Easy Keys for OWL

Bijan Parsia, Ulrike Sattler, and Thomas Schneider

School of Computer Science, University of Manchester, UK
{bparsia,sattler,schneider}@cs.man.ac.uk

Abstract. One of the commonly requested features for OWL is some form of key support, generally phrased as allowing inverse-functional datatype properties. For a variety of technical reasons, these were not included in OWL DL (though they were available in OWL Full). OWL 2—a revision to OWL which is under development by the W3C—introduces a form of key representation, so-called "Easy Keys", which avoids much of the implementational pain of general inverse-functional datatype properties, "dumbs down" nicely to weaker languages (like RDFS), and can be specified (or even implemented) in terms of DL-safe rules. The Easy Key approach also allows for a form of compound key. In this paper we explicate the design and rationale of Easy Keys, sketch a decision procedure for them, and compare them with alternative possibilities, including general inverse-functional datatype properties.

1 Introduction

Identification constraints, keys or inverse-functional datatype properties, are all ways of expressing, by means of a description, that two or more individuals are necessarily identical. These constraints differ in their expressive power and their "behavior", e.g., whether their violation leads to an error, an inconsistency, or the "quiet" identification of the individuals in question. Regardless of these details, identification constraints are of vital importance to many applications. Declaring, for example, the datatype property `hasSSN` as inverse-functional would mean that any two individuals sharing the `hasSSN` same value are inferred to be identical. For OWL, key reasoning in general can be unfeasibly difficult [1–3] and, to the best of our knowledge, no existing OWL DL reasoner handles inverse-functional datatype properties in a complete way. Consider the ontology $\mathcal{O}$ from Figure 1 about records of persons in a certain country's tax office.

```
1  ObjectProperty: hasParent   Characteristics: InverseFunctional
2  Class: Person   hasKey hasSSN
     SubClassOf: hasParent some Person,  hasSSN some int[>41, <43]
3  Individual: Mary   Types: Person, inv(hasParent) only owl:Nothing
4  Individual: Sheevah   Types: Person
```

Fig. 1. Example ontology $\mathcal{O}$

- Axiom 1 expresses a strict single-child policy of the country in question. (This might seem a bit artificial, but for the explanations below we need a property that naturally creates long chains and can be inverse-functional.)
- Axiom 2 states that the datatype property `hasSSN` (which stands for "social security number") is a key for the class `Person`, and that every person has a parent and an SSN that is an integer between 41 and 43. In particular, each instance of `Person` sits in an infinite `hasParent` chain, possibly looping.
- Axioms 3 and 4 say that `Mary` and `Sheevah` are instances of `Person`, and `Mary` is childless—i.e., sits at the beginning of such an infinite chain.

Suppose now that the key constraint in Axiom 2 is interpreted by stating that `hasSSN` is inverse-functional if restricted to `Person`:

(R_1) For all individuals x, y of type `Person`, if x and y agree on their `hasSSN` value, then $x = y$. Under this reading, $\mathcal{O}$ has no models: Axiom 3 says that `Mary` is a `Person`. According to Axiom 2, she has a parent x who is a `Person`. Axiom 2 then implies that both have SSN 42. Since SSN is a key for `Person`, `Mary` and x coincide. Hence, Mary is her own parent, which contradicts the last part of Axiom 3.

An analogous contradiction would occur if we replaced the range of `hasSSN` with "`int`[> 0, $< 1,000,000$]". Then, in order to show unsatisfiability of $\mathcal{O}$, we would need to create $1,000,000$ ancestors of Mary's before we could reason that two of them collapse and that this would induce a cycle in the `hasParent` chain and therefore contradict Axiom 1.

Hence, if interpreted as a general inverse-functional datatype property, a key constraint can require very large, yet finite, interpretations and makes reasoning computationally difficult. We believe that this expressivity is not required by applications and is too difficult for users to make effective use of. Instead of tackling the general problem of reasoning with keys in OWL, we propose a restricted version of keys—*EasyKeys*—which meets important use cases and is practical to implement. This is achieved by incorporating DL-safety [4] into (R_1):

(R_2) For all named individuals x, y of type `Person`, if x and y agree on their named `hasSSN`-values, then $x = y$. Both occurrences of "named" in this condition restrict the key constraint to *explicit data*: key reasoning is only triggered for *named* individuals and values (i.e., those occurring in the ontology) and not for *generated* ones (i.e., those generated by the reasoner). Under this reading, $\mathcal{O}$ now has models, for instance the one consisting of an infinite chain of instances of `Person`, headed by `Mary`, plus a separate individual `Sheevah`. Each instance of `Person` has a `hasSSN`-successor of value 42, although this is not explicit in $\mathcal{O}$ for any of the named individuals. Hence $\mathcal{O}$ is consistent, but does not entail any equality between named individuals. However, there is another model of $\mathcal{O}$ where `Mary` and `Sheevah` coincide.

The missing entailment is already problematic, but—even worse—it can be restored by adding a completely unrelated fact, see Section 4.1. Therefore, we need to relax DL-safety and obtain:

(R_3) For all named individuals x, y of type `Person`, if x and y agree on their `hasSSN`-values, then $x = y$. The omission of the second occurrence of "named" in this condition restricts the models of $\mathcal{O}$ to those where `Mary` and `Sheevah` coincide. Hence, $\mathcal{O}$ is still consistent, but now entails `Mary` = `Sheevah`.

This reading seems the most natural for $\mathcal{O}$, and we claim that the relaxation of DL-safety does not lead to computational difficulties. We will base EasyKeys' semantics on reading (R_3) and discuss this choice in more detail in Section 4.

Since we are designing a feature of OWL 2, we want EasyKeys to fit in with other Semantic Web languages such as the various OWL 2 Profiles[1] and even RDF(S) (perhaps extended with Datalog like rules). We started with the following desiderata in mind when designing EasyKeys:

(D_1) Be useful enough to be worth specifying.
(D_2) Have a low impact on implementations, at least in common cases.
(D_3) Be easy to specify clearly and standardize.

In our experience (and in the experience of users we interviewed), applications merely require that key constraints are checked on *explicit data*, see (R_2) and (R_3). This alone is a big and effective reduction in expressivity since it allows us to use a variant of DL-safe SWRL rules [4] to specify (and even implement) key semantics. The DL-safety condition (roughly, that variables in rule atoms range only over individuals named in the ontology) is sufficient to render SWRL rules decidable and hopefully feasible in practice. Furthermore, DL-safe rules degrade gracefully as we step down the overall expressiveness of the language, making EasyKeys intuitively feasible for RDF(S) systems with Datalog rule support. These facts strongly suggest that key constraints with semantics based on (R_2) or (R_3) squarely hit the application, implementation, and specification desiderata.

In this paper, we will discuss the intuitions for EasyKeys and their desired properties. We will define a suitable notion of key constraints and explain how to extend standard tableau algorithms to incorporate them. Finally, we will discuss design alternatives for OWL based keys.

Related Work. In the literature, key constraints and, more general, functional dependencies have been added to description logics of different expressivity, largely being interpreted according to reading (R_1). This was unproblematic in the absence of datatypes, where complexity of reasoning was hardly affected [5–10]. But it led to undecidability in the presence of datatypes even for the description logic $\mathcal{ALC}$ [1],[2] which is far less expressive than the designated OWL-2 DL $\mathcal{SROIQ}$ [11]. The latter result suggests to carefully balance the amount of expressivity added to OWL by key constraints. We are confident that EasyKeys meet this requirement.

[1] `http://www.w3.org/TR/owl2-profiles/`

[2] In fact, this result was proven for $\mathcal{ALC}$ extended by a very simple datatype $\mathcal{D}$ that has an efficiently decidable satisfiability problem.

2 EasyKey Intuitions

Keys in general have (or may have) the following properties.

(P_1) *If two individuals x and y have the same key values, then $x = y$.*

(P_2) *Integrity constraint: missing key values raise an error.* Individuals which may have a key *must* have a key. This is easily seen in the case of relational database rows.

(P_3) *Functionality constraint: entities have only one key.* This can be interpreted weakly (in any model, a given entity has at most one key) or strictly (each entity has a known key, the same one in every model).

Property (P_1) is essential—it states the definition of a key. Property (P_2) is not expressible directly in first-order logic (being non-monotonic), thus in neither OWL nor OWL plus DL-safe rules. There are proposals for adding non-monotonic features to a language like OWL (e.g., [12–14]) and even one that is an extension of DL-safe rules [15]. However, none of these OWL-tailored approaches have gotten significant traction with implementors or users. This alone makes adopting a solution to (P_2) as part of EasyKeys a violation of Desideratum (D_3). Not only would we have to build momentum for one of the alternative mechanisms, but we would have to consider the effect of those mechanisms on a wide range of problems beyond missing keys. So we forgo (P_2) for the moment.

Property (P_3) in its weak sense (in any model, a given entity has at most one key) can be expressed in OWL as a `maxCardinality 1` statement, and this also works in the known-individual/value case. However, the strict sense (each entity has a known key, the same one in every model) is not expressible in first-order logic. Of course, while we may impose functionality on keys (at least of a certain type), we do not need to build that into EasyKeys. For example, it is easy to imagine that certain classes of entities may have multiple keys, even of the same sort: email addresses may be keys for people in many datasets and people can have multiple email addresses. We can even more easily imagine that entities may have keys of diverse sorts, e.g., a book may have an ISBN and a Library of Congress catalog number.

Property (P_1) can be expressed as a DL-safe rule according to (R_2) if we assume the necessary DL-safety restrictions. For example, DL-safe Rule (1) expresses that `hasSSN` is a key property.

$$\texttt{hasSSN}(x,k),\ \texttt{hasSSN}(y,k) \ \rightarrow\ x = y \tag{1}$$

$$\texttt{hasSSN}(x,k),\ \texttt{hasSSN}(y,k),\ \texttt{Person}(x),\ \texttt{Person}(y) \ \rightarrow\ x = y \tag{2}$$

$$\texttt{hasSSN}(x,s),\ \texttt{age}(x,a),\ \texttt{hasSSN}(y,s),\ \texttt{age}(y,a) \ \rightarrow\ x = y \tag{3}$$

Key properties can also be restricted to certain classes, e.g., DL-safe Rule (2) expresses that `hasSSN` is a key property for the class `Person`. We can also capture *compound keys*, where several properties together act as a key. For example, we can express that having the same social security number *and* being the same age (we might reuse the SSN every year/generation) implies being the same individual via DL-safe Rule (3).

Thus, in principle, if OWL 2 had a suitable rule language, then users could directly encode their key constraints as rules. Between the OWLED taskforce on DL-safe SWRL rules[3] and the Rule Interchange Format working group[4] of the W3C (not to mention the increasing number of DL-safe rule implementations), it is pretty clear that some sort of more or less standard rule extension to OWL 2 will emerge in the next year. However, even if we had in hand a standard, widely deployed rule language, requiring users to use rules to express key constraints is a suboptimal choice for several reasons:

1. Such rules are not intention-revealing: they do not clearly signal that their purpose is to express a key constraint. Having some sort of key-specific constructor would make the intention of the expression clear to users. It would make a similar difference if, say, the source code of a computer program contained case distinctions expressed by `switch` statements as opposed to nested `if` statements. It is equally common in natural, programming and formal language to use defined constructs rather than their full definition. If this were not the case, how could we search an ontology (or program) for occurrences of transitivity statements or key constraints (or case distinctions)?
2. Using rules for key constraints requires a fair bit of sophistication and care in their formulation. One must keep track of the variables, their relations, and which sorts of atoms go where. Compare this with the transitivity operator: one could express transitive properties as a (general) SWRL rule, but at the price of having to coin three variables and understand how to set up the rule. Though the latter may seem trivial, it is not: writing out transitivity statements as rules is error-prone and obfuscating.
3. It may be possible, as in the transitivity case, to provide special support in reasoners for keys. This would be, however, more difficult to implement if first one needed to analyze the rule base for "key like" rules.

These considerations led us to conclude that it was worth introducing special syntax for keys with the restricted, DL-safe rule-esque semantics into OWL 2.

3 EasyKey Details

In this section, we will define key constraints that meet the above intuitions, and explain how to obtain a tableau algorithm that incorporates all features of OWL and key constraints. This will be achieved by combining existing tableau algorithms for $\mathcal{SROIQ}$—the designated OWL 2 description logic—with existing tableau algorithms for DLs with datatypes and key constraints.

We aim at making this theoretical part as accessible to users and implementors as possible without being imprecise. Therefore we will only introduce those details that are absolutely necessary, and refer the reader to the literature for the missing parts. Readers who are primarily interested in modeling issues can safely skip to Section 4. For a formal introduction into OWL, $\mathcal{SROIQ}$, and DLs with datatypes see [16, 17, 1].

[3] `http://wiki.webont.org/page/DL_Safe_SWRL_Rules`
[4] `http://www.w3.org/2005/rules/`

3.1 Datatype Basics

Datatypes in OWL 2 are restrictions of concrete domains [1]. A *datatype* $\mathcal{D}$ is a pair $(\Delta_{\mathcal{D}}, \Phi_{\mathcal{D}})$, where $\Delta_{\mathcal{D}}$ is a set and $\Phi_{\mathcal{D}}$ is a set of predicate names. Each predicate name $P \in \Phi_{\mathcal{D}}$ is associated with an arity n and an n-ary predicate $P^{\mathcal{D}} \subseteq \Delta_{\mathcal{D}}^n$. For example, if we drop the range restriction from Axiom 2 of $\mathcal{O}$ in Figure 1, we can use $\mathcal{D} = \mathbb{Z}$ (the integers) and $\Phi_{\mathcal{D}} = \{\leqslant, \geqslant, <, >, \leqslant_0, \dots\}$.

Decision procedures for description logics with datatypes usually invoke datatype reasoners. These decide whether a conjunction c of predicates from $\mathcal{D}$ over some set of variables is satisfiable. To ensure the existence of an algorithm for this satisfiability check, $\mathcal{D}$ has to be *admissible* [18]. Essentially, this property requires the set $\Phi_{\mathcal{D}}$ of predicates of $\mathcal{D}$ to contain a unary predicate $\top_{\mathcal{D}}$, to be closed under negation, and it requires satisfiability of $\mathcal{D}$-conjunctions to be decidable.

In the presence of key constraints, standard decision procedures use a variation of this datatype reasoner: in case c is satisfiable, the reasoner chooses a solution s of c and outputs the set of variables that take on the same values in s. To ensure the existence of such an algorithm, $\mathcal{D}$ has to be *key-admissible* [1]. This means that $\mathcal{D}$ is admissible and there exists an algorithm that decides satisfiability of $\mathcal{D}$-conjunctions c and, in the positive case, nondeterministically outputs an equivalence relation $\sim$ on the set of variables V used in c such that there exists a solution s for c with the following property: for all $v, v' \in V$, $s(v) = s(v')$ if and only if $v \sim v'$. From now on, we will always assume key-admissibility of $\mathcal{D}$, which is not too strong a restriction because it is already satisfied if $\mathcal{D}$ is admissible and has an equality predicate.

3.2 EasyKey Syntax and Semantics

A *key constraint* is a statement of the form

$$C \ \texttt{hasKey} \ (p_1, \dots, p_n, d_1, \dots, d_m), \tag{4}$$

where $p_1, \dots, p_n$ are simple object properties, $d_1, \dots, d_m$ are datatype properties, and C is a class description. The requirement of p_i being simple is necessary in order to ensure decidability [17, 2]. The semantics of (4) in an ontology $\mathcal{O}$ can be defined in two ways, using the following abbreviation.

$$\alpha = C(x) \wedge C(y) \wedge \bigwedge_{i=1,\dots,n} \big(p_i(x, z_i) \wedge p_i(y, z_i)\big) \wedge \bigwedge_{i=1,\dots,m} \big(d_i(x, v_i) \wedge d_i(y, v_i)\big)$$

1. An interpretation $\mathcal{I}$ is a *g-model of* $\mathcal{O}$ *and* (4) if $\mathcal{I}$ is a model of $\mathcal{O}$ and satisfies (5), which corresponds to reading (R_1) from Section 1.

$$\forall xy \big[\exists z_1 \dots z_n v_1 \dots v_m \,.\, \alpha \ \rightarrow \ x = y\big] \tag{5}$$

2. An interpretation $\mathcal{I}$ is an *e-model of* $\mathcal{O}$ *and* (4) if $\mathcal{I}$ is a model of $\mathcal{O}$ and satisfies the following property, where HU is a predicate holding of all elements of the Herbrand universe, i.e., HU holds of all individuals named in $\mathcal{O}$.

$$\forall xy \Big[\exists z_1 \dots z_n v_1 \dots v_m \Big[\alpha \wedge \mathrm{HU}(x) \wedge \mathrm{HU}(y) \wedge \bigwedge_{i=1,\dots,n} \mathrm{HU}(z_i)\Big] \ \rightarrow \ x = y\Big] \tag{6}$$

Had we included conjuncts $\mathrm{HU}(v_i)$ for each datatype variable v_i, this statement could have been captured by a DL-safe rule and would correspond to reading (R_2). Without these additional conjuncts, (6) corresponds to (R_3). We will discuss in Section 4 why we forgo the "missing" portion of DL-safety.

We call key constraints with semantics (5) *general key constraints* and those with semantics (6) *EasyKey constraints*. The former are a generalization of inverse-functional datatype properties (IFDTPs), whose problematic behavior w.r.t. reasoning they inherit, see Section 4.3. EasyKey constraints, on the contrary, are orthogonal to IFDTPs, and we will see next why the can be viewed as being computationally harmless.

In the following, we distinguish between *abstract* individuals/variables (instances of classes) and *concrete* ones (representing datatype values).

3.3 EasyKey Reasoning

It suffices to restrict our attention to class satisfiability because all other standard DL reasoning problems can be reduced to this problem [19]. We will show how to extend the tableau algorithm for $\mathcal{SROIQ}$ from [17] to incorporate datatypes and EasyKey constraints. This will be achieved by picking the necessary parts from existing tableau algorithms for $\mathcal{SHOQK}(D)$ [1] and $\mathcal{SHOQ}(D_n)$ [20]. Since our extension of the $\mathcal{SROIQ}$ tableau algorithm will not affect its complexity, reasoning with EasyKey constraints is not harder than without.

We will not fully develop the algorithm here because the necessary technical details would repeat large parts of the algorithms in [1, 17] and obstruct the view on the essential parts. However, this section specifies *all* components needed to implement an extension of an OWL 2 reasoner that treats EasyKey constrains.

We expect the reader to be familiar with tableau algorithms as in [21, 19, 17] and start from the $\mathcal{SROIQ}$ tableau algorithm from [17], including its rules, its blocking condition, and its pruning and merging technique. Recall that the *NN*-rule from this algorithm creates new individuals, but EasyKey constraints only affect abstract nodes[5] labelled with *named* individuals.

Before we show how to reason in the presence of EasyKey constraints, let us recall reasoning with non-inverse-functional datatype properties from [22, 20]. This is done with the following extension of standard tableau algorithms. To each abstract node x, we associate a set $\mathrm{DC}(x)$ containing *datatype constraints* of the form $(x, P^{\mathcal{D}}(t_1, \ldots, t_n))$, where the t_i are concrete successor nodes of x, and P is an n-ary datatype predicate. The constraints to go into the $\mathrm{DC}(x)$ are obtained via tableau expansion rules that have been given in [20] as the datatype counterparts of the usual $\forall$, $\exists$, $\geqslant$, $\leqslant$, and *choose* rules from standard tableau algorithms [22]. If number restrictions with datatype properties are not required, the $\forall$ and $\exists$ rules suffice. They are given in Figure 2. The instructions "update $\sim$" in these rules are only necessary below, in the presence of key constraints. The $\geqslant$, $\leqslant$, and *choose* rules are way more technical, see [20].

[5] Abstract/concrete nodes in a tableau are nodes associated with abstract/concrete individuals.

```
∃-rule:  if $\exists g_1, \ldots, g_n.P \in \mathcal{L}(x)$, $x$ is not blocked, and
           there are no $g_i$-successors $x_i$ with $P(x_1, \ldots, x_n) \in \mathrm{DC}(x)$
         then *add a $g_i$-successor of $x$* for each $1 \leqslant i \leqslant n$,
              for $y_i$ the $g_i$-successor of $x$, add $P(y_1, \ldots, y_n)$ to $\mathrm{DC}(x)$, and
              *update* $\sim$

∀-rule:  if $\forall g_1, \ldots, g_n.P \in \mathcal{L}(x)$, $x$ is not blocked, and
           there are $g_i$-successors $y_i$ of $x$ with $P(y_1, \ldots, y_n) \notin \mathrm{DC}(x)$
         then add $P(y_1, \ldots, y_n)$ to $\mathrm{DC}(x)$, and
              *update* $\sim$
```

Fig. 2. Two completion rules for $\mathcal{SHOQ}(D_n)$.

For instance, if the abstract node x contains the label $\exists$`height`, `weight`.$>$ expressing that the weight of the individual associated with node x is greater than its height, then the datatype $\exists$-rule will create `weight`- and `height`-successors y and z and add the constraint $y > z$ to $\mathrm{DC}(x)$. When detecting clashes, $\mathrm{DC}(x)$ is tested for satisfiability by a datatype reasoner (DTR), whose negative answer is a clash for x. (This requires $\mathcal{D}$ to be admissible.) It suffices to invoke the DTR once after all completion rules have been exhaustively applied [18, 1].

EasyKey constraints, as opposed to inverse-functional datatype properties, do not impair tableau termination because they only affect *named individuals*. The general extension of standard tableau algorithms to key constraints can be taken from [1] and is as follows.

As above, datatype constraints are collected and checked for satisfiability. But since key constraints may cause abstract nodes labelled by named individuals to merge, a separate construction and treatment of each $\mathrm{DC}(x)$ does not suffice. Instead, for all abstract nodes x labelled with *named* individuals, all constraints from labels in $\mathcal{L}(x)$ have to be put into a global set DC of constraints.

The DTR that runs on DC has to answer more than "yes" if DC is satisfiable. In addition, it has to nondeterministically output conditions under which DC is satisfied, namely an equivalence relation $\sim$ on the set of variables occurring in DC such that $\mathrm{DC}^{\sim} \cup \{y \neq z \mid y \not\sim z\}$ is satisfiable. Here $\mathrm{DC}^{\sim}$ denotes DC where each occurrence of a variable y is replaced by some fixed representative of y's equivalence class. The existence of such a DTR is ensured by the key-admissibility of $\mathcal{D}$.

After a positive answer of the DTR, abstract nodes are merged according to $\sim$. As a consequence, concrete nodes might have to be merged as well—but they might have incoming edges labelled with datatype properties that are involved in key constraints. Therefore, DC needs to be updated, and the process of running the DTR plus merging has to be iterated. Since, in each iterative step, the "new" relation $\sim$ is a superset of the "old" one, this procedure terminates. This is explained in more detail in [1]. In particular, since it suffices to consider $\sim$ restricted to those abstract nodes that are labelled by named individuals, the number of iterations is at most quadratic in the size of the ontology.

Furthermore, the size of DC is polynomial in the size of the ontology: for each abstract node x labelled with a named individual, x will only contribute as many constraints to DC as it has outgoing edges. The number of the latter is bounded by the sum of the numbers that occur in any number restriction in x's labels. Hence x contributes at most $m \cdot n$ constraints, where m is the maximal number that occurs in number restrictions in x's labels, and n is the count of these restrictions. Since m, n, and the number of named individuals are bounded by the size of the ontology, we obtain a cubic upper bound for the size of DC.

4 Key Design Alternatives

Why are these keys easy? The reason is the usage of HU in the above definition (6) of the semantics: key constraints only act on abstract individuals explicitly present in our ontology, but not on those abstract individuals whose existence is only implied. And key constraints "act" by inferring new equalities.

In (6), the predicate HU is applied to all *abstract* variables, but not to *concrete* ones. This is so because, for the former, it is (a) necessary for decidability (DL-safety), and (b) it is easy to determine for which elements HU holds—but it would be more problematic for datatype values and is not needed for decidability.

For this subtlety in the usage of HU, EasyKeys cannot be captured by DL-safe rules, according to the definition of the latter in [4]. Furthermore, the standard-resolution based proof in [4] that OWL with rules is decidable would not go through with our relaxed version of DL-safety. (But it might work for a less standard way of resolution.) However, our relaxation of DL-safety does not significantly impair standard tableau algorithms and hence "preserves" this particular method of showing decidability. Section 3.3.

4.1 Why Not Easiest Keys?

The easiest keys would be according to the interpretation (R_2) from Section 1. This special case of DL-safe rules would mean to handle the abstract and concrete variables in precisely the same way. One could then preprocess the ontology, looking for explicit concrete values, and treat them as the only substitution candidates for the variables. This case would even simplify the operation of the datatype reasoner DTR (see Section 3.3): for each generated or named abstract individual, only the set DC(x) needs to be evaluated independently, while, for EasyKeys, the datatype constraints for the named abstract individuals are collected in the global set DC and evaluated.

Unfortunately, easiest keys lead to some rather counterintuitive cases. Recall the example ontology $\mathcal{O}$ from Figure 1 and the explanation why $\mathcal{O}$ does not entail `Mary` = `Sheevah` under (R_2). Let us now add the following assertion to $\mathcal{O}$:

```
5  Individual: Demeter   Types: hasAge value 42
```

This is a *completely* unrelated fact—it does not say anything about `Demeter`'s type or `hasSSN`-value. But since now 42 is explicit in $\mathcal{O}$, $\mathcal{O}$ entails `Sheevah` =

```
1  Class: Person   hasKey hasSSN
2  Individual: Mary   Types: Person, hasSSN integer[>41, <43]
3  Individual: Sheevah   Types: Person
```

Fig. 3. Example ontology $\mathcal{O}$'

`Mary`. A similar anomaly can be observed for $\mathcal{O}'$ from Figure 3. This ontology does not entail `Sheevah`=`Mary`, but it will if we replace Axiom 2 by 2'.

```
2'  Individual: Mary   Types: Person, hasSSN value 42
```

Such anomalies are the primary reason for adopting slightly less easy keys. They cause quite some implementation challenge since large, but finite, enumerations of key values, e.g., `integer`[>0, <1000], can yield nontrivial interactions.

4.2 When Do Easiest Keys and EasyKeys Coincide?

There is a sense in which EasyKeys' extra expressivity covers currently somewhat improbable corner cases. However, the consequences of not covering those cases well seem quite negative: intuitively and *wildly* irrelevant assertions can induce key-based identification. Furthermore, the underspecification of user-defined datatypes in OWL 1 might have been a reason why this sort of case has not emerged. OWL 2 has much more robustly specified datatype support, so we can anticipate that ontology developers will be writing more and more complex data constraints. It is exactly when users go beyond what is easy to understand that helpful semantics with good reasoning support becomes essential.

However, there are two cases where EasyKeys and the easiest keys coincide, namely (1) when data values drawn from finite datatypes are not used as keys (e.g., if the range restriction in Axiom 2 of $\mathcal{O}$ in Figure 1 is replaced with "[>0]") or (2) when all values of key properties are given explicitly. If the modeler takes care to satisfy at least one of these properties, a much simpler "look and merge" algorithm can be employed. Thus, EasyKeys have good pay-as-you-go behavior, in principle, and we expect that implementations will take advantage of that. The restriction is also fairly easy to understand and syntactically enforce.

4.3 Why Not Full Inverse-Functional Datatype Properties?

If you restrict yourself to single key properties (i.e., no composite keys), then our EasyKeys yield a subset of the entailments you can get with inverse-functional datatype properties (IFDTPs). There are two main problems with IFDTPs:

1. While decidable, they are very hard to reason with. In fact, we are not aware of any implementations or any implementor who relishes the challenge. We have discussed in Section 3.3 that, without IFDTPs, datatype reasoning can be confined to a single individual (asserted or generated) at a time. That is, you gather all the data constraints applicable to the individual x and pass

them to the datatype reasoner DTR. If the DTR finds them consistent and x is generated, you mark x as checked and can (largely) forget about it. With IFDTPs, you lose this locality—*all* (named and generated) individuals with datatype properties may need to be checked at any time. Since, in OWL, not all of the potential key values may be explicit, this involves maintaining a large constraint system DC that is constantly updated. Because generated individuals have to be checked as well, termination of the algorithm will be difficult to guarantee.
2. It is not clear that modelers would know what to do with such a powerful feature, or even if they would want such power in most cases. Clearly, they want keys, but it is not at all clear that they want IFDTPs.

So, in essence, we would have a lot of implementation work (and research!) for a not obvious gain. It is much easier to deliver robust, reasonably scalable support for EasyKeys which, for most users, is a dominating consideration.

What do we miss with EasyKeys? Basically, we miss the ability to entail subsumptions through key constraints (at least, directly through key constraints; if we have nominals, for example, we can force subsumptions in specific cases). This is essentially the difference between full SWRL rules and DL-safe SWRL rules. The major use case we could think of for this is aligning database schemas. If you could reduce database schemas to OWL classes (including the key constraints as IFDTPs), then your reasoner could show whether the two schemas were equivalent or disjoint. If one had keys that were integers over 1000, and the other had keys that were integers from 100 to 9999, we would get an inconsistency.

To our minds, this is a very specialized application which, in fact, has never occurred. If it does, one can always appeal to IFDTPs in OWL Full as an extension. Indeed, one nice feature of EasyKeys is that they do not interfere with general IFDTP support: on the one hand, they can be seen as a special case, on the other, they can live side by side in an ontology just as DL-safe rules may live side by side with corresponding transitivity axioms.

5 Conclusion and Future Work

In this paper we have presented a design for helpful, yet principled, key support for OWL like languages. Our basic proposal has been accepted by the OWL WG as part of the forthcoming OWL 2, and we expect it shall make it into the next public working drafts. We have also discussed, in some detail, various considerations that went into the design of EasyKeys and the tradeoffs we made from the implementational and modeling perspectives. We have also presented a direct algorithm for EasyKeys plus some indications on how optimized implementations may be built. Our investigation into EasyKeys has also led us to a new account of data property atom support in DL-safe rules.

For future work, we plan to finish our running implementation of EasyKeys and to study, more precisely, the pay-as-you-go behavior, particularly as one scales up the data. We also plan to study their use in applications to determine whether we in fact made the right tradeoffs.

References

1. Lutz, C., Areces, C., Horrocks, I., Sattler, U.: Keys, nominals, and concrete domains. J. of Artificial Intelligence Research **23** (2005) 667–726
2. Nguyen, T.D.T., Thanh, N.L.: Integrating identification constraints in web ontology. In Cardoso, J., Cordeiro, J., Filipe, J., eds.: ICEIS (1). (2007) 338–343
3. Penwill, C.D.: A tableaux decision procedure for $\mathcal{SHIQ}(D_n)$ and $\mathcal{SHOIQ}(D_n)$. Master's thesis, School of Computer Science, University of Manchester (2006)
4. Motik, B., Sattler, U., Studer, R.: Query answering for OWL-DL with rules. J. of Web Semantics **3**(1) (2005) 41–60
5. Borgida, A., Weddell, G.E.: Adding uniqueness constraints to description logics (preliminary report). In: Proc. of DOOD-97. Volume 1341 of LNCS. (1997) 85–102
6. Calvanese, D., De Giacomo, G., Lenzerini, M.: Keys for free in description logics. In: Proc. of DL 2000, CEUR (`http://ceur-ws.org/`) (2000) 79–88
7. Khizder, V.L., Toman, D., Weddell, G.E.: On decidability and complexity of description logics with uniqueness constraints. In: Proc. of ICDT-01. Volume 1973 of LNCS. (2001) 54–67
8. Toman, D., Weddell, G.E.: On attributes, roles, and dependencies in description logics and the Ackermann case of the decision problem. In: Proc. of DL 2001, CEUR (`http://ceur-ws.org/`) (2001)
9. Toman, D., Weddell, G.E.: On the interaction between inverse features and path-functional dependencies in description logics. In: Proc. of IJCAI-05. (2005) 603–608
10. Toman, D., Weddell, G.E.: On path-functional dependencies as first-class citizens in description logics. In: Proc. of DL 2005, CEUR (`http://ceur-ws.org/`) (2005)
11. Lutz, C., Milicic, M.: Description logics with concrete domains and functional dependencies. In: Proc. of ECAI 2004. (2004) 378–382
12. Baader, F., Hollunder, B.: Embedding defaults into terminological knowledge representation formalisms. In: Proc. of KR-92, Morgan Kaufmann (1992) 306–317
13. Donini, F., Nardi, D., Rosati, R.: Description logics of minimal knowledge and negation as failure. ACM Trans. Comput. Log. **3**(2) (2002) 177–225
14. Motik, B., Horrocks, I., Sattler, U.: Bridging the gap between OWL and relational databases. In: Proc. of WWW-07. (2007) 807–816
15. Motik, B., Rosati, R.: A faithful integration of description logics with logic programming. In: Proc. of IJCAI-07. (2007) 477–482
16. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. J. of Web Semantics **1**(1) (2003) 7–26
17. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible $\mathcal{SROIQ}$. In: Proc. of KR-06. (2006) 57–67
18. Baader, F., Hanschke, P.: A schema for integrating concrete domains into concept languages. In: Proc. of IJCAI-91, Sydney (1991) 452–457
19. Horrocks, I., Sattler, U.: A tableau decision procedure for $\mathcal{SHOIQ}$. J. of Automated Reasoning **39** (2007) 249–276
20. Pan, J.Z., Horrocks, I.: Semantic web ontology reasoning in the $\mathcal{SHOQ}(\mathbf{D_n})$ description logic. In: Proc. of DL 2002. CEUR (`http://ceur-ws.org/`) (2002) 53–62
21. Horrocks, I., Sattler, U., Tobies, S.: Practical reasoning for very expressive description logics. Logic Journal of the IGPL **8**(3) (May 2000) 239–264
22. Horrocks, I., Sattler, U.: Ontology reasoning in the $\mathcal{SHOQ}(D)$ description logic. In Nebel, B., ed.: Proc. of IJCAI-01, Morgan Kaufmann (2001) 199–204