Benchmarking Semantic Reasoning on Mobile Platforms: Towards Optimization Using OWL2 RL

William Van Woensel[a,*] and Syed Sibte Raza Abidi[a]
[a]*NICHE Research Group, Faculty of Computer Science, Dalhousie University, 6050 University Ave, Halifax, NS B3H 4R2, Nova Scotia, Canada*

Abstract. Mobile hardware has advanced to a point where apps may consume the Semantic Web of Data, as exemplified in domains such as mobile context-awareness, m-Health, m-Tourism and augmented reality. However, recent work shows that the performance of ontology-based reasoning, an essential Semantic Web building block, still leaves much to be desired on mobile platforms. This presents a clear need to provide developers with the ability to benchmark mobile reasoning performance, based on their particular application scenarios, i.e., including reasoning tasks, process flows and datasets, to establish the feasibility of mobile deployment. In this regard, we present a mobile benchmark framework called MobiBench to help developers to benchmark semantic reasoners on mobile platforms. To realize efficient mobile, ontology-based reasoning, OWL2 RL is a promising solution since it (a) trades expressivity for scalability, which is important on resource-constrained platforms; and (b) provides unique opportunities for optimization due to its rule-based axiomatization. In this vein, we propose selections of OWL2 RL rule subsets for optimization purposes, based on several orthogonal dimensions. We extended MobiBench to support OWL2 RL and the proposed ruleset selections, and benchmarked multiple OWL2 RL-enabled rule engines and OWL reasoners on a mobile platform. Our results show significant performance improvements by applying OWL2 RL rule subsets, allowing performant reasoning for small datasets on mobile systems.

Keywords: mobile computing; OWL2 RL; rule-based reasoning; OWL reasoning; reasoning optimization

1. Introduction

Advances in mobile technologies have enabled mobile applications to consume semantic data, with the goal of e.g., collecting context- [52], [59] and location-related data [6], [50], achieving augmented reality [42], [58], performing recommendations [60], accessing linked biomedical data (m-Health) [40] and enabling mobile tourism [28]. Automated reasoning, an essential Semantic Web pillar, involves the inference of useful information based on the semantics of ontology constructs, domain-specific if-then rules, or both. Giving the availability of advanced mobile technology and large volumes of mobile-accessible semantic data, we hence consider it opportune to investigate the potential of semantic reasoning on mobile, resource-constrained platforms. In light of recent empirical work [11], [26], which indicates that mobile reasoning performance still leaves much to be desired, we choose to focus on benchmarking and optimizing mobile semantic reasoning. In particular, we discern a clear need for benchmarking specific application scenarios, including reasoning task (e.g., ontology or rule-based reasoning), process flow (e.g., frequent vs. incremental reasoning) and rule- and datasets, as it will allow mobile developers to make more informed decisions – for instance, in case of poor performance of their particular application scenario, they may choose hybrid solutions that combine mobile- and server-deployed reasoning [2], [56].

In traditional Semantic Web reasoning applications, OWL2 DL is the most popular representation and reasoning approach. Regarding resource-constrained systems however, it has been observed that OWL2 DL

* Corresponding author. E-mail: william.van.woensel@gmail.com.

is too complex and resource-intensive to achieve scalability [11], [26]. Reflecting this, most mobile semantic reasoners i.e., tailored to resource-constrained systems, instead focus on rule-based OWL axiomatizations, such as custom entailment rulesets [1], [29] or OWL2 RL rulesets [37], [48]. Indeed, OWL2 RL is an OWL2 profile with a stated goal of scalability, partially axiomatizing the OWL2 RDF-based semantics as a set of rule axioms. Further, a rule-based axiomatization allows easily adjusting reasoning complexity to the application scenario [48] or avoiding resource-heavy inferences [8], [44], by applying subsets of rule axioms. In contrast, transformation rules used in tableau-based DL reasoning are often hardcoded, making it hard to de-select them at runtime [48]. Also, most classic DL optimizations improve performance at the cost of memory [11], which is limited in mobile devices. At the same time, as only a partial axiomatization, OWL2 RL does not guarantee completeness for TBox reasoning [37]; and places syntactic restrictions on ontologies to ensure all correct inferences. Nevertheless, we find this expressivity trade-off acceptable in case it would render semantic reasoning feasible on resource-constrained platforms.

In this regard, our objective is three-fold: **(1)** developing a mobile reasoning benchmark framework (called MobiBench) that allows developers to evaluate the performance of reasoning on mobile platforms (reasoning times, memory usage), for specific scenarios and using standards-based rule- and datasets. Key features of MobiBench include a uniform, standards-based rule and data interface across reasoning engines, as well as its extensibility and cross-platform nature, allowing benchmarks to be applied across multiple platforms;
(2) optimizing semantic reasoning on mobile platforms, by studying the following three OWL2 RL rule subset selections: (i) *Equivalent OWL2 RL rule subset*, which leaves out logically equivalent rules; i.e., rules of which the results are covered by other rules; (ii) *Purpose and reference-based subsets*, which divides rule subsets based on their purpose and referenced data; and (iii) *Removal of resource-heavy rules* that have a large performance impact – although this will result in missing certain inferences, we feel that developers should be able to weigh their utility vs. computational cost; and
(3) performing mobile reasoning benchmarks, which measure the performance of the materialization of ontology inferences, using the AndroJena and RDFStore-JS rule systems loaded with different OWL2 RL ruleset selections, as well as three OWL2 DL reasoners (HermiT, JFact and Pellet). We note that, although the proposed OWL2 RL subset selections were construed and evaluated in the context of resource-constrained platforms, they may be applied in any kind of computing environment.

This paper is built on previous work, which presented a clinical benchmark [55] and an initial version of the Mobile Benchmark Framework [54], which only supplied an API, restricted benchmarking to rule-based reasoning, and did not attempt optimizations or applications of OWL2 RL.

The paper is structured as follows. Section 2 introduces the MobiBench framework, presenting its architecture and main components, and Section 3 discusses how mobile developers can utilize MobiBench. In Section 4, we shortly discuss the OWL2 RL profile and our reasons for focusing on it, and detail its implementation as a ruleset. Section 5 elaborates on our selection of OWL2 RL rule subsets for optimization purposes. Section 6 presents and discusses the benchmarks we performed using MobiBench. We review related work in Section 7, and end with conclusions and future work in Section 8.

2. Mobile Benchmark Framework

The goal of the MobiBench benchmark framework is to allow studying and comparing reasoning performance on mobile platforms, given particular application scenarios, including reasoning task, process flow and rule- and datasets. An important focus lies on extensibility, with clear extension points allowing different rule and data formats, tasks and flows to be plugged in. Moreover, given the multitude of mobile platforms currently in use (e.g., Android, iOS, Windows Phone, BlackBerry), MobiBench was implemented as a cross-platform system.

Fig. 1 shows the architecture overview of the MobiBench framework. The API supplies third parties with direct access to the MobiBench functionality. To facilitate developers in running benchmarks, the Automation Support allows automating large numbers of benchmarks, and comprises (1) a remote *Automation Client*, which generates a set of benchmark configurations; and (2) an *Automation Web Service* on the device that invokes the *API* for each configuration. This setup avoids re-deploying MobiBench for each benchmark (i.e., with a new hard-coded configuration); and even allows benchmarking without physical access to the device. The Analysis Tools aggregates the benchmark results, including

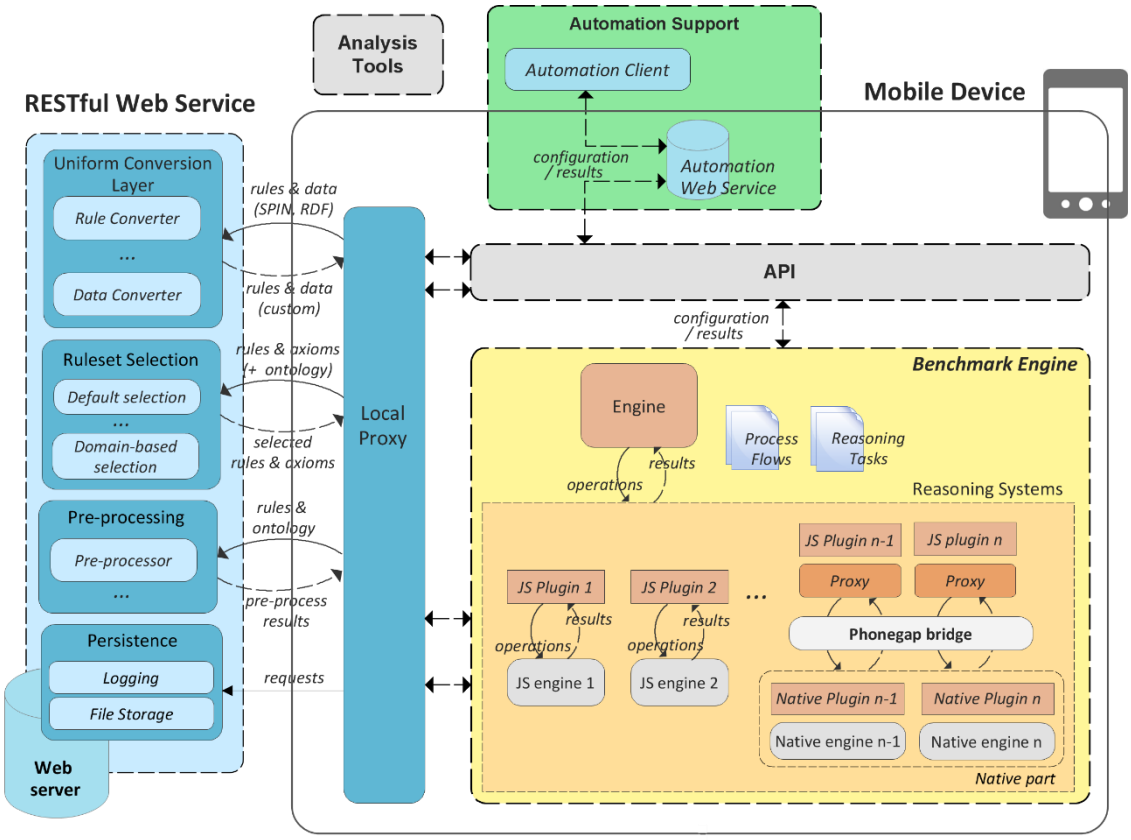

Fig. 1. MobiBench Framework Architecture.

reasoning times and memory dumps, into CSV files. The core of the framework, the Benchmark Engine, can perform different *reasoning tasks*, using different *process flows*, to better align benchmarks with real-world scenarios. Any *Reasoning System* can be plugged into this component by implementing the uniform plugin interface.

To support OWL2 RL, MobiBench was extended with the following services: **(a)** Uniform Conversion Layer, to cope with the myriad of rule (and data) formats currently supported by rule-based reasoners; **(b)** Pre-Processing Service, which pre-processes the ruleset and ontology if required (e.g., to support n-ary rules); and **(c)** Ruleset Selection Service, which automatically applies OWL2 RL subset selections to optimize ontology-based reasoning.

A remote *RESTful Web Service*, deployed on a server (e.g., the developer's PC), comprises these services, and also hosts some utility services to persist benchmark output (Persistence Support). A Local Proxy component acts as an intermediary between the mobile system and the remote Web service.

For portability across platforms, MobiBench was implemented in JavaScript (JS) and deployed using Apache Cordova [62] for mobile platforms and JDK8 Nashorn [68] for PC (this version is used for testing), which allows native, platform-specific parts to be plugged in. We note that this also allows MobiBench to easily benchmark JavaScript reasoners, which are usable in mobile websites or cross-platform, JavaScript-based apps (e.g., developed using Apache Cordova, Appcelerator Titanium [64]) with a write-once, deploy-everywhere philosophy. We currently rely on Android as the deployment platform, since most reasoners are either developed for Android or written in Java (which facilitates porting to Android), but MobiBench could be easily deployed on other platforms as well. The MobiBench framework can be found online [51].

In the subsections below, we elaborate on the main MobiBench components, namely the Uniform Conversion Layer (Section 2.1), Ruleset Selection Service (Section 2.2), Pre-Processing Service (Section 2.3) and Benchmark Engine (Section 2.4);

and indicate extension points for each component (see parts on *Extensibility*). Section 3 shows how developers can utilize the benchmark framework.

2.1. Uniform Conversion Layer

The goal of the *Uniform Conversion Layer* is to handle the multitude of rule (and data) formats currently supported by rule-based reasoners. It supplies a uniform, standards-based resource interface across reasoning engines, which dynamically converts the input to their supported formats. The major benefit of this layer is that it allows developers to re-use a single rule- and dataset across different reasoners.

A range of semantic rule standards are currently in use, including the Semantic Web Rule Language (SWRL) [24], Web Rule Language (WRL) [3], Rule Markup Language (RuleML) [12], and SPARQL Inferencing Notation (SPIN) [32]. Some reasoners also introduce their own custom formats (e.g., Apache Jena) or rely on non-Semantic Web syntaxes (e.g., Datalog: IRIS, PocketKRHyper). When benchmarking multiple systems, this multitude of formats prevents direct re-use of a single rule- and dataset. We chose to support SPIN rules and RDF data as standard input formats; Section 2.1.1 shortly discusses SPIN and our reasons for choosing it.

Since the only available SPIN API is developed for the Java Development Kit (JDK) [31], conversion functions are deployed on an external Web service. To convert incoming SPIN rules, the SPIN API is utilized to generate an Abstract Syntax Tree (AST), which is then visited by a *Rule Converter* to convert the rule. Section 2.1.2 discusses our current converters, and how new converters can be plugged in. To convert incoming RDF data, a *Data Converter* can utilize Apache Jena [4] to query and manipulate the data.

2.1.1. SPIN

SPIN is a SPARQL-based rule and constraint language, which provides a natural, object-oriented way of dealing with constraints and rules associated with RDF(S)/OWL classes. In the object-oriented design paradigm, classes define the structure of objects (i.e., attributes) together with their behavior, which includes creating/changing objects (rules) and ensuring a consistent object state (constraints). Similarly, SPIN allows directly associating locally-scoped rules and constraints to their related RDF(S)/OWL classes, using properties such as *spin:rule* and *spin:constraint*.

To serialize rules and constraints, SPIN relies on SPARQL [19], a W3C standard with sufficient expressivity to represent both queries and general-purpose rules and constraints. SPARQL is supported by most Semantic Web systems, and is well known by Semantic Web developers. As such, this rule format is more likely to be easily comprehensible to developers. Further, relying on SPIN also simplifies support for our current rule engines (see below).

2.1.2. Rule and Data Conversion

Regarding rule-based reasoners, our choice for SPIN greatly reduces conversion effort for systems with built-in SPARQL support. *RDFStore-JS* supports INSERT queries from SPARQL 1.1/Update [19], which are easy to obtain from SPIN rules in their SPARQL query syntax. Both *AndroJena* and *RDFQuery* support a triple-pattern like syntax, which likewise makes conversion from SPIN straightforward. Other rule engines lack built-in Semantic Web support, and require more significant conversion effort. Two systems, namely *PocketKrHyper* and *IRIS*, accept Datalog rules and facts in a Prolog-style input syntax. For these cases, we utilize the same first-order representation as in the W3C OWL2 RL specification [13], namely *T(s, p, o)* (since predicates may also be variables, a representation such as *predicate(subject, object)* is not an option in non-HiLog).

Currently, our converters support SPIN functions that represent primitive comparators (greater, equal, etc.) and logical connectors in FILTER clauses. Advanced SPARQL query constructs, such as (not-)exists, optional, minus and union, are not yet supported. None of the OWL reasoners (Section 2.4.1) required (data) conversion, since they can consume serializations of OWL in RDF out of the box.

Extensibility To plug in a new resource format, developers can create a new converter class implementing the uniform converter interface. The class is then added to a configuration file (*spin2s.txt* / *rdf2s.txt*), used by the Web service to dynamically load converter class definitions at startup. Each converter identifies its own format via a unique ID, allowing to match incoming conversion requests to the correct converter.

2.2. Ruleset Selection Service

To optimize OWL2 RL reasoning on mobile platforms, the *Ruleset Selection Service* automatically applies the OWL2 RL ruleset selections presented in

this paper (Section 5), given one or more selection criteria. Indeed, due to its rule-based axiomatization, the OWL2 RL profile greatly facilitates applying subsets of axioms. In Section 5, we discuss relevant selection criteria in detail, such as logical equivalence with other rules, and subsets based on purpose and reference. As before, since the only available API for SPIN (i.e., the input rule format) [31] is developed for Java, this component is deployed in the Web service.

The *Default Selection* function selects an OWL2 RL subset, given a list of selection criteria indicating rules and axioms to leave out, replace or add. The *Domain-based Selection* function leaves out rules that are not relevant to a given ontology – i.e., rules that will not yield any additional inferences (Section 5.2.2). Typically, a ruleset selection is performed once, before reasoning takes place; and in case of ontology updates that require re-executing the selection (e.g., schema updates; Section 5.2.1). Hence, the usefulness of selections will depend on whether the ontology is prone to frequent, relevant updates at runtime. This is especially true in our current setup, where this requires re-invoking the remote service at runtime, causing considerable overhead. By deploying the service directly on the mobile device, and even integrating it with the reasoner, this drawback could be mitigated (see future work).

Extensibility: To support a new selection criterion that requires an a priori analysis of the ontology, developers can create a new subclass of the *DomainBasedSelection* class. Else, the developer can simply add a new subfolder under the *owl2rl/* folder in MobiBench, which keeps a list of rules and axioms to be removed, replaced or added.

2.3. Pre-Processing Service

The *Pre-processing Service* performs pre-processing of the ruleset and target ontology to support OWL2 RL-based reasoning, if required. In particular, the service implements 3 solutions to support n-ary rules (see Section 4.2.3): (1) *instantiate the rules*, based on schema assertions found in the ontology; (2) *normalize (or "binarize") the input ontology* to only contain binary versions of the n-ary assertions, and apply binary versions of the rules; and (3) *replace each rule by 3 auxiliary rules*.

When applying solutions (1) and (2), pre-processing needs to occur initially and each time the ontology is updated. Solution (3) does not have this drawback, but infers *n+1* intermediary inferences for each "complete" inference for an *n*-ary assertion, which do not follow from the OWL2 RL semantics. The choice between these solutions thus depends on the scenario, e.g., whether the ontology is prone to frequent updates. As before, deploying this service on the mobile device could alleviate these drawbacks (see future work). Currently, it is deployed on the Web service since only a Java SPIN API is available.

Extensibility: To support a new pre-processing mechanism, developers can create a new subclass of the *PreProcessor* class. In case the mechanism requires ontology analysis (cfr. solutions (1), (2)), *OntologyBasedPreProcessor* should be subclassed.

2.4. Benchmark Engine

The Benchmark Engine performs benchmarks of reasoning engines, following a particular reasoning setup. A reasoning setup includes a reasoning task and process flow. By supporting different setups, and allowing new ones to be plugged in, benchmarks can be better aligned to real-world scenarios.

In Section 2.4.1, we elaborate on the currently supported reasoning engines. Next, we discuss the available reasoning tasks (Section 2.4.2) and process flows (Section 2.4.3), as well as the supported benchmark measurement criteria (Section 2.4.4).

2.4.1. Reasoning Engines

Below, we categorize currently supported engines according to their reasoning support. The engines not indicated as Android systems, excluding the JavaScript (JS) engines, were manually ported to Android. In this categorization, we consider rule engines as any system that can calculate the deductive closure of a ruleset, i.e., execute a ruleset and output resulting inferences (not necessarily limited to this).

Rule-based systems

AndroJena [61] is an Android-ported version of Apache Jena [4]. It supplies a rule-based reasoner, which supports both forward and backward chaining, respectively based on the RETE algorithm [17] and SLG resolution [15].

RDFQuery [70] is a JavaScript RDF store that performs queries using a RETE network, and implements a naïve reasoning algorithm.

RDFStore-JS [71] is a JavaScript RDF store, supporting SPARQL 1.0 and parts of SPARQL 1.1. We extended this system with naïve reasoning, accepting rules as SPARQL 1.1 INSERT queries.

IRIS (Integrated Rule Inference System) [9] is a Java Datalog engine meant for Semantic Web

applications. The system relies on bottom-up evaluation combined with Magic Sets [7].

PocketKrHyper [45] is a J2ME first-order theorem prover based on a hyper tableaux calculus, and is meant to support mobile semantic apps. It supplies a DL interface that accepts DL expressions and transforms them into first-order logic.

OWL reasoners

AndroJena supplies an OWL reasoner, which implements OWL Lite (incompletely) and supports *full*, *mini* and *micro* modes that indicate custom expressivities; and an RDFS reasoner, similarly with *full*, *default* and *simple* modes. For details, we refer to the Jena documentation [63].

The *ELK* reasoner [27] supports the OWL2 EL profile, and performs (incremental) ontology classification. Further, Kazakov et al. [26] has demonstrated that it can take advantage of multi-core CPUs of modern mobile devices.

HermiT [18] is an OWL2 DL reasoner based on a novel hypertableaux calculus, and is highly optimized for performing ontology classification.

JFact [66] is a Java port of the FaCT++ reasoner, which implements a tableau algorithm and supports OWL2 DL expressivity.

Pellet [46] is a DL reasoner with sound and complete support for OWL2 DL, featuring a tableaux reasoner. It also supports incremental classification.

In Section 6.3, we list the reasoning engines utilized in our benchmarks.

Extensibility: To support a new JS reasoner, the developer writes a JS *plugin* object, which implements a uniform reasoner interface and specifies the accepted rule and data format, the process subflow (if any) dictated by the engine (Section 2.4.3), and its available settings (e.g., reasoning scope (OWL, RDFS)). To rule out communication, console output, etc. influencing measurements, each plugin captures its own fine-grained result times using our *ExperimentTimer API*. Any required JavaScript libraries, as indicated by the plugin, are automatically loaded. Developers register their plugins in an *engine.json* file.

For native engines, the developer similarly implements a native plugin class, and supplies a skeleton JS plugin. The system wraps this skeleton plugin with a proxy object that delegates invocations to the native plugin over the Cordova bridge (see Fig. 1). In practice, native (Android) reasoners often have large amounts of dependencies, some of which may be conflicting (e.g., different versions of the same library). To circumvent this issue, we package each engine and its dependencies as jar-packaged .dex files, which are automatically loaded at runtime. For more details, we refer to our online documentation [51].

2.4.2. Reasoning Tasks

Currently, we support three reasoning tasks. Fig. 2 illustrates the dependencies between these tasks.

1) Rule-based materializing inference: Computing the deductive closure of a ruleset for a dataset, and adding all inferences to the dataset.

2) OWL2 materializing inference: Given an ontology, materialize all inferences based on an OWL2 expressivity (e.g., OWL2 Full, OWL2 DL, OWL Lite, or some other reduced expressivity). This task can also be performed by rule engines, e.g., using the rules axiomatizing the OWL2 RL semantics. Fig. 2 shows two types of OWL inference: *"built-in"* inference of any kind (e.g., OWL2 DL, QL, Lite, etc.), which only requires an input ontology; and *OWL2 RL reasoning*, which uses a rule engine and accepts both an OWL2 RL ruleset and ontology as input.

Regarding our choice for materializing inferences vs. reasoning per query (e.g., via resolution methods such as SLG [15]), we note that each have their advantages and drawbacks on mobile platforms. Prior to data access, the former involves an expensive pre-processing step that may significantly increase the dataset scale, which is problematic on mobile platforms, but then leaves query answering purely depending on speed of data access. In contrast, the latter incurs a reasoning overhead for each query that depends on dataset scale and complexity. Another materialization drawback is that inferences need to be (re-)computed whenever new data becomes available. For instance, Motik et al. [37] combine materialization with a novel incremental reasoning algorithm, to efficiently update previously drawn conclusions. To allow benchmarking such incremental methods, our framework supports an "incremental reasoning" process flow (Section 2.4.3). For the purposes of this paper, we chose to focus on a materialization approach, although supporting resolution-based reasoning is considered future work. We note that many Semantic Web rule-based reasoners, including DLEJena [36], SAOR [23], OwlOntDb [16] and RuQAR [5], also follow a materialization approach.

3) Service matching: Checks whether a user goal, which describes the services the user is looking for, matches a service description. In its rule-based implementation, a pre- or post-condition / effect from one description (e.g., goal) acts as a rule; and its counterpart condition from the other (e.g., service)

serves as a dataset, which is done by "freezing" variables, i.e., replacing them by constants. A match is found when rule execution infers the consequent. We note that this rule-based task can easily be enhanced with ontology reasoning – i.e., by including an OWL2 RL ruleset with the match rule(s), and relevant ontology elements in the match dataset – which is one of the additional advantages of utilizing a rule-based OWL axiomatization. In mobile settings, service matching enables mobile apps to locate useful services in a smart environment, with all necessary computation taking place on the mobile platform (see e.g., [53]). While our benchmarks do not measure the performance overhead of service matching, this is considered future work.

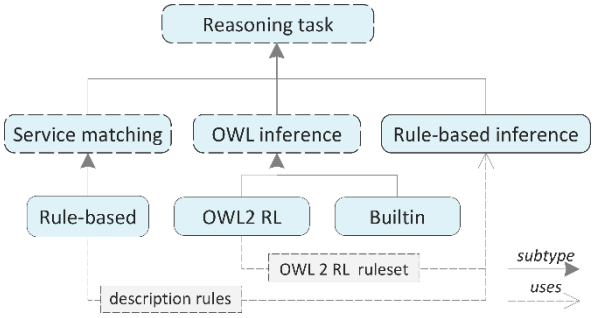

Fig. 2. Reasoning types.

Extensibility: Reasoning tasks are implemented as JS classes, with a hierarchy as shown in Fig. 2. A new reasoning task class needs to implement the *inference* function, which realizes the task by either directly invoking the reasoner interface (see Section 2.4.1), delegating to another task class (e.g., *Rule-based inference*) or to a subflow (see Section 2.4.3 – *Extensibility*). The *Reasoning task* super class provides functions such as checking conformance, collecting result times, and logging inferences. A new task file should be listed in *tasks.json*.

2.4.3. Process Flows

To better align benchmarks with real-world use cases, MobiBench supports several process flows, which dictate the times at which operations (e.g., load data, execute rules / perform reasoning) are performed. From previous work [54], [55], and in line with our choice for materializing inferences, we identified two useful process flows:

Frequent Reasoning: in this flow, the system stores all incoming facts directly in a data store (which possibly also includes an initial dataset). To generate new inferences, reasoning is periodically applied to the entire datastore. Concretely, this entails loading a reasoning engine with the entire datastore each time a certain timespan has elapsed, applying reasoning, and storing new inferences into the datastore.

Incremental Reasoning: here, the system applies reasoning for each new fact (currently, MobiBench only supports monotonic reasoning, and thus does not deal with deletions). In this case, the reasoning engine is first loaded into memory (possibly with an initial dataset). Then, reasoning is (re-)applied for each incoming fact, whereby the new fact and possible inferences are added to the dataset. Some OWL reasoners directly support incremental reasoning, such as ELK and Pellet. As mentioned, Motik et al. [37] implemented an algorithm to optimize this kind of reasoning, initially presented by Gupta et al. [21].

Further, we note that each reasoner dictates a *subflow*, which imposes a further ordering on reasoning operations. In case of OWL inference (implemented via e.g., tableau reasoning), data is typically first loaded into the engine, and then an inference task is performed (*LoadDataPerform-Inference*). Similarly, *RDFQuery*, *RDFStore-JS* and *AndroJena* first load data and then execute rules. For the *IRIS* and *PocketKrHyper* engines, rules are first loaded (e.g., to build the Datalog KB), after which the dataset is loaded and reasoning is performed (*LoadRulesDataExecute*). For more details, we refer to previous work [54].

Extensibility: Process flows are implemented as JS classes. Each main process flow is listed in *flows.json*, and will call a reason task at certain times (e.g., frequent vs. incremental) and with particular parameters (e.g., entire dataset vs. new fact). A subflow is specific to a particular reasoning task (see Section 2.4.2). A *Reason task* may thus utilize a subflow class behind-the-scenes, in case multiple subflows are possible. When called, a subflow class executes the uniform reasoning functions (e.g., load-data, execute) in the appropriate order.

2.4.4. Measurement Criteria

The Benchmark Engine allows studying and comparing the metrics listed below.

Performance:

Loading times: time needed to load data and rules, ontologies, etc. into the engine.

Reasoning times: time needed to infer new facts or check for entailment.

Memory consumption: total memory consumed by the engine after reasoning. Currently, it is not feasible

to measure this criterium for non-native engines; we revisit this issue in Section 6.4.

Conformance:

The *Benchmark Engine* allows to automatically compare inferences to the expected output for conformance checking (Section 5.4). As such, MobiBench allows investigating the completeness and soundness of inference as well (cfr. [20]).

Other related works focus on measuring the fine-grained performance of specific components [34], such as large joins, Datalog recursion and default negation. In contrast, MobiBench aims to find the most suitable reasoner on a mobile platform given an application scenario (e.g., reasoning setup, dataset). Our performance metrics support this objective.

We further note that the performance of the remotely deployed services, i.e., the Uniform Conversion Layer (Section 2.1), Ruleset Selection (Section 2.2) and Pre-Processing (Section 2.3) services are not measured. The Uniform Conversion Layer will not be included in actual reasoning deployments since it only aims to facilitate benchmarking; e.g., for production systems, rulesets can be converted a priori and then stored locally. Regarding the Ruleset Selection and Pre-Processing services, we note that these services are invoked once, before reasoning takes place; and then each time a *relevant* ontology update (e.g., schema update) occurs at runtime, i.e., which requires re-executing the operation. In scenarios where such updates may take place, we currently do not utilize selections or pre-processing options that would require re-invoking the service (see Section 5.2.1) at runtime. Therefore, and in light of future work to improve these services (e.g., by directly integrating them with the reasoner), we do not measure their performance.

Finally, we note that this paper focuses in particular on performance times and memory consumption on mobile platforms. Clearly, battery usage is an important aspect on mobile platforms as well. In the state of the art, a recent study [39] reported a near linear relation between consumed energy and OWL reasoning time, meaning that energy usage estimates, based on our captured performance times, could already be realistic. Nevertheless, future work involves supporting battery measurements as well.

3. Using MobiBench for Benchmarking

While the previous section indicated how MobiBench can be extended by third-party developers, this section describes how developers can utilize MobiBench for benchmarking. Developers may run benchmarks programmatically (Section 3.1) or use the automation support (Section 3.2). To aggregate benchmark results into summary CSV files, developers can utilize the analysis tools (Section 3.3). For more detailed instructions, we refer to our online documentation [51].

3.1. Programmatic Access

To execute benchmarks programmatically, developers call the MobiBench's *execBenchmark* function with a configuration object, specifying options for reasoning and resources. Below, we show an example (Code 1):

```
config: {
  engine: 'androjena', nrRuns: 10, warmupRun: true,
      dumpHeap: true,
  reasoning: {
    task: 'ontology_inference',
    mechanism: {
      ontology_inference: {
        type: 'owl2rl', dependency: 'rule_inference'
      },
      rule_inference: {
        mainFlow: 'frequent',
        subFlow: 'load_data_exec_rules'
      } } },
  resources: {
    ontology: {
      path: 'res/owl/data/0.nt',
      type:'data', format:'RDF', syntax:'N-TRIPLE'
    },
    owl2rl : {
      axioms: {
        path: 'res/owl/owl2rl/full/axioms.nt',
        type:'data', format:'RDF', syntax:'N-TRIPLE'
      },
      rules: {
        path: 'res/owl/owl2rl/full/rules.spin',
        type: 'rules', format: 'SPIN' },
     preprocess: 'inst-rules',
     selections: [ 'inf-inst', 'entailed' ]
  },
  confPath: 'res/owl/conf/ontology_inference/0.nt'
  outputInf: 'res/output/ontology_inference/...'
  id: '...' }
```

Code 1. Example benchmark configuration object.

This object specifies the unique engine id, the number of experiment runs, possibly including a "warmup" run (not included in the collected metrics), and whether memory usage should be measured (*dumpHeap*). The *reasoning* part indicates the high-level reasoning task (i.e., *ontology_inference*) and concrete mechanism (i.e., *owl2rl*), as well as details on dependency tasks (i.e., *rule_inference*), including its main and sub process flow.

The *resources* section lists the resources to be used in the benchmark; in this case, an ontology and OWL2 RL axioms and rules. Further, the section specifies

that the *inst-rules* pre-processing method (i.e., instantiate rules; Section 4.2.3, (1)) should be applied, as well as selections *inf-inst* (i.e., *inference-instance* subset) and *entailed* (i.e., leaving out logically redundant rules) (Section 5). Both involve calling the respective services on the Web service. It may also indicate the path for storing inferences (*outputInf*); as well as the expected reasoning output (*confPath*), to allow for automatic conformance checking.

3.2. Automation Support

Due to the potential combinatorial explosion of configuration options, including engines and their possible settings, resources and OWL2RL subsets, manually writing configurations quickly becomes impractical. For that purpose, we implemented an *Automation Support* component.

This solution includes an *Automation Client*, deployed on a server or PC, which generates a set of benchmarks based on an automation configuration; and communicates these benchmarks over HTTP with the *Automation Web Service* on the mobile device, which locally invokes the MobiBench *API* and returns the benchmark results. In the *Automation Client* code, developers specify ranges of configuration options, whereby each possible combination will be used to run a benchmark. Code 2 shows (abbreviated) example code for running a set of OWL2 RL benchmarks:

```
1.  OWL2RLRunConfig config = new OWL2RLRunConfig();
2.  config.setTask("owl_inference", "owl2rl");
3.  config.select({ "entailed" },
       { "inf-inst", "entailed", "domain-based" });
5.  config.addDataset("ore", 0, 188); ...
```

Code 2. Example automation configuration.

In this case, one subset leaves out entailed, logically redundant rules (*entailed*), and the second applies the *inf-inst* (i.e., inference-instance subset), *entailed* and *domain-based* (i.e., selecting a domain-based subset) selections. Both rulesets are applied on all benchmark ontologies, creating a total of 378 benchmarks.

3.3. Analysis Tools

To deal with large amounts of benchmark results, the MobiBench *Analysis Tools* assemble benchmark results into a CSV file. This file lists the performance results and memory usages per configuration; including process flow and reasoning task, rule subsets, engine-specific options, and datasets.

Further, the *Analysis Tools* include a utility function to easily compare performance times of two reasoning configurations (e.g., different OWL2 RL subsets), and output both the individual (i.e., per benchmark ontology) and total (i.e., aggregated) differences in performance. The *Analysis Tools* are available both as source code and a command line utility. See our online documentation [51] for more information.

4. OWL2 RL Realization

We argue that OWL2 RL is a promising solution for ontology-based reasoning on resource-constrained devices, as it targets scalability at the expense of expressivity; while its rule-based axiomatization also provides unique opportunities for optimization, as discussed in Section 5. Although its reduced expressivity leads to a lack of completeness of TBox reasoning [37] and places syntactic restrictions on ontologies, we find this trade-off acceptable if it would lead to ontology-based reasoning becoming feasible on resource-constrained platforms.

In this section, we discuss our realization of the OWL2 RL profile. First, we shortly discuss the OWL2 RL profile (Section 4.1), and then elaborate on our practical implementation (Section 4.2).

4.1. OWL2 RL Profile

The OWL2 Web Ontology Language Profiles document [13] introduces three OWL2 profiles, namely OWL2 EL, OWL2 QL and OWL2 RL. By restricting ontology syntax and reducing expressivity, these profiles can more efficiently 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. Reasoning in OWL2 RL has been found to be decidable, in particular, PTIME-complete with regards to data and taxonomic complexity, and co-NP-complete (PTIME-complete for atomic class expressions) regarding combined complexity [14]. Using OWL2 RL, reasoning systems can be implemented using standard rule engines. The W3C specification [13] presents the OWL2 RL axiomatization as a set of universally quantified, first-order implications over a ternary predicate T, which stands for a generalization of RDF triples. In addition to regular inference rules, OWL2 RL includes rules that are always applicable (i.e., without antecedent), and consistency-checking rules (i.e., with consequent *false*). Below, we exemplify each type of rules (namespaces omitted for brevity) for later reference.

Code 3 shows a "regular" inference rule that types resources based on the *subClassOf* construct:

$$T(?c_1, subClassOf, ?c_2), T(?x, type, ?c_1) \rightarrow T(?x, type, ?c_2)$$

Code 3. Rule classifying resources (*#cax-sco*).

The second type of rule lacks an antecedent and is thus always applicable. E.g., the rule in Code 4 indicates that each built-in OWL2 RL annotation property has the *owl:AnnotationProperty* type:

$$T(?ap, type, AnnotationProperty)$$

Code 4. Rule typing annotation properties (*#prp-ap*).

Thirdly, the consistency-checking rule in Code 5 checks whether an instance of a restriction, indicating a maximum cardinality of 0 on a particular property, participates in said property. If so, the ontology is flagged as inconsistent.

$$T(?x, maxCardinality, 0), T(?x, onProperty, ?p), \\ T(?u, type, ?x), T(?u, ?p, ?y) \rightarrow false$$

Code 5. Rule based on maxCardinality restriction to check consistency (*#cls-maxc1*).

4.2. Practical Realization of OWL2 RL

To implement the OWL2 RL axiomatization for general-purpose rule engines, where no particular internal support can be assumed, three types of rules may pose problems: 1) rules that require internal datatype support; 2) rules that are always applicable; and 3) rules referring to lists of elements. Below, we present these issues and our solutions, and we end with a description of our final ruleset implementation.

4.2.1. Rules requiring datatype support

The datatype inference rule *#dt-type2* (Code 6) requires literals with data values from a certain value space to be typed with the datatype of that value space (e.g., typing an integer "42" with *xsd:int*):

$$T(?lt, type, ?dt)$$

Code 6. Rule typing each literal with its corresponding datatype (*#dt-type2*).

Similarly, a second rule (*#dt-not-type*) flags an inconsistency when a literal is typed with the wrong datatype. Two other datatype rules (*#dt-eq* and *#dt-diff*) indicate equality and inequality of literals based on their values; which requires differentiating literals from URIs, to avoid these rules to fire for URI resources as well. These four rules thus require built-in support for RDF datatypes and literals, meaning they cannot be consistently implemented across arbitrary rule engines. Therefore, we chose to leave these rules out of our OWL2 RL ruleset. Related work, including DLEJena [36], the SPIN OWL ruleset by Knublauch [30] and OWLIM OWL2 RL ruleset [8] also do not include datatype rules.

4.2.2. Always-applicable rules

A number of OWL2 RL rules lack an antecedent, and are thus always applicable. One subset of these rules lack variables (e.g., specifying that *owl:Thing* has type *owl:Class*), and may thus be directly represented as axiomatic triples to accompany the OWL2 RL ruleset. A second subset comprises "quantified" variables in the consequent; e.g., stating that each annotation property has type *owl:AnnotationProperty* (Code 4). Likewise, these were implemented by axioms that properly type each annotation property (built-in for OWL2 [22]) and datatype property (supported by OWL2 RL [13]).

4.2.3. Rules referencing element lists

This set of rules includes so-called *n-ary* rules, which refer to a finite list of elements. A first subset (L1) of these rules lists restrictions on individual list elements (*#eq-diff2*, *#eq-diff3*, *#prp-adp*, *#cax-adc*, *#cls-uni*). For instance, rule *#eq-diff2* flags an ontology inconsistency if two equivalent elements of an *owl:AllDifferent* construct are found.

In contrast, rules from the second subset (L2) include restrictions referring to all list elements (*#prp-spo2*, *#prp-key*, *#cls-int1*), and a third ruleset (L3) yields inferences for all list elements (*#cls-int2*, *#cls-oo*, *#scm-int*, *#scm-uni*). E.g., for (L2), rule *#cls-int1* infers that *y* is an instance of an intersection in case it is typed by *each* intersection member class; for (L3), for any union, rule *#scm-uni* (Code 8) infers that *each* member class is a subclass of that union.

To support rulesets (L1) and (L3), we added two list-membership rules (Code 7) that recursively link each element to preceding list cells, eventually linking the first cell to all list elements:

$$T(?l, first, ?m) \rightarrow T(?l, hasMember, ?m) \quad \textbf{a)}$$

$$T(?l_1, rest, ?l_2), T(?l_2, hasMember, ?m) \rightarrow \\ T(?l_1, hasMember, ?m) \quad \textbf{b)}$$

Code 7. Two rules for inferring list membership.

Using these rules, *#scm-uni* (L3) may be formulated as follows (Code 8):

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

Code 8. Rule inferring subclasses based on union membership (*#scm-uni*).

Since the supporting rules (Code 7) eventually link all list elements to the first list cell (i.e., *?l*) using *hasMember* assertions, the rule yields inferences for all union member classes.

However, extra support is required for (L2). For these kinds of n-ary rules, we supply three solutions, each with their own advantages and drawbacks:

(1) *Instantiate the rules* based on n-ary assertions found in the ontology. Per OWL2 RL rule, this generates a separate rule for each related n-ary assertion, by constructing a list of the found length and instantiating variables with concrete schema references. E.g., a property chain axiom P with properties P_{1-3} will yield the following rule (Code 9):

$$T(?u1, P1, ?u2), T(?u2, P2, ?u3), T(?u3, P3, ?u4) \rightarrow T(?u1, P, ?u4)$$

Code 9. Instantiated rule supporting a specific property chain axiom (*#prp-spo2*).

Some related works apply this approach (a.k.a. "rule-templates") for any n-ary rule [38], or even all (applicable) OWL2 RL rules [5], [36], [37].

A drawback of this approach is that it requires pre-processing the ruleset for each ontology, and whenever it changes. Although our selections also include a pre-processing option (Section 5.2), this is only needed for optimization. Of course, the severity of this drawback depends on the frequency of ontology updates. In addition, it yields an extra rule for each relevant assertion, potentially inflating the ruleset. On the other hand, instantiated rules contain less variables, and may also reduce the need for joins, as for *#prp-spo2* (see also [37]). Further, in case no related assertions are found, no rules will be added the ruleset. Future work includes studying the application of this approach to all rules (Section 8).

(2) *Normalize (or "binarize") the input ontology* to only contain binary versions of relevant n-ary assertions. E.g., an n-ary intersection can be converted to a set of binary intersections as follows (Code 10), with I representing the original, n-ary intersection; I_i representing a binary intersection; and C_i standing for a constituent class of the original n-ary intersection:

$$I = C_1 \cap C_2 \cap \ldots \cap C_n \equiv$$
$$I = C_1 \cap I_2 \wedge I_2 = C_2 \cap I_3 \wedge \ \ldots \wedge \ I_{n-1} = C_{n-1} \cap C_n$$

Code 10. Binary version of an n-ary intersection.

With the binary version of *#cls-int1* (Code 11):

$$T(?c, intersectionOf, ?x_1), T(?x_1, first, ?c_1), T(?x_1, rest, ?x_2), T(?x_2, first, ?c_2), T(?y, type, ?c_1), T(?y, type, ?c_2) \rightarrow ?T(?y, type, ?c)$$

Code 11. Binary version of rule *#cls-int1*.

This rule may be considered recursive, since it both references and infers the same kind of assertion (i.e., $T(?y, type, ?c)$). Applying this rule on a set of binary assertions $I, I_2, \ldots, I_{n-1}$ (see Code 10) yields the following for any resource R (Code 12), with R_t representing the resource's set of all types, and, as before, I_i representing a binary intersection and C_i standing for a constituent intersection class:

$$\begin{cases} \{C_{n-1}, C_n\} \subset R_t \ \rightarrow R_t = \ \mathrm{R}_t + I_{n-1} \\ \{C_{i-1}, I_i\} \subset R_t \ \rightarrow R_t = R_t + I_{i-1} \ \ (n-1 \geq i \geq 1) \\ \{C_1, I_2\} \subset R_t \ \rightarrow R_t = R_t + I \end{cases}$$

Code 12. Inferences when applying binary *#cls-int1*.

In doing so, the rule travels up the chain of binary intersections, until it finally infers type *I* for *R*.

It is not hard to see how this approach only works for recursive rules. Rule *#prp-key* is not a recursive rule, since it infers equivalence between resources but does not refer to such relations. So, this approach only works for rules *#prp-spo2* and *#cls-int1* from (L2). Another drawback is that, similar to (1), it requires pre-processing for each ontology and its updates. In particular, each relevant n-ary assertion needs to be replaced by $n-1$ binarized versions. Further, to support a complete, single n-ary inference, this solution generates a total of $n-1$ inferences. While these are sound inferences, they may be considered to "crowd" (i.e., expand) the dataset.

(3) *Replace each rule* from (L2) by *3 auxiliary rules*. Bishop et al. [8] suggested this solution for OWLIM, based on a W3C note [41]. In this solution, a first auxiliary rule starts at the end of any list, and infers an intermediary assertion for the last element (cell *n*). Starting from the first inference, a second rule travels up the list structure by inferring the same kind of assertions for cells i ($n > i \geq 0$). In case the first cell is related to a relevant n-ary assertion (e.g., intersection, property chain), a third auxiliary rule generates the original, n-ary inference. See Bishop et al. [8] or our online documentation [51] for details.

A distinct advantage of this approach is that, in contrast to (1) and (2), it does not rely on pre-processing. However, each complete, single n-ary inference requires a total of *n+1* inferences, and these do not follow from OWL2 RL semantics (instead, they ensue from custom, auxiliary rules). As such, they can be considered to not only "crowd" but also "pollute" the dataset with unsound inferences. Bishop et al. [8] internally flag these inferences so they are skipped in query answering. Developers may want to support a similar mechanism when adopting this solution.

4.2.4. OWL2 RL Realization Outcome

Based on all observations from Section 4, we collected an OWL2 RL ruleset implementation written in the SPARQL Inferencing Notation (SPIN), based on an initial ruleset created by Knublauch [30]. This

initial ruleset relies on built-in Apache Jena functions to implement the rules from Section 4.2.3. Such built-in support cannot be assumed for arbitrary rule engines, which are targeted by our ruleset. Also, it does not specify axioms (Section 4.2.2). Our ruleset contains 69 rules and 13 supporting axioms, and can be found in Appendix A. This ruleset includes the two list-membership rules (Code 7) for n-ary rules from sets (L1) and (L3) (Section 4.2.3). To add support for a particular solution for (L2), our Web service needs to be contacted (Section 2.3) to pre-process the necessary rules or ontology, and/or add the rules (e.g., binary versions, auxiliary rules) to the ruleset. Note that our evaluation does not compare the performance of these n-ary rule solutions; this is future work.

In Section 5.4, we discuss options for checking conformance with OWL2 RL semantics.

5. OWL2 RL Optimization

This section discusses OWL2 RL ruleset selections with the goal of optimizing ontology-based reasoning. We note that, while this solution was construed and evaluated for resource-constrained platforms, it may be applied in any kind of computing environment. We consider three selections: leaving out redundant rules (Section 5.1), dividing the ruleset based on rule purpose and references (Section 5.2), and removing resource-heavy rules (Section 5.3). We note that most[1] selections represent a best-effort in reducing the size of the OWL2 RL ruleset, and do not necessarily optimize the ruleset for all types of systems. Although the total number of rules is reduced, some selections involve removing or replacing specific rules by more general rules, which could negatively impact performance. Our evaluation (Section 6) compares the effects of each subset selection.

For the purpose of these selections, we introduce the terms *owl2rl-schema-completeness* and *owl2rl-instance-completeness*, to indicate when a selection respectively derives all *schema inferences* and *instance inferences* covered by the OWL2 RL axiomatization. Although OWL2 RL reasoning infers all ABox inferences over OWL2 RL-compliant ontologies, it does not cover all TBox inferences dictated by the OWL 2 semantics [33], [37], hence our introduction of these specialized terms. Further, we discuss conformance with the OWL2 RL W3C specification (Section 5.4).

5.1. Equivalent OWL2 RL subset

As mentioned by the OWL2 RL specification [13], the presented ruleset is not minimal, as certain rules are implied by others. The stated goal of this redundancy is to make the semantic consequences of OWL2 constructs self-contained. Although this is appropriate from a conceptual standpoint, this redundancy is not useful when optimizing reasoning.

Aside from rules that are entailed by other rules (Section 5.1.1), opportunities also exist to leave out specialized rules by introducing extra axioms (Section 5.1.2) or replacement by generalized rules (Section 5.1.3). Some inference rules may also be considered redundant at the instance level, since they do not contribute to inferring instances (Section 5.1.4).

5.1.1. Entailments between OWL2 RL rules

A first set of rules is entailed by *#cax-sco* (see Code 3), each time combined with a second inference rule. For instance, *#scm-uni* (see Code 8) indicates that each class in a union is a subclass of that union. Together, these two rules entail the *#cls-uni* rule (Code 13). This rule infers that each instance of a union member is an instance of the union itself:

$$T(?c, unionOf, ?x), T(?x, hasMember, ?cl), \\ T(?y, type, ?cl) \rightarrow T(?y, type, ?c)$$

Code 13. Rule that infers membership to OWL unions (*#cls-uni*).

Code 14 shows that the rule *#cls-uni*, for each instantiation of the input variables, is covered by *#scm-uni* + *#cax-sco*:

$$T(?c, unionOf, ?x), T(?x, hasMember, ?cl) \rightarrow \\ T(?cl, subClassOf, ?c) \quad \textbf{a)}$$

$$T(?cl, subClassOf, ?c), T(?y, type, ?cl) \rightarrow \\ T(?y, type, ?c) \quad \textbf{b)}$$

Code 14. Entailment of *#cls-uni* by *#scm-uni*, *#cax-sco*.

Applying *#scm-uni* on two premises from *#cls-uni* returns inference (a). Then, *#cax-sco* is applied on the remaining premise, together with (a). This yields the inference in (b), which equals the *#cls-uni* consequent. As such, this rule may be left out without losing expressivity. Similarly, it can be shown that rules *#cls-int2*, *#cax-eqc1 and #cax-eqc2* are entailed by *#cax-sco*, each time combined with a schema-based rule.

[1] Aside from the selection presented in Section 5.3, as it focuses in particular on leaving out *resource-heavy* rules.

A second set of inference rules is entailed by the *#prp-spo1* rule, each time combined with rules that indicate equivalence between *owl:equivalent [Class|Property]* and *rdfs:sub[Class|Property]Of*. Similar to *#cax-sco*, *#prp-spo1* (Code 15) infers that resources related via a sub property are also related via its super property:

$$T(?p_1, subPropertyOf, ?p_2), T(?x, ?p_1, ?y) \rightarrow T(?x, ?p_2, ?y)$$

Code 15. Rule that infers new resource relations *(#prp-spo1*).

E.g., the *#scm-eqp1* (Code 16) rule indicates that two equivalent properties are also sub properties:

$$T(?p_1, equivalentProperty, ?p_2) \rightarrow T(?p_1, subPropertyOf, ?p_2), T(?p_2, subPropertyOf, ?p_1)$$

Code 16. Rule inferring sub properties (*#scm-eqp1*).

These two rules collectively entail the rule *#prp-eqp1* (Code 17). This rule infers that, for two equivalent properties, any resources related via the first property are also related via the second property:

$$T(?p_1, equivalentProperty, ?p_2), T(?x, ?p_1, ?y) \rightarrow T(?x, ?p_2, ?y)$$

Code 17. Rule for property membership (*#prp-eqp1*).

This entailment is shown by Code 18:

$$T(?p_1, equivalentProperty, ?p_2) \rightarrow T(?p_1, subPropertyOf, ?p_2) \quad \textbf{a)}$$

$$T(?p_1, subPropertyOf, ?p_2), T(?x, ?p_1, ?y) \rightarrow T(?x, ?p_2, ?y) \quad \textbf{b)}$$

Code 18. Entailment of *#prp-eqp1* by *#scm-eqp1*,*#prp-spo1*.

By applying *#scm-eqp1* on the first premise from *#prp-eqp1*, the inference from (a) is returned. Applying *#prp-spo1* on this inference and the remaining premise yields (b), which equals the *#prp-eqp1* consequent. This rule may thus be left out. Rule *#prp-eqp2* is similarly equivalent to these two rules.

Other rules are covered by single rule. The *#eq-trans* rule (Code 19) indicates the transitivity of *owl:sameAs*:

$$T(?x, sameAs, ?y), T(?y, sameAs, ?z) \rightarrow T(?x, sameAs, ?z)$$

Code 19. Rule indicating transitivity of owl:sameAs (*#eq-trans*)

This rule is entailed by *#eq-rep-o* (Code 20), which indicates that, for any triple, subject resources are related to any resource equivalent to the object:

$$T(?o, sameAs, ?o_2), T(?s, ?p, ?o) \rightarrow T(?s, ?p, ?o_2)$$

Code 20. Rule inferring new relations via owl:sameAs (*#eq-rep-o*)

By partially materializing the premise of *#eq-rep-o*, Code 21 shows how this rule entails *#eq-trans*:

$$T(?y, sameAs, ?z), T(?x, sameAs, ?y) \rightarrow T(?x, sameAs, ?z)$$

Code 21. Entailment of *#eq-tra*ns by *#eq-rep-o*.

When executing the *#eq-rep-o* rule on suitable data, the *?p* variable is instantiated with *owl:sameAs*, thus covering each possible inference of *#eq-trans*.

Finally, we note that some rules could potentially be removed, depending on type assertions found in the dataset. Rules *#cls-maxqc4* & *#cls-svf2* support restrictions that apply to *owl:Thing*, and thus do not require objects to be typed with the restriction class (since each resource is implicitly already an *owl:Thing*). Related rules *#cls-maxqc3* & *#cls-svf2* support restrictions that apply to a particular class, and thus require related objects to be typed with the restriction class. Since *owl:Thing* is the supertype of each class (*#scm-cls* rule), and each instance is typed by its class's supertype (*#cax-sco* rule, Code 3), any instance will be typed as *owl:Thing*. Therefore, executing the second ruleset on restrictions relating to *owl:Thing* could produce the same inferences. However, *#cax-sco* requires each instance to be explicitly typed, which often is not the case in practice. Therefore, we opted to leave these rules in the ruleset.

We note that our online documentation [51] discusses all rule equivalences in detail. In total, this selection involved leaving out 7 redundant rules.

5.1.2. Extra supporting axiomatic triples

In other cases, extra axiomatic triples can be introduced to allow for entailment by existing rules. For instance, the rule *#eq-sym* (Code 22) explicitly encodes the symmetry of the *owl:sameAs* property:

$$T(?x, sameAs, ?y) \rightarrow T(?y, sameAs, ?x)$$

Code 22. Rule indicating owl:sameAs symmetry (*#eq-sym*).

By adding an axiom stating that *owl:sameAs* has type *owl:SymmetricProperty*, Code 23 shows that any inferences generated by the *#eq-sym* rule are covered by the *#prp-symp* rule:

$$T(?p, type, SymmetricProperty), T(?x, ?p, ?y) \rightarrow T(?y, ?p, ?x)$$
$$T(sameAs, type, SymmetricProperty)$$

Code 23. Rule implementing property symmetry *(#prp-symp*) and supporting axiom.

Similarly, *#prp-inv2* is entailed by *#prp-symp* with an extra axiom, together with the *#prp-inv1* rule.

Rules *#scm-spo* and *#scm-sco*, implementing the transitivity of *rdfs:subPropertyOf* and *rdfs:subClassOf*, respectively, are entailed by *#prp-trp* with supporting axioms (Code 24):

$$T(?p, type, TransitiveProperty), T(?x, ?p, ?y), T(?y, ?p, ?z) \rightarrow T(?x, ?p, ?z)$$
$$T(subPropertyOf, type, TransitiveProperty)$$
$$T(subClassOf, type, TransitiveProperty)$$

Code 24. Transitivity rule *(#prp-trp*) and supporting axioms.

In doing so, 4 rules can be left out, at the expense of adding 4 new supporting axioms.

5.1.3. New generalized OWL2 RL rules

Opportunities also exist to generalize multiple rules into a single rule, combined with supporting axioms. We observe that rules *#eq-rep-p* (Code 25) and *#prp-spo1* (see Code 15) are structurally very similar:

$$T(?p, sameAs, ?p_2), T(?s, ?p, ?o) \rightarrow T(?s, ?p_2, ?o)$$

Code 25. Rule inferring new relations via owl:sameAs (*#eq-rep-p*).

Therefore, both rules can be generalized into a single rule, with accompanying axioms (Code 26):

$$T(?p_1, ?p, ?p_2), T(?p, type, SubLink), T(?s, ?p_1, ?o) \rightarrow T(?s, ?p_2, ?o)$$
$$sameAs\ type\ SubLink\ .$$
$$subPropertyOf\ type\ SubLink\ .$$

Code 26. Rule covering *#eq-rep-p* and *#prp-spo1* (*#prp-sl*) and supporting axioms.

In fact, several rules are structurally very similar, and may be pairwise generalized into a single rule with supporting axioms: rules *#scm-hv* and *#scm-svf2*; *#scm-avf1* and *#scm-svf1; #eq-diff2* and *#eq-diff3*; *#prp-npa1* and *#prp-npa2*; and *#cls-com* and *#cax-dw* (see [51] for details). We note that the same solution could also be applied for rule *#prp-eqp1* (Code 17) but this rule had already been removed (Code 18). In doing so, we left out 12 specialized rules while adding 6 general rules and 12 supporting axioms. After applying these selections, 52 rules remain and 16 axioms are added.

5.1.4. Equivalence with instance-based rules

So-called "stand-alone" schema inferences, which extend the ontology but do not impact the set of instances, may also be considered redundant, at least at the instance level. E.g., *#scm-dom1* (Code 27) infers that properties also have as domain the super types of their domains:

$$T(?p, domain, ?c_1), T(?c_1, subClassOf, ?c_2) \rightarrow T(?p, domain, ?c_2)$$

Code 27. Rule inferring super class domains (*#scm-dom1*).

Although this information may be a useful addition to the ontology, the new schema element will not result in new instance inferences. Code 28 shows that its resulting instance inferences are already covered by rules *#prp-dom* (a) and *#cax-sco* (b):

$$T(?s, ?p, ?o), T(?p, domain, ?c_1) \rightarrow T(?s, type, ?c_1) \quad \textbf{a)}$$
$$T(?s, type, ?c_1), T(?c_1, subClassOf, ?c_2) \rightarrow T(?s, type, ?c_2) \quad \textbf{b)}$$

Code 28. Two rules yielding same instances as *#scm-dom1*.

Thus, any variable *?s* will already be typed with super classes of the property's domain, regardless of the inferences generated by *#scm-dom1*. Similarly, rules *#scm-rng1*, *#scm-dom2* and *#scm-rng2* will not yield any new instances. By leaving out these 4 rules, this selection retains *owl2rl-instance-completeness* but clearly breaks *owl2rl-schema-completeness*.

5.2. Purpose- and reference-based subsets

In this section, we discuss selections based on purpose and reference. We differentiate between selections independent of the domain (Section 5.2.1) and that leverage domain knowledge (Section 5.2.2).

5.2.1. Domain-independent ruleset selection

Many (e.g., context-aware [37]) scenarios only involve adding or updating ABox (instance) statements at runtime, meaning that TBox reasoning may be restricted to design/startup time and whenever the ontology changes, with ABox reasoning being re-applied when new instances are added. Reflecting this, most OWL2 RL reasoners focus on separating TBox from ABox reasoning [5], [16], [23], [36], [37]. Further, data generated by the system may have a smaller likelihood of being inconsistent, thus reducing (or even removing) the need for continuous consistency checking as well.

Consequently, an opportunity exists to divide our OWL2 RL ruleset into 2 major subsets according to purpose; 1) *inference ruleset*, comprising inference rules (53 rules), and 2) *consistency-checking ruleset*, containing rules for checking consistency (18 rules[2]). The *inference ruleset* can further be subdivided along both purpose and reference, into 1.1) *instance ruleset*, consisting of rules inferring only instance assertions, while referring to both instance and schema elements (32 rules); and 1.2) *schema ruleset*, comprising rules only referencing schema elements (23 rules[2]). Since the consistency-checking ruleset only contains rules referring to both instance and schema elements, it cannot be further subdivided.

In this approach, *inference-schema* is applied on the ontology, initially and whenever the ontology changes, to materialize all schema inferences. When new instances are added, only *inference-instance* is applied on the instance assertions and materialized

[2] These rule subsets both include the two membership rules (Section 2.2.3), making them cumulatively larger.

schema. As shown in our evaluation (Section 6), executing only *inference-instance* has the potential to improve performance. Below, we show that this process still produces a complete materialization.

Definition 1. We define S as the set of all schema assertions (i.e., TBox) and I the set of all instance assertions (i.e., ABox) with $S \cap I = \emptyset$, and $A = S \cup I$ the set of all assertions. We further define schema ruleset α and instance ruleset β as follows, with $IR = \alpha \cup \beta$ the set of all inference rules in OWL2 RL:

$$\alpha = \{\, r \mid \forall c \in body(r), \quad \forall a \in A : match(a, c) \rightarrow a \in S \,\} \qquad \textbf{(1)}$$
$$\beta = \{\, r \mid \forall i \in infer(r, A) : i \in I \,\}$$

Where r is a rule from the OWL2 RL ruleset, $body(r)$ returns all clauses in the body of rule r, $match(a, c)$ returns true if assertion a matches a body clause c, and $infer(r, A)$ returns all inferences yielded by rule r on the set of assertions A. In other words, ruleset α includes rules for which each body clause is only matched by assertions from S, and ruleset β includes rules that only infer assertions from I. These conditions can be easily confirmed for our OWL2 RL rulesets [51]. Further, $k^*(X)$ denotes the deductive closure of ruleset k on assertions X (i.e., returning X extended with any resulting inferences).

Theorem 1. The deductive closure of IR on the union of any ontology schema O ($O \subseteq S$) and dataset D ($D \subseteq I$) is equivalent to the deductive closure of β on the union of materialized schema $\alpha^*(O)$ (i.e., including schema inferences) and dataset D:

$$(\alpha \cup \beta)^*(O \cup D) \equiv \beta^*(\alpha^*(O) \cup D) \qquad \textbf{(2)}$$

It is easy to see why this equivalence holds. Compared to the left operand, the set of assertions on which ruleset α is applied no longer includes inferences from β (since its deductive closure is now calculated separately), nor assertions from *D*. But this does not affect the deductive closure of α, since α only matches assertions from S with $S \cap I = \emptyset$, whereas $D \subseteq I$ and β only infers $i \in I$ (see Definition 1). ■

In the same vein, the *consistency-checking* ruleset needs to be applied on a dataset with all inferences materialized using the *inference* ruleset. It can be similarly shown that applying only *consistency-checking* on such a dataset will not result in losing any consistency errors. We note that related work often uses a separate OWL reasoner for materializing the schema [5], [16], [36]. Although this is a viable approach, we argue that this is not optimal for mobile platforms as it requires deploying two resource-heavy components (i.e., an OWL reasoner and rule engine).

At the same time, it is clear that the utility of separately applying these subsets depends on the frequency of ontology (schema) updates, since each update requires re-materializing the (schema) inferences. Although ontology changes are typically infrequent compared to instance data, this depends on the concrete scenario. In general, we define an ontology as stable when (a) it is not subject to relevant changes at runtime, i.e., changes that require re-executing the selection; or (b) these changes occur so infrequently that it remains advantageous to apply the selection. We note that the "relevancy" of a change, as well as what constitutes a "re-execution" of the selection, depends on the ruleset selection. In case of *inference-instance*, a relevant change involves a schema update, which requires re-materializing the ontology schema at runtime. In case an ontology is not stable in the context of a ruleset selection, it should not be applied. An ontology is considered volatile when it is not stable. Bobed et al. use a similar definition for static ontology properties [10].

5.2.2. Domain-based ruleset selection

By leveraging domain (i.e., ontology) knowledge, rules that do not reference the ontology and will thus not yield any inferences, may be left out as well, yielding a *domain-based* rule subset.

Manually determining such a domain-based ruleset is quite cumbersome and error-prone. Firstly, one can clearly not just check whether constructs referenced by the rule are present; e.g., the ontology may contain *owl:subClassOf* constructs, but the premise of *#scm-eqc2* requires two classes to be subclasses of each other, which is less likely. Secondly, some rules may be indirectly triggered by other rules, meaning that checking inferences per individual rule is insufficient.

Consequently, Tai et al. [48] describe a "selective rule loading" algorithm to determine this ruleset. As a type of naïve forward-chaining algorithm, it executes each rule sequentially on the initial dataset, adding any inferences. In case a rule yields results, it is added to the selective ruleset. This process continues until no more inferences are generated. We implemented this algorithm in the MobiBench framework (Section 2.2). Similar to before, the applicability of this ruleset selection depends on the "stability" of the ontology. In this case, relevant changes not only include schema updates but also insertions of certain data patterns, i.e.,

sets of instance assertions (e.g., reciprocal *owl:subClassOf* relations would make the *#scm-eqc2* rule relevant); these will require re-calculating the ruleset (with its associated overhead) at runtime.

5.3. Removal of inefficient rules

Rule *#eq-ref* (Code 29), inferring that each resource is equivalent to itself, greatly bloats the dataset:

$$T(?s,?p,?o) \rightarrow T(?s,sameAs,?s),$$
$$T(?p,sameAs,?p),T(?o,sameAs,?o)$$

Code 29. Rule inferring that each unique resource is equivalent to itself (#eq-ref).

For each unique resource, this rule creates a new statement indicating the resource's equivalence to itself. Consequently, 3 new triples are generated for each triple with unique resources, resulting in a worst-case 4x increase in dataset size (!). One could argue that there is limited practical use in materializing these statements; and it is unlikely that their absence will affect other inferences (there is one case where this may happen; see [51]). If needed, the system could e.g., be adapted to support them virtually. Therefore, we feel that developers should at least be allowed to weigh the utility of this rule versus its computational cost. We note that some production-strength OWL reasoners, such as SwiftOWLIM, have configuration options available to disable such rules as well [25].

After applying all selections cumulatively (aside from purpose- and reference-based subsets), this leaves a ruleset of 51 rules; 18 rules less than the original ruleset. Our evaluation (Section 6) studies the performance of separately and cumulatively applying these selections.

5.4. Conformance testing

To check the conformance of our original OWL2 RL ruleset and its subset selections (Sections 5.1–5.3), standard OWL2 RL conformance tests should be applied. However, many test cases listed on the W3C OWL2 Web Ontology Language Conformance page for the OWL2 RL profile [47] are not actually covered by OWL2 RL (as confirmed by one of its major contributors on the W3C mailing list [72]). Therefore, we used the OWL2 RL conformance test suite presented by Schneider et al. [43]. We note that some of these tests had to be left out, either due to the limitations of the original OWL2 RL ruleset (Section 4.2; e.g., lack of datatype support), or due to difficulties testing conformance. We detail these cases in our online documentation [51].

The original OWL2 RL ruleset (Section 4.2), as well as its conformant subsets (Sections 5.1.1–5.1.3), pass this conformance test suite. As a sanity-check regarding domain-independent ruleset selection (Section 5.2.1), the result of sequentially applying *inference-schema* rules, *inference-instance* rules and *consistency-checking* rules also passes the conformance tests. As expected, the selections presented in Section 5.1.4 (*Equivalence with instance-based rules*) loses *owl2rl-schema-completeness*, and Section 5.3 (*Removal of inefficient rules*) fully breaks conformance with the test suite. Finally, we note that conformance of the domain-based ruleset selection (Section 5.2.2) cannot be checked using this test suite, since this subset only includes rules specific to the domain ontology (while the test suite clearly checks all OWL2 RL rules). Instead, conformance of this rule subset was tested by collecting the inferences of the full ruleset (when applied on our evaluation ontologies; Section 6), and comparing them to the output of the domain-based rule subset.

6. Mobile Reasoning Benchmark Results

This section presents the benchmark results for materializing ontology inferences on mobile platforms, obtained using MobiBench.

6.1. Reasoning Task

Our benchmarks cover OWL2 materializing inference. We note that, although rule-based reasoning is not benchmarked separately, it is used to implement this task (Section 2.4.2). Our goals include:

1) Measuring the performance impact of our OWL2 RL subset selections (Section 5). To that end, we utilize two rule-based systems (Section 6.3).

2) Using the best performing OWL2 RL rulesets, benchmarking the best-effort performance of the two rule-based systems under different orthogonal cases: "stable" vs. "volatile" ontologies (Section 5.2); and OWL2 RL-conformant vs. non-conformant rulesets. In addition, we benchmark three OWL2 DL reasoners (Section 6.3) and compare the benchmark results.

Currently, we chose to only apply the *Frequent Reasoning* process flow; since most systems either support incremental reasoning only partially (e.g., Pellet: only incremental classification), or not at all. This means they will have virtually identical performance for incremental reasoning steps.

6.2. Benchmark Resources

To benchmark our reasoning task, we rely on the validated resources listed below (available for download at our online documentation [51]), including OWL ontologies (Section 6.2.1) and rulesets for OWL2 RL reasoning (Section 6.2.2).

6.2.1. OWL2 Ontologies

OWL 2 RL Benchmark Corpus [35]: Matentzoglu et al. extracted this corpus from repositories including the Oxford Ontology repository [69], the Manchester OWL Corpus (MOWLCorp) [67], and BioPortal [49], a comprehensive repository of biomedical ontologies. The corpus contains ontologies from clinical and biomedical fields (ProPreo, ACGT, SNOMED), linguistic and cognitive engineering (DOLCE) and food and wine domains (Wine), thus covering a range of use cases for ontology-based reasoning.

To suit the constrained resources of mobile platforms, we extracted ontologies with 500 statements or less from this corpus, resulting in 189 benchmark ontologies (total size: ca. 9Mb). By focusing on OWL2 RL ontologies, all ontology constructs are supported by all evaluated reasoners, i.e., OWL2 RL and DL.

In Section 6.6, benchmark ontologies are ordered 0–188, with an ontology's cardinal number indicating its relative OWL2 RL reasoning performance.

6.2.2. OWL2 RL Rulesets

To study the effects of OWL2 RL subset selections on performance (Section 5), we created multiple benchmark rulesets using our *Ruleset Selection Service* (Section 2.2). We summarize each selection below, and indicate their label used in the benchmark results. Note that, when discussing the benchmark results, the "+" symbol indicates applying one or more selections on the OWL2 RL ruleset.

All selections from (1) guarantee OWL2 RL conformance, i.e., they are complete under the OWL2 RL semantics (Section 5.4); whereas the *inst-ent* selection from (2) still guarantees *owl2rl-instance-completeness* (Section 5.1.4), i.e., ensuring all *instances* are inferred under the OWL2 RL semantics.

(1) Conformant selections

- *entailed*: leave out logically redundant rules (Section 5.1.1);
- *extra-axioms*: add extra supporting axioms, which allows leaving out specific rules (Section 5.1.2);
- *gener-rules*: add generalized rules, each replacing two or more specialized rules (Section 5.1.3);
- *inf-inst*: retain inference rules referring to both instance and schema elements (Section 5.2.1);
- *inf-schema*: retain inference rules referring only to schema elements (Section 5.2.1);
- *consist*: retain only consistency-checking rules (Section 5.2.1);
- *domain-based*: leave out rules not referenced by the ontology (Section 5.2.2).

(2) Non-conformant selections

- *inst-ent*: leave out schema-based rules not yielding extra instance inferences (Section 5.1.4);
- *ineff:* leave out inefficient rules (Section 5.3).

To support n-ary rules from (L2) (Section 4.2.3) we chose to only apply solution (1), i.e., *instantiating the ruleset*. This was done for all benchmarks, i.e., all benchmark results were obtained with a ruleset that can deal with any n-ary rule. Since the benchmark ontology corpus (Section 6.2.1) only contains 18 intersections in total (with no property-chain or has-key assertions), we chose a solution that leaves out these rules in case no related n-ary assertions are found. Due to the low number of relevant assertions in this corpus, comparing the performance impact of different solutions would not make much sense (this is future work). We also note that ontologies with intersections were manually extended with relevant instance assertions, so inferences would be made based on the (instantiated) *#cls-int1* rule.

6.3. Benchmarked Systems

In order to focus on our main goals (Section 6.1), namely studying the performance impact of the proposed OWL2 RL rule subsets and best-effort performances in light of the proposed optimizations, we limit ourselves to benchmarking only two rule-based systems (AndroJena, RDFStore-JS). An exhaustive comparison of all systems would warrant its own paper and is thus considered out of scope. For this purpose, we chose the best-performing native Android system (AndroJena) and JavaScript system (RDFStore-JS). Our reason for including a JavaScript system is because they are interesting from a development perspective; i.e., they can be directly used by cross-platform, JavaScript-based mobile apps (e.g., deployed using Apache Cordova). We put these results side-by-side with performance results for Hermit, JFact, and Pellet, the only three OWL2 DL reasoners currently supported by our framework.

6.4. Benchmark Measurements

Benchmarks capture the metrics discussed in Section 2.4.4, including loading and reasoning times and memory consumption. Regarding memory, Android Java heap dumps are used to accurately obtain memory usage of native Android engines. However, regarding JavaScript engines, heap dumps can only capture the entire memory size of the native WebView (used by Apache Cordova to run JavaScript on native platforms), not individual components. Although Chrome DevTools [65] is more fine-grained, it only records memory inside the mobile Chrome browser. Therefore, memory measurements were only possible for native Android reasoners.

6.5. Benchmark Hardware

To perform the benchmarks, we used an LG Nexus 5 (model LG-D820), with a 2.26 GHz Quad-Core Processor and 2Gb RAM. This device runs Android 6, which grants Android apps 192Mb of heap space. During the experiments, the device was connected to a power supply.

6.6. Benchmarking Results and Discussion

This section presents and discusses the benchmarks for OWL materializing inference. We show the results for individually benchmarking OWL2 RL ruleset selections for AndroJena (Section 6.6.1) and RDFStore-JS (Section 6.6.2), and summarize these results in Section 6.6.3. In Section 6.6.4, we present the best performing OWL2 RL rule subsets, given different requirements and scenarios, and set them side by side with benchmarks of OWL2 DL reasoners (HermiT, JFact and Pellet). Unless indicated otherwise, result times include ontology loading, reasoning, and inference collection.

6.6.1. AndroJena Benchmarking Results

Figures 3-5 show the performance of OWL2 RL ruleset selections for AndroJena. Fig. 3 shows that leaving out logically redundant rules (+*entailed*, i.e., applying the *entailed* selection) has a slight positive impact on performance (avg. ca. -180ms), whereas also replacing specific rules by extra axioms and general rules (+ *entailed*, *extra-axioms*, *gener-rules*) performs slightly worse (avg. ca. +180ms). This was a possibility, since this selection introduces more general, i.e., less constrained, rules (e.g., less able to leverage internal data indices). Applying a domain-specific ruleset (+*entailed*, *domain-based*) supplies a much larger performance gain (avg. ca. -0,78s). The *inf-inst* selection improves performance even more (avg. ca. -1s). The *ineff* selection loses completeness but shows the highest cumulative gain (avg. ca. -1,3s).

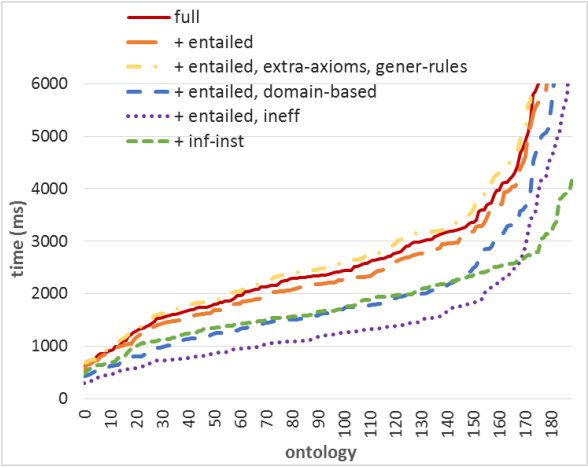

Fig. 3. *AndroJena*: OWL2 RL selections (*full*)[3].

Although the *inf-inst* selection shows promise, it requires materializing schema inferences using the *inf-schema* subset, initially and in case of ontology updates. Also, when consistency needs to be checked, the *consist* ruleset needs to be separately executed. Next, we discuss the performance of *inf-schema* and *consist*, and the effect of ruleset selections on *inf-inst*.

Fig. 4 shows the performance of materializing schema inferences (*inf-schema*). As was the case before, ruleset selections may be applied on this subset. Similar to the *full* case, replacing specific rules with extra axioms and general rules (+*extra-axioms*, *gener-rules*) reduces performance (avg. ca. +250ms, compared to *inf-schema*). For *inf-schema*, a non-conformant selection is leaving out rules inferring schema inferences that do not yield extra instances (*inst-ent*, Section 5.1.4), which slightly improves performance (avg. ca. -80ms). Since *entailed* and *ineff* do not include schema-only rules, they cannot be applied here. Applying *domain-based*, alone and when combined with *inst-ent* (+*inst-ent*, *domain-based*),

[3] Some figures chop off peaks to avoid skewing the graph. The full average results can be found at [51].

similarly improves performance slightly (avg. ca. -50ms and -100ms, respectively).

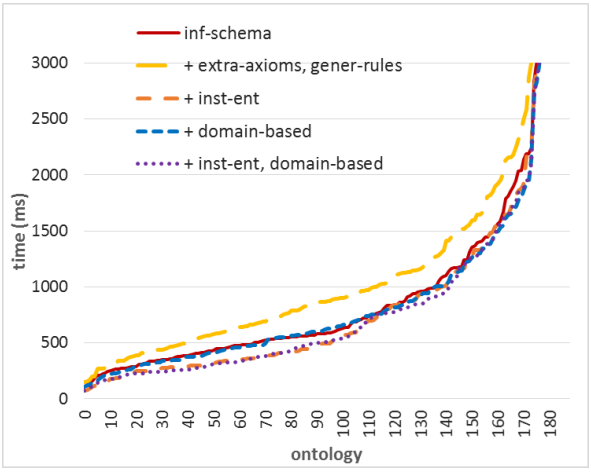

Fig. 4. *AndroJena*: OWL2 RL selections (*inf-schema*).

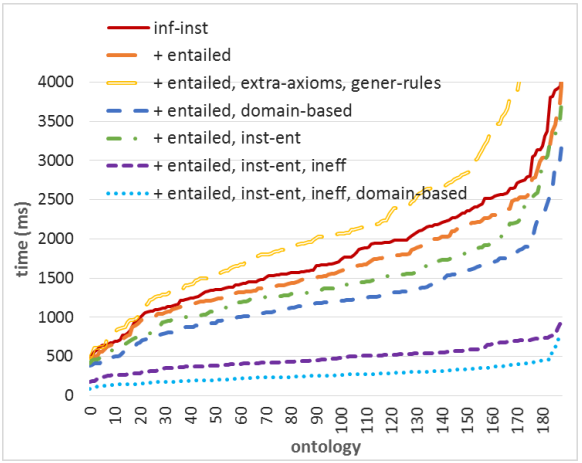

Fig. 5. *AndroJena*: OWL2 RL selections (inf-inst).

However, we note that when applying *domain-based* on the *inf-schema* subset, the *domain-based* selection would need to reconstruct the *inf-schema* ruleset for each ontology update; and the ruleset is then utilized only once[4], i.e., to materialize schema inferences in the updated ontology. Its suitability here thus depends on the performance of the *domain-based* selection, which is not measured as part of these benchmarks as it is deployed on a Web service[5].

After materializing the ontology with schema inferences, instance-related rules (*inf-inst*) are applied whenever new instances are added. When consistency needs to be checked, the *consist* ruleset selection is applied on a materialized set of schema and instance assertions (avg. ca. 420ms). We note that the only applicable selection for *consist*, i.e., *gener-rules*, results in very similar performance (avg. ca. 430ms).

Fig. 5 shows that, similar to the *full* case, leaving out redundant rules (+*entailed*) results in small improvements (avg. ca. -145ms, compared to *inf-inst*). Additionally replacing specific rules by extra axioms and general rules (+ *entailed*, *extra-axioms*, *gener-rules*) similarly leads to performance loss (avg. ca. +0,5s), while selecting a domain-based subset (+*entailed*, *domain-based*) results in gains (avg. ca. -0,5s). Regarding non-conformant cases, a first option is to execute the rule subset when the ontology schema is instead materialized by *inst-ent*, which is smaller since it lacks some schema elements (i.e., not yielding extra instances). This scenario (+ *entailed*, *inst-ent*) improves performance by avg. ca. -340ms. Also removing inefficient rules (+ *entailed*, *inst-ent*, *ineff*) increases performance by avg. ca. -1,3s. Combining all selections yields reductions of avg. ca. -1,5s.

6.6.2. RDFStore-JS Benchmarking Results

Figures 6-8 show OWL2 RL subset performances for RDFStore-JS. Fig. 6 shows that, similar to AndroJena, *entailed* yields only slightly better performance (avg. ca. -100ms), whereas *entailed*, *extra-axioms* and *gener-rules* collectively result in worse performance (avg. ca. +0,85s). At the same time, compared to AndroJena, also applying *domain-based* yields much higher performance gains (avg. ca. -5,8s), while *inf-inst* (avg. ca. -1,3s) and *ineff* (avg. ca. -1,9s) have a smaller comparative impact.

Fig. 7 shows the performance of materializing schema inferences (*inf-schema*). As for AndroJena, replacing specific rules (+*extra-axioms*, *gener-rules*) reduces performance (avg. ca. +380ms, compared to *inf-schema*), while leaving out "instance-redundant" rules (*inst-ent*) improves performance to a larger extent (avg. ca. -270ms). As before, we note that *entailed* and *ineff* are not applicable here. Utilizing *domain-based*, individually and combined with *inst-ent* (+*inst-ent*, *domain-based*) results in the largest improvements in performance (avg. ca. -0,46s and

[4] Except for scenarios where e.g., the ontology needs to be re-materialized at each startup.

[5] Future work involves studying mobile deployment, see Section 8.

-0,5s, respectively), although, as mentioned, the suitability of *domain-based* could be questioned here.

Fig. 8 shows the results of the *inf-inst* rule subsets, applied on an ontology materialized with schema inferences. In contrast to AndroJena and the *full* case for RDFStore-JS, collectively applying *entailed*, *extra-axioms* and *gener-rules* improves performance (avg. ca. -0,8s), and thus exceeds the performance gained by only +*entailed* (avg. ca. -180ms). Similar to *full* (Fig. 6), the *domain-based* selection (+*entailed*, *domain-based*) performs much better (avg. ca. -4,5s).

Considering non-conformant selections, applying the rule subset when instead materializing the ontology schema using *inst-ent* (+*entailed*, *inst-ent*) increases performance by avg. ca. -430ms (compared to *inf-inst*). Also applying the *ineff* selection (+*entailed*, *inst-ent*, *ineff*) significantly improves performance (avg. ca. -3,8s). Combining all selections reduces reasoning times by avg. ca. -5,5s. Finally, the *consist* ruleset yields a performance of avg. ca. 2,1s, with +*gener-rules* (only applicable selection) performing slightly better (avg. ca. -160ms).

6.6.3. Benchmarking Results Summary

Overall, we observe that the *entailed* selection has a relatively small performance impact, with reductions from -1,2% (rdfstore-js: *full*) to -8% (androjena: *inf-inst*). Utilizing *extra-axioms* and *gener-rules* typically results in (slightly) worse performance; which is not wholly unexpected, seeing how it replaces specific rules with more general ones (e.g., with more joins and less ability to leverage internal data indices). In some cases however, these selections perform better: i.e., when executing *inf-inst* (-21%) on RDFStore-JS.

In case of a stable ontology, additional conformant optimizations exist. Executing the *inf-inst* ruleset on a materialized ontology results in performance increases from -17% (rdfstore-js) to -36% (androjena) compared to the *full* ruleset. Here, applying the best-performing, conformant selection (i.e., *inf-inst*+*entailed*+*domain-based)* yields huge optimizations, up to -72% (rdfstore-js) compared to the non-selection case.

When dropping the conformance requirement, utilizing the *inst-ent* selection yields slight improvements in performance for *inf-schema*; 8% (androjena) and 15% (rdfstore). Re-using the smaller materialized ontology optimizes the *inf-inst* selection as well, up to -12% (androjena). Putting it all together, selection +*inf-inst, entailed, inst-ent, domain-based, ineff* yields dramatic improvements, as much as -90% (androjena) compared to the *full* case.

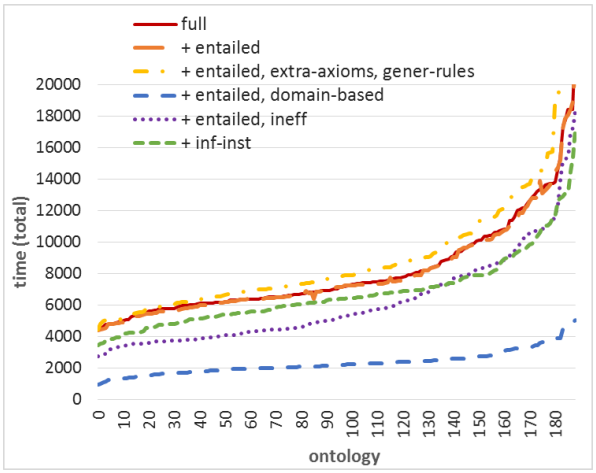

Fig. 6. *RDFStore-JS*: OWL2 RL selections (*full*).

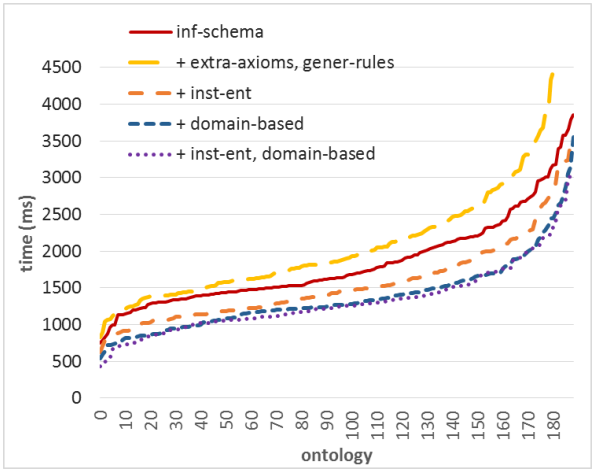

Fig. 7. *RDFStore-JS*: OWL2 RL selections (*inf-schema*).

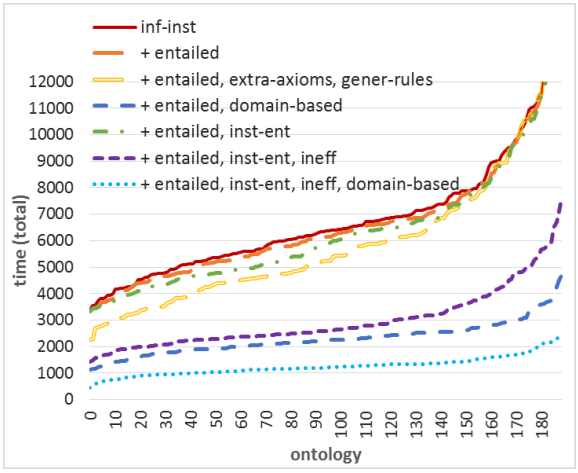

Fig. 8. *RDFStore-JS*: OWL2 RL selections (*inf-inst*).

6.6.4. Best Overall Performance

Table 1 shows the best-effort performances of the rule engines: for the full, original OWL2 RL ruleset (*original*, for reference); when applying best-performing conformant (*conformant*) and non-conformant (*non-conformant*) rule subsets; and for cases where the domain ontology frequently faces changes (*volatile ontology*) that rule out certain selections, and cases where such changes are not likely to occur (*stable ontology*) (Section 5.2.1). For brevity, we only consider a single "volatile" case, i.e., where frequent ontology updates rule out both the *domain-based* selection and separation of *inf-inst, inf-schema* and *consist* selections. For the "stable" case, times for a priori materializing the schema (*inf-schema*), inferring new instances (*inf-inst*), and consistency checking times (*consist*) are shown. Based on results from the previous section, we chose the best-performing ruleset selections for each case (see table). Both total times and constituent loading and reasoning times are indicated. Further, the table sets these results side by side with the overall performance of HermiT, Pellet and JFact, well-known OWL2 DL reasoners. These systems perform reasoning with higher complexity (OWL2 DL), which yields extra schema (TBox) inferences not covered by the OWL2 RL rule axiomatization [33], [37]. We confirmed that the OWL2 RL and OWL2 DL reasoners infer the same ABox inferences. Clearly, any comparison should take this schema incompleteness issue into account.

In line with expectations, the table shows that AndroJena, as a native Android system and featuring a non-naïve, RETE-based forward chainer, greatly outperforms RDFStore-JS, which we manually outfitted with naïve reasoning (Section 2.4.1). As shown before, the ruleset selection suiting volatile ontologies and guaranteeing conformance (*entailed*) performs only slightly better. However, if the ontology is considered stable, the conformant *inf-inst* selection supplies huge relative gains (avg. ca. 1,6s (55%) – 5,8s (72%)) compared to the *original* case, respectively for AndroJena and RDFStore-JS (percentage indicates the proportion of time gained w.r.t. *original*). At the same time, *inf-schema* yields a comparatively lower, but certainly not negligible, overhead, which is incurred for each ontology update. As mentioned, since applying the *domain-based* selection on *inf-schema* would not be advantageous in most scenarios, it is not

Table 1. Best overall performances (avg) (ms)

<table>
<tr><th colspan="6">OWL2 RL*</th><th colspan="2">OWL2 DL**</th></tr>
<tr><td rowspan="6">AndroJena</td><td>original</td><td>2819 (88 | 2731)</td><td colspan="3"></td><td rowspan="3">Hermit</td><td rowspan="3">21111</td></tr>
<tr><td></td><td>volatile ontology</td><td colspan="3">stable ontology</td></tr>
<tr><td rowspan="2">conformant</td><td>full</td><td>inf-schema</td><td>inf-inst</td><td>consist</td></tr>
<tr><td>2639 (90 | 2549)
+ entailed</td><td>1001 (69 | 932)</td><td>1245 (187 | 1058)
+ entailed, domain-based</td><td rowspan="3">418
(195 | 223)</td></tr>
<tr><td rowspan="2">non-conformant</td><td>full</td><td>inf-schema</td><td>inf-inst</td><td rowspan="3">Pellet</td><td rowspan="3">6978</td></tr>
<tr><td>1547 (93 | 1455)
+ entailed, ineff</td><td>919 (65 | 854)
inst-ent</td><td>272 (165 | 106)
+ entailed, domain-based, ineff, inst-ent</td></tr>
<tr><td rowspan="6">RDFStore-JS</td><td>original</td><td>8120 (618 | 7502)</td><td colspan="3"></td></tr>
<tr><td></td><td>volatile ontology</td><td colspan="3">stable ontology</td><td rowspan="5">JFact</td><td rowspan="5">7034</td></tr>
<tr><td rowspan="2">conformant</td><td>full</td><td>inf-schema</td><td>inf-inst</td><td>consist</td></tr>
<tr><td>8022 (620 | 7402)
+ entailed</td><td>1831 (536 | 1296)</td><td>2304 (566 | 1738)
+ entailed, domain-based</td><td rowspan="3">1947
(1282 | 665)</td></tr>
<tr><td rowspan="2">non-conformant</td><td>full</td><td>inf-schema</td><td>inf-inst</td></tr>
<tr><td>6168 (583 | 5586)
+ entailed, ineff</td><td>1561 (511 | 1050)
+ inst-ent</td><td>1255 (1080 | 176)
+ entailed, domain-based, ineff, inst-ent</td></tr>
</table>

* : [*total-time*] ([*load-time*] | [*reason-time*] ; applied selections are shown, if any.
**: *total-time*

applied here. In contrast, the best-performing conformant *inf-inst* ruleset requires the *domain-based* ruleset selection, which needs to be re-calculated for each ontology update and thus adds an extra overhead (not included here). Hence, we only apply this configuration for stable ontologies, i.e., not faced by frequent updates that would require such re-calculations at runtime. Similarly, the cost of *consist* is not negligible; the frequency of applying the ruleset depends on the scenario.

When dropping conformance, we find significant performance improvements even for volatile ontologies (avg. ca. 1,3s (45%) – 1,9s (24%)). For non-conformant reasoning in stable ontologies, the performance gain of *inf-inst* is tremendous (avg. ca. 2,5s (90%) – 6,9s (85%)). Regarding OWL2 DL reasoners, Pellet and JFact have comparable performance (around avg. ca. 7s) with HermiT being a clear outlier (avg. ca. 21s).

Table 2 shows memory usage for each engine (aside from the JavaScript-based RDFStore-JS; see Section 6.4). JFact uses the least amount of memory, i.e., only 585Kb, making it a suitable choice overall (see Table 1) for mobile platforms. Nevertheless, all memory usages appear acceptable (at least on Android), seeing how each Android app receives a 192Mb max. heap.

Table 2: Memory usage (Kb)

AndroJena	HermiT	Pellet	JFact
6242	13543	12832	585

7. Related Work

In the state of the art on rule-based OWL reasoning, most works focus on separating TBox from ABox reasoning [5], [16], [23], [36], [37]. In most cases, a separate OWL reasoner is utilized to compute and materialize schema inferences [5], [16], [36]. However, this is inadvisable on mobile platforms, since it necessitates deploying two (resource-heavy) reasoner systems, i.e., an OWL reasoner and rule engine. After this separate schema reasoning step, some works [5], [36], [37] proceed with a rule-template approach; where OWL2 RL rules are instantiated based on the materialized input ontology. In particular, multiple instantiated rules are created for each rule, replacing schema variables by concrete schema references. We support a similar solution to support certain n-ary rules, and applied it in our benchmarks. Implementing and benchmarking this as an optimization for all rules is considered future work. Tai et al. [48] propose a selective rule loading algorithm, which composes an OWL2 RL ruleset depending on the input ontology. In our benchmarks, we found that this domain-based rule selection can significantly improve performance.

Bobed et al. [11] presented a set of comprehensive benchmarks on Android devices for a number of DL reasoners, focusing on classification and consistency checking in the OWL2 DL and OWL2 EL profiles. The authors manually ported the OWL reasoners to Android and detailed their porting efforts. It was found that reasoning on PC was between 1.5 and 150 times faster than on Android, with the number of out-of-memory errors increasing on Android as well. Similarly, Kazakov et al. [26] found orders of magnitude difference between PC and Android reasoning times. Nonetheless, Bobed et al. found some promising trends: reasoners on the new Android RunTime (ART), which features ahead-of-time compilation, can be around 2 times faster than in Dalvik. In prior work [57], the same team also found a performance increase of ca. 30% between Android devices only 1 year apart.

While the benchmarks presented by Bobed et al. are comprehensive and informative, our work goes beyond the benchmarking of existing reasoners by presenting 1) a freely available, cross-platform benchmark framework for evaluating mobile reasoner performance, so others may perform detailed benchmarks given their application scenarios; and 2) selections of OWL2 RL rule subsets to optimize mobile reasoning, accompanied by comprehensive benchmarks that show their performance effects.

As mentioned, Patton et al. [39] reported that due to the single-threaded nature of most reasoners, a near linear relation exists between energy usage and computing time for OWL inferences on mobile systems. As such, energy usage estimates, based on reasoning times, could be realistic. Regardless, future work involves measuring battery usage as well.

8. Conclusion and Future Work

This paper presented the following contributions:

- The MobiBench cross-platform, extensible mobile benchmark framework, for evaluating mobile reasoning performance. Given a reasoning setup, including process flow, reasoning task, ruleset (if any) and ontology, developers can use MobiBench to benchmark reasoners on mobile platforms, and thus

find the best system for the job. The large differences in performance between reasoners and scenarios, as observed in our benchmarks, clearly point towards the need for such a framework. To facilitate the developer's job, the framework includes a uniform conversion layer, ruleset selection and pre-processing services, as well as automation and analysis tools. We indicated the extensibility for each component, allowing developers to easily plug in new variants.

- A selection of OWL2 RL subsets, with the goal of optimizing reasoning performance on mobile systems. Orthogonally, these methods include OWL2 RL-conformant vs. non-conformant selections; and selections suiting "stable" vs. "volatile" ontologies. Our benchmarks showed that, depending on ontology volatility and need for conformity, these selections can greatly improve performance.

- Mobile benchmarks, which measure reasoning performance when materializing ontology inferences; focusing on the impact of different OWL2 RL ruleset selections, as well as the computational cost of best-performing OWL2 RL rulesets for particular scenarios and systems. We put these performance results side-by-side with the performance of 3 OWL2 DL reasoners. Depending on the scenario, we found that OWL2 RL reasoning can be greatly optimized.

Despite the presented work, as well as advancements reported in the state of the art, scalable mobile performance remains elusive. A huge gap still looms between PC and mobile reasoning times. Therefore, future work includes integrating additional optimization methods into MobiBench, such as utilizing rule templates for all rules. Optimizing and porting domain-specific rule selection to the mobile platform, in light of its positive impact on performance, is also an avenue of future work. Similarly, we aim to deploy pre-processing solutions for n-ary rules directly on the mobile device, and compare their performance on an ontology corpus featuring large amounts of n-ary assertions. Measuring energy consumption, an important aspect for mobile systems, is also part of future work.

Our major focus in this paper was on materializing ontology inferences. Reasoning per query (via e.g., SLG) may also have its merits on mobile platforms, since it does not require a priori materialization. Studying its performance on mobile systems is considered a major avenue of future work. We also aim to study the utility of semantically enhancing service matching, one of the supported reasoning tasks (Section 2.4.2), by weighting the extra found matches against the ensuing performance overhead. Finally, identifying additional OWL2 RL rule subsets for particular reasoning tasks (such as instance checking and realization) is also viewed as future work.

References

[1] S. Ali and S. Kiefer, "microOR --- A Micro OWL DL Reasoner for Ambient Intelligent Devices," in *Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing*, 2009, pp. 305–316.

[2] N. Ambroise, S. Boussonnie, and A. Eckmann, "A Smartphone Application for Chronic Disease Self-Management," in *Proceedings of the 1st Conference on Mobile and Information Technologies in Medicine*, 2013.

[3] J. Angele *et al.*, "Web Rule Language (W3C Member Submission 2005)," 2005. [Online]. Available: http://www.w3.org/Submission/WRL/.

[4] Apache, "Apache Jena." [Online]. Available: https://jena.apache.org/. [Accessed: 28-Jul-2017].

[5] J. Bak, M. Nowak, and C. Jedrzejek, "RuQAR: Reasoning Framework for OWL 2 RL Ontologies," in *The Semantic Web: ESWC 2014 Satellite Events, Anissaras, Crete, Greece, May 25-29, 2014, Revised Selected Papers*, 2014, vol. 8798, pp. 195–198.

[6] C. Becker and C. Bizer, "DBpedia Mobile: A Location-Enabled Linked Data Browser.," in *LDOW*, 2008, vol. 369.

[7] C. Beeri and R. Ramakrishnan, "On the Power of Magic," in *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1987, pp. 269–284.

[8] B. Bishop and S. Bojanov, "Implementing OWL 2 RL and OWL 2 QL Rule-Sets for OWLIM.," in *OWLED*, 2011, vol. 796.

[9] B. Bishop and F. Fischer, "IRIS - Integrated Rule Inference System," in *Proceedings of the 1st Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, 2008.

[10] C. Bobed, F. Bobillo, S. Ilarri, and E. Mena, "Answering Continuous Description Logic Queries: Managing Static and Volatile Knowledge in Ontologies," *Int. J. Semant. Web Inf. Syst.*, vol. 10, no. 3, pp. 1–44, Jul. 2014.

[11] C. Bobed, R. Yus, F. Bobillo, and E. Mena, "Semantic reasoning on mobile devices: Do Androids dream of efficient reasoners?," *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 35, pp. 167–183, 2015.

[12] H. Boley, S. Tabet, and G. Wagner, "Design Rationale of RuleML: A Markup Language for Semantic Web Rules," in *Proc. Semantic Web Working Symposium*, 2001, pp. 381–402.

[13] D. Calvanese *et al.*, "OWL2 Web Ontology Language Profiles (Second Edition)," 2012. [Online]. Available: http://www.w3.org/TR/owl2-profiles/#OWL_2_RL. [Accessed: 28-Jul-2017].

[14] D. Calvanese *et al.*, "OWL 2 Web Ontology Language Profiles (Second Edition): Computational Properties," 2012. [Online]. Available: https://www.w3.org/TR/owl2-profiles/#Computational_Properties.

[15] W. Chen and D. S. Warren, "Towards Effective Evaluation of General Logic Programs," in *The 12th ACM Symposium on Principles of Database Systems (PODS)*, 1993.

[16] R. U. Faruqui and W. MacCaull, "OwlOntDB: A Scalable

Reasoning System for OWL 2 RL Ontologies with Large ABoxes," *Found. Heal. Inf. Eng. Syst.*, vol. 7789, pp. 105–123, 2013.

[17] C. L. Forgy, "Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem," *Artif. Intell.*, vol. 19, no. 1, pp. 17–37, 1982.

[18] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "HermiT: An OWL 2 Reasoner," *J. Autom. Reason.*, vol. 53, no. 3, pp. 245–269, 2014.

[19] W. S. W. Group, "SPARQL 1.1 Overview (W3C Recommendation 21 March 2013)," 2013. [Online]. Available: http://www.w3.org/TR/sparql11-overview/.

[20] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *Web Semant. Sci. Serv. Agents World Wide Web*, vol. 3, no. 2, pp. 158–182, 2005.

[21] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining Views Incrementally," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993, pp. 157–166.

[22] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, and S. Rudolph, "OWL 2 Web Ontology Language Primer (Second Edition)," 2012. [Online]. Available: http://www.w3.org/TR/owl2-primer/. [Accessed: 28-Jul-2017].

[23] A. Hogan and S. Decker, "On the Ostensibly Silent `W' in OWL 2 RL," in *Proceedings of the 3rd International Conference on Web Reasoning and Rule Systems*, 2009, pp. 118–134.

[24] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML (W3C Member Submission 21 May 2004)," 2004. [Online]. Available: http://www.w3.org/Submission/SWRL/.

[25] M. Karamfilova and B. Bishop, "SwiftOWLIM Reasoner," 2011. [Online]. Available: https://confluence.ontotext.com/display/OWLIMv35/SwiftOWLIM+Reasoner#SwiftOWLIMReasoner-PerformanceOptimizationsinRDFSandOWLSupport.

[26] Y. Kazakov and P. Klinov, "Experimenting with ELK Reasoner on Android," in *Proceedings of the 2nd International Workshop on OWL Reasoner Evaluation, Ulm, Germany, July 22, 2013*, 2013, pp. 68–74.

[27] Y. Kazakov, M. Krötzsch, and F. Simančík, "The Incredible ELK: From Polynomial Procedures to Efficient Reasoning with EL Ontologies," *J. Autom. Reason.*, vol. 53, no. 1, pp. 1–61, 2014.

[28] C. Keller, R. Pöhland, S. Brunk, and T. Schlegel, "An Adaptive Semantic Mobile Application for Individual Touristic Exploration," in *HCI (3)*, 2014, pp. 434–443.

[29] T. Kim, I. Park, S. J. Hyun, and D. Lee, "MiRE4OWL: Mobile Rule Engine for OWL," in *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, 2010, pp. 317–322.

[30] H. Knublauch, "OWL 2 RL in SPARQL using SPIN." [Online]. Available: http://composing-the-semantic-web.blogspot.ca/2009/01/owl-2-rl-in-sparql-using-spin.html. [Accessed: 28-Jul-2017].

[31] H. Knublauch, "The TopBraid SPIN API," 2014. [Online]. Available: http://topbraid.org/spin/api/.

[32] H. Knublauch, J. A. Hendler, and K. Idehen, "SPIN - Overview and Motivation (W3C Member Submission 22/02/2011)," 2011. [Online]. Available: http://www.w3.org/Submission/spin-overview/.

[33] M. Krötzsch, "The Not-So-Easy Task of Computing Class Subsumptions in OWL RL," Springer, Berlin, Heidelberg, 2012, pp. 279–294.

[34] S. Liang, P. Fodor, H. Wan, and M. Kifer, "OpenRuleBench: An Analysis of the Performance of Rule Engines," in *Proceedings of the 18th International Conference on World Wide Web*, 2009, pp. 601–610.

[35] N. Matentzoglu, S. Bail, and B. Parsia, "A Snapshot of the OWL Web," in *The Semantic Web – ISWC 2013 – 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I*, 2013, pp. 331–346.

[36] G. Meditskos and N. Bassiliades, "DLEJena: A Practical Forward-chaining OWL 2 RL Reasoner Combining Jena and Pellet," *Web Semant.*, vol. 8, no. 1, pp. 89–94, Mar. 2010.

[37] B. Motik, I. Horrocks, and S. M. Kim, "Delta-reasoner: A Semantic Web Reasoner for an Intelligent Mobile Platform," in *Proceedings of the 21st International Conference Companion on World Wide Web*, 2012, pp. 63–72.

[38] M. O'Connor and A. Das, "A Pair of OWL 2 RL Reasoners," in *OWL: Experiences and Directions Workshop 2012*, 2012.

[39] E. W. Patton and D. L. McGuinness, "A Power Consumption Benchmark for Reasoners on Mobile Devices," in *13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014.*, 2014, vol. 8796, pp. 409–424.

[40] E. Puertas, M. L. Prieto, and M. De Buenaga, "Mobile Application for Accessing Biomedical Information Using Linked Open Data," in *Proceedings of the 1st Conference on Mobile and Information Technologies in Medicine*, 2013.

[41] D. Reynolds, "OWL 2 RL in RIF (Second Edition)," 2013. [Online]. Available: http://www.w3.org/TR/rif-owl-rl/.

[42] V. Reynolds, M. Hausenblas, A. Polleres, M. Hauswirth, and V. Hegde, "Exploiting linked open data for mobile augmented reality," in *W3C Workshop: Augmented Reality on the Web*, 2010.

[43] M. Schneider and K. Mainzer, "A Conformance Test Suite for the OWL 2 RL RDF Rules Language and the OWL 2 RDF-Based Semantics," in *6th International Workshop on OWL: Experiences and Directions*, 2009.

[44] C. Seitz and R. Schönfelder, "Rule-Based OWL Reasoning for Specific Embedded Devices," in *10th International Semantic Web Conference, Bonn, Germany, Proceedings, Part II*, 2011, vol. 7032, pp. 237–252.

[45] A. Sinner and T. Kleemann, "KRHyper - In Your Pocket," in *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings*, 2005, vol. 3632, pp. 452–457.

[46] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A Practical OWL-DL Reasoner," *Web Semant.*, vol. 5, no. 2, pp. 51–53, Jun. 2007.

[47] M. Smith, I. Horrocks, M. Krotzsch, and B. Glimm, "OWL 2 Web Ontology Language Conformance (Second Edition)," *W3C Recommendation*, 2012. [Online]. Available: http://www.w3.org/TR/owl2-test/.

[48] W. Tai, J. Keeney, and D. O'Sullivan, "Resource-constrained reasoning using a reasoner composition approach," *Semant. Web*, vol. 6, no. 1, pp. 35–59, 2015.

[49] The National Center for Biomedical Ontology, "BioPortal," 2016. [Online]. Available: http://bioportal.bioontology.org/. [Accessed: 28-Jul-2017].

[50] M. Wilson, A. Russell, D. A. Smith, A. Owens, and M. C.

Schraefel, “mSpace Mobile: A Mobile Application for the Semantic Web,” in *User Semantic Web Workshop, ISWC2005*, 2005.

[51] W. Van Woensel, “MobiBench Online Documentation,” 2016. [Online]. Available: https://niche.cs.dal.ca/materials/mobi_bench/.

[52] W. Van Woensel, S. Casteleyn, E. Paret, and O. De Troyer, “Mobile Querying of Online Semantic Web Data for Context-Aware Applications,” *IEEE Internet Comput. Spec. Issue (Semantics Locat. Serv.*, vol. 15, no. 6, pp. 32–39, 2011.

[53] W. Van Woensel, M. Gil, S. Casteleyn, E. Serral, and V. Pelechano, “Adapting the Obtrusiveness of Service Interactions in Dynamically Discovered Environments,” in *Proceedings of the 9th International Conference on Mobile and Ubiquitous Systems*, 2012, pp. 250–262.

[54] W. Van Woensel, N. Al Haider, A. Ahmad, and S. S. R. Abidi, “A Cross-Platform Benchmark Framework for Mobile Semantic Web Reasoning Engines,” in *13th International Semantic Web Conference, Riva del Garda, Italy. Proceedings, Part I*, 2014, pp. 389–408.

[55] W. Van Woensel, N. Al Haider, P. C. Roy, A. M. Ahmad, and S. S. Abidi, “A Comparison of Mobile Rule Engines for Reasoning on Semantic Web Based Health Data,” in *2014 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2014)*, 2014, pp. 126–133.

[56] W. Van Woensel, P. C. Roy, S. Abidi, and S. S. Abidi, “A Mobile & Intelligent Patient Diary for Chronic Disease Self-Management,” in *15th World Congress on Health and Biomedical Informatics*, 2015.

[57] R. Yus, C. Bobed, G. Esteban, F. Bobillo, and E. Mena, “Android goes Semantic: DL Reasoners on Smartphones,” in *Proceedings of the 2nd International Workshop on OWL Reasoner Evaluation, Ulm, Germany*, 2013, pp. 46–52.

[58] S. Zander, C. Chiu, and G. Sageder, “A computational model for the integration of linked data in mobile augmented reality applications,” in *Proceedings of the 8th International Conference on Semantic Systems*, 2012, pp. 133–140.

[59] S. Zander and B. Schandl, “A framework for context-driven RDF data replication on mobile devices,” in *Proceedings of the 6th International Conference on Semantic Systems*, 2010, p. 22:1--22:5.

[60] C. Ziegler, “Semantic web recommender systems,” in *In Proceedings of the Joint ICDE/EDBT Ph.D. Workshop 2004 (Heraklion*, 2004, pp. 78–89.

[61] “AndroJena.” [Online]. Available: https://github.com/lencinhaus/androjena. [Accessed: 28-Jul-2017].

[62] “Apache Cordova.” [Online]. Available: https://cordova.apache.org/.

[63] “Apache Jena Inference Support.” [Online]. Available: https://jena.apache.org/documentation/inference/. [Accessed: 28-Jul-2017].

[64] “Appcelerator Titanium.” [Online]. Available: http://www.appcelerator.com/mobile-app-development-products/.

[65] “Chrome DevTools.” [Online]. Available: https://developer.chrome.com/devtools.

[66] “JFact.” [Online]. Available: http://jfact.sourceforge.net/.

[67] “Manchester OWL Repository.” [Online]. Available: http://mowlrepo.cs.manchester.ac.uk/datasets/mowlcorp/. [Accessed: 16-Jun-2016].

[68] “Nashorn JavaScript Engine.” [Online]. Available: http://www.oracle.com/technetwork/articles/java/jf14-nashorn-2126515.html.

[69] “Oxford Ontology repository.” [Online]. Available: http://www.cs.ox.ac.uk/isg/ontologies/. [Accessed: 16-Jun-2016].

[70] “RDFQuery.” [Online]. Available: https://code.google.com/p/rdfquery/wiki/RdfPlugin.

[71] “RDFStore-JS.” [Online]. Available: http://github.com/antoniogarrote/rdfstore-js.

[72] “W3C Forum Post on OWL2 RL test cases.” [Online]. Available: http://lists.w3.org/Archives/Public/public-owl-dev/2010AprJun/0074.html.