Optimizing Semantic Reasoning on Memory-Constrained Platforms using the RETE Algorithm

William Van Woensel[1] and Syed Sibte Raza Abidi[1]

[1]Faculty of Computer Science, Dalhousie University, Halifax, Canada
{william.van.woensel, raza.abidi}@dal.ca

Abstract. Mobile hardware improvements have opened the door for deploying rule systems on ubiquitous, mobile platforms. By executing rule-based tasks locally, less remote (cloud) resources are needed, bandwidth usage is reduced, and local, time-sensitive tasks are no longer influenced by network conditions. Further, with data being increasingly published in semantic format, an opportunity arises for rule systems to leverage the embedded semantics of semantic, ontology-based data. To support this kind of ontology-based reasoning in rule systems, rule-based axiomatizations of ontology semantics can be utilized (e.g., OWL 2 RL). Nonetheless, recent benchmarks have found that any kind of semantic reasoning on mobile platforms still lacks scalability, at least when directly re-using existing (PC- or server-based) technologies. To create a tailored solution for resource-constrained platforms, we propose changes to RETE, the mainstay algorithm for production rule systems. In particular, we present an adapted algorithm that, by selectively pooling RETE memories, aims to better balance memory usage with performance. We show that this algorithm is well-suited towards many typical Semantic Web scenarios. Using our custom algorithm, we perform an extensive evaluation of semantic, ontology-based reasoning, using our custom RETE algorithm and an OWL2 RL ruleset, both on the PC and mobile platform.

Keywords: RETE, OWL2 RL, rule-based reasoning, OWL reasoning, reasoning optimization.

1 Introduction

Using structured domain knowledge, production rule systems realize a diversity of tasks in domains such as business, science and healthcare. Knowledge is formulated as a set of productions (i.e., if-then rules) together with a set of assertions. In the healthcare domain, production rule systems are often at the core of Clinical Decision Support Systems (CDSS), which aid in diagnosis, prognosis and treatment tasks [1]. To semantically structure health data, a variety of biomedical ontologies (e.g., see BioPortal [2]) and clinical health terminologies (e.g., SNOMED-CT [3]) are available. In order to improve decision support accuracy, a CDSS can leverage the embedded semantics of semantic health data, such as subclass, transitive and symmetric relations between drugs, illnesses and treatments. Performing this kind of ontology-based, semantic reasoning

in (production) rule systems requires a rule-based axiomatization of ontology semantics. The W3C OWL2 RL profile [4] is highly relevant, since it partially axiomatizes the OWL2 RDF-based semantics [5] as a set of high-level, abstract IF-THEN rules.

There is a growing demand to deploy production rule systems, such as clinical decision support systems, directly on mobile, resource-constrained platforms. Examples include clinical, time-sensitive tasks to be performed directly on mobile consumer devices [6], and sensor networks pushing reasoning down to the device layer to cope with unstable communication [7]. However, recent benchmarks [8, 9] show that the mobile performance of existing, desktop- or server-based reasoners still leaves much to be desired. It may be noted that, although modern mobile consumer devices are outfitted with 2Gb of RAM or more, single apps are only assigned max. 192Mb on Android; whereas devices in sensor networks may even feature much less memory.

To optimize semantic, ontology-based reasoning in rule systems, we propose a novel version of the RETE algorithm, a well-known algorithm for production rule systems, which aims to better balance memory usage with performance. The RETE algorithm uses *alpha nodes* to represent rule premises, with *alpha memories* keeping matching premise facts (tokens). Our proposed $RETE_{pool}$ algorithm is based on the observation that generic rule premises, which occur frequently in OWL2 RL, result in a large duplication of data in alpha memories. For instance, the same set of tokens can match OWL2 RL premises *<?x ?p ?y>*, *<?c owl:unionOf ?x>* and *<?c rdf:type owl:Class>*. An extreme example is a "wildcard" premise, i.e., with variables at all positions, which will effectively duplicate the data from all other premises. By pooling a selection of alpha memories into a single shared memory, the $RETE_{pool}$ algorithm aims to reduce duplication of data in RETE. We note that $RETE_{pool}$ is well-suited towards Semantic Web settings that typically involve an existing, multi-purpose RDF store. We integrated this algorithm into Apache Jena [10] and AndroJena [11], a port of Apache Jena for mobile (Android) platforms. We present an extensive evaluation of semantic reasoning, using a rule system featuring the $RETE_{pool}$ algorithm and an OWL2 RL ruleset, both on PC and mobile platform (Android).

The paper is structured as follows. First, Section 2 summarizes our OWL2 RL ruleset, which implements the OWL2 RL specification and is used to realize semantic reasoning. In Section 3, we summarize and exemplify the RETE algorithm. Section 4 presents the $RETE_{pool}$ algorithm. In Section 5, we extensively evaluate semantic reasoning using $RETE_{pool}$ and our OWL2 RL ruleset. In Section 6, we discuss relevant state of the art, and Section 7 presents conclusions and future work.

2 OWL2 RL Ruleset

To realize semantic reasoning on mobile, resource-constrained platforms, we rely on the W3C OWL2 RL profile. The OWL2 Web Ontology Language Profiles document [4] introduces multiple OWL2 profiles, which are optimized to handle specific application scenarios. The OWL2 RL profile is aimed at balancing expressivity with reasoning scalability, and presents a partial, rule-based axiomatization of OWL2 RDF-Based Semantics [5]. As only a partial axiomatization, OWL2 RL does not guarantee completeness for TBox reasoning [12]; and places syntactic restrictions on ontologies to

ensure all correct inferences. Nevertheless, this trade-off seems acceptable when targeting scalable reasoning on resource-constrained platforms.

Based on the OWL2 RL specification, we created a concrete OWL2 RL ruleset that is re-usable by any arbitrary rule engine, which means no particular internal support (e.g., for datatypes or lists) can be assumed. Below, we focus on 3 non-trivial issues that occur when attempting to create an OWL2 RL ruleset:

(1) A pair of rules (*#dt-type2* and *#dt-not-type*) support RDF datatype semantics, by inferring types and flagging inconsistencies based on the datatype of a literal's value space (e.g., typing integer "42" with *xsd:int*). Two other rules (*#dt-eq* and *#dt-diff*) indicate equality and inequality of literals, which requires differentiating literals from URLs (to avoid these rules firing for URL resources as well). Since built-in support for RDF datatypes and literals cannot be assumed for arbitrary systems, we left out these rules. Related work, including DLEJena [13] and the SPIN [14] and OWLIM [15] OWL2 RL rulesets, also do not include datatype rules. Others opted to leave out datatype rules due to their significant performance issues [16].

(2) Another set of rules lacks an antecedent and are thus always applicable. Some of these rules lack variables (e.g., specifying that *owl:Thing* has type *owl:Class*), and were represented as axiomatic triples accompanying the ruleset. Other rules comprise "quantified" variables in the consequent; e.g., stating that each annotation property has type *owl:AnnotationProperty*. Similarly, these were implemented by creating an axiom for each annotation (OWL2 [17]) and datatype property (OWL2 RL [4]).

(3) *N-ary rules* refer to a finite list of elements; a first subset **(L1)** places restrictions on a limited number of list elements (e.g., *#eq-diff2*); a second subset **(L2)** places restrictions on all elements (e.g., *#cls-int1*), and a third ruleset **(L3)** yields inferences for all list elements (e.g., *#scm-uni*). Rulesets **(L1)** and **(L3)** can be supported by adding two auxiliary list-membership rules (Rule 1), which link each list element to all preceding list cells; meaning the first cell will be directly linked to all elements.

$$T(?l, first, ?m) \rightarrow T(?l, hasMember, ?m)$$
$$T(?l_1, rest, ?l_2), T(?l_2, hasMember, ?m) \rightarrow T(?l_1, hasMember, ?m)$$

Rule 1. Two rules for inferring list membership.

E.g., using these rules, *#scm-uni* **(L3)** may be formulated as follows (Rule 2; note that Rule 3 similarly belongs to (L3)):

$$T(?c, unionOf, ?l), T(?l, hasMember, ?cl) \rightarrow T(?cl, subClassOf, ?c)$$

Rule 2. Rule inferring subclasses based on union membership (*#scm-uni*).

Multiple solutions are possible for n-ary rules from **(L2)**. We chose a solution that *replaces* each (L2) rule by a set of auxiliary rules [15], which infer intermediary assertions for each list cell i ($0 \leq i < n$), and, based on these inferences, finally generates the n-ary inference if the first cell is related to an (L2) assertion. We note that this is the only solution that does not require pre-processing the ruleset or ontology for per ontology update, compared to e.g., instantiating (L2) rules based on n-ary assertions [18], or "binarizing" (i.e., converting all n-ary assertions to binary ones).

Based on these considerations, we created an OWL2 RL ruleset written in the SPARQL Inferencing Notation (SPIN) based on an initial ruleset created by Knublauch [14]. This initial ruleset did not specify axioms, and relied on built-in Apache Jena

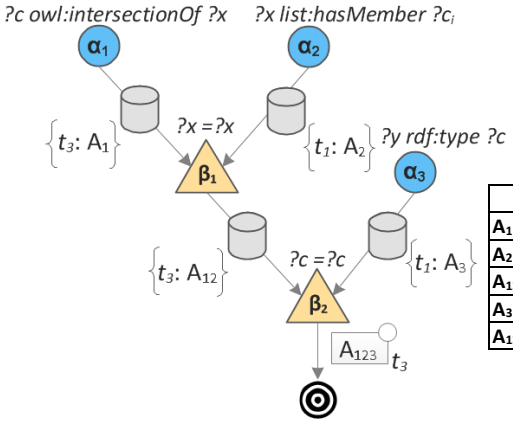
functions to implement n-ary rules. Our final ruleset contains 78 rules and 43 supporting axioms, and can be found online [19]. We checked the conformance of the OWL2 RL ruleset using the OWL2 RL conformance test suite by Schneider et al. [20]. We note that some of these tests had to be left out, either due to the limitations of our OWL2 RL ruleset or difficulties testing conformance. We detail these cases online [19].

3 Using RETE for Reasoning on RDF

Production rule systems operate by matching production conditions to a set of assertions, and then adding / removing assertions based on the production's actions [21]. For this purpose, the RETE algorithm sets up a network consisting of *alpha nodes* for each condition (i.e., intra-condition check), and *beta nodes* to join shared variables between these conditions (i.e., inter-condition check). Each alpha node, and all but the last beta node, is linked to a *memory* keeping the results of these checks. A rule ends with a *terminal node*, which represents the actions. To create a RETE network, the *right input* of each beta node is linked to an alpha node, and its *left input* to the previous beta node, or if none exists, the first alpha node (cfr. Ishida [22]). When reasoning over RDF data, an intra-condition check matches a triple pattern (or FILTER expression) to an RDF triple token [7, 23, 24]. Below, we show the RETE network for the *#cls-int2* OWL2 RL rule, and describe the reasoning process when new facts (tokens) enter the network.

$$T(?c, intersectionOf, ?x), T(?x, hasMember, ?c_i), T(?y, type, ?c) \rightarrow T(?y, type, ?c_i)$$

Rule 3. Rule inferring resource types based on intersection members (#cls-int2).



RETE network tokens

	?c	*?x*	*?ci*	*?y*
A1	<cls-A>	**<lst-A>**	null	null
A2	null	**<lst-A>**	<cls-C1>	null
A12	**<cls-A>**	<lst-A>	<cls-C1>	null
A3	**<cls-A>**	null	null	<inst-1>
A123	<cls-A>	<lst-A>	<cls-C1>	<inst-1>

Fig. 1. Example RETE structure (rule *#cls-int2*).

At time t_1, triple tokens A_2 and A_3 enter the network, which are matched by alpha nodes α_2 and α_3 and inserted into their memories. At time t_2, incoming token A_1 is matched to alpha node α_1, and inserted into its memory. Since nodes α_1 and α_2 are connected by beta node β_1, the new token triggers a join attempt between the left-input A_1 token and the right-input A_2 token. As shown in the token table, tokens A_1 and A_2 have the same value for shared variable *?x* (i.e., *<lst-A>*), leading to a successfully joined

token A_{12} that is added to the β_1 memory. With this new token at its left input, node β_2 attempts a join with right input token A_3. Both tokens have the same value for shared variable *?c* (i.e., <*cls-A*>), leading to a successfully joined token A_{123}.This token reaches the terminal node, which will use the instantiated variables to infer a new fact, i.e., <*inst-1*> *rdf:type* <*cls-*C_1>.

A standard RETE optimization is to re-use alpha nodes (and memories) when the same premise occurs multiple times, and beta nodes in case rules share the first two or more premises (else, the contents of the beta memories may differ). Re-using nodes and memories reduces the number of match and join operations, and avoids duplicate storage of tokens. To speed up the joining process, the most restrictive conditions (i.e., alpha nodes) are often placed first, and Cartesian products are avoided [7, 22, 24]. Alpha and beta memories are typically indexed to allow for hashed joins [10, 24, 25].

4 The RETE$_{pool}$ Algorithm

A default RETE optimization involves re-using alpha memories for identical premises (Section 3), which reduces data duplication. Nevertheless, we observe that generic rule premises (i.e., with more than 1 variable) typically still lead to large duplications of data in alpha memories. For instance, the memory related to premise <*?x ?p ?y*> will effectively duplicate all data from the memory of <*?p rdf:type owl:ObjectProperty*>; and both memories will overlap with the memory of <*?y rdf:type ?c*>. This is especially apparent in the OWL2 RL ruleset with its many generic premises. An extreme example are wildcard premises, which are found in the OWL2 RL ruleset and match all tokens. As such, they effectively duplicate data from all other alpha memories. Furthermore, we note that many Semantic Web applications involve an existing RDF store, which is used to load data into the rule system but for other purposes as well (e.g., querying). Alpha memories will always duplicate (parts of) this RDF store, thus presenting a second, orthogonal level of data duplication.

We present the *RETE$_{pool}$* algorithm, which pools alpha memories into a single shared memory. As a result, duplicate tokens, i.e., tokens occurring in multiple alpha memories, are only stored once in a single memory. In doing so, data duplication in alpha memories is effectively avoided. In scenarios with an existing RDF datastore, *RETE$_{pool}$* can directly re-use this store as the shared memory; thus avoiding both internal duplication, as well as duplication between the RETE structure and RDF store. Below, we discuss the implementation of the *RETE$_{pool}$* algorithm.

4.1 Implementation of RETE$_{pool}$

The *RETE$_{pool}$* algorithm utilizes *virtual* alpha memories (cfr. Hanson [26]) in the RETE network, which keep a mask on the single, shared memory that represents the related premise (e.g., <*?c rdf:type ?t*>). For instance, an RDF store may be used as the shared memory, as is done in our evaluation (Section 5).

New tokens are added to the shared memory, and injected at suitable alpha nodes into the network (see Fig. 1). In this process, a beta node will attempt to join the new

token with other tokens from its input alpha memory. Joining two tokens implies they have the same value(s) for the shared variable(s). Hence, join operations can be performed by searching the alpha memory for tokens matching the shared variable(s) value(s). In our case, the virtual memory's mask is extended with these values, and then used as a search constraint on the shared memory. By extending the mask, only tokens that match the alpha node premise will be returned. This is illustrated in Fig. 2. At time t_3, token A_{12} is used to extend the virtual memory mask with the token's value of shared variable *?c*, leading to search constraint S = *?*, P = *rdf:type*, O = <*cls-A*>.

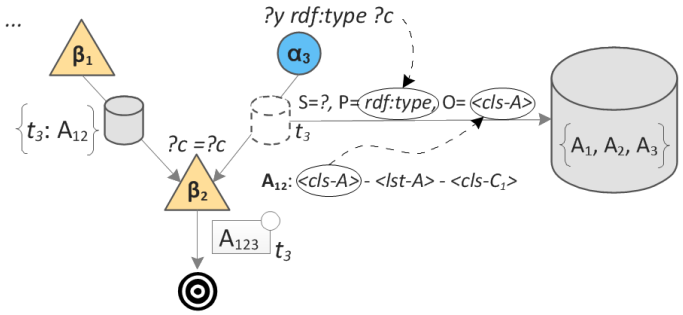


Fig. 2. Usage of the shared memory in RETE$_{pool}$.

Since each join involves accessing a (very) large shared memory, instead of a relatively small alpha memory, it is clear that this algorithm optimizes memory at the cost of performance. This is confirmed by our evaluation (Section 5). We note that the issue that *RETE$_{pool}$* aims to solve, i.e., data duplication in alpha memories, clearly depends on premise selectivity. By only utilizing a virtual alpha memory for overly generic, non-restrictive premises, we may thus better balance memory usage with runtime performance. To that end, *RETE$_{pool}$* allows configuring a selectivity threshold t_s ($0 < t_s \leq 1$). In case the premise selectivity (i.e., number of matching facts) equals or exceeds t_s, a virtual memory will be utilized, otherwise a regular memory. Our evaluation studies the effects of different values for t_s on memory and performance.

Below, we discuss an additional issue that arises when an existing, pre-loaded RDF store is being re-used.

4.2 Reciprocal Join Issue

Many Semantic Web applications will start out with a pre-loaded RDF datastore, which will be used to inject data into the rule system. When utilizing *RETE$_{pool}$*, an opportunity exists to re-use this datastore as a single shared memory. In this case, each virtual alpha memory will initially be fully "populated", since it references the pre-loaded RDF dataset (Section 4.1). This is illustrated in Fig. 3.

At time t_0, token A_1 is injected into the network, and joins with token A_2 from node α_2 (already present in its virtual memory). At time t_1, token A_2 is injected and similarly joins with token A_1 (also present) – thus performing a second, redundant (and reciprocal) join. Later on, in case token A_{12} was stored twice in memory β_1, four joins would

take place – two for each stored version of A_{12}. Hence, a single successful rule firing requires an exponential $2^{(|\mathcal{A}_r|-1)}$ joins, with $\mathcal{A}_r = \{\, \alpha_i \in \mathcal{R}_r \,\}$ the set of alpha nodes in network $\mathcal{R}_r$ (for any rule r). In case duplicate tokens are not stored (e.g., duplicate checking takes place), a single rule firing would require $(|\mathcal{A}| - 1) \times 2$ joins. Without reciprocal joins, a single successful rule firing requires only $|\mathcal{A}_r| - 1$ joins.

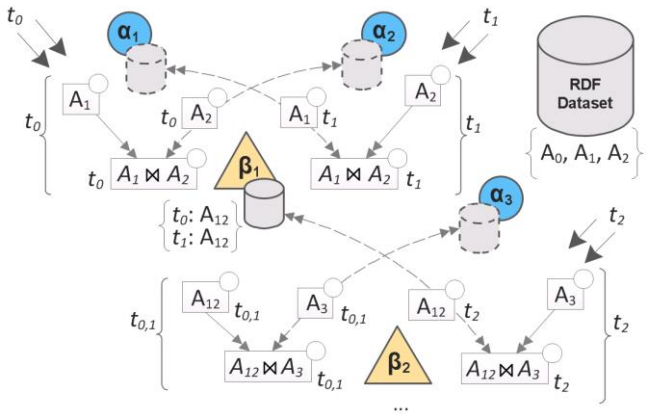


Fig. 3. Example where the reciprocal-join issue occurs in RETE$_{pool}$.

Since this issue only occurs during the first reasoning cycle, we introduced a custom reasoning process for this cycle. In this process, only tokens are injected that match the first alpha node α_1. As the token travels through the network, all possible joins will be attempted, since all tokens are already at their virtual alpha memories, while avoiding reciprocal joins. In Fig. 3, by only injecting token A_1, a second, redundant join with A_2 will be avoided, which in turn avoids redundant joins later on in the network.

5 Semantic Reasoning Benchmarks

This section presents benchmark results for ontology-based reasoning, using a rule system and an OWL2 RL ruleset. For the rule system, we benchmarked multiple configurations of the *RETE$_{pool}$* algorithm, and compared the results to a baseline RETE algorithm. To test the performance impact on resource-constrained platforms, we ran each benchmark on a PC as well as a mobile device.

Below, we discuss the setup (Section 5.1) and present the benchmarks (Section 5.2).

5.1 Benchmark Setup

5.1.1 Baseline System

We extended the original Apache Jena RETE implementation [27] with standard optimizations, including the re-use of RETE nodes and memories, memory indexing and join ordering (see Section 3). These optimizations are considered standard-practice in

modern RETE systems [7, 24, 25, 28], making the extended system an appropriate baseline. We also copied these extensions to Jena's Android port, i.e., AndroJena [11]. In the benchmark results, this baseline system is referred to as $RETE_{base}$. We implemented the $RETE_{pool}$ algorithm on top of this baseline implementation.

5.1.2 OWL2 RL Ontologies and Ruleset

Our benchmarks were executed on the BioPortal ontologies from the OWL 2 RL Benchmark Corpus [29]. On PC, reasoning over 3 of the 45 ontologies took longer than 10 minutes (our cut-off time) for any configuration, so these were left out. For the 42 remaining ontologies, the number of statements range from 246 to 57310 (avg. 6684), and their file sizes (N3 format) range from 24 Kb to 5852 Kb (avg. 642 Kb). For the mobile benchmarks, we considered a subset of 34 BioPortal ontologies, since the other 11 ontologies either caused out-of-memory exceptions, or ran longer than 10 minutes. For these ontologies, the number of statements range from 246 to 7291 (avg. 2199) and their file sizes (N3 format) from 24 Kb to 838 Kb (avg. 210 Kb). As a result, we note that average performance times for PC and mobile are not directly comparable. We detail each set of ontologies in our online documentation [19]).

As the benchmark ruleset, we utilized the OWL2 RL ruleset introduced in Section 2. This ruleset contains 78 rules and 43 supporting axioms, and can be found online [19]. As mentioned, we checked the conformance of this ruleset using the OWL2 RL conformance test suite by Schneider et al. [20], and detail the results online [19].

5.1.3 Benchmark Platforms

Benchmarks were performed on two platforms:

(1) *PC*: Lenovo Thinkpad T530, with a dual-core Intel Core i7-3520M CPU (2.9Ghz), 8Gb RAM and a 64 bit infrastructure, running Windows 7.0 (Service Pack 1).

(2) *Mobile*: LG Nexus 5 (model LG-D820), with a 2.26 GHz Quad-Core Processor and 2Gb RAM. This device runs Android 6, which grants apps 192Mb of heap space.

During the benchmarks, both devices were connected to a power supply.

5.1.4 $RETE_{pool}$ Configurations

We benchmarked the following algorithms and configurations (regular memories are utilized for beta nodes).

(A.i) $RETE_{base}$: a regular memory is utilized for each alpha node.

(A.ii) $RETE_{full\text{-}pool}$: a virtual alpha memory is utilized for each alpha node.

(A.iii) $RETE_{part\text{-}pool}$: a virtual alpha memory is only utilized in case its premise selectivity exceeds the configured threshold: 0.1 – 0.25 – 0.5 – 0.75 – 1.

Further, we consider two orthogonal scenarios for $RETE_{pool}$:

(S.i.) A shared memory is introduced for the sole purpose of supporting $RETE_{pool}$;
(S.ii.) An existing RDF store is re-used as the shared memory pool.

We utilized the Apache Jena RDF store as the shared memory. Each configuration was benchmarked in a one-shot reasoning scenario, i.e., with a single reasoning cycle over each of the benchmark ontologies. For $RETE_{X\text{-}pool}$ configurations, the shared RDF store was pre-loaded with the ontology, after which a custom reasoning process took place (see Section 4.2). This allowed us to estimate premise selectivity by using the number of matched tokens from the actual benchmark ontology.

5.1.5 Benchmark Metrics

We measure the following performance metrics:

(P.1) *Network compilation time*: time needed to compile the network, including selecting memory indices and deciding the best join order.
(P.2) *Reading time*: time needed for the system to read and parse the data.
(P.3) *Reasoning time*: time needed to complete the first reasoning cycle.
(P.4) *Initialization time*: time needed to load the ontology into the shared memory. As mentioned, for $RETE_{X\text{-}pool}$ configurations, the shared memory was pre-loaded with the ontology, after which reasoning took place (Section 5.1.4).

In addition, we collect the following memory-related metrics:

(M.1) *Number of alpha memories*: the total number of alpha memories, differentiating between different types (i.e., regular vs. virtual memories).
(M.2) *Alpha memory size*: the total size (Kb) taken up by the set of alpha memories.
(M.3) *Total memory size*: the total size (Kb) taken up by the set of alpha memories and the shared memory pool.
(M.4) *RDF nodes size*: the total size (Kb) taken up by RDF node data. The contents of an RDF graph are kept in RDF node objects. By considering this memory size separately, these are not accidentally counted towards only (M.2) or (M.3).
(M.5) *Shared memory size*: the total size (Kb) taken up by the shared memory, if any.

For evaluating the (Z.ii) *shared memory pool* setup, we also measure the following:

(M.6) *Memory operations*: the number and performance overhead of memory operations, including updating the shared memory and individual RETE memories.

To obtain actual memory usage (Kb), we performed heap dumps on PC and Android. Per configuration, a single heap dump was taken after reasoning over the ontology that yielded the min., median and max. number of alpha memory tokens, respectively.

5.2 Benchmark Results

Table 1 shows the memory usage for $RETE_{base}$ (i.e., the baseline system) and the $RETE_{pool}$ configurations. In particular, it shows the memory usage for the min., median and max. ontology (see Section 5.1.5). The table lists the number of alpha memories (M.1), alpha memory sizes (M.2), and (for ease of reference) the total memory size, which includes alpha memories and the shared memory (if any) (M.2). Further, the table shows the reasoning performances (P.3).

The following memory metrics are identical for all configurations (Kb): (M.4) *RDF nodes size*:5.1.5 median: 800 (min: 203 – max: 26760); (M.5) *Shared memory size*: median: 562 (min: 98 – max: 26070). Further, that the following performance metrics (ms) are identical: (P.1) *Network compilation time*: avg. ca. 5 (PC), 71 (mobile); (P.2) *Reading time*: avg. ca. 73 (PC), 696 (mobile). As mentioned, a separate data loading step took place for $RETE_{X\text{-}pool}$ configurations (P.4). This amounts to 11ms for median (min: 3ms – max: 670ms) on PC, and 99ms for median (min: 15ms – max: 1034ms).

Note that $RETE_{part\text{-}pool}$ configurations with $t_s = 0.1$ and $t_s = 0.25$; and $t_s = 0.75$ and $t_s = 1$ are pairwise identical, so we only present $t_s = 0.1$ and $t_s = 1$.

Table 1. Memory usage (Kb) and reasoning performance (ms)
*: $RETE_{part\text{-}pool}$ (t_s), †: **r** = regular, **v** = virtual,
: median (min – max), *: average (min – max).

version*	**memory usage (Kb)****			**reasoning performance*** (ms)** (P.3)	
	# α mem. (M.1)†	*α mem. size* (M.2)	*total mem. size* (M.3)	*PC*	*mobile*
$RETE_{base}$	**#r**: 46 **#v**: 0	1487 (263 – 52789)	1487 (263 – 52789)	15705 (18 - 322352)	24968 (1051 - 199974)
$RETE_{full\text{-}pool}$	**#r**: 0 **#v**: 46	20 (20 – 20)	582 (118 – 26090)	51905 (51 - 1187570)	69573 (2903-542670)
$RETE_{part\text{-}pool}$ (0.1)	**#r**: 42 **#v**: 4	482 (100 – 13757)	1044 (198 – 39874)	16303 (27 - 340194)	29287 (1526 - 212573)
$RETE_{part\text{-}pool}$ (0.5)	**#r**: 43 **#v**: 3	859 (161 – 14675)	1421 (259 – 40745)	16475 (23 - 338109)	26444 (1145 - 202646)
$RETE_{part\text{-}pool}$ (1)	**#r**: 44 **#v**: 2	891 (166 – 30067)	1453 (264 – 56137)	17715 (25 - 365843)	25203 (1115 - 198404)

We first observe that only 46 alpha memories are required for a total of 78 OWL2 RL rules, due to the re-use of alpha (and beta) nodes and memories where possible (Section 3). Memory savings for $RETE_{pool}$ depend on the concrete application scenario: i.e., whether (S.i.) a separate shared memory needs to be introduced, or (S.i.) an existing, multi-purpose RDF store can be re-used for this purpose.

Scenario (S.i.)

Even in scenario (S.i.), we expect significant memory savings since duplication of data in alpha memories is either avoided entirely ($RETE_{full\text{-}pool}$) or partially ($RETE_{part\text{-}pool}$). Indeed, memory savings for $RETE_{full\text{-}pool}$ are huge, with a ca. 60% for the median ontology (min: 55%, max: 50%). As expected, the number of regular alpha memories (M.1) increases together with threshold t_s. This causes data duplication among individual memories, as well as with the shared memory; thus increasing the total memory usage (M.3). For $RETE_{part\text{-}pool}$ (0.1), memory savings drop to ca. 30% for median (min: 25%, max: 25%); for $RETE_{part\text{-}pool}$ (0.5), ca. 4% (min: 1%, max: 23%); and for $RETE_{part\text{-}pool}$ (1), memory usage is similar for median and min., and increases for max.

At the same time, we observe that $RETE_{full\text{-}pool}$ greatly reduces performance; by factors of avg. ca. 3,3 and 2,8 for PC and mobile, respectively[1]. We expect $RETE_{part\text{-}pool}$ to strike a better memory/performance balance, with performance improving as threshold t_s increases. Indeed, performance improves greatly on mobile (e.g., $t_s = 0.1$: avg. ca. 58%), and approaches $RETE_{base}$ as t_s increases. On PC, we observe the same effect (e.g., $t_s = 0.1$: avg. ca. 69%) at least until $t_s \geq 0.5$, after which performance (slightly) worsens. This is likely due to excessive garbage collection; when leaving out the 7 most memory-intensive ontologies, $RETE_{part\text{-}pool}$ performance is constant for all t_s at avg. ca. 1,25s ($RETE_{base}$ = 1.15s, $RETE_{full\text{-}pool}$ = 3.3s).

We observe that $RETE_{part\text{-}pool}$ (0.1) effects the best memory/performance balance. Compared to $RETE_{base}$, its saves memory by 30% for median (min: 25%, max: 25%) whereas it is ca. 4.3s slower on mobile and 0.6s slower on PC. While $RETE_{part\text{-}pool}$ (0.5) only incurs a penalty of ca. 1.5s on mobile and 0.8s on PC, its memory savings are significantly lower, i.e., 4% for median (min: 1%, max: 23%). At the same time, we note that an extra initialization time is incurred for all $RETE_{X\text{-}pool}$ configurations (P.4).

Scenario (S.ii.)

In scenario (S.ii.), since we are re-using an RDF store that is utilized for other purposes as well, the size of the shared memory is not counted towards our total memory usage. In that case, memory savings by $RETE_{full\text{-}pool}$ are tremendous, using only ca. 0,04% (max) to ca. 7,6% (min) compared to $RETE_{base}$. In fact, $RETE_{part\text{-}pool}$ (1), which utilizes the most memory, still only takes up avg. ca. 57% of $RETE_{base}$.

For $RETE_{part\text{-}pool}$ (0.1): memory savings include 68% for median (min: 62%, max: 74%); for $RETE_{part\text{-}pool}$ (0.5), savings constitute 42% for median (min: 39%, max: 72%); for $RETE_{part\text{-}pool}$ (1), savings include 40% for median (min: 37%, max: 43%).

In this case, $RETE_{part\text{-}pool}$ (0.5) seems preferable as it greatly improves performance (ca. 65%) on mobile (PC is only slightly slower), while, in this scenario, memory savings are significant as well. Further, the performance gains by $RETE_{part\text{-}pool}$ (1) do not seem comparable to its increased memory usage. As before, we note that any $RETE_{X\text{-}pool}$ configuration also incurs an extra initialization time (P.4).

6 Related Work

To realize ontology-based reasoning, many mobile reasoners, i.e., targeting resource-constrained platforms, utilize rule-based OWL axiomatizations; such as custom entailment rulesets [30, 31] or OWL2 RL rulesets [7, 12]. For instance, MiRE4OWL [32] and μOR [31] apply a custom entailment ruleset; Seitz et al. [16] load the CLIPS engine with the OWL2 RL ruleset; and Tai et al. [7] and BaseVISor [33] rely on rules implementing pD* semantics. In general, by focusing on subsets of rule axioms, rule-based axiomatizations allow easily adjusting reasoning complexity to the application scenario [7], or avoiding resource-heavy inferences [16, 15]. In contrast, transformation rules used in tableau-based DL reasoning are often hardcoded, making it hard to de-select

[1] Note that performance times for PC and mobile are not directly comparable (Section 5.1.2).

them at runtime [7]. Also, most classic DL optimizations improve performance at the cost of memory, which is limited in mobile devices [9].

To deal with data duplication caused by generic rule premises in OWL2 RL, we presented the *RETE$_{pool}$* algorithm, which utilizes virtual alpha memories that act as masks (or views) on a large, shared dataset. This concept was first introduced by Hanson for the Ariel system [26, 34]. Later on, Hanson et al. [35] presented a set of optimizers that choose an efficient Gator [36] network, possibly including virtual alpha memories, based on database size, predicate selectivity and update frequency distribution, among others. In this paper, we implemented this concept to realize more memory-efficient, semantic ontology-based reasoning on mobile platforms. We further consider typical Semantic Web scenarios, where an RDF store is already available and possibly pre-loaded with data, as well as the issues ensuing from such a setup. As opposed to Hanson et al., our evaluation focuses in particular on how memory and performance may be balanced by using different selectivity thresholds t_s.

Some approaches [12, 13, 37] support a different solution for dealing with generic rule premises, as they occur in rule-based axiomatizations such as OWL2 RL. In this solution, a first step materializes all schema-related inferences in the ontology (e.g., using a separate OWL reasoner), which is then followed by a rule instantiation step. Based on the materialized schema, the second step creates multiple concrete rules for each generic rule by replacing schema variables by concrete schema references. This kind of approach deals poorly with ontology schema updates, and is thus only suitable for scenarios where such updates do not occur (or occur very infrequently). As a different solution to optimizing mobile, semantic reasoning, Tai et al. [7] present a selective rule loading algorithm, which composes a pD* ruleset based on ontology expressivity; and a two-phase RETE construction process, which utilizes selectivity information from the first phase to optimize join sequences in the second phase. Komazec et al. [38] integrated a special ε network into RETE to optimize RDFS entailments.

We note that other production rule algorithms aside from RETE exist. Miranker [21] proposed the TREAT algorithm, which, instead of storing join results, re-calculates results of intermediate joins when required. In doing so, TREAT avoids the memory and maintenance overhead of beta memories. The Gator [36] and RETE* [39] algorithms generalize RETE and TREAT, treating both as special cases. The more recent PHREAK algorithm, introduced by the well-known Drools production system [25], is based on RETE but incorporates lazy and goal-oriented aspects, inspired by LEAPS [40] and RETE/UL [41]. As our work focuses on reducing data duplication in alpha memories, which is an issue that, to the best of our knowledge, potentially affects all these approaches and algorithms, it can be considered complementary to these efforts.

7 Conclusions and Future Work

In this paper, we presented the *RETE$_{pool}$* algorithm which, by pooling a particular selection of RETE alpha memories, aims to balance memory usage with performance. We illustrated how this algorithm is well-suited for many typical Semantic Web scenarios, which typically utilize an existing, multi-purpose RDF store. We performed an

extensive set of benchmarks, which evaluated semantic, ontology-based reasoning using our OWL2 RL ruleset and multiple configurations of the algorithm, both on PC and mobile platforms. In line with expectations, the $RETE_{pool}$ algorithm drastically reduces memory usage. By configuring selectivity thresholds, i.e., where virtual alpha memories are only used in case estimated selectivity exceeds a threshold, we were better able to balance memory savings with performance overhead.

Our evaluation has a number of limitations. Firstly, our solution and evaluation focuses specifically on semantic reasoning using the OWL2 RL ruleset, which includes many generic rule premises. For other rulesets with more concrete premises, utilizing $RETE_{pool}$ will likely lead to smaller memory savings. Hence, future work includes running additional benchmarks to test the usefulness of this approach for other types of rulesets. Secondly, premise selectivity was estimated based on the actual number of tokens matched from the benchmark ontology. Clearly, this will not be possible in incremental reasoning scenarios, where only a very limited amount of initial data is available. As a result, future work involves utilizing other kinds of selectivity estimates (e.g., based on SPO position). Finally, to avoid the main pitfall of $RETE_{pool}$ – i.e., accessing a large shared memory for each join attempt – future work involves creating a more fine-grained memory strategy. We observe that alpha memories will often completely subsume other memories, depending on premise structure: e.g., premise <*?c rdf:type ?t*> subsumes premise <*?c rdf:type owl:Class*>. By constructing a nested memory structure, a subsuming memory could directly access the data of subsumed memories, while still reducing duplication.

References

1. Hussain, S., Abidi, S.S.R.: An Ontology-based Framework for Authoring and Executing Clinical Practice Guidelines for Clinical Decision Support Systems. J. Inf. Technol. Healthc. 8 (2008).
2. The National Center for Biomedical Ontology: BioPortal, http://bioportal.bioontology.org/.
3. SNOMED International: SNOMED-CT, http://www.snomed.org/snomed-ct.
4. Calvanese, D., Carroll, J., De Giacomo, G., Hendler, J., Herman, I., Parsia, B., Patel-Schneider, P.F., Ruttenberg, A., Sattler, U., Schneider, M.: OWL2 Web Ontology Language Profiles (Second Edition), http://www.w3.org/TR/owl2-profiles/#OWL_2_RL.
5. Carroll, J., Herman, I., Patel-Schneider, P.F.: OWL 2 Web Ontology Language RDF-Based Semantics (Second Edition), https://www.w3.org/TR/owl2-rdf-based-semantics/.
6. Van Woensel, W., Roy, P.C., Abidi, S., Abidi, S.S.: A Mobile & Intelligent Patient Diary for Chronic Disease Self-Management. 15th World Congress on Health and Biomedical Informatics. , Sao Paulo, Brazil (2015).
7. Tai, W., Keeney, J., O'Sullivan, D.: Resource-constrained reasoning using a reasoner composition approach. Semant. Web. 6, 35–59 (2015).
8. Yus, R., Bobed, C., Esteban, G., Bobillo, F., Mena, E.: Android goes Semantic: DL Reasoners on Smartphones. Proceedings of the 2nd International Workshop on OWL Reasoner Evaluation, Ulm, Germany. pp. 46–52 (2013).
9. Bobed, C., Yus, R., Bobillo, F., Mena, E.: Semantic reasoning on mobile devices: Do

Androids dream of efficient reasoners? Web Semant. Sci. Serv. Agents World Wide Web. 35, 167–183 (2015).
10. Apache: Apache Jena, https://jena.apache.org/.
11. AndroJena, https://github.com/lencinhaus/androjena.
12. Motik, B., Horrocks, I., Kim, S.M.: Delta-reasoner: A Semantic Web Reasoner for an Intelligent Mobile Platform. Proceedings of the 21st International Conference Companion on World Wide Web. pp. 63–72. ACM, New York, NY, USA (2012).
13. Meditskos, G., Bassiliades, N.: DLEJena: A Practical Forward-chaining OWL 2 RL Reasoner Combining Jena and Pellet. Web Semant. 8, 89–94 (2010).
14. Knublauch, H.: OWL 2 RL in SPARQL using SPIN, http://composing-the-semantic-web.blogspot.ca/2009/01/owl-2-rl-in-sparql-using-spin.html.
15. Bishop, B., Bojanov, S.: Implementing OWL 2 RL and OWL 2 QL Rule-Sets for OWLIM. In: Dumontier, M. and Courtot, M. (eds.) OWLED. CEUR-WS.org (2011).
16. Seitz, C., Schönfelder, R.: Rule-Based OWL Reasoning for Specific Embedded Devices. 10th International Semantic Web Conference, Bonn, Germany, Proceedings, Part II. pp. 237–252. Springer (2011).
17. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S.: OWL 2 Web Ontology Language Primer (Second Edition), http://www.w3.org/TR/owl2-primer/.
18. O'Connor, M., Das, A.: A Pair of OWL 2 RL Reasoners. OWL: Experiences and Directions Workshop 2012 (2012).
19. Online Documentation: Shared RETE Memory, https://niche.cs.dal.ca/materials/rete-shared-mem/.
20. Schneider, M., Mainzer, K.: A Conformance Test Suite for the OWL 2 RL RDF Rules Language and the OWL 2 RDF-Based Semantics. 6th International Workshop on OWL: Experiences and Directions (2009).
21. Miranker, D.P.: TREAT: A Better Match Algorithm for AI Production Systems (Long Version). University of Texas at Austin, Austin, TX, USA (1987).
22. Ishida, T.: An optimization algorithm for production systems. IEEE Trans. Knowl. Data Eng. 6, 549–558 (1994).
23. Miranker, D., Depena, R., Jung, H., Sequeda, J.F., Reyna, C.: Diamond: A SPARQL Query Engine, for Linked Data Based on the Rete Match. Artificial Intelligence meets the Web of Data Workshop, co-located at ECAI (2012).
24. Özacar, T., Öztürk, Ö., Ünalir, M.O.: Optimizing a Rete-based Inference Engine using a Hybrid Heuristic and Pyramid based Indexes on Ontological Data. J. Comput. 2, (2007).
25. Red Hat: Drools - RETE Algorithm, https://docs.jboss.org/drools/release/latest/drools-docs/html_single/#_reteoo.
26. Hanson, E.N.: Rule Condition Testing and Action Execution in Ariel. SIGMOD Rec. 21, 49–58 (1992).
27. Apache Jena Inference Support, https://jena.apache.org/documentation/inference/.
28. Friedman-Hill, E.J.: Jess - RETE Algorithm, http://www.jessrules.com/docs/71/rete.html.
29. Matentzoglu, N., Bail, S., Parsia, B.: A Snapshot of the OWL Web. The Semantic Web – ISWC 2013 – 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I. pp. 331–346 (2013).
30. Kim, T., Park, I., Hyun, S.J., Lee, D.: MiRE4OWL: Mobile Rule Engine for OWL. Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference

Workshops. pp. 317–322. IEEE Computer Society, Washington, DC, USA (2010).
31. Ali, S., Kiefer, S.: microOR --- A Micro OWL DL Reasoner for Ambient Intelligent Devices. Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing. pp. 305–316. Springer-Verlag, Berlin, Heidelberg (2009).
32. Kim, T., Park, I., Hyun, S.J., Lee, D.: MiRE4OWL: Mobile Rule Engine for OWL. 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops. pp. 317–322. IEEE (2010).
33. Matheus, C.J., Baclawski, K., Kokar, M.M.: BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rules. Rules and Rule Markup Languages for the Semantic Web, Second International Conference on. pp. 67–74 (2006).
34. Hanson, E.N.: The Design and Implementation of the Ariel Active Database Rule System. IEEE Trans. Knowl. Data Eng. 8, 157–172 (1996).
35. Hanson, E.N., Bodagala, S., Chadaga, U.: Trigger Condition Testing and View Maintenance Using Optimized Discrimination Networks. IEEE Trans. Knowl. Data Eng. 14, 261–280 (2002).
36. Hanson, E., Hasan, M.S.: Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing. (1993).
37. Bak, J., Nowak, M., Jedrzejek, C.: RuQAR: Reasoning Framework for OWL 2 RL Ontologies. The Semantic Web: ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers. pp. 195–198. Springer (2014).
38. Komazec, S., Cerri, D.: Towards Efficient Schema-Enhanced Pattern Matching over RDF Data Streams. 1st International Workshop on Ordering and Reasoning (OrdRing'11) (2011).
39. Wright, I., Marshall, J.: The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. Int. J. Intell. Games Simul. 2(1), 36–48 (2003).
40. Miranker, D.P., Brant, D.A., Lofaso, B., Gadbois, D.: On the Performance of Lazy Matching in Production Systems. Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 1. pp. 685–692. AAAI Press (1990).
41. Doorenbos, R.B.: Combining Left and Right Unlinking for Matching a Large Number of Learned Rules. In: Hayes-Roth, B. and Korf, R.E. (eds.) Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1. pp. 451–458. AAAI / MIT Press (1994).