OWL Datatypes: Design and Implementation

Boris Motik and Ian Horrocks

University of Oxford, UK

Abstract. We analyze the datatype system of OWL and OWL 2, and discuss certain nontrivial consequences of its definition, such as the extensibility of the set of supported datatypes and complexity of reasoning. We also argue that certain datatypes from the list of normative datatypes in the current OWL 2 Working Draft are inappropriate and should be replaced with different ones. Finally, we present an algorithm for datatype reasoning. Our algorithm is modular in the sense that it can handle any datatype that supports certain basic operations. We show how to implement these operations for number and string datatypes.

1 Introduction

The Web Ontology Language (OWL) has been phenomenally successful, and the OWL DL version of OWL is nowadays routinely used for conceptual modeling in fields as diverse as biomedicine, clinical sciences, astronomy, geography, aerospace and defence. OWL DL is grounded in *description logics* (DLs) [1]—a family of theoretically well-understood knowledge representation formalisms. The popularity of OWL DL is largely due to the availability of practically effective reasoners[1] that can be used in applications.

Applications of OWL often use properties with values such as strings and integers. OWL therefore supports *datatypes*, a simplified version of the concrete domain approach [2] that can be combined with most DLs in a decidable way [5, 7]. A *datatype* can be seen as a unary predicate with a built-in interpretation; for example, the *xsd:integer* datatype is interpreted as the set of all integer values. Particular data values can be denoted at the syntax level using *constants*. Properties in OWL DL are separated into *object properties*, interpreted as relations between pairs of individuals, and *data properties*, interpreted as relations between individuals and data values. Data properties can be used in axioms such as "the range of the *a:name* property is *xsd:string*," or "each instance of the *a:Person* class must have an *xsd:integer* data value for the *a:age* property."

Practical experience with OWL DL has revealed several shortcomings of its datatype system. The datatypes in OWL DL are modeled after XML Schema [3], which provides a rich set of datatypes; however, only *xsd:string* and *xsd:integer* are normative in OWL DL, which is often not sufficient for applications. Furthermore, OWL DL provides no portable means for restricting datatypes, as in "a person who is 70 or older" or "the values of the *a:name* property should

[1] `http://www.cs.man.ac.uk/~sattler/reasoners.html`

be a string not containing a whitespace." Various OWL DL tools, such as the RACER reasoner [4], have provided proprietary solutions to these problems; however, there is currently little or no compatibility between extensions provided by different tools.

In order to address these shortcomings, as well as some shortcomings unrelated to datatypes, a W3C Working Group[2] has recently been established with the goal of developing a major revision of the language called OWL 2. The current OWL 2 Working Draft[3] lists most XML Schema datatypes as normative, and it supports XML Schema *facets* for restricting the range of built-in datatypes. For example, the minExclusive facet can be applied to *xsd:integer* to obtain a subset of integers larger than a particular value.

Extensions of DLs with concrete domains and datatypes have received in-depth theoretical treatment [2, 9, 10, 5, 7]. Furthermore, datatype groups [11] provide an architecture for integrating different datatypes, and the OWL-Eu approach [12] provides a way to restrict datatypes using expressions. These works assume that datatype reasoning can be performed using an external *datatype checking* procedure. Standard DL tableau calculi can then be extended to handle datatypes by invoking the datatype checker as an oracle.

Less attention has so far been paid to actual datatype checking algorithms. Datatype checking is a constraint satisfaction problem; however, general CSP algorithms [14] are typically too general and complex than what is necessary for datatype reasoning. Furthermore, the list of normative datatypes in the current OWL 2 Working Draft has been selected without paying attention to their suitability and implementability in a datatype checker.

In this paper we therefore formally define the OWL 2 datatype system, review its design, and investigate the problem of implementing a suitable datatype checker. Our analysis reveals several nonobvious consequences of the current design. In particular, we show that datatype checking in OWL 2 is NP-hard in the general case, but may become trivial in many (hopefully typical) cases. We also argue that certain datatypes listed as normative in the current OWL 2 Working Draft may be unsuitable from both a modeling and implementation perspective, and we suggest several changes to the Working Draft that address the problems identified. We then present a modular datatype checking algorithm that can support any datatype for which it is possible to implement a small set of basic operations that we call a *datatype handler*. Finally, we discuss how to implement datatype handlers for number and string datatypes.

The results of this paper thus provide important guidance for the designers of OWL 2, by pointing out potential design mistakes that could make implementation difficult, and for implementors of OWL 2 reasoners, by showing how to implement a suitable datatype checker.

[2] `http://www.w3.org/2007/OWL/wiki/OWL_Working_Group`
[3] `http://www.w3.org/TR/owl2-syntax/`

2 Preliminary Definitions

In this section, we present the formal definitions underlying the datatype system of OWL 2. For simplicity, we focus here only on unary datatypes; our definitions can be extended to n-ary datatypes as in [11]. We assume the reader to be familiar with the basics of DL syntax and semantics [1].

The central notion in the datatype system of OWL is the *datatype map*. In OWL 2, datatype maps can additionally support *facets*—expressions that can be applied to a datatype to restrict its interpretation.

Definition 1. *A* datatype map *is a 4-tuple* $\mathcal{D} = (N_D, N_C, N_F, \cdot^{\mathcal{D}})$*, where*

- N_D *is a set of* datatypes*,*
- N_C *is a function assigning a set of* constants $N_C(d)$ *to each* $d \in N_D$*,*
- N_F *is a function assigning a set of* facets $N_F(d)$ *to each* $d \in N_D$*,*
- $\cdot^{\mathcal{D}}$ *is a function assigning a* datatype interpretation $d^{\mathcal{D}}$ *to each datatype* $d \in N_D$*, a* facet interpretation $f^{\mathcal{D}} \subseteq d^{\mathcal{D}}$ *to each facet* $f \in N_F(d)$*, and a* data value $v^{\mathcal{D}} \in d^{\mathcal{D}}$ *to each constant* $v \in N_C(d)$*.*

By a slight abuse of notation, let $N_C = \bigcup_{d \in N_D} N_C(d)$*; the intended usage of* N_C *should be clear from the context.*

A facet expression *for a datatype* $d \in N_D$ *is a formula* φ *built using propositional connectives over the elements from* $N_F(d) \cup \{\top_d, \bot_d\}$*. The function* $\cdot^{\mathcal{D}}$ *is extended to facet expressions for* d *by setting, for* $f_{(i)} \in N_F(d)$*,* $\top_d^{\mathcal{D}} = d^{\mathcal{D}}$*,* $\bot_d^{\mathcal{D}} = \emptyset$*,* $(\neg f)^{\mathcal{D}} = d^{\mathcal{D}} \setminus f^{\mathcal{D}}$*,* $(f_1 \wedge f_2)^{\mathcal{D}} = f_1^{\mathcal{D}} \cap f_2^{\mathcal{D}}$*, and* $(f_1 \vee f_2)^{\mathcal{D}} = f_1^{\mathcal{D}} \cup f_2^{\mathcal{D}}$*.*

In the rest of this paper we additionally assume that the datatypes in N_D are pairwise disjoint—that is, that $d_1, d_2 \in N_D$ and $d_1 \neq d_2$ imply $d_1^{\mathcal{D}} \cap d_2^{\mathcal{D}} = \emptyset$. As we show in Sections 3 and 5, this leads to no loss of generality, simplifies our reasoning algorithm, and allows for a modular treatment of different datatypes. Our datatypes are, in this respect, comparable to datatype groups [11].

For example, $\mathcal{D}$ might be a datatype map with $N_D = \{str, real\}$, where $str^{\mathcal{D}}$ and $real^{\mathcal{D}}$ are the set of all strings and real numbers, respectively. The sets $N_C(str)$ and $N_C(real)$ would then contain all string constants and all decimal representations of real numbers. Finally, the set $N_F(real)$ might contain the facet int, interpreted as the set of all integers, and facets of the form $<_w$, $>_w$, $\leq_w$, and $\geq_w$ for each decimal number w. Thus, the facet expression $int \wedge >_{12} \wedge <_{15}$ would represent the integers 13 and 14.

We next show how to extend a description logic $\mathcal{DL}$ with a datatype map. We omit the definition of the concepts and axioms of $\mathcal{DL}$, and present only the concepts and axioms of the combined language that involve datatypes. We first introduce *data ranges*—expressions over the predicates in $\mathcal{D}$—and then show how to integrate data ranges into $\mathcal{DL}$ concepts and axioms.

Definition 2. *Let* $\mathcal{D} = (N_D, N_C, N_F, \cdot^{\mathcal{D}})$ *be a datatype map. The set of* data ranges *for* $\mathcal{D}$ *is the smallest set that contains* $\top_{\mathcal{D}}$*,* d*,* $d[\varphi]$*,* $\{v_1, \ldots v_n\}$*,* $\overline{dr}$*, for* $d \in N_D$*,* φ *a facet expression for* d*,* $v_i \in N_C$*, and* dr *a data range.*

Table 1. Model-Theoretic Semantics of $\mathcal{DL}$+$\mathcal{D}$

Semantics of Data Ranges	
$(\top_{\mathcal{D}})^{\mathcal{D}} = \triangle^{\mathcal{D}}$	$(d[\varphi])^{\mathcal{D}} = \varphi^{\mathcal{D}}$
$(\{v_1, \ldots, v_n\})^{\mathcal{D}} = \{v_1^{\mathcal{D}}, \ldots, v_n^{\mathcal{D}}\}$	$\overline{dr}^{\,\mathcal{D}} = \triangle^{\mathcal{D}} \setminus dr^{\mathcal{D}}$

Semantics of Concepts	Semantics of Axioms
$(\forall U.dr)^{\mathcal{I}} = \{x \mid \forall y : \langle x, y\rangle \in U^{\mathcal{I}} \rightarrow y \in dr^{\mathcal{D}}\}$	$\mathsf{Dis}(U_1, U_2) \Rightarrow U_1^{\mathcal{I}} \cap U_2^{\mathcal{I}} = \emptyset$
$(\exists U.dr)^{\mathcal{I}} = \{x \mid \exists y : \langle x, y\rangle \in U^{\mathcal{I}} \wedge y \in dr^{\mathcal{D}}\}$	$U_1 \sqsubseteq U_2 \Rightarrow U_1^{\mathcal{I}} \subseteq U_2^{\mathcal{I}}$
$(\leq n\, U.dr)^{\mathcal{I}} = \{x \mid \sharp\{y \mid \langle x, y\rangle \in U^{\mathcal{I}} \wedge y \in dr^{\mathcal{D}}\} \leq n\}$	$U(a, v) \Rightarrow \langle a^I, v^{\mathcal{D}}\rangle \in U^{\mathcal{I}}$
$(\geq n\, U.dr)^{\mathcal{I}} = \{x \mid \sharp\{y \mid \langle x, y\rangle \in U^{\mathcal{I}} \wedge y \in dr^{\mathcal{D}}\} \geq n\}$	

Note: $\sharp N$ is the number of elements in a set N.

Let $\mathcal{DL}$ be a description logic, defined over a set of individuals N_I, and let N_{DP} be a set of data properties *disjoint from each of the sets of symbols used in $\mathcal{DL}$. The logic $\mathcal{DL}$+$\mathcal{D}$, obtained by extending $\mathcal{DL}$ with $\mathcal{D}$, is defined as follows. The set of concepts of $\mathcal{DL}$+$\mathcal{D}$ extends the set of concepts of $\mathcal{DL}$ with* datatype concepts *of the form $\exists U.dr$, $\forall U.dr$, $\geq n\, U.dr$, and $\leq n\, U.dr$, for $U \in N_{DP}$, n a nonnegative integer, and dr a data range for $\mathcal{D}$. The set of axioms of $\mathcal{DL}$+$\mathcal{D}$ extends the set of axioms of $\mathcal{DL}$ with* data property disjointness axioms $\mathsf{Dis}(U_1, U_2)$, data property inclusion axioms $U_1 \sqsubseteq U_2$, *and* data property assertions $U(a, v)$, *for $U_{(i)} \in N_{DP}$, $a \in N_i$, and $v \in N_C$.*

An interpretation *for $\mathcal{DL}$+$\mathcal{D}$ is a triple $\mathcal{I} = (\triangle^{\mathcal{I}}, \triangle^{\mathcal{D}}, \cdot^{\mathcal{I}})$, where $\triangle^{\mathcal{I}}$ and $\triangle^{\mathcal{D}}$ are nonempty disjoint sets such that $d^{\mathcal{D}} \subseteq \triangle^{\mathcal{D}}$ for each $d \in N_D$, and $\cdot^{\mathcal{I}}$ is a function assigning to each concept C, property R, and individual a of $\mathcal{DL}$, and to each data property $U \in N_{DP}$, interpretations $C^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}}$, $R^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$, $a^{\mathcal{I}} \in \triangle^{\mathcal{I}}$, and $U^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{D}}$ respectively. The functions $\cdot^{\mathcal{D}}$ and $\cdot^{\mathcal{I}}$ are extended to data ranges and datatype concepts as shown in Table 1. For $\mathcal{DL}$+$\mathcal{D}$ knowledge bases $\mathcal{K}$ and $\mathcal{K}'$, an interpretation $\mathcal{I}$ is a* $\mathcal{D}$-model *of $\mathcal{K}$, written $I \models_{\mathcal{D}} \mathcal{K}$, if all axioms of $\mathcal{DL}$ are satisfied in $\mathcal{I}$ as specified by $\mathcal{DL}$, and the additional axioms are satisfied in $\mathcal{I}$ and $\mathcal{D}$ as specified in Table 1; furthermore, $\mathcal{K}$* $\mathcal{D}$-entails *$\mathcal{K}'$, written $\mathcal{K} \models_{\mathcal{D}} \mathcal{K}'$, if $I \models_{\mathcal{D}} \mathcal{K}'$ whenever $I \models_D \mathcal{K}$.*

3 The Architecture of the Datatype System of OWL 2

We now explain the rationale behind and some nontrivial consequences of the definitions presented in Section 2.

3.1 Openness of the Domain in a Datatype Map

A number of approaches for adding datatypes to DLs are based on the framework of concrete domains [2], in which the set $\triangle^{\mathcal{D}}$ is fixed in advance; for example, $\triangle^{\mathcal{D}}$ can be fixed to the set of all integers. Datatype groups [11] are similar to datatype maps, but they also fix the set $\triangle^{\mathcal{D}}$ as the union of the interpretations of all datatypes in the group. The semantics of OWL DL [13] is somewhat

ambiguous regarding this point: it says that $\triangle^{\mathcal{D}}$ *contains* the interpretation of all datatypes, but it is not clear whether $\triangle^{\mathcal{D}}$ must be *exactly* that set (the intention was, however, that $\triangle^{\mathcal{D}}$ should *not* be fixed[4]).

The set of datatypes in OWL 2 should be extensible: future versions of OWL might want to add support datatypes that will not be normative in OWL 2, and implementations might also want to support custom datatypes. Extensibility, however, is impossible if the set $\triangle^{\mathcal{D}}$ is fixed in advance. For example, the axiom $\alpha = \top \sqsubseteq \forall U. <_5 \sqcup \exists U.real$ is a tautology w.r.t. a datatype map $\mathcal{D}$ where $\triangle^{\mathcal{D}}$ is fixed to the set of all real numbers: if some individual a is connected by U to a data value, this value must be a real number, so $\exists U.real(a)$ is satisfied; if a has no U-connections, then $\forall U. <_5 (a)$ is satisfied. The axiom α, however, is not a tautology w.r.t. a datatype map $\mathcal{D}'$ in which $\triangle_{\mathcal{D}'}$ is fixed to contain all real numbers and all strings: an individual a might be connected only to a string, which makes neither disjunct in α satisfied. This example shows that, in general, the consequences of a knowledge base $\mathcal{K}$ can depend on datatypes that are not mentioned in $\mathcal{K}$ at all—clearly an undesirable situation.

Therefore, in $\mathcal{DL}$+$\mathcal{D}$ (and in OWL 2) the set $\triangle^{\mathcal{D}}$ can be *any* set that at least contains the interpretations of all datatypes in $\mathcal{D}$. The following proposition shows that $\mathcal{K}$ can in fact be interpreted by considering only those datatypes explicitly mentioned in $\mathcal{K}$. Therefore, in the rest of this paper we simply talk of *models* and *entailment* instead of $\mathcal{D}$-models and $\mathcal{D}$-entailment.

Proposition 1. *Let $\mathcal{D}_1 = (N_{D_1}, N_{V_1}, N_{F_1}, \cdot^{\mathcal{D}_1})$ be a datatype map and $\mathcal{K}$ and $\mathcal{K}'$ $\mathcal{DL}$+$\mathcal{D}_1$ knowledge bases. For each datatype map $\mathcal{D}_2 = (N_{D_2}, N_{V_2}, N_{F_2}, \cdot^{\mathcal{D}_2})$ such that $N_{D_1} \subseteq N_{D_2}$, $N_{V_1}(d) \subseteq N_{V_2}(d)$ and $N_{F_1}(d) \subseteq N_{F_2}(d)$ for each $d \in N_{D_1}$, and $\cdot^{\mathcal{D}_2}$ coincides with $\cdot^{\mathcal{D}_1}$ on the elements from $\mathcal{D}_1$, we have $\mathcal{K} \models_{\mathcal{D}_1} \mathcal{K}'$ iff $\mathcal{K} \models_{\mathcal{D}_2} \mathcal{K}'$.*

Proof. We show the contrapositive: $\mathcal{K} \not\models_{\mathcal{D}_2} \mathcal{K}'$ iff $\mathcal{K} \not\models_{\mathcal{D}_1} \mathcal{K}'$. The ($\Rightarrow$) direction is trivial. For the ($\Leftarrow$) direction, let $\mathcal{I} = (\triangle^{\mathcal{I}}, \triangle^{\mathcal{D}}, \cdot^{\mathcal{I}})$ be an interpretation such that $\mathcal{I} \models_{\mathcal{D}_1} \mathcal{K}$ and $\mathcal{I} \not\models_{\mathcal{D}_1} \mathcal{K}'$. We construct $\mathcal{I}' = (\triangle^{\mathcal{I}'}, \triangle^{\mathcal{D}'}, \cdot^{\mathcal{I}'})$ such that $\triangle^{\mathcal{I}'} := \triangle^{\mathcal{I}}$ and $\cdot^{\mathcal{I}'} := \cdot^{\mathcal{I}}$, and $\triangle^{\mathcal{D}'}$ is obtained from $\triangle^{\mathcal{D}}$ by adding the interpretations of all datatypes from $N_{\mathcal{D}_2} \setminus N_{\mathcal{D}_1}$. Extending $\triangle^{\mathcal{D}}$ to $\triangle^{\mathcal{D}'}$ can change only the interpretation of complemented data ranges; hence, for each data range dr over $\mathcal{D}_1$, we have $dr^{\mathcal{D}} \subseteq dr^{\mathcal{D}'}$. Since $U^{\mathcal{D}'} = U^{\mathcal{D}}$ for each $U \in N_{DP}$, for each concept C of $\mathcal{DL}$+$\mathcal{D}_1$, we have $C^{\mathcal{I}'} = C^{\mathcal{I}}$; but then, $\mathcal{I}' \models_{\mathcal{D}_2} \mathcal{K}$ and $\mathcal{I}' \not\models_{\mathcal{D}_2} \mathcal{K}'$. □

3.2 Giving Names to Data Ranges

The OWL 2 Working Group is currently considering whether to extend the language with the ability to explicitly name commonly used data ranges. For example, the axiom $Teens \equiv real[int \wedge >_{12} \wedge <_{20}]$ would give a name *Teens* to the set of integers between 12 and 20, which could then be used in concept definitions such as $Teenager \equiv \exists hasAge.Teens$. The syntax and the semantics of such axioms can be formalized as shown in the following definition.

[4] Personal communication with Peter F. Patel-Schneider.

Definition 3. *Let* $\mathcal{D} = (N_D, N_C, N_F, \cdot^{\mathcal{D}})$ *be a datatype map, and let* N_N *be a set of* datatype names*, disjoint with* N_D*,* N_C*, and* N_F*. The set of data ranges of* $\mathcal{D}$ *is extended such that each* $dn \in N_N$ *is a data range. A* datatype naming axiom *then has the form* $dn \equiv dr$*, where* $dn \in N_N$ *and* dr *is a data range. An interpretation* $\mathcal{I} = (\triangle^{\mathcal{I}}, \triangle^{\mathcal{D}}, \cdot^{\mathcal{I}})$ *interprets each* $dn \in N_N$ *as* $dn^{\mathcal{D}} \subseteq \triangle^{\mathcal{D}}$*. Furthermore,* $\mathcal{I}$ *satisfies a datatype naming axiom* $dn \equiv dr$ *iff* $dn^{\mathcal{D}} = dr^{\mathcal{D}}$*.*

Although seemingly innocuous, datatype naming axioms can easily invalidate Proposition 1. For example, let $\mathcal{K} = \{A \equiv \mathit{real},\ \ A \equiv \top_{\mathcal{D}}\}$. The datatype naming axioms in $\mathcal{K}$ behave similarly to general concept inclusions, allowing us to fix the set $\triangle^{\mathcal{D}}$ to the set of real numbers. Thus, $\mathcal{K}$ is satisfiable w.r.t. a datatype map that contains only real numbers, but it becomes unsatisfiable as soon as we extend the datatype map with a datatype disjoint with *real*.

Datatype naming axioms thus seem to be too expressive in general: datatypes are fully described by the datatype map, so allowing for additional axioms about the datatypes is likely to be undesirable. This problem can be solved by requiring datatype naming axioms $dn \equiv dr$ to be acyclic [1]: each name dn may occur in at most one such axiom, and it may neither directly nor indirectly be used in dr. With such restrictions, each datatype name dn can be unfolded—that is, it can be (recursively) replaced with its definition. Thus, datatype names can always be eliminated from a knowledge base, so Proposition 1 still holds.

3.3 Disjointness of Datatypes in a Map

In Definition 1 and in OWL 2, the datatypes in a datatype map need not be disjoint. Without losing generality, however, we can assume the contrary: two nondisjoint datatypes d_1 and d_2 can be replaced with a datatype d_{1+2} that is interpreted as a union of d_1 and d_2 and that provides d_1 and d_2 as facets. For example, a datatype system with strings, real numbers, and integers can be formalized as a datatype map in which strings and real numbers are datatypes, and integers are modeled as a facet for the real numbers.

Assuming that datatypes in a map are disjoint allows us to obtain a modular algorithm for datatype reasoning. In particular, only four basic operations (see Definition 5) are needed to support a datatype d in our datatype reasoning algorithm; if d is disjoint from any other datatype in the map, these operations need not consider any of the other supported datatypes. For example, in Sections 5.2 and 5.3 we present datatype handlers for numbers and strings, respectively. Although the handler for numbers needs to know about all kinds of numbers, it does not need to know about strings and vice versa. Therefore, the notion of a datatype provides us with a natural modularization boundary for reasoning.

3.4 The Semantics of Complemented Data Ranges

Complemented data ranges have been added to $\mathcal{DL}{+}\mathcal{D}$ mainly to support the representation of axioms in negation-normal form. In (1), for example, the concept $\exists \mathit{hasAge}.\mathit{real}[<_{18}]$ is implicity negated, which becomes visible if the axiom

is brought into negation-normal form, cf. (2).

$$\exists \mathit{hasAge}.\mathit{real}[<_{18}] \sqsubseteq \mathit{YoungPerson} \tag{1}$$

$$\top \sqsubseteq \forall \mathit{hasAge}.\overline{\mathit{real}[<_{18}]} \sqcup \mathit{YoungPerson} \tag{2}$$

Note that the semantics of complemented data ranges is defined w.r.t. the entire set $\triangle^{\mathcal{D}}$; hence, the interpretation of $\overline{\mathit{real}[<_{18}]}$ includes all real numbers greater than or equal to 18, as well all data values that are not numbers—that is, $\overline{\mathit{real}[<_{18}]}^{\mathcal{D}} = \overline{\mathit{real}}^{\mathcal{D}} \cup \mathit{real}[\geq_{18}]^{\mathcal{D}}$. This may seem counterintuitive, but it is necessary if $\mathcal{DL}$+$\mathcal{D}$ is to have a standard first-order semantics. For example, if $\overline{\mathit{real}[<_{18}]}$ were interpreted as "the set of real numbers greater than or equal to 18," then $\forall \mathit{hasAge}.\overline{\mathit{real}[<_{18}]}$ would not be the complement of $\exists \mathit{hasAge}.\mathit{real}[<_{18}]$, which would invalidate the basic assumptions of first-order logic.

3.5 Reasoning with a Datatype Map

Several tableau algorithms for DLs extended with datatypes have been proposed [2, 10, 5, 7, 11]; we illustrate them using the ABox $\mathcal{A}$, shown in (3). First, standard tableau expansion rules are used to expand $\mathcal{A}$ to $\mathcal{A}'$, shown in (4). To obtain a model from $\mathcal{A}'$, one must check whether individuals t_1 and t_2 can be assigned values from $\triangle^{\mathcal{D}}$ in a consistent way. For this purpose, algorithms such as [7] invoke an external *datatype checker*—an oracle that decides satisfiability of conjunctions of the form $d_1(x_1) \wedge \ldots \wedge d_n(x_n)$, where d_i are datatypes. In our example, $\varphi = \{5\}(x_1) \wedge \mathit{int}[>_4 \wedge <_6](x_2)$ is satisfied by an assignment $x_1 = x_2 = 5$, so we can conclude that $\mathcal{A}'$ and $\mathcal{A}$ are satisfiable.

$$\mathcal{A} = \{ \quad \exists U_1.\{5\}(a), \;\; \exists U_2.\mathit{int}[>_4 \wedge <_6](a) \quad \} \tag{3}$$

$$\mathcal{A}' = \mathcal{A} \cup \{ \quad U_1(a, t_1), \;\; \{5\}(t_1), \;\; U_2(a, t_2), \;\; \mathit{int}[>_4 \wedge <_6](t_2) \quad \} \tag{4}$$

Although the set $\triangle^{\mathcal{D}}$ in a datatype map is not fixed (see Section 3.1), a tableau algorithm for $\mathcal{DL}$ can be combined with a datatype checker in much the same way as in [2, 10, 5, 7, 11]. A minor problem in $\mathcal{DL}$+$\mathcal{D}$ arises due to disjointness of data properties. Assume that the knowledge base shown in (3) also contains the axiom $\mathsf{Dis}(U_1, U_2)$. The assignment $x_1 = x_2 = 5$ is the only one satisfying φ; however, setting t_1 and t_2 equal to 5 clearly invalidates $\mathsf{Dis}(U_1, U_2)$. A similar problem has been observed and solved in a slightly different context [10]. Roughly speaking, the solution is to derive the inequality $t_1 \not\approx t_2$ whenever $U_1(s, t_1)$ and $U_2(s, t_2)$ are derived and U_1 and U_2 are disjoint. In our example, deriving $t_1 \not\approx t_2$ gives rise to a conjunction $\varphi' = \varphi \wedge x_1 \not\approx x_2$, which is clearly unsatisfiable. The model construction therefore fails, and we can correctly conclude that $\mathcal{A} \cup \{\mathsf{Dis}(U_1, U_2)\}$ is unsatisfiable.

The following definition formalizes the datatype checking problem, as applicable in our setting, and introduces some useful notation. For convenience, we treat conjunctions of datatype assertions as sets.

Definition 4. *Let $\mathcal{D} = (N_D, N_C, N_F, \cdot^{\mathcal{D}})$ be a datatype map and N_V a set of* variables *disjoint with N_C. A* $\mathcal{D}$-conjunction *is a finite set of assertions of the form $dr(t)$ and $t_1 \not\approx t_2$, for dr a data range over $\mathcal{D}$ and $t_{(i)} \in N_V \cup N_C$. A $\mathcal{D}$-conjunction Γ is* $\mathcal{D}$-satisfiable *if a set $\triangle^{\mathcal{D}}$ and a mapping $\delta : N_V \cup N_C \to \triangle^{\mathcal{D}}$ exist such that (*i*) $d^{\mathcal{D}} \subseteq \triangle^{\mathcal{D}}$ for each $d \in N_D$, (*ii*) $\delta(c) = c^{\mathcal{D}}$ for each $c \in N_C$, (*iii*) $dr(t) \in \Gamma$ implies $\delta(t) \in (dr)^{\mathcal{D}}$, and (*iv*) $t_1 \not\approx t_2 \in \Gamma$ implies $\delta(t_1) \neq \delta(t_2)$.*

Let Γ be a $\mathcal{D}$-conjunction. Each assertion $t_1 \not\approx t_2$ in Γ should also be read as $t_2 \not\approx t_1$—that is, the $\not\approx$ predicate has built-in symmetry. For x a variable, $\Gamma/_{-x}$ is the result of deleting all assertions in Γ that contain x; for t a variable or a constant, $\Gamma/_{x \mapsto t}$ is the result of replacing x with t in all assertions in Γ; finally, let $\mathsf{cv}(\Gamma, x) = \{x' \mid x' \not\approx x \in \Gamma\}$ and $\mathsf{cc}(\Gamma, x) = \{c \mid c \not\approx x \in \Gamma\}$.

Boolean combinations of facets are thus dealt with in the datatype checker rather than the tableau algorithm, since knowledge about datatypes and facets can be used to optimize the handling of common cases (see Sections 5.2 and 5.3).

3.6 Complexity of Datatype Checking

We now turn our attention to the complexity of datatype reasoning, and show that datatype checking is intractable in general (Proposition 2), but that an important case exists in which the problem becomes trivial (Proposition 3).

Proposition 2. *Checking $\mathcal{D}$-satisfiability of a $\mathcal{D}$-conjunction is* NP*-hard.*

Proof. The proof is by reduction from the NP-hard Graph 3-Colorability problem: for a finite undirected graph $G = (V, E)$, decide whether it is possible to label each vertex in V with a number from the set $\{1, 2, 3\}$ such that adjacent vertices are not labeled with the same number.

For $G = (V, E)$ a finite graph, let x_i be a variable uniquely assigned to each vertex $i \in V$, and let Γ_G be the following $\mathcal{D}$-conjunction, where $dr = \{1, 2, 3\}$:

$$\Gamma_G = \bigcup_{i \in V} \{\ dr(x_i)\ \} \cup \bigcup_{\langle i,j \rangle \in E} \{\ x_i \not\approx x_j\ \}$$

It is easy to see that G is 3-colorable if and only if Γ_G is $\mathcal{D}$-satisfiable. □

The proof of Proposition 2 requires inequality predicates, which are already necessary for the proper handling of number restriction datatype concepts. Such concepts, however, generate sets of pairwise unequal variables, which may be easier to handle; for example, it is not trivial to see if they can encode Graph 3-Colorability. In contrast, $\mathcal{D}$-conjunctions of the form used in the proof of Proposition 2 can be obtained using only axioms of the form $\exists U.dr(a)$ and $\mathsf{Dis}(U_1, U_2)$. Proposition 2 thus suggests that data property disjointness axioms might make datatype checking harder in practice.

Proposition 3. *Let Γ be a $\mathcal{D}$-conjunction and x a variable such that (*i*) x occurs in Γ in exactly one assertion of the form $dr(x)$,[5] (*ii*) $x \not\approx x \notin \Gamma$, and*

[5] If Γ does not contain such assertion, we can always take $dr = \top_{\mathcal{D}}$.

(iii) $\sharp dr^{\mathcal{D}} \geq \sharp\mathsf{cv}(\Gamma, x) + \sharp\mathsf{cc}(\Gamma, x) + 1$.[6] *Then, Γ is $\mathcal{D}$-satisfiable if and only if $\Gamma/_{-x}$ is $\mathcal{D}$-satisfiable.*

Proof. The ($\Rightarrow$) direction is obvious. For the ($\Leftarrow$) direction, assume that a set $\triangle^{\mathcal{D}}$ and a mapping δ of variables in $\Gamma/_{-x}$ to the elements of $\triangle^{\mathcal{D}}$ exist such that $\Gamma/_{-x}$ is satisfied. By *(iii)*, the set $dr^{\mathcal{D}} \setminus \{c^{\mathcal{D}} \mid c \not\approx x \in \Gamma\} \setminus \{\delta(x') \mid x' \in \mathsf{cv}(\Gamma, x)\}$ contains at least one element that is not mapped to a variable in $\mathsf{cv}(\Gamma, x)$ or a constant in $\mathsf{cc}(\Gamma, x)$. Let $\delta(x)$ be an arbitrarily selected element from this set. By *(i)* and *(ii)*, all assertions in $\Gamma \setminus \Gamma/_{-x}$ are of the form $dr(x)$, $x \not\approx x'$ with $x' \in \mathsf{cv}(\Gamma, x)$, or $x \not\approx c$ with $c \in \mathsf{cc}(\Gamma, x)$. Clearly, δ satisfies Γ. □

4 Selecting the Set of Datatypes for OWL 2

The current OWL 2 Working Draft[7] contains a normative list of datatypes and facets, most of which are taken from XML Schema [3]. Although these datatypes may be quite useful in XML applications, some of them do not seem appropriate for a logic-based language such as OWL 2.

4.1 String-Based Datatypes

The base string datatype in OWL 2 is *xsd:string*, and it is equipped with facets $\mathsf{length}\ n$, $\mathsf{minLength}\ n$, $\mathsf{maxLength}\ n$, which restrict the length of a string, and $\mathsf{pattern}\ re$, which restricts the form of a string to the regular expression re. We present algorithms for handling strings and all these facets in Section 5.3. XML Schema includes a number of string-derived datatypes [3], which can be seen as shortcuts for the pattern facet with particular values. The *xsd:anyURI* datatype represents Uniform Resource Locators (URIs). The OWL 2 specification needs to clarify whether this datatype is a subset of *xsd:string*.

4.2 Numbers

XML Schema provides a multitude of datatypes for numbers: *xsd:decimal* represents arbitrarily long numbers in decimal notation, *xsd:integer* represents unbounded integers, and *xsd:double* and *xsd:float* represent floating-point numbers in double and single precision, respectively. Other numeric datatypes are derived from these base ones by imposing various restrictions; for example, *xsd:nonNegativeInteger* represents all nonnegative integers. The supported facets are $\mathsf{minInclusive}\ x$, $\mathsf{minExclusive}\ x$, $\mathsf{maxInclusive}\ x$, and $\mathsf{maxExclusive}\ x$, which restrict the range of numbers, and $\mathsf{pattern}\ re$, which restricts numbers to those whose string representation matches the regular expression re.

These datatypes exhibit a number of different problems. First, *xsd:decimal* is not closed under division, so it does not provide a suitable basis for possible extensions of OWL 2 with arithmetic. Second, the floating-point datatypes have

[6] $\sharp S$ denotes the cardinality of the set S.

[7] `http://www.w3.org/TR/2008/WD-owl2-syntax-20080411/`

a very large but finite number of values, and can also exhibit complex behavior due to rounding of values that cannot be exactly represented; these features could lead to unexpected inferences, and they might be a source of inefficiency in implementations. Third, the pattern facet seems to be of limited utility for number types, and it might place an unreasonable burden on implementers as it allows for data ranges such as "all decimal numbers greater than 5 that match a particular regular expression."

In view of these problems, we propose that the OWL 2 datatypes *xsd:decimal*, *xsd:double*, and *xsd:float* be replaced with a new datatype *owl:real*, interpreted as the set of all real numbers. Clearly, not all data values in the interpretation of *owl:real* can be represented using a constant (i.e., a finite string over a finite alphabet). This should, however, not pose a problem in practice: one could define constants for all rational numbers, and possibly also for "important" irrational numbers such as π or e. The *xsd:integer* datatype can be supported as a facet of *owl:real*, as can the other facets apart from pattern. In Section 5.2 we present a reasoning algorithm for this datatype.

4.3 Date and Time

XML Schema provides the *xsd:dateTime* datatype, interpreted as a set of time points in the Gregorian calendar. A number of other datatypes represent possibly recurring intervals and time points. For example, *xsd:date* represents intervals of length one day; *xsd:time* represents an instance in time that recurs each day; and *xsd:gMonthDay* represents a Gregorian date that recurs every year.

Reasoning about recurring time points and intervals is difficult due to their complex and ill defined semantics: the recurrences are irregular due to exceptions such as leap years; furthermore, the occurrence of future time points cannot be determined in advance due to leap seconds, which are introduced into the calendar by the International Earth Rotation and Reference Systems Service as necessary. Therefore, only the *xsd:dateTime* datatype seems amenable to implementation, and it can be handled by techniques similar to the ones for numbers.

5 Reasoning with Datatypes in OWL 2

While the principles of integrating a datatype checker with a tableau algorithm are well understood, little attention has been paid in the literature to the details of actual datatype checking algorithms. We next present such an algorithm that is extensible w.r.t. the set of supported datatypes and show its correctness.

5.1 An Extensible Datatype Checking Algorithm

We first identify the basic operations that are needed to support a particular datatype in a datatype map. In Sections 5.2 and 5.3, we discuss how to implement these operations for the real and string datatypes, respectively.

Definition 5. *Let $\mathcal{D} = (N_D, N_C, N_F, \cdot^{\mathcal{D}})$ be a datatype map where the interpretations of different datatypes are pairwise disjoint. A* datatype handler *for a datatype $d \in N_D$ is a 4-tuple $(\mathsf{minc}_d, \mathsf{enu}_d, \mathsf{in}_d, \mathsf{eq}_d)$ of functions where, for each data range dr of the form $d[\varphi]$,*

- $\mathsf{minc}_d(dr, n) = \mathsf{true}$ *for an integer n iff $dr^{\mathcal{D}}$ contains at least n elements,*
- $\mathsf{enu}_d(dr)$ *is defined only if $dr^{\mathcal{D}}$ is finite, and it is a set $\{c_1, \ldots, c_n\}$ such that $dr^{\mathcal{D}} = \{c_1^{\mathcal{D}}, \ldots, c_n^{\mathcal{D}}\}$—that is, $\mathsf{enu}_d(dr)$ returns a finite set of constants that enumerate the interpretation of dr,*
- $\mathsf{in}_d(c, dr) = \mathsf{true}$ *for $c \in N_C(d)$ iff $c^{\mathcal{D}} \in dr^{\mathcal{D}}$, and*
- $\mathsf{eq}_d(c_1, c_2) = \mathsf{true}$ *for $c_1, c_2 \in N_C(d)$ iff $c_1^{\mathcal{D}} = c_2^{\mathcal{D}}$.*

Our algorithm for checking $\mathcal{D}$-satisfiability of a $\mathcal{D}$-conjunction Γ consists of Procedures 1 and 2. For convenience, we assume that all data ranges in Γ are of the form $d[\varphi]$ (i.e., data ranges without facet expressions are represented as $d[\top_d]$). The letter c (possibly with subscripts or superscripts) denotes a constant. We additionally use the following two auxiliary functions: for c_1 and c_2 constants, $\mathsf{eq}(c_1, c_2) = \mathsf{true}$ iff c_1 and c_2 are constants of the same datatype d and $\mathsf{eq}_d(c_1, c_2) = \mathsf{true}$; furthermore, for c a constant and $d[\varphi]$ a data range, $\mathsf{in}(c, d[\varphi]) = \mathsf{true}$ iff c is a constant of the datatype d and $\mathsf{in}_d(c, d[\varphi]) = \mathsf{true}$.

We next explain the intuition behind Procedure 1. First, Γ is checked for trivial unsatisfiability (lines 1–3), after which all complemented enumerations are rewritten using inequalities (lines 4–6) in order to simplify the rest of the algorithm. Lines 7–22 form the core part of the algorithm. For each variable x in Γ, the set of data ranges in which x occurs is normalized (line 8)—that is, it is reduced to $*$ (meaning that x is trivially satisfiable), a data range of the form $d[\varphi]$, or a finite enumeration $\{c_1, \ldots, c_n\}$. In the second case, a call is made to the datatype handler to see whether Proposition 3 is applicable (line 9); if so, the variable x is removed from Γ (line 11). If $D = d[\varphi]$ and the test in line 9 fails, then the interpretation of D is finite, so it is enumerated (line 13). Thus, by line 15, D is either $*$ or a finite enumeration. If D is empty, then Γ unsatisfiable (lines 15–16). If D is not empty, the normalized data range D is reintroduced into Γ (lines 17–21): if D is a singleton, then x must be assigned the only value in D (line 18); otherwise, all original data range assertions involving x are replaced with $D(x)$ (line 20). By line 23, therefore, in all assertions of the form $dr(x) \in \Gamma$, for dr a nonempty enumerated data range.

Lines 23–33 try to further simplify Γ, first by considering all assertions not containing variables (lines 23–30), and then by applying Proposition 3 to the remaining assertions (lines 31–33). All that now remains is to check if Γ is satisfied for at least one assignment of values to variables and constants. This part of the algorithm is nondeterministic, and can be implemented using search. To reduce the search space, Γ is first decomposed into variable-disjoint subsets (line 34), each of which is tested for $\mathcal{D}$-satisfiability independently (lines 35–43).

Procedure 2 normalizes a set of data ranges S to a finite enumeration of constants of the form $\{c_1, \ldots, c_n\}$, a single data range of the form $d[\varphi]$, or $*$ (meaning that the corresponding variable is trivial to satisfy). Lines 1–13 handle

Procedure 1 $\mathcal{D}$-SATISFIABLE(Γ)

Require: a $\mathcal{D}$-conjunction Γ containing assertions of the form $dr(t)$ and $t_1 \not\approx t_2$

if $\overline{\top_{\mathcal{D}}}(t) \in \Gamma$ for t a variable or a constant, or $x \not\approx x \in \Gamma$ **then**
  return false
end if
for each $\overline{\{c_1, \ldots, c_n\}}(t) \in \Gamma$ for t a variable or a constant **do**
  $\Gamma := (\Gamma \setminus \{\ \overline{\{c_1, \ldots, c_n\}}(t)\ \}) \cup \{t \not\approx c_1, \ldots, t \not\approx c_n\}$
end for
for each variable x occurring in Γ **do**
  $D := \text{NORMALIZE}(\{dr \mid dr(x) \in \Gamma\})$
  if $D = *$, or $D = d[\varphi]$ and $\mathsf{minc}_d(D, \sharp\mathsf{cv}(\Gamma, x) + \sharp\mathsf{cc}(\Gamma, x) + 1) = \mathsf{true}$ **then**
    $\Gamma := \Gamma/_{-x}$ ▷ apply Proposition 3 to x
    $D := *$
  else if $D = d[\varphi]$ **then**
    $D := \mathsf{enu}_d(D)$
  end if
  if $D = \emptyset$ **then**
    return false
  else if $D = \{c\}$ **then**
    $\Gamma := (\Gamma \setminus \{dr(x) \mid dr(x) \in \Gamma\})/_{x \mapsto c}$
  else if $D \neq *$ **then**
    $\Gamma := (\Gamma \setminus \{dr(x) \mid dr(x) \in \Gamma\}) \cup \{D(x)\}$
  end if
end for
for each $\alpha \in \Gamma$ that does not contain a variable **do**
  if $\alpha = c_1 \not\approx c_2$ and $\mathsf{eq}(c_1, c_2) = \mathsf{true}$, or
    $\alpha = d[\varphi](c)$ and $\mathsf{in}(c, d[\varphi]) = \mathsf{false}$, or $\alpha = \overline{d[\varphi]}(c)$ and $\mathsf{in}(c, d[\varphi]) = \mathsf{true}$, or
    $\alpha = \{c_1, \ldots, c_n\}(c)$ and $\mathsf{eq}(c, c_i) = \mathsf{false}$ for each $1 \leq i \leq n$ **then**
    return false
  end if
  $\Gamma := \Gamma \setminus \{\ \alpha\ \}$
end for
while Γ contains some $\{c_1, \ldots, c_n\}(x)$ such that $n \geq \sharp\mathsf{cv}(\Gamma, x) + \sharp\mathsf{cc}(\Gamma, x) + 1$ **do**
  $\Gamma := \Gamma/_{-x}$ ▷ apply Proposition 3 to x
end while
decompose Γ into nonempty mutually disjoint subsets $\Gamma_1, \ldots, \Gamma_n$ such that no Γ_i and Γ_j, $i \neq j$, have variables in common
for each $1 \leq i \leq n$ **do**
  if an assignment δ to variables and constants in Γ_i such that
    $\delta(c) = c$ for each constant c, and
    $\{c_1, \ldots, c_m\}(x) \in \Gamma$ implies $\delta(x) = c_i$ for some $1 \leq i \leq m$, and
    $t_1 \not\approx t_2 \in \Gamma$ implies $\mathsf{eq}(\delta(t_1), \delta(t_2)) = \mathsf{false}$
   does not exist **then**
    return false
  end if
end for
return true

Procedure 2 NORMALIZE(S)

Require: a nonempty set of data ranges S of the form $d[\varphi]$, $\overline{d[\varphi]}$, or $\{c_1, \ldots, c_n\}$

1: **if** S contains a data range of the form $\{c_1, \ldots, c_n\}$ **then**
2: $R := \{c_1, \ldots, c_n\}$
3: **for each** $c \in R$ **do**
4: **for each** $dr \in S$ **do**
5: **if** $dr = \{c'_1, \ldots, c'_m\}$ and $\mathsf{eq}(c, c'_i) = \mathsf{false}$ for each $1 \leq i \leq m$, or
6: $dr = d[\varphi]$ and $\mathsf{in}(c, d[\varphi]) = \mathsf{false}$, or
7: $dr = \overline{d[\varphi]}$ and $\mathsf{in}(c, d[\varphi]) = \mathsf{true}$ **then**
8: $R := R \setminus \{c\}$
9: **end if**
10: **end for**
11: **end for**
12: **return** R
13: **end if**
14: **if** S contains no data range of the form $d[\varphi]$ **then**
15: **return** $*$
16: **else if** S contains data ranges $d_1[\varphi_1]$ and $d_2[\varphi_2]$ such that $d_1 \neq d_2$ **then**
17: **return** $\emptyset$
18: **end if**
19: **let** d be the datatype of all the data ranges in S of the form $d[\varphi]$
20: $\psi := \top_d$
21: **for each** $dr \in S$ **do**
22: **if** $dr = d[\varphi]$ **then**
23: $\psi := \psi \wedge \varphi$
24: **else if** $dr = \overline{d[\varphi]}$ **then**
25: $\psi := \psi \wedge \neg\varphi$
26: **end if**
27: **end for**
28: **return** $d[\psi]$

the case where S contains at least one enumerated data range: the result is then an enumeration containing only those constants c_i that are contained in all data ranges in S. If S does not contain at least one positive data range (line 14), then the corresponding variable can be assigned a fresh distinct data value not contained in the interpretation of any of the datatypes (note that $\triangle^{\mathcal{D}}$ can be any set that contains the interpretations of all the datatypes in $\mathcal{D}$). If S contains two positive data ranges with different datatypes (line 16), then S is clearly unsatisfiable. Lines 21–27 then combine the facet expressions in all the data ranges in S. Note that the simplification of complemented data ranges in line 25 is possible because $\overline{d[\varphi]}^{\mathcal{D}} = \overline{d}^{\mathcal{D}} \cup \overline{d[\neg\varphi]}^{\mathcal{D}}$ and, since S contains at least one data range of the form $d[\varphi]$, no data value can be in both $d[\varphi]^{\mathcal{D}}$ and $\overline{d}^{\mathcal{D}}$.

The correctness of our algorithm follows easily from Proposition 3. Therefore, we only sketch the proof of the following theorem.

Theorem 1. *$\mathcal{D}$-*SATISFIABLE(Γ) *returns* true *if and only if Γ is $\mathcal{D}$-satisfiable.*

Proof (Sketch). It is easy to see that, for each $D = \text{NORMALIZE}(S)$, the following holds (†): $D = *$ if and only of all data ranges in S are of the form $\overline{d[\varphi]}$; otherwise,

$$D^{\mathcal{D}} = \bigcap_{dr \in S_p} dr^{\mathcal{D}} \cap \bigcap_{\overline{d[\varphi]} \in S \setminus S_p} (d[\neg\varphi])^{\mathcal{D}}, \text{ where}$$
$$S_p = \{dr \in S \mid dr \text{ is not of the form } \overline{d[\varphi]}\}.$$

We now prove the claim of this theorem. If Γ is $\mathcal{D}$-satisfiable, then the condition in line 1 cannot be satisfied. Furthermore, the transformation in line 5 clearly preserves $\mathcal{D}$-satisfiability of Γ. Consider now the iteration over the variables occurring in Γ in lines 7–22. In lines 9–11, assertions containing x can be deleted from Γ without affecting its $\mathcal{D}$-satisfiability if either $D = *$, or if $D = d[\varphi]$ and Proposition 3 is applicable; in the former case, this is because (†) tells us that we can interpret x as an arbitrary unique element not contained in any datatype in $\mathcal{D}$. If $D = d[\varphi]$ and Proposition 3 is not applicable, then $d[\varphi]$ must be finite and equal to the enumeration obtained in line 13; hence, in lines 15–22 we can either return false (if the enumeration is empty), replace x with c in Γ (if $D(x)$ is a singleton enumeration $\{c\}$), or replace all assertions of the form $dr(x)$ in Γ with a single assertion $D(x)$, all of which preserve the $\mathcal{D}$-satisfiability of Γ. Lines 23–30 detect obvious inconsistencies involving assertions in Γ that do not contain variables; clearly, this does not affect the $\mathcal{D}$-satisfiability of Γ. Lines 31–33 apply Proposition 3 again, which by definition preserves the $\mathcal{D}$-satisfiability of Γ. Since all data ranges in Γ are now finite enumerations, only a finite number of assignments need to be considered, and lines 34–44 will detect if one of these satisfies Γ. □

In practice, the number of variables in a $\mathcal{D}$-conjunction Γ is likely to be of the same order of magnitude as the numbers occurring in number restrictions, which are usually quite small. Furthermore, data ranges are rarely constrained to small interpretations in practice, so test (9) is likely to succeed. The satisfiability of such a Γ can thus be decided without the need to enumerate data ranges and perform combinatorial reasoning. The performance of our algorithm in practice is thus mainly limited by the efficiency of minc_d, which, as we discuss next, can be efficiently implemented for numbers and strings.

5.2 A Datatype Handler for Numbers

To implement a datatype handler for the *owl:real* datatype of OWL 2, here abbreviated as *real*, we devise an efficient representation of the interpretation of a facet expression ψ. In particular, we represent fragments of this interpretation as *intervals* of the form $t[l, u]$, $t[l, u)$, $t(l, u]$, and $t(l, u)$, where $t \in \{real, int, \overline{int}\}$ determines the *type* of the interval, l is either $-\infty$ or a real number, and u is either $+\infty$ or a real number, such that $l \leq u$. Such an interval represents the set of all real numbers of type t between l and u; the round parenthesis means that the end-point is not included, and the square parenthesis means that the end-point is included in the set. By taking into account that $int \cap \overline{int} = \emptyset$, it is straightforward to define the intersection $\alpha \cap \beta$ of intervals α and β.

We can now represent the interpretation of each facet expression ψ using a set of intervals S_ψ, inductively defined as follows:

$$\begin{aligned} S_{<_w} &= \{real(-\infty, w)\} & S_{>w} &= \{real(w, +\infty)\} \\ S_{\leq_w} &= \{real(-\infty, w]\} & S_{\geq w} &= \{real[w, +\infty)\} \\ S_{int} &= \{int(-\infty, +\infty)\} & S_{\neg\psi} &= \{real(-\infty, +\infty) \cap \alpha \mid \alpha \in S_\psi\} \\ S_{\psi_1 \vee \psi_2} &= S_{\psi_1} \cup S_{\psi_2} & S_{\psi_1 \wedge \psi_2} &= \{\alpha \cap \beta \mid \alpha \in S_{\psi_1} \text{ and } \beta \in S_{\psi_2}\} \end{aligned}$$

In practice, it is beneficial to ensure that each S_ψ does not contain overlapping, empty, or adjoining intervals; the sets S_ψ can then be efficiently implemented by storing interval end-points in a sorted array.

The function $\mathsf{minc}_{real}(real[\varphi], n)$ can then be implemented by computing S_φ, checking whether it consists only of finite integer-restricted intervals, and if so, comparing their total length to n. Similarly, the function $\mathsf{enu}_{real}(real[\varphi])$ can be implemented by computing S_φ, checking whether it consists of only finite integer-restricted intervals, and if so, enumerating all the relevant integers. The function $\mathsf{in}_{real}(c, real[\varphi])$ can be implemented in a straightforward manner if φ is a facet, and it can be computed recursively for φ a general expression by taking into account the standard semantics of propositional connectives. Finally, the function $\mathsf{eq}_{real}(c_1, c_2)$ can be implemented by normalizing the lexical representation of c_1 and c_2 and then comparing the result.

5.3 A Datatype Handler for Strings

We now discuss the implementation of the datatype handler for the *xsd:string* datatype of OWL 2, here abbreviated as *str*. The function $\mathsf{eq}_{str}(c_1, c_2)$ can be implemented as an identity. The function $\mathsf{in}_{str}(d[\varphi], c)$ can be implemented in a straightforward manner if φ is a facet (if φ is a regular expression, membership of a string in a regular language can be checked as in [6]), and it can be computed recursively for a general expression φ in the obvious way.

The implementations of $\mathsf{minc}_{str}(d[\varphi], n)$ and $\mathsf{enu}_{str}(d[\varphi])$ differ based on the facets used in φ. If φ contains no regular expressions—that is, if the only restrictions are on the length of the string—then the intervals of allowed string lengths can be computed as in Section 5.2. If φ contains regular expressions, then each of the facets in φ can be represented using a finite state automaton [6]. (Note that the languages of all strings longer or shorter than some integer are regular.) Regular languages are closed under Boolean connectives so using, say, the results from [6], one can compute a finite state automaton $\mathcal{A}_\varphi$ accepting the language $\mathcal{L}_\varphi$ of φ. The next step is to test whether $\mathcal{L}_\varphi$ is finite. This can be done by identifying states that can occur on a path between the starting and the accepting states of $\mathcal{A}_\varphi$ and checking whether these states can occur in a loop. If the language is finite, it can be enumerated by identifying all finite paths between the starting and the accepting states of $\mathcal{A}_\varphi$.

We note that checking emptiness for intersections of regular languages is known to be PSPACE-complete [8]. This source of complexity, however, has been well studied in the literature and several optimization techniques are available.

6 Conclusion

In this paper, we have formalized the datatype system of OWL 2 and have discussed some nontrivial consequences of our definitions. Furthermore, we have discussed the normative datatypes listed in the current OWL 2 Working Draft and have proposed some modifications to the list. Finally, we have presented a general algorithm for datatype checking—the basic reasoning problem involving datatypes. Our algorithm is applicable to any datatype for which a small set of basic operations can be implemented. We have also discussed how to realize these operations for strings and numbers. The main challenge for our future work is to implement these algorithms in our reasoner and test them in practice.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2nd edition, August 2007.
2. F. Baader and P. Hanschke. A Scheme for Integrating Concrete Domains into Concept Languages. In *Proc. IJCAI 91*, pages 452–457, Sydney, Australia, 1991.
3. P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes, Second Edition, October 28 2004. `http://www.w3.org/TR/xmlschema-2/`.
4. V. Haarslev and R. Möller. RACER System Description. In *Proc. IJCAR 2001*, pages 701–706, Siena, Italy, 2001.
5. V. Haarslev, R. Möller, and M. Wessel. The Description Logic $ALCNH_{R+}$ Extended with Concrete Domains: A Practically Motivated Approach. In *Proc. IJCAR 2001*, pages 29–44, Siena, Italy, 2001.
6. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2nd edition, November 24 2000.
7. I. Horrocks and U. Sattler. Ontology Reasoning in the $\mathcal{SHOQ}(\mathbf{D})$ Description Logic. In *Proc. IJCAI 2001*, pages 199–204, Seattle, WA, USA, 2001.
8. D. Kozen. Lower Bounds for Natural Proof Systems. In *FOCS '77*, pages 254–266, Providence, RI, USA, 1977.
9. C. Lutz. *The Complexity of Reasoning with Concrete Domains*. PhD thesis, Teaching and Research Area for Theoretical Computer Science, RWTH Aachen, Germany, 2002.
10. C. Lutz, C. Areces, I. Horrocks, and U. Sattler. Keys, Nominals, and Concrete Domains. *Journal of Artificial Intelligence Research*, 23:667–726, 2005.
11. J. Z. Pan and I. Horrocks. Web Ontology Reasoning with Datatype Groups. In *Proc. ISWC 2003*, pages 47–63, Sanibel Island, FL, USA, 2003.
12. J. Z. Pan and I. Horrocks. OWL-Eu: Adding customised datatypes into OWL. *Journal of Web Semantics*, 4(1):29–39, 2006.
13. P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language: Semantics and Abstract Syntax, W3C Recommendation, February 10 2004. `http://www.w3.org/TR/owl-semantics/`.
14. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier Science Inc., New York, NY, USA, 2006.