Knowledge Representation on the Web revisited: the Case for Prototypes

Michael Cochez[1,2,4], Stefan Decker[1,2], and Eric Prud'hommeaux[3]

[1] Fraunhofer Institute for Applied Information Technology FIT
DE-53754 Sankt Augustin, Germany
{stefan.decker,michael.cochez}@fit.fraunhofer.de

[2] RWTH Aachen University, Informatik 5
DE-52056 Aachen, Germany

[3] World Wide Web Consortium (W3C)
Stata Center, MIT
eric@w3.org

[4] University of Jyvaskyla,
Department of Mathematical Information Technology
FI-40014 University of Jyvaskyla, Finland

Abstract. Recently, RDF and OWL have become the most common knowledge representation languages in use on the Web, propelled by the recommendation of the W3C. In this paper we examine an alternative way to represent knowledge based on Prototypes. This Prototype-based representation has different properties, which we argue to be more suitable for data sharing and reuse on the Web. Prototypes avoid the distinction between classes and instances and provide a means for object-based data sharing and reuse.

In this paper we discuss the requirements and design principles for Knowledge Representation based on Prototypes on the Web, after which we propose a formal syntax and semantics. We further show how to embed knowledge representation based on Prototypes in the current Semantic Web stack and report on an implementation and practical evaluation of the system.

Keywords: Linked Data, Knowledge Representation, Prototypes

1 Introduction and Motivation

In earlier days of Knowledge Representation, *Frames* [19,20] and Semantic Networks [23] were accepted methods of representing static knowledge. These had no formal semantics but subsequent works (e.g., KL-ONE [2]) introduced reasoning with concepts, roles, and inheritance, culminating in Hayes's 1979 [10] formalization of Frames. This formalization included instances formalized as elements of a domain (individuals) and classes (or concepts) as sets in a domain (unary predicates). This formalization was subsequently used as a basis for Description Logics (DL) and the investigation of expressiveness vs. tractability [16], which lead to Description Logic systems and reasoners such as SHIQ [11] and FaCT [12]. Finally, the Semantic Web effort led to the combination of Description Logics with Web Technologies such as RDF [6], which subsequently evolved into the Web Ontology Language OWL [13]. However, the formalization of Frames only covered some modeling primitives which were in use at the time. Specifically Prototype-based systems, which do not make a distinction between instances and classes, did not get much attention for knowledge representation (cf. Karp [15]). Exceptions exists, for instance, THEO [21], which is a Frame Knowledge Representation System deviating from the — now common — instance–class distinction by using only one type of frame, with the authors arguing that the distinction between instances and classes is not always well defined. Also several programming Languages based on prototypes were successfully developed (SELF [27], JavaScript [8] and others), but the notion of Prototypes as a Knowledge Representation mechanism was not formalized and remained unused in further developments. As noted in [24], these knowledge representation mechanisms may now be again relevant for applications. In this paper we develop a syntax and formal semantics for a language based on prototypes for the purpose of enabling knowledge representation and knowledge sharing on the Web. We argue that such a system has distinctive advantages compared to other representation languages.

This paper is augmented by a separate technical report in which we detail the software which we wrote to support prototype knowledge representation. [3] The report also includes experiments which show how the system performs in a web environment.

2 A Linked Prototype Layer on the Web

2.1 Idea and Vision

Tim Berners-Lee stated the motivation for creating the Web as:

> The dream behind the Web is of a common information space in which we communicate by sharing information. [5]

We aim to optimize the sharing and reuse of structured data. Currently, on the Semantic Web, this sharing is typically achieved by either querying a SPARQL endpoint or downloading a graph or an ontology. We call this *vertical sharing*: top-down sharing where a central authority or institution shares an ontology or graphs. We would like to enable *horizontal sharing*: sharing between peers where individual pieces of instance data can be used and reused. Note that this mode of sharing appears much closer to the intended spirit of the Web. Languages like OWL evolved driven by the AI goal of intelligent behavior and sound logical reasoning [14]. They don't emphasize or enable horizontal sharing - the sharing and reuse of individual objects in a distributed environment. Rather, their goal is to represent axioms and enable machines to reason. Imagine a prototype, for example, an `Oil Painting` with properties and values for those properties, that lives at a particular addressable location on the Web. This prototype `Oil Painting` can be reused in a number of different ways (see Figure 1):

[5] `https://www.w3.org/People/Berners-Lee/ShortHistory.html`

– First, by specializing the the `Oil Painting` prototype(i.e., using it as a template by linking to it), and either specializing or changing its properties. For example, whereas the `Oil Painting` has a value `Canvas` for its `surface` property, the `Arnolfini Portrait` prototype has the value `Oak Panel`. However, the value for the creator property (`Jan van Eyck`) remains the same. To accomplish this, current Semantic Web infrastructure would require one to copy the initial object to a new object before changing its properties. Note, however, that a this also means that the newly created object looses its heritage, meaning that it will not receive any updates which are made to object in the inheritance chain later on.
– Second, by either directly or indirectly referring to it as a value of a property. For instance, in Figure 1 the prototype `National Gallery` has a property `displays`, which links to the prototype `Arnolfini Portrait`, which is based on the `Oil Painting` prototype. This usage of entities is currently also possible using RDF. (But, see also the discussion in section 4.3.)

These two ways to reusing objects on the Web create a distributed network of interlinked objects, requiring horizontal as well as vertical sharing:

– Vertical sharing is enabled by specializing an object or prototype. The prototype that is being specialized defines the vocabulary and structure for the new object, realizing the task of ontologies. For example, a museum can publish a collection of prototypes that describe the types of artifacts on display (e.g., `Oil Painting`), which can then be used to describe more specific objects.
– Horizontal sharing is enabled by reusing prototypes and only changing specific attributes or linking to other prototypes as attribute values. For example, a specific oil painting by painter Jan van Eyck can be used as a template by describing how other oil paintings differ from it, or a specific oil painting can be the attribute value for the `National Gallery` prototype. This creates a network of prototypes across the Web.

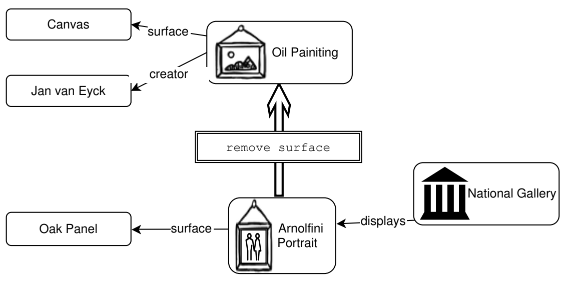


Fig. 1: The figure shows three prototypes and different relations between them. The Arnolfini Portrait is a specialization of the Oil Painting but also displayed at the National Gallery, London.

2.2 Requirements

In the previous section, we presented a vision for a prototype layer on the Web. In this section we discuss requirements for the linked prototype layer. Some of these requirements are based on actual tasks that user communities want to perform while others are based on desirable principles of the World Wide Web.

The linked prototype layer must primarily enable *sharing and reuse of knowledge*. Sharing and reuse of knowledge requires an explicit distributed network of entities. In particular we desire means to share vertically (i.e., provide a central vocabulary or ontology that many can refer to) and share horizontally (i.e., provide concrete reusable entities). Further, it must be possible for the knowledge to evolve over time and anyone should be able to define parts of the network. This implies that central authority should be avoided as much as possible. Preferably, the realization of the prototype layer should be achieved using facilities which the Semantic Web already provides, such as RDF and IRIs, in order to leverage existing data resources. Finally, the designed system should still retain a certain level of familiarity.

2.3 Design Principles

While designing the prototype-based system, we were inspired by design principles, such as the KISS Principle (as defined in [28]), and *worse-is-better* (as coined by R.P. Gabriel [9]). On the intersection between these principles lies the idea of simplicity. The worse-is-better approach encourages dropping parts of the design that would cause complexity or inconsistency.

Our goal was explicitly not to enable sophisticated reasoning, but rather provide a simple object or prototype layer for the Web.

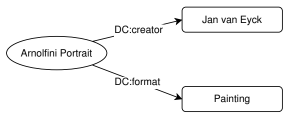
We use the idea of prototypes as suggested in early Frame Systems [15] as well as in current programming languages such as Javascript [8]. Prototypes fulfill the requirements to support the reusabilty and horizontal shareability since it is possible to just refer to an existing prototype that exists elsewhere on the Web, ensuring horizontal shareability. Furthermore a collection of prototypes published by an authority can still serve the function of a central ontology, ensuring vertical shareability.

3 Prototypes

In this section we introduce our approach for knowledge representation on the web, based on prototypes. First, we provide an informal overview of the approach, illustrating the main concepts. Then we introduce a formal syntax and semantics.

3.1 Informal Presentation

To illustrate the prototype system we use an example about two Early Netherlandish painters, the brothers *van Eyck*. First, we look at a simple representation of the *Arnolfini Portrait* in fig. 2.[6] This figure contains the prototype of the portrait which is derived from the empty prototype ($\mathtt{P}_\emptyset$, see section 3.2) and has two properties. The first property is `dc:creator` and has value `Jan van Eyck`[7]. The second property describes the format of the artwork. We also display the example using a concrete syntax.



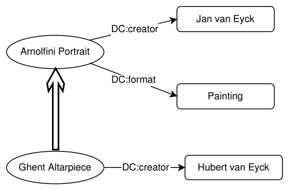
(a) Graphical Representation

```
example:Arnolfini_Portrait
base proto:P_0
add dc:creator example:Jan_Van_Eyck
add dc:format example:Painting
```

(b) Concrete Syntax

Fig. 2: The prototype representation of the Arnolfini Portrait

Next we will start making use of the prototype nature of the representation. Starting from the Arnolfini Portrait, we derive the *Ghent Alterpiece*. This painting was created by the same painter, but also his brother *Hubert van Eyck* was involved in the creation of the work. Figure 3 illustrates how this inheritance works in practice; we create a prototype for the second work and indicate that its base is the first one (using the big open arrow). Then, we add a property asserting that the other brother is also a creator of the work. The resulting prototype has the properties we defined directly as well as those inherited from its base.



(a) Graphical Representation

```
example:Ghent_Altarpiece
base example:Arnolfini_Portrait
add dc:creator example:Hubert_Van_Eyck
```

(b) Concrete Syntax

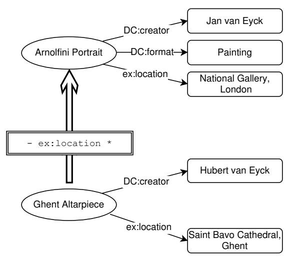
Fig. 3: Deriving the prototype representation of the Ghent Altarpiece from the Arnolfini Portrait

[6] In the illustrations, we loosely write identifiers like `Arnolfini Portrait` for prototypes, properties and their values. However, the proposed systems requires the use of IRIs for identifiers, just like RDF. The concrete syntax examples reflect this. Note that our syntax does not support prefixes as supported by RDF Turtle syntax. If we write `dc:creator` we mean an IRI with scheme `dc`.

[7] For illustrative purposes we use different graphical shapes for the prototypes under consideration and the values of their properties. However, as will become clear in the sections below, all values are themselves prototypes.

Often, there will be a case where the base prototype has properties which are not correct for the derived prototype. In the example shown in fig. 4 we added the `example:location` property to the Arnolfini Portrait with the value `National Gallery, London`. The Ghent Altarpiece is, however, located in the `Saint Bavo Cathedral, Ghent`. Hence, we first remove the `example:location` property from the Arnolfini Portrait before we add the correct location to the second painting. In effect, the resulting prototype inherits the properties of its base, can remove unneeded ones, and add its own properties as needed.

Another way to arrive at the same final state would be to derive from a base without any properties and add all the properties needed. The predefined empty prototype (`proto:P_0`) has no properties. All other prototypes derive from an already existing prototype; circular derivation is not permitted. Now, we will let the prototype which we are creating derive directly from the empty prototype and add properties. This flattening of inherited properties produces the prototype's *fixpoint*. The fixpoint of the prototype created in fig. 4 can be found in fig. 5.



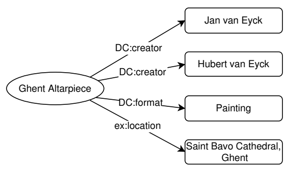
(a) Graphical Representation

```
example:Arnolfini_Portrait
base proto:P_0
add dc:creator example:Jan_Van_Eyck
add dc:format example:Painting
add example:location example:National_Gallery

example:Ghent_Altarpiece
base example:Arnolfini_Portrait
rem example:location *
add dc:creator example:Hubert_Van_Eyck
add example:location example:Saint_Bavo
```

(b) Concrete Syntax

Fig. 4: Removing properties while deriving the Ghent Altarpiece from the Arnolfini Portrait



(a) Graphical Representation

```
example:Ghent_Altarpiece
base proto:P_0
add dc:creator example:Jan_Van_Eyck
add dc:format example:Painting
add dc:creator example:Hubert_Van_Eyck
add example:location example:Saint_Bavo
```

(b) Concrete Syntax

Fig. 5: The result of removing properties while deriving the Ghent Altarpiece from the Arnolfini Portrait.

In the proposed system we apply the closed world and the unique name assumptions. If the system used the open world assumption and one would ask whether the Arnolfini Portrait is located in Beijing, the system would only be able to answer that it does not know. In a closed world setting, the system will answer that the painting is not in Beijing. This conclusion is not based on the fact that the system sees that the painting is located in England, but because of the fact that there is no indication that it would be in Beijing. Under the *non*-unique name assumption, the system would not be able to answer how many paintings it knows about. Instead, it would only be able to tell that there are one or more. Without the unique name assumption, the resource names `Ghent Altarpiece` and `Arnolfini Portrait` may refer to the same real-world instance.

3.2 Formal presentation

The goal of this section is to give a formal presentation of the concepts discussed in the previous section. We separate the formal definition into two parts. First, we define the syntax of our prototype language. Then, we present the semantic interpretation and a couple of definitions which we used informally above.

Prototype Syntax In this section we define the formal syntax of prototype-based knowledge bases. We define a set of syntactic material first, before we define the language.

Definition 1 (Prototype Expressions). *Let ID be a set of absolute IRIs according to RFC 3987 [7] without the IRI `proto:P_0`. The IRI `proto:P_0` is the empty prototype and will be denoted as $\mathtt{P}_\emptyset$. We define expressions as follows:*

- *Let $p \in ID$ and $r_1, \ldots, r_m \in ID$ with $1 \leq m$. An expression $(p, \{r_1, \ldots, r_m\})$ or $(p, *)$ is called a* simple change expression. *p is called the simple change expression ID, or its property. The set $\{r_1, \ldots, r_m\}$ or $*$ are called the* values of the simple change expression.
- *Let $id \in ID$ and $base \in ID \cup \mathtt{P}_\emptyset$ and add and remove be two sets of simple change expressions (called change expressions) such that each simple change expression ID occurs at most once in each of the add and remove sets and $*$ does not occur in the add set. An expression $(id, (base,\ add,\ remove))$ is called a* prototype expression. *id is called the prototype expression ID.*

Let $\mathtt{PROTO}$ be the set of all prototype expressions. The tuple $PL = (\mathtt{P}_\emptyset, ID, \mathtt{PROTO})$ is called the Prototype Language.

Informally, a prototype expression contains the parts of a prototype which we introduced in the previous subsection. It has an id, a base (a reference to the prototype it derives from), and a description of the properties which are added and removed.

As an example, we could write down the example of fig. 4 using this syntax. The prototype expression of the Arnolfini Portrait would look like this:

$$(\texttt{example:Arnolfini\_Portrait}, (\texttt{proto:P\_0},$$
$$\{(\texttt{dc:creator}, \{\texttt{example:Jan\_Van\_Eyck}\}),$$
$$(\texttt{dc:format}, \{\texttt{example:Painting}\}),$$
$$(\texttt{example:location}, \{\texttt{example:National\_Gallery}\})\},$$
$$\emptyset))$$

.

The prototype for the Altarpiece would be written down as follows:

$$(\texttt{example:Ghent\_Altarpiece}, (\texttt{example:Arnolfini\_Portrait},$$
$$\{(\texttt{dc:creator}, \{\texttt{example:Hubert\_Van\_Eyck}\}),$$
$$(\texttt{example:location}, \{\texttt{example:Saint\_Bavo}\})\},$$
$$\{(\texttt{example:location}, *)\}))$$

.

This syntax is trivially transformable into the concrete syntax which we used in fig. 4b and the other examples in the previous subsection.

Definition 2 **(*dom*).** *The domain of a finite subset $S \subseteq \texttt{PROTO}$, i.e., $dom(S)$ is the set of the prototype expression IDs of all prototype expressions in S.*

Definition 3 (Grounded). *Let $PL = (\texttt{P}_\emptyset, ID, \texttt{PROTO})$ be the Prototype Language. Let $S \subseteq \texttt{PROTO}$ be a finite subset of $\texttt{PROTO}$. The set $\mathcal{G}$ is defined as:*

1. $\texttt{P}_\emptyset \in \mathcal{G}$
2. If there is a prototype $(id, (base, add, remove)) \in S$ and $base \in \mathcal{G}$ then $id \in \mathcal{G}$.
3. $\mathcal{G}$ is the smallest set satisfying (1) and (2).

S is called grounded iff $\mathcal{G} = dom(S) \cup \{\texttt{P}_\emptyset\}$. This condition ensures that all prototypes derive (recursively) from $\texttt{P}_\emptyset$ and hence ensures that no cycles occur.

To illustrate how cycles are avoided by this definition, imagine that $S = \{(A, (\texttt{P}_\emptyset, \emptyset, \emptyset)), (B, (C, \emptyset, \emptyset)), (C, (B, \emptyset, \emptyset)), \}$. What we see is that there is a cycle between B and C. If we now construct the set $\mathcal{G}$, we get $\mathcal{G} = \{\texttt{P}_\emptyset, A\}$ while $dom(S) \cup \{\texttt{P}_\emptyset\} = \{A, B, C, \texttt{P}_\emptyset\}$, and hence the condition for being grounded is not fulfilled.

Definition 4 (Prototype Knowledge Base). *Let $PL = (\texttt{P}_\emptyset, ID, \texttt{PROTO})$ be the Prototype Language. Let $KB \subseteq \texttt{PROTO}$ be a finite subset of $\texttt{PROTO}$. KB is called a* Prototype Knowledge Base *iff 1) KB is grounded, 2) no two prototype expressions in KB have the same prototype expression ID, and 3) for each prototype expression $(id, (base, add, remove)) \in KB$, each of the values of the simple change expressions in add are also in $dom(KB)$.*

Definition 5 **(R).** *Let KB be a prototype knowledge base and $id \in ID$. Then, the resolve function R is defined as: $R(KB, id) =$ the prototype expression in KB which has prototype expression ID equal to id. .*

Prototype Semantics

Definition 6 (Prototype-Structure). *Let SID be a set of identifiers. A tuple $pv = (p, \{v_1, \dots, v_n\})$ with $p, v_i \in SID$ is called a Value-Space for the ID-Space SID. A tuple $o = (id, \{pv_1, \dots, pv_m\})$ with $id \in SID$ and Value-Spaces $pv_i, 1 \leq i \leq m$ for the ID-Space SID is called a Prototype for the ID-Space SID. A Prototype-Structure $O = (SID, OB, I)$ for a Prototype Language PL consists of an ID-Space SID, a Prototype-Space OB consisting of all Prototypes for the ID-Space SID and an interpretation function I, which maps IDs from PL to elements of SID.*

Definition 7 (Herbrand-Interpretation).
Let $O = (SID, OB, I_h)$ be a Prototype-Structure for the prototype language $PL = (\mathtt{P}_\emptyset, ID, PROTO)$. I_h is called a Herbrand-Interpretation *if I_h maps every element of ID to exactly one distinct element of SID.*

As per the usual convention used for Herbrand-Interpretations, we assume that ID and SID are identical.

Next, we define the meaning of the constituents of a prototype. We start with the interpretation functions I_s and I_c which give the semantic meaning of the syntax symbols related to change expressions. These functions (and some of the following ones) are parametrized (one might say contextualized) by the knowledge base. This is needed to link the prototypes together.

Definition 8 (I_s). *Interpretation for the values of a simple change expression Let KB be a prototype knowledge base and v the values of a simple change expression. Then, the interpretation for the values of the simple change expression $I_s(KB, v)$ is a subset of SID defined as follows:*

$$\begin{array}{c} SID, \text{ if } v = * \\ \{I_h(r_1), I_h(r_2), \dots, I_h(r_n)\}, \text{ if } v = \{r_1, \dots, r_n\} \end{array}$$

Definition 9 (I_c). *Interpretation of a change expression. Let KB be a prototype knowledge base and a function $ce = \{(p_1, vs_1), (p_2, vs_2), \dots\}$ be a change expression with $p_1, p_2, \dots \in ID$ and the vs_i be values of the simple change expressions. Let $W = ID \setminus \{p_1, p_2, \dots\}$. Then, the interpretation of the change expression $I_c(KB, ce)$ is a function defined as follows (We will refer to this interpretation as a change set, note that this set defines a function):*

$$\{(I_h(p_1), I_s(KB, vs_1)), (I_h(p_2), I_s(KB, vs_2)), \dots\} \cup \bigcup_{w \in W} \{(I_h(w), \emptyset)\}$$

Next, we define J which defines what it means for a prototype to have a property.

Definition 10 (J). *The value for a property of a prototype. Let KB be a prototype knowledge base and $id, p \in ID$. Let $R(KB, id) = (id, (b, r, a))$ (the* *resolve function applied to id). Then the value for the property p of the prototype id, i.e., $J(KB, id, p)$ is:*

$$\begin{array}{c} I_c(KB, a)(I_h(p)), \ \textit{if } b = \mathtt{P}_\emptyset \\ (J(KB, b, p) \setminus I_c(KB, r)(I_h(p))) \cup I_c(KB, a)(I_h(p)), \ \textit{otherwise} \end{array}$$

Informally, this function maps a prototype and a property to *1*) the set of values defined for this property in the base of the prototype *2*) minus what is in the remove set *3*) plus what is in the add set.

As an example, let us try to find out what the value for the creator of the Ghent Altarpiece described in the example of the previous subsection would evaluate to assuming that these prototypes were part of a Prototype Knowledge Base KB. For brevity we will write `example:Ghent_Altarpiece` as GA, `example:Arnolfini_Portrait` as AP, `dc:creator` as creator, `example:Jan_Van_Eyck` as JVE, and `example:Hubert_Van_Eyck` as HVE.

Concretely, we have to evaluate $J(KB, GA, creator) = (J(KB, AP, creator) \setminus I_c(KB, \emptyset)(creator)) \cup I_c(KB, add)(creator)$ where add is the add change set of the GA prototype expression. First we compute the recursive part, $J(KB, AP, creator) = I_c(KB, add_{ap})(creator) = \{(creator, \{JVE\}), \dots\}(creator) = \{JVE\}$. Where add_{ap} is the add change set of the AP prototype expression. The second part (what is removed) becomes $I_c(KB, \emptyset)(creator) = \emptyset$. The final part (what this prototype is adding) becomes $I_c(KB, add)(creator) = \{(creator, \{HVE\}), \dots\}(creator) = \{HVE\}$. Hence, the original expression becomes $(\{JVE\} \setminus \emptyset) \cup \{HVE\} = \{JVE, HVE\}$ as expected.

Definition 11 (FP). *The interpretation of a prototype expression is also called its fixpoint. Let $pe = (id, (base, add, remove)) \in KB$ be a prototype expression. Then the interpretation of the prototype expression in context of the prototype knowledge base KB is defined as $FP(KB, pe) = (I_h(id), \{(I_h(p), J(KB, id, p)) | p \in ID, J(KB, id, p)) \neq \emptyset\})$, which is a Prototype.*

Definition 12 (I_{KB}:Interpretation of Knowledge Base). *Let $O = (SID, OB, I_h)$ be a Prototype-Structure for the Prototype Language $PL = (\mathtt{P}_\emptyset, ID, PROTO)$ with I_h being a* Herbrand-Interpretation*. Let KB be a Prototype-Knowledge Base. An interpretation I_{KB} for KB is a function that maps elements of KB to elements of OB as follows: $I_{KB}(KB, pe) = FP(KB, pe)$*

This concludes the definition of the syntactic structures and semantics of prototypes and prototype knowledge bases. For the semantics, we have adopted Herbrand-Interpretations, which are compatible with the way RDF is handled in SPARQL.

4 Inheritance

Our discussion of inheritance is based on the work by Lieberman [17], Cook et al. [5], de la Rocque Rodriguez [25], and Taivalsaari [26]. The combination of these works provides a wide overview of different forms of inheritance. Despite the fact that the focus of these works is on object oriented programming (OOP) we chose them because prototype-based systems are much more developed in OOP than in knowledge representation. Many of the OOP concepts and concerns also apply to how inheritance mechanisms can be applied in Knowledge Representation.

Broadly speaking, inheritance means that an entity receives properties from another one because of a relation between the two. Two types of inheritance are common: class-based and prototype-based. In class-based systems there is a distinction between objects and classes. An object is an instantiation of a class or, as some say, a class is a blueprint for an object. A new class can be inherited from another one and will typically inherit all properties and methods from the base or parent class. The values associated with these properties are typically defined in the context of the instances. Prototype-based systems on the other hand only have one type of things: prototypes. A new prototype can be made by *cloning* an existing prototype (i.e., the base). The freshly created object now inherits from the earlier defined one and the values are defined directly on the prototypes. As we argued above, we chose the prototype-based inheritance to allow for both horizontal and vertical sharing. In the next sections we will describe the consequences of the choice of property based inheritance.

4.1 Prototype Dynamics

There are essentially two ways to achieve prototype-based inheritance. The first one, *concatenation*, would copy all the content from the original object to the newly created one and apply the needed changes to the copy. The second one, *delegation*, keeps a reference to the original object and only stores the changes needed in the newly created object. We decided to follow the second option (for now) because it more closely resembles what one would expect from a system on the web. Instead of centralizing all information into one place, one links to information made available by others. This type of inheritance makes it possible to automatically make use of enhancements made in the base prototypes. Furthermore, the option of making a copy of the object one extends from is still available; we will discuss this further in section 4.3. Note that this is also a space-time trade-off. Copying will occupy more space, but make look-up faster while delegation will be slower, but only the parts which have been changed have to be stored. Another option is to get parts of both worlds by caching frequently used prototypes for a set amount of time. In this case, one may retrieve outdated values. In our technical report [3], we describe a possible approach towards caching using existing HTTP mechanisms.

When parts of a knowledge base are not in the control of the knowledge engineer who is adding new information, it might be tempting to recreate certain prototypes to make sure that the prototypes one is referring to do not change over time, rendering the newly added information invalid.

4.2 A Prototype is-not-a Class

In class-based object oriented languages, deriving a class A from a base class B usually implies that an instance of A can be used wherever an object of type B is expected. In other words, the objects instantiated from the classes A and B follow the Liskov substitution principle [18]. Since class-based object-orientation is currently most common in popular programming languages, one might be tempted to emulate classes in a prototype-based language. Imagine, for instance, that we want to create a prototype *employee* to represent an employee of a company. One might be tempted to give this *employee* a property *name*, with some default value since all employees will have a name in the end. However, this is not necessary, or even desired, when working with prototype-based systems. Instead, the *employee* should only have properties with values which all or most employees have in common, like for example the company they work for. Any more specific properties should instead be put on the employees themselves. Moreover, the fact that a prototype derives from the created *employee* does not have any implication beyond the inherited properties. Put another way, there is no *is-a* relation between a concrete employee and the *employee* prototype from which it was derived. This is also clearly visible from the fact that a derived prototype has the ability to remove properties from the base. Moreover, any other prototype with the properties needed to qualify for being an employee can be seen as an employee; independently from whether it derives from the *employee* prototype or not. Next, we will discuss what it means to be 'seen' as an employee.

4.3 Object boundaries

Applications usually need to work with data with predictable properties. For instance, the employees from the example in the previous section need to have a name, gender, birthday, department, and social security number in order for the application to work. Hence, there is a need to specify the properties a prototype needs to have in order to be used for a specific application. This idea is not new and has also been identified in other knowledge representation research. Named Graphs are often used for this purpose, but they don't capture shared ownership or inheritance. Further, resource shapes[8] and shape expressions [22] have the core idea of determining whether a given RDF graph complies with a specification. The main goal of these is checking some form of constraints, but they could as well be used to identify instances in a dataset.

This need has been identified in many places in OOP literature. An object oriented programming language which allows variables to contain any object which fulfills a given interface definition is said to have a structural type system. Recent examples of programming languages with such type system include OCaml and Go, but to our knowledge the first programming language to use it was Emerald [1] and later School [25]. In these languages, if objects have a given set of operations (according to what they called an *abstract type* in Emerald, *type*

[8] `https://www.w3.org/Submission/2014/SUBM-shapes-20140211/`

in School, or *interface* in Go), they would be treated a being an instance of, or assignable to, a variable of that type.

One of the arguments against structural type systems is that it might happen that an object has the properties (or methods) of the type by accident. We can envision this happening in OOP because the names of methods have little semantic meaning connected to them (does the write() method write something to the disk or to the printer?). However, in a Semantic Web setting, the property names are themselves IRIs and chosen carefully not to clash with existing names (a `http://xmlns.com/foaf/0.1/workplaceHomepage` will always be 'The workplaceHomepage of a person is a document that is the homepage of a organization that they work for.'[9]). In other words the property names in the system under consideration in this paper do in principle not suffer from this problem.

5 Future Work

Since most past work in the research community has been focused on class-based knowledge representation, there are still many areas unexplored related to prototype-based knowledge representation on the web.

5.1 Relation to RDF and OWL

In this paper, we are suggesting a knowledge sharing language based on prototypes. Future work will need to investigate how to layer the prototype language on top of RDF. While most of the conversion and layering should be straightforward (e.g., the IRI of a prototype expression would also be the IRI of the RDF resource), some challenges remain. For example, one would need to define a protocol working on RDF graphs in order to locate and interact with a prototype. However, we believe that these challenges can be overcome.

5.2 A Hint of Class?

In this paper, we presented prototypes as a possible alternative to class-based systems such as OWL for Knowledge representation on the Web - at least for the purpose of scalable Knowledge Sharing. However, both ways - prototypes and class-based representations, have different use cases and reasons to exist: OWL is focusing on enabling reasoning whereas prototypes are focusing on enabling Knowledge Sharing. Exploring the exact boundaries of their respective use cases still remains a topic for future work.

Another interesting future research path would be the discovery of 'hidden' classes in the knowledge base. A hidden class would be formed by a group of objects with similar characteristics. These classes would be automatically discovered, perhaps with techniques like Formal Concept Analysis (FCA) [29], by

[9] definition of `foaf:workplaceHomepage`from `http://xmlns.com/foaf/spec/`

collecting a large number of prototypes from the Web. Another approach to this would be to perform a hierarchical clustering of the prototypes with a scalable technique as proposed in [4]. After this clustering, it might be possible to extract a class hierarchy from the generated dendrogram.

5.3 Variations, Evaluations and Large Scale Benchmarks

The prototype system introduced in this paper is only an initial exploration. There are numerous variations possible by making different choices for the inheritance model (e.g. concatenation, multiple inheritance, etc.), the allowed values (intervals, literals, etc.), and solutions for resolving the values for non-local prototypes. These choices will have different implications for implementations and good evaluation metrics and large scale benchmarks should be designed to compare them. We presented initial work in this direction in a technical report [3] and publicly available software `https://github.com/miselico/knowledgebase` (LGPLv3). We benchmarked the system using several synthetic data sets and observed that the theoretical model presented offers the scalability needed for use in production environment in a typical distributed web architecture.

6 Conclusions

During the last decade, Knowledge Representation (KR) research has been dominated by W3C standards whose development was influenced by the state of the mind that researchers in the involved research communities had at the time of creation. Several choices which were made which have far reaching consequences on the way knowledge representation is done on the Web today.

In this paper we tried to take a step back and investigate another option for KR which, in our opinion, has properties more suitable to deliver on the goals of horizontal and vertical sharing. Concretely, we introduced a system in which everything is represented by what we call prototypes and the relations between them. Prototypes enable both vertical sharing by the inheritance mechanism and horizontal sharing by direct reference to any prototype. We provided a possible syntax and semantics for the Prototype system and performed experiments with an implementation. The experiments showed that the proposed system easily scales up to millions of prototypes. However, many question still remain to be answered. First and foremost, this kind of Knowledge Representation needs to get traction on the Web, which is a considerable challenge - but one we believe can be achieved based on early feedback we obtained. Furthermore, a larger deployment of this kind of system would need a clear mechanism for resolving non-local prototypes. We did some experiments in this direction in a technical report using existing web technologies like HTTP for this, but still there are many options to investigate. We would like to see what kind of options others come up with to introduce useful parts of class-based systems into the prototype world. Finally, we hinted towards finding 'hidden' classes in the Prototype system. This would not only be an academic exercise, but would be very useful to be able to compress knowledge base representations and reduce communication costs. We hope that this paper contributes constructively to the field of Knowledge Representation on the Web and that in the future, more researchers will explore different directions to see how far we can reach.

Acknowledgments

Stefan Decker would like to thank Pat Hayes, Eric Neumann, and Hong-Gee Kim for discussions about Prototypes and Knowledge Representation in general.

Michael Cochez performed parts of this research at the Industrial Ontologies Group of the University of Jyväskylä, Finland, and at the Insight Centre for Data Analytics in Galway, Ireland.

References

1. Black, A.P., Hutchinson, N.C., Jul, E., Levy, H.M.: The development of the emerald programming language. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. pp. 11–1–11–51. HOPL III, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1238844.1238855
2. Brachman, R.J.: A structural paradigm for representing knowledge. Tech. Rep. BBN Report 3605, Bolt, Beraneck and Newman, Inc., Cambridge, MA (1978)
3. Cochez, M., Decker, S., Prud'hommeaux, E.G.: Knowledge representation on the web revisited: Tools for prototype based ontologies. In: arXiv (2016), https://arxiv.org/abs/1607.04809, arXiv:1607.04809 [cs.AI]
4. Cochez, M., Mou, H.: Twister tries: Approximate hierarchical agglomerative clustering for average distance in linear time. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 505–517. ACM (2015)
5. Cook, W.R., Hill, W., Canning, P.S.: Inheritance is not subtyping. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 125–135. POPL '90, ACM, New York, NY, USA (1990), http://doi.acm.org/10.1145/96709.96721
6. Decker, S., Fensel, D., van Harmelen, F., Horrocks, I., Melnik, S., Klein, M., Broekstra, J.: Knowledge representation on the web. In: Proc. of the 2000 Description Logic Workshop (DL 2000). CEUR (http://ceur-ws.org/), vol. 33, pp. 89–98 (2000)
7. Duerst, M., Suignard, M.: Internationalized resource identifiers (iris). RFC 3987, RFC Editor (January 2005), http://www.rfc-editor.org/rfc/rfc3987.txt, http://www.rfc-editor.org/rfc/rfc3987.txt
8. European Computer Manufacturers Association and others: Standard ecma-262 ecmascript 2015 language specification. June (2015)
9. Gabriel, R.: The rise of "worse is better". Lisp: Good News, Bad News, How to Win Big 2, 5 (1991)
10. Hayes, P.J.: The logic of frames. In: Metzing, D. (ed.) Frame Conceptions and Text Understanding, pp. 46–61. Walter de Gruyter and Co., Berlin, Germany (1979)
11. Horrocks, I., Sattler, U., Tobies, S.: Practical reasoning for expressive description logics. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99). pp. 161–180. No. 1705 in Lecture Notes in Artificial Intelligence, Springer (1999)

12. Horrocks, I.: Using an expressive description logic: FaCT or fiction? In: Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98). pp. 636–647 (1998)
13. 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), 7–26 (2003)
14. Israel, D.J., Brachman, R.J.: Some remarks on the semantics of representation languages. In: Brodie, M.L., Mylopoulos, J., Schmidt, J.W. (eds.) On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages, pp. 119–142. Springer, New York (1984)
15. Karp, P.D.: The design space of frame knowledge representation systems. Tech. rep., SRI International Artificial Intelligence (1993)
16. Levesque, H.J., Brachman, R.J.: A fundamental tradeoff in knowledge representation and reasoning (revised version). In: Brachman, R.J., Levesque, H.J. (eds.) Readings in Knowledge Representation, pp. 41–70. Kaufmann, Los Altos, CA (1985)
17. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications. pp. 214–223. OOPLSA '86, ACM, New York, NY, USA (1986), `http://doi.acm.org/10.1145/28697.28718`
18. Liskov, B.: Keynote address - data abstraction and hierarchy. SIGPLAN Not. 23(5), 17–34 (Jan 1987), `http://doi.acm.org/10.1145/62139.62141`
19. Minsky, M.: A framework for representing knowledge. Tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA (1974)
20. Minsky, M.: A framework for representing knowledge. In: Haugeland, J. (ed.) Mind Design: Philosophy, Psychology, Artificial Intelligence, pp. 95–128. MIT Press, Cambridge, MA (1981)
21. Mitchell, T.M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M., Schlimmer, J.C.: Theo: A framework for self-improving systems. Architectures for intelligence pp. 323–356 (1991)
22. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.: Shape expressions: an rdf validation and transformation language. In: Proceedings of the 10th International Conference on Semantic Systems. pp. 32–40. ACM (2014)
23. Quillian, M.R.: Semantic memory. Tech. rep., DTIC Document (1966)
24. Rector, A.L.: Defaults, context, and knowledge: Alternatives for owl-indexed knowledge bases. In: Altman, R.B., Dunker, A.K., Hunter, L., Jung, T.A., Klein, T.E. (eds.) Pacific Symposium on Biocomputing. pp. 226–237. World Scientific (2004), `http://dblp.uni-trier.de/db/conf/psb/psb2004.html#Rector04`
25. Rodriguez, N.d.l.R., Ierusalimschy, R., Rangel, J.L.: Types in school. SIGPLAN Not. 28(8), 81–89 (Aug 1993), `http://doi.acm.org/10.1145/163114.163125`
26. Taivalsaari, A.: On the notion of inheritance. ACM Computing Surveys (CSUR) 28(3), 438–479 (1996)
27. Ungar, D., Smith, R.B.: Self. In: Ryder, B.G., Hailpern, B. (eds.) Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007. pp. 1–50. ACM (2007), `http://doi.acm.org/10.1145/1238844.1238853`
28. Victor, T., Dalzell, T.: The concise new Partridge dictionary of slang and unconventional English. Routledge (2007)
29. Wille, R.: Formal concept analysis as mathematical theory of concepts and concept hierarchies. In: Formal Concept Analysis, pp. 1–33. Springer (2005)