_i) \in \mathfrak{p}(p_i)$ for all $i \in \{1, \ldots, n\}$. In case $\text{comp}(f) \cap E = W$ is nonempty, define $\mu : W \to C$ alongside $h^\star$ (i.e., $\mu(\_\texttt{:y}) = h^\star(v^\exists_{\texttt{y}})$ for each $\_\texttt{:y} \in W$). For each atomic formula $g_j = s_j\ \ p_j\ \ o_j$ of f, we get $\mathfrak{M} \models g_j\mu^c$ since $tr(h^\star(s_j), h^\star(p_j), h^\star(o_j)) \in \mathcal{M}$ implies $(h^\star(s_j), h^\star(o_j)) \in \mathfrak{p}(h^\star(p_j))$ and, thus, $(\mathfrak{a}(s_j\mu^c), \mathfrak{a}(o_j\mu^c)) \in \mathfrak{p}(\mathfrak{a}(p_j\mu^c))$. This argument holds for every atomic formula g_j of f, implying $\mathfrak{M} \models F$. The converse direction uses the same argumentation backwards, constructing $h^\star$ from μ.
In case (b), we have $F = \{f\}$ with $f = \{e_1\}\texttt{=>}\{e_2\}$ and $\mathcal{T}(F) = \{\forall\mathbf{x}.\ \varphi \to \exists\mathbf{z}.\ \psi\}$ where φ and ψ are translated conjunctions from e_1 and e_2. Let $\mathfrak{I}(\mathcal{M})$ be a model of F. To show that $\mathcal{M}$ is a model of $\mathcal{T}(F)$, it suffices to prove, for each match h of the rule, the existence of an extension $h^\star$ (of h), such that $h^\star(\psi) \subseteq \mathcal{M}$. Let h be a match for the body of the rule and the body of the rule is a conjunction of atoms. Then σ with $\sigma(\texttt{?x}) = h(v^\forall_{\texttt{x}})$ for each universal variable in e_1 is a substitution, such that $\mathfrak{I}(\mathcal{M}) \models e_1\sigma^c$. In order to prove this claim, let $s\ \ p\ \ o$ be a triple in e_1. Hence, $tr(s, p, o) \in \varphi$ and, by the choice of h, $tr(h(s), h(p), h(o)) \in \mathcal{M}$. This implies that $(h(s), h(o)) \in \mathfrak{p}(h(p))$, which also implies $(s\sigma^c, o\sigma^c) \in \mathfrak{p}(o\sigma^c)$. 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. As $\mathfrak{I}(\mathcal{M})$ is a model of f, there is a substitution $\mu : \text{comp}(e_2) \cap E \to C$, such that $\mathfrak{I}(\mathcal{M}) \models e_2\sigma^c\mu^c$. Define $h^\star := h \cup \{w \mapsto \mu(w) \mid w \in \text{comp}(e_2) \cap E\}$. It holds that $h^\star$ satisfies match h since for each atomic formula $s_i\ \ p_i\ \ o_i$ of e_2, we get $\mathfrak{a}(\mu(\sigma(s_i)), \mu(\sigma(o_i))) \in \mathfrak{p}(\mathfrak{a}(\mu(\sigma(p_i))))$ implying $tr(\mu(\sigma(s_i)), \mu(\sigma(p_i), \mu(\sigma(o_i)))) \in \mathcal{M}$ and $h^\star(\mathbb{T}(x)) = \mu(\sigma(x))$ $(x \in \{s_i, p_i, o_i\})$ 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, h is a satisfied match via $h^\star$. As before, the construction can be inverted, obtaining μ from $h^\star$ and σ from h, which completes the proof for this case.

Step: Let $F = \{f_1, f_2, \ldots, f_k, f_{k+1}\}$ be a set of $N3^\exists$ formulae in PNF. By induction hypothesis, $\mathcal{M}$ is a model of $\mathcal{T}(\{f_1, f_2, \ldots, f_k\})$ iff $\mathfrak{I}(\mathcal{M})$ is a model of $\{f_1, f_2, \cdots, f_k\}$. Also by induction hypothesis, $\mathcal{M}$ is a model of $\mathcal{T}(\{f_{k+1}\})$ iff $\mathfrak{I}(\mathcal{M})$ is a model of $\{f_{k+1}\}$. Thus, $\mathcal{M}$ is a model of $\mathcal{T}(F)$ iff it is a model of $\mathcal{T}(\{f_1 f_2 \cdots f_k\})$ and of $\mathcal{T}(\{f_{k+1}\})$ iff $\mathfrak{I}(\mathcal{M})$ is a model of $\{f_1 f_2 \cdots f_k\}$ and of $\{f_{k+1}\}$ iff $\mathfrak{I}(\mathcal{M})$ is a model of F.

Our correctness proof also considers completeness since, otherwise, a trivial translation function would have sufficed: Let $\mathcal{T}_0$ be a function mapping all $N3^\exists$ formulae to the empty rule set (i.e., $\emptyset$): All equivalent $N3^\exists$ formulae are mapped to the same (i.e., equivalent) rule set $\emptyset$, but also pairs of non-equivalent formulae yield the same translation. Having the stronger criterion between N3 and existential rules allows us to soundly use the translation function $\mathcal{T}$ in practice.

Theorem 6 *For PNFs F and G of $N3^{\exists}$ formulae, $F \equiv G$ iff $\mathcal{T}(F) \leftrightarrows \mathcal{T}(G)$.*

Proof: We prove soundness and completeness separately.

Soundness: If $F \equiv G$, then $\mathcal{M}$ is a ground model of $\mathcal{T}(F)$ iff $\mathfrak{I}(\mathcal{M})$ is a model of F (by Lemma 5) iff $\mathfrak{I}(\mathcal{M})$ is a model of G (by assumption) iff $\mathcal{M}$ is a ground model of $\mathcal{T}(G)$ (again by Lemma 5). Hence, $\mathcal{T}(F) \leftrightarrows_g \mathcal{T}(G)$ which implies $\mathcal{T}(F) \leftrightarrows \mathcal{T}(G)$ (by Lemma 4).

Completeness: If $F \not\equiv G$, then there is an interpretation $\mathfrak{M} = (\mathfrak{D}, \mathfrak{a}, \mathfrak{p})$, such that (w.l.o.g.) $\mathfrak{M} \models F$ and $\mathfrak{M} \not\models G$. **Claim:** By using a Herbrand argument once more, there is an interpretation $\mathfrak{M}_g = (C, \mathfrak{b}, \mathfrak{q})$ such that (1) for each set of $N3^{\exists}$ formulae H in PNF, $\mathfrak{M}_g \models H$ if, and only if, $\mathfrak{M} \models H$ and (2) there is an instance $\mathcal{M}_g$ such that $\mathfrak{I}(\mathcal{M}_g) = \mathfrak{M}_g$. If the claim holds, we obtain that $\mathfrak{M} \models F$ iff $\mathfrak{M}_g \models F$ (by (1)) iff $\mathcal{M}_g \models \mathcal{T}(G)$ for ground $\mathcal{M}_g$ with $\mathfrak{I}(\mathcal{M}_g) = \mathfrak{M}_g$ (by Lemma 5). $\mathcal{M}_g \not\models \mathcal{T}(G)$ since, otherwise, $\mathfrak{M} \models G$ (using the previous chain of arguments backwards) which contradicts our choice of $\mathfrak{M}$. Thus, $\mathcal{T}(F) \not\leftrightarrows_g \mathcal{T}(G)$ implying $\mathcal{T}(F) \not\leftrightarrows \mathcal{T}(G)$ (by Lemma 4).

The claim remains to be shown: For $\mathfrak{M} = (\mathfrak{D}, \mathfrak{a}, \mathfrak{p})$, define $\mathfrak{M}_g = (C, \mathfrak{b}, \mathfrak{q})$ by (a) $\mathfrak{b}(c)$ is the identity on C and (b) $\mathfrak{q}(p) := \{(s, o) \mid (\mathfrak{a}(s), \mathfrak{a}(o)) \in \mathfrak{p}(\mathfrak{a}(p))\}$ for all $p \in C$. Instance $\mathcal{M}_g := \{tr(s, p, o) \mid (s, o) \in \mathfrak{q}(p)\}$ has property (2) (i.e., $\mathfrak{I}(\mathcal{M}_g) = \mathfrak{M}_g$). We show that $\mathfrak{M} \models H$ iff $\mathfrak{M}_g \models H$ for arbitrary sets H of $N3^{\exists}$ formulae in PNF by induction on the number of pieces $|H| = k$.

Base: There are two cases to consider for $k = 1$ and $H = \{f\}$: f is an atomic piece $g_1 \ \cdots \ g_l$ and f is an N3 rule $\{e_1\}$=>$\{e_2\}$. In case f is an atomic piece, then $\mathfrak{M} \models f$ iff $\mathfrak{M} \models f\mu^c$ for some $\mu : \text{comp}(f) \cap E \to C$ iff for each atomic formula $g = s \ p \ o$ in f, $(\mathfrak{a}(\mu(s)), \mathfrak{a}(\mu(o))) \in \mathfrak{p}(\mathfrak{a}(\mu(p)))$ (all by the semantics of $N3^{\exists}$) iff $(\mathfrak{b}(\mu(s)), \mathfrak{b}(\mu(o))) \in \mathfrak{q}(\mathfrak{b}(\mu(p)))$ for each atomic formula g of f (by construction of $\mathfrak{M}_g$) iff $\mathfrak{M}_g \models f\mu^c$ iff $\mathfrak{M}_g \models f$ (by the semantics of $N3^{\exists}$).

In case f is an N3 rule $\{e_1\}$=>$\{e_2\}$, $\mathfrak{M} \models f$ iff for each substitution $\sigma : U \to C$ with $\mathfrak{M} \models e_1\sigma^c$, there is a substitution $\mu : \text{comp}(e_2) \cap E \to C$ such that $\mathfrak{M} \models e_2\sigma^c\mu^c$. Let $\sigma : U \to C$ and $\mu : \text{comp}(e_2) \cap E \to C$ be substitutions. $\mathfrak{M} \models e_1\sigma^c$ iff $(\mathfrak{a}(\sigma(s)), \mathfrak{a}(\sigma(o))) \in \mathfrak{p}(\mathfrak{a}(\sigma(p)))$ for each atomic formula $s \ p \ o$ in e_1 (by the semantics of $N3^{\exists}$) iff $(\mathfrak{b}(\sigma(s)), \mathfrak{b}(\sigma(o))) \in \mathfrak{q}(\mathfrak{b}(\sigma(q)))$ for each atomic formula $s \ p \ o$ of e_1 iff $\mathfrak{M}_g \models e_1\sigma^c$. The same argumentation can be used to argue for $\mathfrak{M} \models e_2\sigma^t\mu^c$ iff $\mathfrak{M}_g \models e_2\sigma^c\mu^c$. Thus, for each $\sigma : U \to C$ for which $\mathfrak{M} \models e_1\sigma^c$ implying a substitution $\mu : \text{comp}(e_2) \cap E \to C$ such that $\mathfrak{M} \models e_2\sigma^c\mu^c$, we get that $\mathfrak{M}_g \models e_1\sigma^c$ and $\mathfrak{M}_g \models e_2\sigma^c\mu^c$, and vice versa.

Step: For $H = \{f_1, \ldots, f_k, f_{k+1}\}$, the induction hypothesis applies to $H' = \{f_1, \ldots, f_k\}$ and $H'' = \{f_{k+1}\}$, meaning that $\mathfrak{M} \models H$ iff $\mathfrak{M} \models H'$ and $\mathfrak{M} \models H''$ (by Lemma 1) iff $\mathfrak{M}_g \models H'$ and $\mathfrak{M}_g \models H''$ (by induction hypothesis) iff $\mathfrak{M}_g \models H$ (by Lemma 1). □

Beyond the correctness of $\mathcal{T}$, we have no further guarantees. As $N3^{\exists}$ reasoning does not necessarily stop, there is no requirement for termination of the chase over translated rule sets. We expect that the similarity between $N3^{\exists}$ and existential rules allows for the adoption of sufficient conditions for finite models, for instance, by means of acyclicity (see Cuenca Grau et al. for a survey).

5 Reasoning with Lists

So far, we discussed $N3^{\exists}$ as a fragment of N3 which can directly be mapped to existential rules. In this section, we detail how $N3^{\exists}$ and our translation to existential rules can be extended towards supporting lists. Lists is a very important concept in N3. 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 our section by discussing different ways to implement list functions in N3.

5.1 N3 Lists

Before introducing them formally, we explain the role of lists in Notation3 Logic by examples. N3 is based on RDF, but, in contrast to RDF, N3 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).                (11)
```

Stating that lucy likes cake, chocolate and tea. If we understand the above as an example of RDF-turtle Beckett and Berners-Lee (2008), the list-notation `( )` is syntactic sugar for:

```
:lucy :likes _:l1.                                   (12)
_:l1 rdf:first :cake; rdf:rest _:l2.
_:l2 rdf:first :chocolate; rdf:rest _l3.
_:l3 rdf:first :tea; rdf:rest rdf:nil.
```

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 clarified further.

In N3, the list in example 11 itself is understood as a resource and not just as syntactic sugar for example 12. The predicates `rdf:first` and `rdf:rest` have a more specific meaning, they stand for the relation between a list and its first element, respectively, a list and its rest list, that is the list, we retrive if we remove the first element. The rule

```
{ (:a :b :c) rdf:first ?x; rdf:rest ?y}=>{?x :and ?y}.    (13)
```

for example, yields

```
:a :and (:b :c).                                     (14)
```

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

If we define the semantics in a naive way, N3'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).                 (15)
```

If we apply the N3 rule

```
{?x :likes ?z. ?y :likes ?z}=>{?x :sharesPreferencesWith ?y}.     (16)
```

on triple 15 and 11, we retrieve[8] that:

```
:lucy :sharesPreferencesWith :tom.     (17)
```

Now, we replace triple 15 by the first-rest combination it stands for, namely

```
:ben :likes _:k1.     (18)
_:k1 rdf:first :cake; rdf:rest _:k2.
_:k2 rdf:first :chocolate; rdf:rest _:k3.
_:k3 rdf:first :tea; rdf:rest rdf:nil.
```

If we again apply rule 16, but this time on the list representations 12 and 18, it is not immedeately evident that we get triple 17 as a result. The lists are represented by the blank nodes `_:l1` and `_:k1`, and it is not immediately evident that these refer to the same list. The original informal N3 specification overcomes the problems caused by the different representations by providing three axioms (Berners-Lee et al. (2008); Berners-Lee and Connolly (2011)) which need to hold for N3 lists:

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 equality[9] (`=`):
`{?L1 rdf:first ?X; rdf:rest ?R. ?L12 rdf:first ?X; rdf:rest ?R.} => {?L1 = ?L2}.`

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

The first axiom guarantees that there is no new informaion added when translating from the native list notation (example 11) to the first-rest noation (example 12). 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 N3 lists and list predicates by means of existential rules. We will come back to that in Section 5.2.

Before introducing the non-basic list predicates, we provide the syntax and semantics of the extension of $N3^{\exists}$ with basic lists. We start with the syntax and extend the grammar provided in Figure 1 as follows:

[8] Of course, we retrive more, namely, that Tom shares preferences with Lucy and that both share preferences with themselves.
[9] Note that this equality is not that same kind of equality that the N3 predicate `log:equalTo` provides. The latter is on syntax and not on semantics level.

- the set `t` of term additionally contains the empty list `()` and the concept `(l)` of list terms, with

```
l ::=
        t
        l t
```

- the set `n` of N3 terms additionally contains the concept `(k)` of N3 list terms, with

```
k ::=
        n
        k n
```

We furthermore need to extend the *application* of a substitution introduced in Section 3.2 by $(\mathtt{t}_1 \dots \mathtt{t}_n)\sigma = (\mathtt{t}_1\sigma \dots \mathtt{t}_n\sigma)$ if $x = (\mathtt{t}_1 \dots \mathtt{t}_n)$ is a list, and the object function $\mathfrak{a}$ of N3 interpretations $\mathfrak{I} = (\mathfrak{D}, \mathfrak{a}, \mathfrak{p})$ as follows: If $\mathtt{t} = (\mathtt{t}_1 \dots \mathtt{t}_n)$ then $\mathfrak{a}(\mathtt{t}) = (\mathfrak{a}(\mathtt{t}_1) \dots \mathfrak{a}(\mathtt{t}_n))$. If $\mathtt{t} = ()$ then $\mathfrak{a}(\mathtt{t}) = ()$.

Note, that with our extension the domain $\mathfrak{D}$ of a model for a graph containing a list term also needs to contain a list of domain elements. However, the amount of lists necessarily contained in $\mathfrak{D}$ is determined by the number of lists which can be produced using the alphabet. It is countable and does not depend on $\mathfrak{D}$ itself. If $\mathfrak{D}$ contains all lists which can be constructed using the interpretations of the N3 terms, then axiom 1 (existence of lists) is fulfilled.

We finish the definition of the semantics of $N3^{\exists}$ with basic lists as follows:

Given an N3 alphabet which contains the list constants `rdf:first` and `rdf:rest`, and an N3 Interpretation $\mathfrak{I} = (\mathfrak{D}, \mathfrak{a}, \mathfrak{p})$. We say that $\mathfrak{I}$ is a model according to the simple list semantics of a formula ϕ, written as $\mathfrak{I} \models_{sl} \phi$ iff $\mathfrak{I} \models \phi$ and for triples containing `rdf:first` or `rdf:rest` in predicate position:

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

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`, N3 contains a few more special predicates which make it easier to handle lists. In our list-extension of $N3^{\exists}$ we include[10] `list:last`, `list:in`, `list:member`, `list:append`, and `list:remove`: `list:last` is used to relate a list to its last argument[11] (`(:a :b :c) list:last :c.`), `list:member` defines the relation between a list and its member (`(:a :b :c) list:member :a, :b, :c.`), `list:in` is the inverse of `list:member` (`:b 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)) 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

[10] 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`.
[11] We give an example of one or more triples (in brackets) which need to be true after each explanation.

the second argument of the subject list of the first argument of the subject list (`((:a :b :a :c) :a) list:remove (:b :c).`).

Note, that N3 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)) list:append ?x}=>{:result :is ?x}.        (19)
```

a reasoner will retrieve

```
:result :is (:a :b :c :d).        (20)
```

But we can also write a rule like

```
{(?x ?y) list:append (:a :b :c)}=>{?x :and ?y}.        (21)
```

which yields

```
() :and (:a :b :c).
(:a) :and (:b :c).
(:a :b) :and ( :c).
(:a :b :c) :and ().
```

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

```
{((:a :b) ?y) list:append (:a :b :c)}=>{:we :get ?y}.        (22)
```

for example, we get

```
:we :get (:c).        (23)
```

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 list:last :c}=>{:we :get ?x}.        (24)
```

where all possible lists having `:c` as last element need to be produced, reasoning engines normally ignore the rule.[12] 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 N3 alphabet which contains the list constants `rdf:first`, `rdf:rest`, `list:in`, `list:member`, `list:append`, `list:last` and `list:remove`, and an N3 Interpretation $\mathfrak{I} = (\mathfrak{D}, \mathfrak{a}, \mathfrak{p})$. We say that $\mathfrak{I}$ is a model according to list semantics of a formula ϕ, written as $\mathfrak{I} \models_l \phi$ iff $\mathfrak{I} \models_{sl} \phi$ and the following conditions hold:

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

[12] To be more precise, the N3 specification comes with so-called argument-modes specifying which arguments need to be instatntiated for the predicate to be called, see also Woensel and Hochstenbach (2023).

- $\mathfrak{I} \models_l s$ `list:remove` o iff $\mathfrak{a}(s) = ((a_1 \ldots a_n)\, b)$ and $\mathfrak{a}(o) = (a_i)_{a_i \neq b}$

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

5.2 Implementing N3 Lists in Exitential 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 *tr* predicate for lists. Instead of $\mathit{tr}(x, \texttt{rdf:first}, y)$ we use an auxiliary binary predicate *first* and write $\mathit{first}(x, y)$. Similarly we use $\mathit{rest}(x, y)$ to denote $\mathit{tr}(x, \texttt{rdf:rest}, y)$. 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 $\mathcal{M}$ of rule set Σ and database $\mathcal{D}$ satisfies

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

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

Towards **existence of lists**, we ensure the necessary existence of the empty list:

$$\rightarrow \quad \mathit{list}(\texttt{rdf:nil}) \tag{25}$$

Given that many rule reasoners operate via materialization of derived facts, we cannot fully implement the criterion of **existence of lists** 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* list l:

$$\mathit{getList}(x, l) \wedge \mathit{list}(l) \quad \rightarrow \quad \exists l'.\ \mathit{list}(l') \wedge \mathit{first}(l', x) \wedge \mathit{rest}(l', l) \tag{26}$$

With this interface in place, we replicate example (11) as follows:

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

This rather cumbersome encoding achieves our goal to implement the **uniquness of lists** criterion. Towards a much simpler encoding, suppose we only take the following rule to obtain the same list as above:

$$\begin{aligned}
\rightarrow \exists l_1, l_2, l_3. \quad & \mathit{list}(l_1) \wedge \mathit{list}(l_2) \wedge \mathit{list}(l_3) \wedge\\
& \mathit{first}(l_1, \texttt{:cake}) \wedge \mathit{rest}(l_1, l_2) \wedge\\
& \mathit{first}(l_2, \texttt{:chocolate}) \wedge \mathit{rest}(l_2, l_3) \wedge\\
& \mathit{first}(l_3, \texttt{:tea}) \wedge \mathit{rest}(l_3, \texttt{rdf:nil})
\end{aligned} \tag{27}$$

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 Sect. 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`:

$$\begin{aligned}\to \exists l_1, l_2, l_3.\quad & \mathit{list}(l_1) \wedge \mathit{list}(l_2) \wedge \mathit{list}(l_3)\wedge \\ & \mathit{first}(l_1, \texttt{:cookies}) \wedge \mathit{rest}(l_1, l_2)\wedge \\ & \mathit{first}(l_2, \texttt{:chocolate}) \wedge \mathit{rest}(l_2, l_3)\wedge \\ & \mathit{first}(l_3, \texttt{:tea}) \wedge \mathit{rest}(l_3, \texttt{rdf:nil})\end{aligned} \tag{28}$$

While the list created by rule (28) is surely distinct from the one created through rule application of (27), they also obtain different sublists. After a restricted chase over rule set $\{(27), (28)\}$ and the empty database, we get two distinct lists l and l' such that $\mathit{first}(l, \texttt{:tea})$, $\mathit{first}(l', \texttt{:tea})$, $\mathit{rest}(l, \texttt{rdf:nil})$, $\mathit{rest}(l', \texttt{rdf:nil})$, contradicting the uniqueness criterion. 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 (26) overcomes this issue by step-wise introducing new list elements. If a sublist already exists, rule creation is not triggered unnecessarily.

Theorem 7 *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* (25) *and* (26)*, 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 (26) and, thereby, determines uniquely first and rest elements for a list term. Thus, predicates *first* and *rest* are functional.

Regarding uniqueness, we observe that, once more, only rule (26) introduces lists together with their functional atoms with *first* and *rest* predicate. Hence, if there were two lists l_1 and l_2 with the same first and rest elements, then the respective chase sequence $\mathcal{D}^0\mathcal{D}^1\mathcal{D}^2\ldots$ 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 $\mathcal{D}^{j+1}$. Now, rule (26) is already satisfied 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. □

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*.

$$\mathit{first}(l, x) \wedge \mathit{rest}(l, \texttt{rdf:nil}) \to \mathit{last}(x, l) \tag{29}$$

$$\mathit{rest}(l, l') \wedge \mathit{last}(y, l') \to \mathit{last}(y, l) \tag{30}$$

$$\mathit{first}(l, x) \to \mathit{isIn}(l, x) \tag{31}$$

$$\mathit{rest}(l, l') \wedge \mathit{isIn}(l', y) \to \mathit{isIn}(l, y) \tag{32}$$

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:

$$\mathit{list}(l) \rightarrow \mathit{append}(l, \mathtt{rdf:nil}, l) \tag{33}$$

Second, if we append lists l_1 and l_2 to get l_3 (i.e., $\mathit{append}(l_3, l_1, l_2)$), 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:

$$\begin{aligned} \mathit{append}(l_3, l_1, l_2) \wedge \mathit{first}(l_2, x) &\rightarrow \mathit{getAppendS}(l_1, x) & (34)\\ \mathit{getAppendS}(l, x) \wedge \mathit{rest}(l, l') &\rightarrow \mathit{getAppendS}(l', x) & (35)\end{aligned}$$

Rule (34) requests a new list that starts with the same elements as l_1 and appends the additional element x. Rule (35) 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`:

$$\begin{aligned} \mathit{getAppendS}(\mathtt{rdf:nil}, x) &\rightarrow \mathit{getList}(x, \mathtt{rdf:nil})\\ \mathit{getAppendS}(\mathtt{rdf:nil}, x) \wedge \mathit{list}(l) \wedge & \\ \mathit{first}(l, x) \wedge \mathit{rest}(l, \mathtt{rdf:nil}) &\rightarrow \mathit{appendS}(l, \mathtt{rdf:nil}, x)\end{aligned} \tag{36}$$

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

$$\begin{aligned} \mathit{getAppendS}(l, x) \wedge \mathit{first}(l, y) \wedge \mathit{rest}(l, l') \wedge \mathit{appendS}(l'', l', x) &\rightarrow \mathit{getList}(y, l'')\\ \mathit{getAppendS}(l, x) \wedge \mathit{first}(l, y) \wedge \mathit{rest}(l, l') \wedge & \\ \mathit{appendS}(l'', l', x) \wedge \mathit{list}(l_\nu) \wedge \mathit{first}(l_\nu, y) \wedge \mathit{rest}(l_\nu, l'') &\rightarrow \mathit{appendS}(l_\nu, l, x)\end{aligned} \tag{37}$$

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 rules 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:

$$\begin{aligned} \mathit{getAppend}(\mathtt{rdf:nil}, l_2) &\rightarrow \mathit{append}(l_2, \mathtt{rdf:nil}, l_2) & (38)\\ \mathit{getAppend}(l_1, l_2) \wedge \mathit{first}(l_1, x) \wedge \mathit{rest}(l_1, l_1') &\rightarrow \mathit{getAppend}(l_1', l_2) & (39)\\ \mathit{getAppend}(l_1, l_2) \wedge \mathit{first}(l_1, x) \wedge \mathit{rest}(l_1, l_1') \wedge & & (40)\\ \wedge \mathit{append}(l_3, l_1', l_2) &\rightarrow \mathit{getList}(x, l_3) & (41)\\ \mathit{getAppend}(l_1, l_2) \wedge \mathit{first}(l_1, x) \wedge \mathit{rest}(l_1, l_1') \wedge & & (42)\\ \mathit{append}(l_3, l_1', l_2) \wedge \mathit{first}(l_3', x) \wedge \mathit{rest}(l_3', l_3) &\rightarrow \mathit{append}(l_3', l_1, l_2) & (43)\end{aligned}$$

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 7 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 Sect. 5.1 as well as an example of list usage inside N3 rules.

Appending Lists. First, recall the following N3 rule (cf. (19)):

```
{((:a :b) (:c :d)) list:append ?x}=>{:result :is ?x}.
```

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

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

After these rules have been used, the lists in example (19) are guaranteed to exist. Next, we can request to append the two lists matched within the rule:

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

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

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

Second, we reconsider rule (21):

```
{(?x ?y) list:append (:a :b :c)}=>{?x :and ?y}.
```

In this example we need to ensure the resulting list exists. Our rule framework (especially rules (33)–(37)) takes care of disecting the list into its fragment. Thus, the example rule can be implemented, once the list `(:a :b :c)` has been created as before, by

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

List Creation in Rules. Last, we consider an N3 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 cwhole examplereates 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)}=>{:h :i (?y ?b)}.          (44)
```

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

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

DA: Ich würde hier noch den Hinweis geben, dass soetwas komplizierter werden kann, wenn Listen im Body voneinander abhängen, z.B. remove und dann append auf derselben Liste, oder eben zweimal append, wie im body

5.3 N3 list predicates as syntactic sugar

As detailed in the previous section, N3 list predicates can be expressed by means of existential rules if reasoning produces the restricted chase. This is particularly interesting in the context of Notation3 Logic: it is well known that the list predicates `list:in`, `list:member`, `list:append`, `list:last` and `list:remove` introduced in subsection 5.1 are only syntactic sugar, and can be expressed using rules in combination with the predicates `rdf:first` and `rdf:rest`. But typically these rules are only written for reasoners supporting backward-chaining, that is, with algorithms which perform reasoning starting from the goal and following rules from head to body till some evidence is found.[13]

[13] This kind of reasoning is very similar to Prolog's SLD resolution (e.g., Nilsson and Maluszynski (1990)).

To better illustrate this, we provide the rules for `list:append` as an example:[14]

```
{(() ?x) list:append ?x}<={ }.                                    (45)
{(?x ?y) list:append ?z}<={?x rdf:first ?a. ?x rdf:rest ?r.        (46)
                           ?z rdf:first ?a. ?z rdf:rest ?q.
                           (?r ?y) 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 (19), the triple in the body of the rule triggers rule (46), 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) rdf:first :a; rdf:rest (:b).                              (47)
?x rdf:first :a; rdf:rest ?q.
((:b) (:c :d)) 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 istantiation also caused that the triples in the following two lines to partly be instantiated. There is not enough information to instantiate the triples from the second line, a reasoner would thus continue with the last triple which again has `list:append` in predicate position. Rule (46) gets called again. This time we retrive:

```
(:b) rdf:first :b; rdf:rest ().                                   (48)
?q rdf:first :b; rdf:rest ?q2.
(() (:c :d)) list:append ?q2. .
```

Again following the rules backwards, we can apply rule (45) to get a value for `?q2`:

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

With this information, we get a binding for `?q` in equation (48):

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

This, again, provides a new binding `?x` in equation (47):

```
(: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.

[14] N3 allows rules to be written in a backwards, that is instead of `A=>B.` we write `B<=A.` The backward notation is usally 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.

In the N3 community, this and other examples are normally used to argue that N3 reasoners should support backwards-reasoning as a way to only produce triples when these are needed to find instances of a goal. Following the findingss of the previous subsection, it is not true that we necessarily need backward rules to support triple production on-demand. Instaed of writing rule (45) and (46), we can also add the triple `(:a :b) :getAppend (:c :d).` to our initial rule (19). 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 section with the exception that we do not need list constructors in N3. If we apply our rules to the fact above, we sucessively 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 partly 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 N3 follows the axioms introduced in section 5.1, the first-rest interpretation of RDF lists is equilvalent to N3's representation of lists as first-class citizens. As a consequence of that, 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.

We make the observation, that 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, here we need to be careful with dependencies between triples. But the mechanism introduced in the previuos 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 $N3^{\exists}$ 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 N3 reasoners provided with the original data.

In order to do this we have implemented $\mathcal{T}$ as a python script that takes an arbitrary $N3^{\exists}$ 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 $N3^{\exists}$ formulae are available on GitHub. Our implementation allows us to compare N3 reasoners with existential rule reasoners, performance-wise. As existential rule reasoners we chose VLog (Carral et al. (2019)), a state-of-the-art

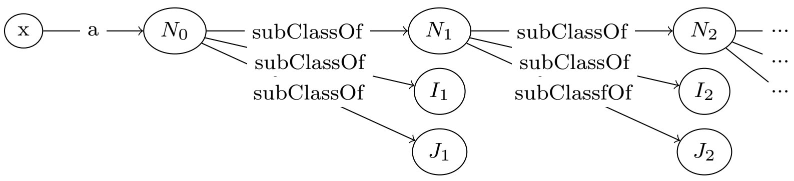

Fig. 2. Structure of the DEEP TAXONOMY benchmark.

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 N3 reasoners we chose cwm (Berners-Lee (2009)) and EYE (Verborgh and De Roo (2015)) which – due to their good coverage of N3 features – are most commonly used. All experiments have been performed on a laptop with 11th Gen Intel Core i7-1165G7 CPU, 32GB 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 N3: 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 N3 format. Predicate names and constants within the dataset become IRIs using the example prefix. An atom like $\mathit{src\_advisor}(\mathit{Student441}, \mathit{Professor8})$ becomes the triple `:Student441 :src_advisor :Professor8.`. For atoms using unary predicates, like $\mathit{TeachingAssistent}(\mathit{Student498})$, we treat `:TeachingAssistent` 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 of unary and binary predicates. For the existential rule

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

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 N3 variables and existentially quantified variables become blank nodes):

$$\texttt{\{}t(B_1)\texttt{.} \cdots t(B_m)\texttt{.}\texttt{\}=>\{}t(H_1)\texttt{.} \cdots t(H_n)\texttt{.}\texttt{\}.}$$

All N3 reasoners have reasoned over the canonical translation of data and rules which was necessary because of the lack of an N3 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 $\mathcal{T}$.

Table 1. *Experimental Results*

Dataset	**# facts**	**# rules**	**cwm**	**EYE-fw**	**EYE-bw**	**VLog**	**Nemo**
DT 1000	1	3001	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

The *Deep Taxonomy benchmark* simulates deeply nested RDFS-subclass reasoning[15]. 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.}=>{ ?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 haeds, full backward reasoning was not 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).

[15] N3 available at: `http://eulersharp.sourceforge.net/2009/12dtb/`.

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 seconds 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.[16] 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[17].

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 N3 reasoners available but also broaden the scope of possible N3 applications.

7 Related work

When originally proposed as a W3C member submission (Berners-Lee and Connolly (2011)), the formal semantics of N3 was only introduced informally. As a consequence, different systems, using N3, interpreted concepts like nested formulae differently (Arndt et al. (2019)). Since then, the relation of N3 to other Web standards has been studied from a use-case perspective (Arndt, Dörthe (2019)) and a W3C Community group has been formed (Woensel et al. (2023)), which recently published the semantics of N3 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 N3's subset RDF, de Bruijn and Heymans (2007) provide a translation to first-order logic and F-Logic using similar embeddings (e.g., a tenary predicate to represent triples) to the ones in this paper, but do not cover rules. Boley Boley (2016) supports N3 in his RuleML Knowledge-Interoperation Hub providing a translation of N3 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

[16] `https://solidproject.org/`.
[17] `https://www.wikidata.org/` *and* `https://www.dbpedia.org/`

the best of our knowledge, a rewriting for N3 as presented in this paper did not exist before. Also, existential rule reasoning engines have not been compared to the existing N3 reasoners.

8 Conclusion

In this paper we studied the close relationship between N3 rules supporting blank node production and existential rules. N3 without special features like built-in functions, nesting of rules, or quotation can be directly mapped to existential rules with tenary predicates. In order to show that, we defined a mapping between $N3^{\exists}$, N3 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 $N3^{\exists}$ to existential rules, (2) reasoning within that framework, and (3) using the inverse mapping to transfer the result back into N3.

We applied that strategy and compared the reasoning times of the N3 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 N3 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 revertible translation between $N3^{\exists}$ and existential rules we increase the number of reasoners (partly) supporting N3 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, N3 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 applied. We provided rules to mimic the list-append function of N3 under the standard chase, which is also implemented in some N3 reasoners. The append function came with rules calling it *on demand* which is very interesting in many situations and, maybe even more important, which was believed by the N3 community to only be possible employing backward reasoning. In that sense we also contributed 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 N3 use cases rely on more powerful N3 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 N3. Another direction of future work could be 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 the handling of existentially quantified rule heads but from other optimization techniques. Fully understanding these differences will help the N3 and the existential rule community 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 N3 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 N3 community, the existential rule reasoning community, and possibly many others.

References

Arndt, D. and Champin, P.-A. July 2023. *Notation3 Semantics.* W3C Community Group Report. Available at https://w3c.github.io/N3/spec/semantics.

Arndt, D. and Mennicke, S. Notation3 as an existential rule language. In Fensel, A., Ozaki, A., Roman, D., and Soylu, A., editors, *Proc. Rules and Reasoning - 7th Int. Joint Conf. (RuleML+RR'23)* 2023, volume 14244 of *LNCS*, pp. 70–85. Springer.

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, Dörthe 2019. *Notation3 as the unifying logic for the semantic web.* PhD 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. and Berners-Lee, T. 14 January 2008. *Turtle – Terse RDF Triple Language.* W3C Team Submission. Available at http://www.w3.org/TeamSubmission/turtle/.

Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., and Tsamoura, E. Benchmarking the chase. In Sallinger, E., den Bussche, J. V., and Geerts, F., editors, *Proc. 36th Symposium on Principles of Database Systems (PODS'17)* 2017, pp. 37–52. ACM.

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 Pract. Log. Program.*, *8*, 3, 249–269.

Boley, H. The ruleml knowledge-interoperation hub. In Alferes, J. J., Bertossi, L., Governatori, G., Fodor, P., and Roman, D., editors, *Rule Technologies. Research, Tools, and Applications* 2016, pp. 19–33. Springer.

Boley, H., Osmun, T. M., and Craig, B. L. Wellnessrules: A web 3.0 case study in ruleml-based prolog-n3 profile interoperation. In Governatori, G., Hall, J., and Paschke, A., editors, *Rule Interchange and Applications* 2009, pp. 43–52, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bonte, P. and Ongenae, F. RoXi: A framework for reactive reasoning. In *The Semantic Web: ESWC 2023 Satellite Events* 2023, volume 13998 of *LNCS*, pp. 159–163. Springer.

Calì, A., Gottlob, G., and Pieris, A. Query answering under non-guarded rules in Datalog+/-. In Hitzler, P. and Lukasiewicz, T., editors, *Proc. 4th Int. Conf. on Web Reasoning and Rule Systems (RR 2010)* 2010, volume 6333 of *LNCS*, pp. 1–17. Springer.

Carral, D., Dragoste, I., González, L., Jacobs, C., Krötzsch, M., and Urbani, J.

VLog: A rule engine for knowledge graphs. In Ghidini et al., C., editor, *Proc. 18th Int. Semantic Web Conf. (ISWC'19, Part II)* 2019, volume 11779 of *LNCS*, pp. 19–35. Springer.

Carral, D. and Krötzsch, M. Rewriting the description logic ALCHIQ to disjunctive existential rules. In Bessiere, C., editor, *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI 2020* 2020, pp. 1777–1783. ijcai.org.

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. *J. of Artificial Intelligence Research*, *47*, 741–808.

de Bruijn, J. and Heymans, S. Logical foundations of (e)rdf(s): Complexity and reasoning. In Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., and Cudré-Mauroux, P., editors, *The Semantic Web* 2007, pp. 86–99. Springer.

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

Ebbinghaus, H.-D., Flum, J., and Thomas, W. 1994. *Semantics of First-Order Languages*, pp. 27–57. Springer.

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.

Harth, A. and Käfer, T. 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)* 2018. CEUR-WS.

Hayes, P., editor 10 February 2004. *RDF Semantics.* W3C Recommendation. Available at `http://www.w3.org/TR/rdf-mt/`.

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

Krötzsch, M., Mehdi, A., and Rudolph, S. Orel: Database-driven reasoning for OWL 2 profiles. In Haarslev, V., Toman, D., and Weddell, G., editors, *Proc. 23rd Int. Workshop on Description Logics (DL'10)* 2010, volume 573 of *CEUR Workshop Proceedings*, pp. 114–124. CEUR-WS.org.

Krötzsch, M., Marx, M., and Rudolph, S. The power of the terminating chase. In Barceló, P. and Calautti, M., editors, *Proc. 22nd Int. Conf. on Database Theory (ICDT'19)* 2019, volume 127 of *LIPIcs*, pp. 3:1–3:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Nilsson, U. and Maluszynski, J. 1990. *Logic, Programming and PROLOG.* John Wiley & Sons, Inc., USA.

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 Pract. Log. Program.*, *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. July 2023. *Notation3 Language.* W3C Community Group Report. Available at `https://w3c.github.io/N3/reports/20230703/`.

Woensel, W. V. and Hochstenbach, P. July 2023. *Notation3 Builtin Functions.* Available at `https://w3c.github.io/N3/reports/20230703/builtins.html`.