RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN
Information Systems
Prof. Dr. Stefan Decker
Prof. Dr. Matthias Jarke

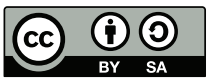
Master Thesis

The Pragmatics and Logic of Knowledge Representation with Prototypes

Martha Hannah Gesche Gierse

September 29, 2017

Advisor: Dr. Michael Cochez
Supervisors: Prof. Dr. Stefan Decker
Prof. Dr. Matthias Jarke



Contents

1 Introduction

A central purpose of the internet is sharing information. That information is usually stored and presented in human-readable form. However, it is often not (easily) processable for a machine. If the information became more machine-readable, it would be easier to write applications that are able to utilize the vast knowledge collected on the web. Knowledge representation mechanisms are used to represent information in machine-processable ways. This movement is called *Semantic Web* [1, 2].

A basic representation used is the Resource Description Framework (RDF) [3]. The RDF describes information in strict subject-predicate-object triples. In contrast to natural language these simplified statements are much better processable for machines.

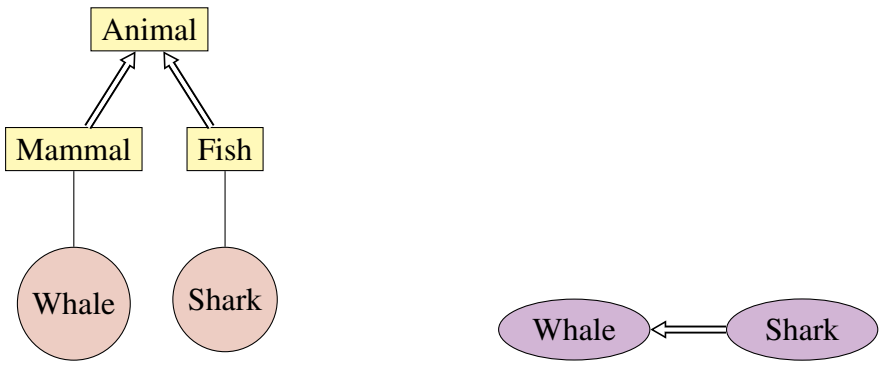
The Web Ontology Language (OWL) [4, 5] is another standard for the Semantic Web. It uses ontologies to model knowledge hierarchically. With this approach, data can be structured in classes and relations can be described. OWL is based on logic and therefore also allows to reason about the represented knowledge.

In the Semantic Web movement, data is shared between different entities. RDF and OWL files are provided online and can be read, downloaded and modified or referenced by other files.

Cochez et al. [6] discuss a distinction between two different modes of sharing: First, *vertical sharing*, where an authority shares a graph or ontology and others use it. The second mode of sharing is *horizontal sharing*, where pieces of data are shared and reused by peers. The authors of [6] describe that currently, sharing on the Semantic Web is rather done vertically than horizontally, even though the spirit of the web is peer-based.

To ease vertical sharing, they introduce a prototype-based approach to knowledge representation on the web. This enables more types of reuse than class-based systems, since with prototype-based inheritance new prototypes can be derived from any kind of existing ones. In a class-based system there is a distinction between classes and instances, and only classes can be used for inheritance.

For example, consider the hierarchy of animals given in Figure 1. Mammals and fish are animals. A whale is a mammal, while a shark is a fish. Figure 1a represents these facts with a class-based approach (as for example used by OWL). The yellow rectangles represent classes and red circles instances. The double arrow shows inheritance and the line



(a) Class-based inheritance and instances.

(b) Prototypical inheritance.

Figure 1: Hierarchy of animals illustrating different approaches to knowledge representation.

indicates that an instance has a class as its type. The idea of the hierarchical representation is that all animals share some properties that mammals and fish inherit. In this representation, whales and sharks are represented as instances and are thereby concrete instances of the classes. Depending on the use-case they could instead be modeled as classes.

In contrast, Figure 1b shows a prototypical representation. The double arrow indicates prototypical inheritance. In contrast to classes and instances, *prototypes* are represented as purple ellipses. Whales and Sharks may have a lot in common because they both live in the water. So, starting from the properties of the whale it might be easy to remove and add a few properties such that they instead represent a shark. This approach to inheritance, where every object may be used to inherit from, is called *prototypical* [7].

In class-based systems the class hierarchy provides useful means for modeling domains and describing knowledge. Ontologies like OWL can also reason about represented knowledge using class specifications. This can be useful for applications using the Semantic Web. For example, terms similar to a search query could be exchanged using ontologies. Furthermore, it enables an agent to reason about class membership.

One possibility to model knowledge is by constraints. For example, the animal classes can be characterized by the following statements: “All mammals feed their babies with milk” and “all fish can breathe underwater”. Notice the constraining factor *all* on the property of feeding and breathing. If a whale was specified as an animal that feeds its babies with milk and cannot breathe underwater, then it could be reasoned that a whale is not a fish. However, with this information it is not possible to conclude that a whale is a mammal since there might be other animals that also feed their babies with milk and are not mammals.

To use data, an application needs methods to ensure that the data is consistent and fits a schema. This can be provided by *integrity constraints*. For example, an application handling animal data may require that each animal has a property that describes in what environment the animal can breathe. One could also define that fish need to be able to breathe underwater. Many of the constraints used to check integrity are similar to the constraints used in knowledge representation. With integrity constraints the focus lies on determining whether given data fits a description. In contrast, knowledge representation can reason about what further information may or may not be possible given the already known facts.

For the Semantic Web it is useful to combine knowledge representation and reasoning capabilities with the possibility to define integrity constraints that ensure that data satisfies defined requirements. In addition, the pursued approach should enable vertical as well as horizontal sharing.

As stated in [6], vertical sharing between peers is not well established with RDF and OWL. To ease vertical sharing they introduce a prototypical approach. This allows to represent data similar to RDF. However, the objects are prototypes and may be reused arbitrarily. The prototypical inheritance proposed even allows to remove properties during inheritance.

In contrast to class-based inheritance, prototypical inheritance does not model the same clear hierarchies. Since the properties of the parent might be changed arbitrarily, parents and children do not necessarily have anything in common. With the loose connection that prototypical inheritance forms the hierarchical representation is lost.

In addition, the approach described in [6] does not feature knowledge representation paradigms beyond the definition of properties. In contrast to OWL it lacks the expressive-

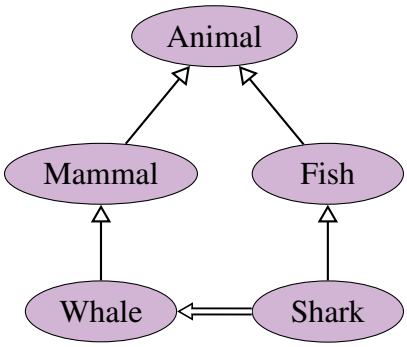


Figure 2: Hierarchy of animals represented through prototypical inheritance with specialization.

ness to define constraints and reason about them. While there are approaches to define integrity constraints for RDF and OWL (e.g., [8, 9, 9]) there are currently none for the prototypical representation in [6].

Thus, the goal of this thesis is to explore possible knowledge representation mechanisms for prototypes. These will then be used to introduce basic reasoning and integrity checking on prototypes. Constraints will be introduced as knowledge representation mechanism since they are common to knowledge representation and integrity checking. These are integrated with the existing prototypical approach. To model hierarchical relationships between prototypes, a new relation called *specialization* will be introduced. It will be based on constraint satisfaction. In contrast to inheritance the relation will be *observed* rather than defined. This means that whenever the constraints of a prototype are satisfied by another prototype, that prototype will be a specialization of the former prototype. It is not necessary to define that they are in a specialization relation. The programming language *Go*[1] has a similar, observant behavior regarding interfaces. A type implements an interface whenever the methods of the interface are present. It is not necessary to explicitly define that a type has an interface.

The animal example from Figure 1b is extended by the specialization relation in Figure 2. In addition to inheritance for reuse, specialization describes the animal hierarchy. The specialization relation is indicated by arrows with a single line and an empty triangle as head ($\rightarrow$). The idea behind specialization is to formulate prototypes that describe requirements for a class. For example, a constraint to describe mammals would be: All mammals feed their babies with milk. Then it can be checked which other prototypes satisfy that requirement. Whales feed their babies with milk while sharks do not. Thus, a whale is a specialization of a mammal whereas a shark is not. Similarly, common properties of fish and mammals can be defined and making it possible to check which prototypes have these common properties. We will choose a part of the constraints present in existing systems to explore the general possibilities for knowledge representation primitives with prototypes.

In the following, we will first explore the background for the thesis in Section 2. Here, we will look at the Semantic Web movement and the differences between class-based and prototypical inheritance in detail. Frames, description logics and integrity constraint systems will be introduced. In later parts of the present work these concepts will be drawn from to decide what requirements are useful for a specialization relation. Section 3 dis-

[1] `https://golang.org/`

cusses which constraints are selected for this thesis, as well as how to represent constraints in the prototype system. The specialization relation is explained and described formally in Section 4. Furthermore, properties of the relation are presented. Section 5 presents the results obtained from an exploratory implementation of the proposed system. This thesis is a first exploration of possible definitions of a specialization relation for prototypes. This opens up many further possibilities, which are discussed in Section 6. Finally, in Section 7 the obtained results are reviewed.

2 Background

Figure 3 shows an overview of related technology and background. This thesis uses three main components:

1. Inheritance as a means to model and reuse objects.

2. Frames and Description Logics as examples of knowledge representation systems and a guide to what kind of knowledge representation paradigms are useful.

3. Integrity constraints as a paradigm for describing data formats in such a way that applications can use them.

Since the goal is to apply the knowledge representation to the Semantic Web, the Semantic Web plays a major role. This can also be seen in the related background. The parts in Figure 3 highlighted in green are closely related to the Semantic Web. An overview of this movement is given first in Section 2.1. The Figure also shows the names of different technologies. These will be explored in more detail in the respective sections.

The differences between prototypical and class-based inheritance are described in Section 2.2. In particular, we will discuss examples of prototypical inheritance and of course the prototypical system for the Semantic Web which this thesis extends, shown in green with a green circle around it.

Section 2.3 then explores Frames and Description Logics. Frames are a knowledge representation system that does not have formal semantics. Description Logics are related to Frames but they have clear semantics. Especially the Web Ontology Language (OWL) is a widely used system that builds on Description Logics. Therefore, we will investigate what kind of knowledge representation paradigms are used there.

Integrity Constraints are means to describe the format of valid data. They are discussed in Section 2.4. In relational databases integrity constraints are used to ensure that updates and new data do not violate defined rules. Later on web technologies such as the Extensible Markup Language (XML) and the Resource Description Framework (RDF) also introduced means to define what schema data should have. This part of the background is related to the goal of describing data formats and finding matching data such that applications can use the data. Knowledge representation paradigms could be used to describe how valid data has to look like. So a selection of systems providing Integrity Constraints are explored to see what kind of constraints they use and how these might be applied to a prototypical approach.

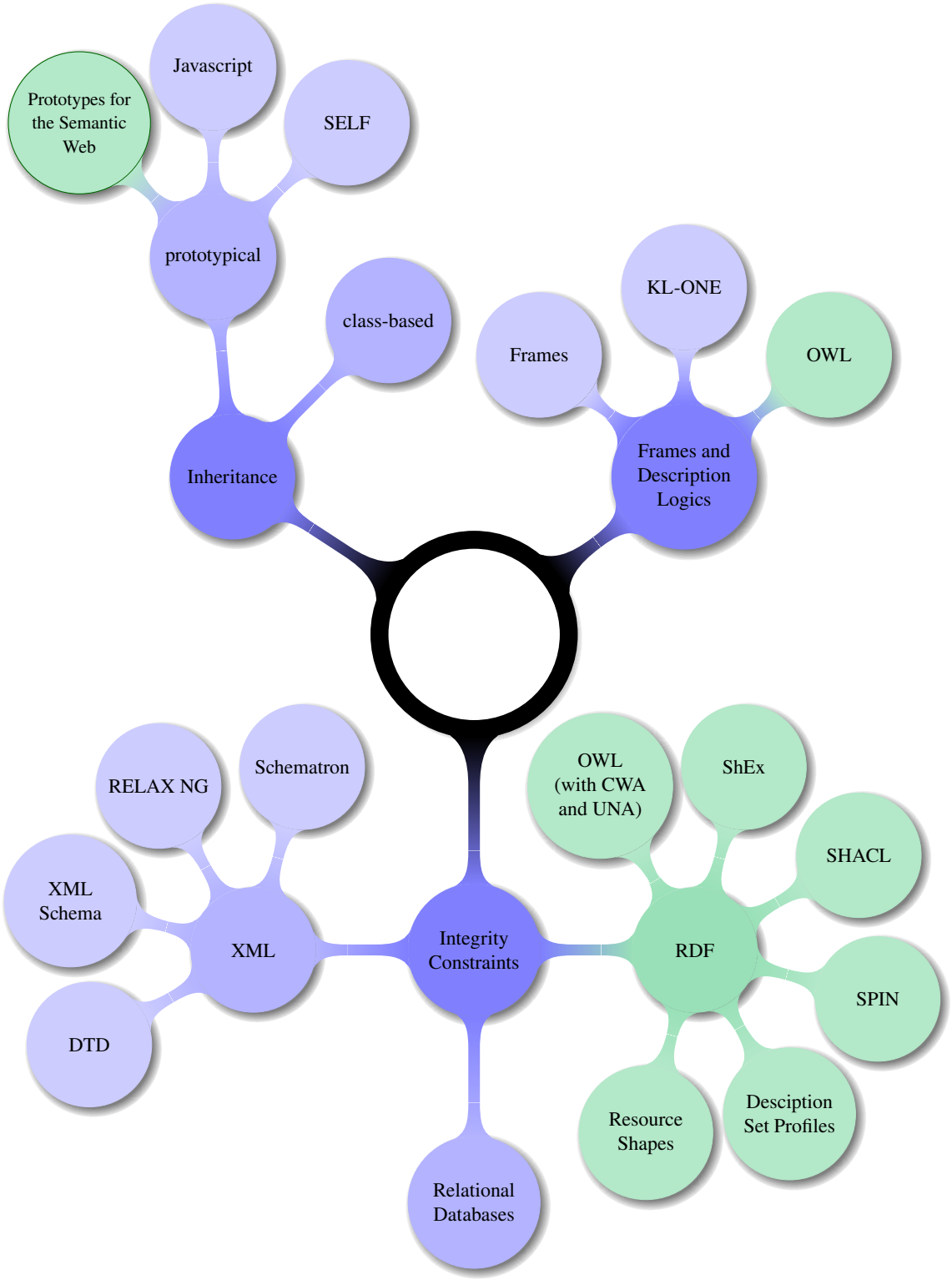


Figure 3: Overview of relevant technology. The green nodes are related to the Semantic Web. The green node with a green circle around it represents the approach that is extended in this thesis.

2.1 Semantic Web

The internet consists of interlinked pages that are mostly accessed through search engines. The idea of the Semantic Web is to enhance information from the internet in a machine-readable and processable format [2]. The World Wide Web Consortium (W3C) makes recommendations for web standards. It also offers recommendations regarding the Semantic Web. The basis of the Semantic Web recommendation is the Resource Description Framework (RDF) [3]. It can be used to describe domains in terms of subject-predicate-object triples. For example, the fact that Tim Berners-Lee was born in London can be expresses as the triple "`TimBerners-Lee isBornIn London`". The parts of the triples are Internationalized Resource Identifiers (IRI). In contrast to the example usually URLs like `http://www.example.org/isBornIn` are used.

An extension of RDF is RDF Schema (RDFS) [10]. It provides basic means to model semantic meaning to RDF triples. With RDFS classes can be defined to structure resources together. In addition IRIs can be defined as properties and the domain and codomain can be restricted to be from some class. For example, the property `isBornIn` can be defined to take as subject resources from the class `Person` and as object either a `City` or `Country`.

RDF and RDFS data sources can be queried by the SPARQL Query Language for RDF (SPARQL) [11]. For example, it could be used to list all persons that are born in London and are computer scientists. The syntax is similar to SQL.

More complicated knowledge can be expressed using the Web Ontology Language (OWL) [12]. Ontologies are hierarchies of concepts. For example, a ontology could express that a mammal is an animal. OWL will be discussed in more detail in Section 2.3.

RDF as well as OWL data can be interlinked. A public ontology maintained by one party can be referenced and extended by another, much like links on webpages. OWL has a strict is-a relation for their classes. Because of this, interlinking or extending ontologies can be difficult. It could be that an ontology is nearly what is needed for an application, but some things need to be changed. Since the is-a relation is strict, the data can be copied over and changed but linking the data and removing parts of the defined concepts is not possible.

To ease sharing and extending a prototypical approach to the Semantic Web has been presented in [6]. Instead of a strict is-a relation prototypical inheritance is used. The next section describes the difference between these kinds of inheritance in more detail.

2.2 Inheritance

In this section different kinds of inheritance are discussed. Inheritance, prototypes and classes are means to model a domain. How (parts of) the world can be described systematically has been explored in many fields.

Philosophers have long since discussed what categories exist in the world and how they can be classified. In philosophy the term *ontology* describes this field of study [13]. One example of things for which a classification is explored are different concepts. The classical theory states that a concept consists of a definition built on other concepts. For example, the concept *bachelor* is said to be *unmarried* and *male* [14]. The idea of classification has influenced object oriented programming languages as well as description logics.

The idea of prototypical inheritance originates from linguistics. In linguistics the prototype theory states that humans form categories of things and that some examples of that

category are more typical for the class while others cannot be clearly classified. For example, it was found by Eleanor Rosch that a typical prototype for the category *furniture* is for example *chair*, while other things, like a *lamp* or a *telephone* were not associated as strongly with the category [15]. With the emerging of the prototype theory in the 1970s an alternative to the strict definitional classification was introduced. Instead of a clear definition, something belongs to a concept, if it has sufficient many properties that are associated with the concept [14].

In computer science it is important to be able to model parts of the world that are relevant for an application. According to [16] inheritance is used to base new objects on existing ones. The mechanism is *incremental*, i.e., the new object (child) inherits properties from the original object (parent) and only new, modified or removed properties need to be defined. Not all kinds of inheritance allow all these steps. For example, the removal of properties is not necessarily allowed in all systems. Inheritance relations are *transitive* [16]. Thus, a child will not only inherit properties from its parent but also from their parents and all other ancestors. In programming languages class-based inheritance is widely used, especially in object oriented programming languages. Class-based inheritance distinguishes between two kinds of objects: classes and instances. Only classes can inherit from another while instances are concrete objects of a class. The authors of [16] regard removing properties from the parent as a variation of inheritance which is an exception rather than the normal case. Some programming languages also use prototypical inheritance mechanisms. In contrast to class-based inheritance, prototypical approaches have only one kind of object, namely a prototype [7]. It is a concrete object but may also serve as blueprint for inheritance. In the following these two mechanisms will be discussed in more detail.

2.2.1 Class-based Inheritance

Class-based inheritance is common in object oriented programming languages. Generally, these approaches distinguish between classes and instances. A class is a definition of an object, while an instance is a concrete thing that has as a class as type. For example consider the simple class diagram shown in Figure 4. The class `Computer` has two variables, namely a processor name and a list of video outputs. It has functions to boot and to shutdown the computer. The `Laptop` class inherits from it, thus it will have all variables and functions from the `Computer` class but also the variable `monitorSize`.

An instance of the `Laptop` class is shown in Figure 5. It has concrete values for the different variables of the `Laptop` class and its parent.

Class-based inheritance is strict. A value or method that a parent possesses is also inherited by every subclass. It can be overwritten but it cannot be erased. Furthermore, inheritance is only formed between classes. Instances cannot be used to inherit from. An instance always needs to belong to a class and it cannot have variables and functions that are not defined in its class.

In the following, a short overview of criticism to class-based inheritance as presented in [7] is described: One objection is that it is not always possible to model the reality based on strict class inheritance. However, it may be sufficient for a given application. Furthermore, for most applications classes can be modeled in many ways with different benefits and drawbacks. Often there is no optimal solution because of trade-offs like reusability and space-efficiency. Modeling class hierarchies also faces the problem that usually classes in

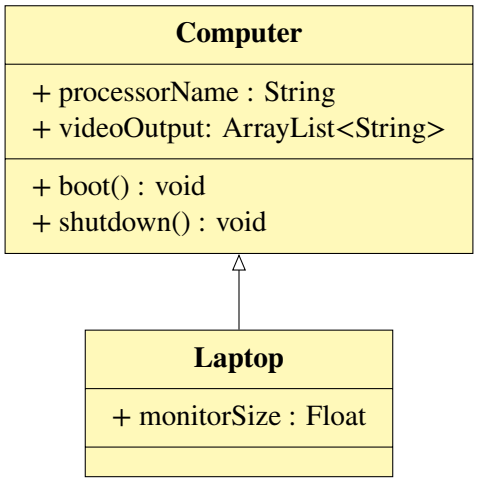


Figure 4: Class diagram of computers

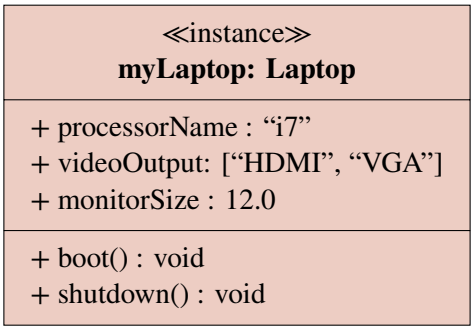


Figure 5: Example instance of the `Laptop` class

the middle of a hierarchy are the most important ones. For example, in a hierarchy of computers there might be more objects that inherit from the `Computer` class. There might be classes for servers, desktop PCs, smartphones and another for embedded computers. And they each in turn could have subclasses of specific models of servers, laptops, smartphones and so on. For many use cases the middle layer of different kind of computers is of most interest. While the computer class might be too general, the others will be to specific. When trying to model a domain it is often clear what classes are needed in the middle. However, the development of a hierarchy cannot easily start with concepts in the middle. Thus, the process of developing class hierarchies needs to be iterative.

2.2.2 Prototypical Inheritance

Prototypical inheritance works different than class-based inheritance. Prototype-based inheritance usually forms an inheritance relation between instances. An instance is seen as a prototype, where others can derive from. However, the relation does not have to be strict. The inheriting prototype may also modify or even remove functionality. Thus, prototypical systems do not have a clear class hierarchy.

When using prototype-based inheritance there are many decisions that have to be made. An overview of questions and possible answers is presented in [17]. The authors focus on programming languages. They discuss what primitive mechanisms, like inheritance, creation, extension or delegation are possible and how they are done. We will only look at the

criteria defined that are relevant to knowledge representation systems. Many criteria are relevant to a programming language and not directly transferable to a knowledge representation system. For example, whether objects or the parent link of an object can be modified dynamically is not of importance in static knowledge representation system. However, the following criteria introduced in [17] are relevant to both programming language and static knowledge representation systems:

1. Is it possible to create elements from nothing? That is, need every object be cloned from an existing one or have a prototypical inheritance relation to another?

2. Is multi-inheritance possible? That is, can one prototype have more than one prototype as its parent?

As examples for systems with prototypical inheritance we will look at two programming languages and the proposed prototypical knowledge representation system for the web upon which this thesis builds.

SELF SELF [18] was designed in the 1980s. It is one of the earliest programming languages that use prototypical inheritance. Another specialty about the language is that it does not allow object variables to be accessed directly. To read or change a variable the object can send a message to itself. The language is named "SELF" because of the many messages that are passed to "self".

With SELF objects can be created from nothing but they may also be cloned. Multi-inheritance is possible. SELF allows to modify properties of an object dynamically and objects can even change their parent dynamically. If an object lacks the functionality to handle a message it delegates the message to its parents. In this fashion variables and functions are resolved upwards in the inheritance chain.

Javascript Another programming languages that uses prototypes is Javascript [19]. The language was originally intended to be used in web browsers where it is very widely used. In addition, also server side applications can be written in Javascript. Beside prototypical inheritance it also features imperative and functional programming paradigms.

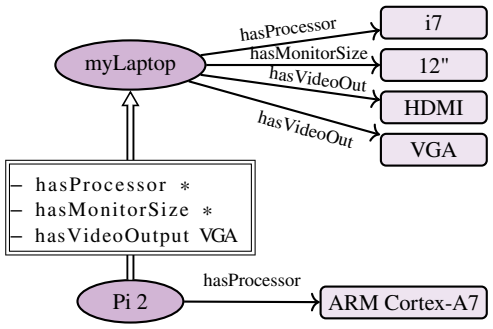
In Javascript each object has a link to another object that is its prototype. The inheritance chain ends when an object has `null` as its prototype. Thus, formally objects are created from `null` and thus not from nothing. The language does not feature multi-inheritance, but there are workarounds. As in SELF, it is possible modify properties dynamically and the parent of the inheritance relation can be changed dynamically. If a property is not found by the object itself, it is delegated to the parent of the object until it is found or the end of the inheritance chain is reached.

Prototypes in Knowledge Representation There are different approaches that use prototypes for knowledge representation. As examples we will discuss two of them. An early example for a prototypical approach in knowledge representation is the framework *Theo* [20]. Theo is based on frames and used to represent beliefs, meta-beliefs and problems to be solved. It uses only one kind of frame to represent all these different parts. The authors argue that it is not always clear what is an instance and what is a class. Thus, they do not make the distinction. The Theo framework stores available methods for interference

as beliefs. One of the possible methods is inheritance, which is applied to search for solutions in the generalizations of a frame. Another approach to knowledge representation with prototypes has been proposed recently [6, 21]. This approach focuses on sharing data in the context of the Semantic Web. Since this thesis will build upon the system presented by Cochez et al. a more detailed overview of the prototype system is given. As the formal syntax and semantics will be needed later they are presented in Section 2.2.4. For now, a general overview of the system is introduced.

In the approach described in [6] a prototype has a unique identifier and a base from which it inherits prototypically. This means that properties and their values can be removed and added. The possibility to change values arbitrarily from the parent is of major interest for the Semantic Web. Exchanging and reusing data can be done in many different ways because of the gained freedom. Since arbitrary changes can be made, data can be freely adapted by inheritance to suite different applications.

With the presented approach a prototype always inherits from another prototype. If a new prototype should not be copied from an existing one, it may inherit from a special empty prototype from which all prototypes recursively inherit. However, formally this means that there is no element creation from nothing. Multi-inheritance is not supported by the approach.



(a) Graphical Representation

```
myLaptop
base proto:P_0
add hasProcessor i7
add hasMonitorSize 12
add hasVideoOut HDMI
add hasVideoOut VGA

ex:Pi_2
base myLaptop
rem hasProcessor *
rem hasMonitorSize *
rem hasVideoOutput VGA
add hasProcessor ARM_Cortex-A7
```

(b) Concrete Syntax

Figure 6: Deriving a Raspberry Pi from a laptop by removing and adding properties.

Figure 6 shows an example of prototypical inheritance. The computer `MyLaptop` is defined with multiple properties. Figure 6a shows a graphical representation. In the figures throughout this thesis prototypes are always represented as purple structures. Thus, the ellipses are prototypes and the single arrows describe property relations. The arrows are labeled with the property name. The nodes pointed to by the arrows represent the values of the property relation. These are the identifiers of prototypes and are thus colored light purple. For example the prototype `myLaptop` has the property `hasVideoOutput` which has `HDMI` as well as `VGA` as values. The values themselves are prototypes but are only referenced to, thus the different kind of node. A Raspberry Pi inherits from the laptop. The inheritance relations is shown by the double arrow and the white box. In the box each property that is removed during inheritance is named. The value of the processor is completely removed (indicated by "*"). Since a Raspberry Pi has no own monitor the respective property is also removed. In addition, one video output is removed, namely,

VGA. In the graphical representation the added values are represented as properties of the Raspberry Pi node.

The concrete syntax shown in Figure 6b starts with the name of the prototype described. Then, the base defines the inheritance relation by naming the parent of the prototype. All prototypes eventually inherit from P_0, a special prototype that is empty. Afterwards the properties removed (indicated by rem) and added (indicated by add) are listed. The Raspberry Pi will have all properties of its parent that were not removed. Thus, it inherits the video output HDMI. Notice, how this approach only uses instances in contrast to the class-based computer hierarchy presented in Section 2.2.1. There is no hierarchy here just objects that can be inherited from and thereby may be arbitrarily modified.

2.2.3 Discussion

Inheritance and classes serve multiple purposes during the design of an application [22]:

> "As part of the high-level design phase, inheritance serves as a means of modeling generalization/specialization relationships. [..] In the low-level design phase, inheritance supports the reuse of an existing class as the basis for the definition of a new class. "

The mixture of specialization and reuse is not necessarily beneficial and the approach presented in this thesis aims to untangle these two parts. Reuse is optimally supported by prototypical inheritance, since everything can be reused and modified in completely unrestricted ways. As a trade-off prototypical inheritance does not have the same expressiveness in modeling generalization/specialization relations. Since prototypes can vary so much from their parent, inheritance does not necessarily convey any information about the classification of the objects that are in inheritance relations to another. With class-based inheritance on the other hand the specialization relation from parent to child is clear.

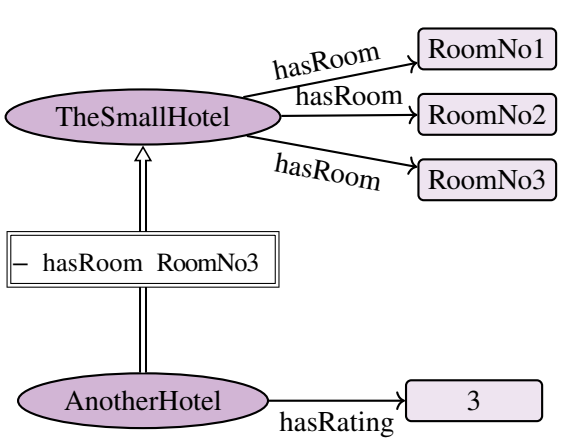
In order to keep reusability advantages of the prototypical system and gain the ability to model specialization, we will propose an additional specialization relation that deals only with this aspect of modeling.

2.2.4 Syntax and Semantics of Prototypes for the Semantic Web

The work in this thesis is an extension of the prototype system described in [6] and has been briefly summarized in Section 2.2.2. The general idea of the prototypical system is that each prototype has an identifier, a base from which it inherits and two sets of properties and values that should be removed, respectively, added. For future reference and extension the syntax and semantics of that system are introduced in the following.

Prototype Syntax Prototypes consist of an identifier, a base (the prototype to inherit from) and a set of add and remove instructions for the inheritance. These instructions will be referred to as simple change expressions that assign a property with a set of values.

Consider the example given in Figure 7. TheSmallHotel has the property hasRoom with three different values. The prototype AnotherHotel inherits from it and removes the value RoomNo3. Furthermore, the property hasRating with value 3 is added. The graph representation shown in Figure 7a is the original graphical representation used in [6]. The prototypes are represented by elliptical nodes. The added values are nodes connected to



(a) Graph Representation

```
TheSmallHotel
base proto:P_0
add hasRoom RoomNo1
add hasRoom RoomNo2
add hasRoom RoomNo3

AnotherHotel
base TheSmallHotel
rem hasRoom RoomNo3
add hasRating 3
```

(b) Concrete Syntax

```
Prototype TheSmallHotel
  property hasRoom
    value RoomNo1,
          RoomNo2,
          RoomNo3

remove hasRoom RoomNo3
add    hasRating 3

Prototype AnotherHotel
  property hasRoom
    value RoomNo1,
          RoomNo2
  property hasRating
    value 3
```

(c) Block Representation

Figure 7: Example of prototypical inheritance in three representations: `AnotherHotel` inherits from `TheSmallHotel` by removing `RoomNo2` and adding a property `hasRating` with value `3`.

the prototypes by edges with the property name as label. Properties and values that are removed are annotated to the arrow indicating inheritance. The concrete prototype syntax (Figure 7b) describes the same example. First, the prototype identifier is written, then the base to inherit from, followed by remove and add expressions. Both notations only describe the changes. In contrast, the block representation shown in Figure 7c describes the results of the inheritance. The information used for the process of inheritance is displayed by remove and add operations that are both written on the inheritance edge. The properties and their values are written in a textual form.

The formal definition introduced in [6] is tightly linked to the concrete syntax.

Definition 2.2.1 (Prototype Expressions [6])**.** *Let ID be a set of absolute IRIs according to RFC 3987 [23] 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 PROTO *be the set of all prototype expressions. The tuple $PL = (\mathtt{P}_\emptyset, ID, \mathtt{PROTO})$ is called the* Prototype Language.

For brevity we assume that the IRIs used here are unique. In a realistical setting longer IRIs or IRIs with prefixes would be used as in RDF. An example for a simple change expression would be:

$$(\mathtt{hasRoom}, \{\mathtt{RoomNo1},\ \mathtt{RoomNo2},\ \mathtt{RoomNo3}\}) \tag{1}$$

And the syntax of `TheSmallHotel` is written as:

$$(\mathtt{TheSmallHotel}, (\mathtt{P}_\emptyset, \{(\mathtt{hasRoom}, \{\mathtt{RoomNo1},\ \mathtt{RoomNo2},\ \mathtt{RoomNo3}\})\}, \emptyset) \tag{2}$$

`AnotherHotel` has this syntax:

$$(\mathtt{AnotherHotel}, (\mathtt{TheSmallHotel}, \{(\mathtt{hasRating}, \{3\})\}, \{(\mathtt{hasRoom}, \{\mathtt{RoomNo3}\})\}) \tag{3}$$

For future definitions the domain of a subset of all prototype expressions is needed:

Definition 2.2.2 (dom [6])**.** *The domain of a finite subset $S \subseteq \mathtt{PROTO}$, i.e., $dom(S)$ is the set of the prototype expression IDs of all prototype expressions in S.*

The domain of the example from Figure 7 is the set {`TheSmallHotel`, `AnotherHotel`}.

In order to ensure that there are no inheritance cycles in a knowledge base and that every inheritance chain originates from $\mathtt{P}_\emptyset$, *groundedness* is introduced.

Definition 2.2.3 (Grounded [6])**.** *Let $PL = (\mathtt{P}_\emptyset, ID, \mathtt{PROTO})$ be the Prototype Language. Let $S \subseteq \mathtt{PROTO}$ be a finite subset of* PROTO*. The set $\mathcal{G}$ is defined as:*

1. $\mathtt{P}_\emptyset \in \mathcal{G}$
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 if and only if $\mathcal{G} = dom(S) \cup \{\mathtt{P}_\emptyset\}$. This condition ensures that all prototypes derive (recursively) from $\mathtt{P}_\emptyset$ and hence ensures that no cycles occur.

So by this definition the set $\mathcal{G}$ for Figure 7 is $\{\mathtt{P}_\emptyset, \mathtt{TheSmallHotel}, \mathtt{AnotherHotel}\}$, since `TheSmallHotel` and `AnotherHotel` are all prototype expression IDs in the example and their respective base is also in $\mathcal{G}$. Since this is the domain of the example plus the empty prototype it follows that the example is grounded.

An example with cycles in it is:

$$S = \{(A, (B, \emptyset, \emptyset)) \\ (B, (A, \emptyset, \emptyset))\}$$

Here the set $\mathcal{G}$ (which is minimal) would only contain $\mathtt{P}_\emptyset$, since neither A nor B could be added to it because their bases are not in G. However, $dom(S) = \{A, B\}$ and thus S is not grounded.

A knowledge base is defined as a grounded subset of all possible prototype expressions. In addition each referenced ID has to be represented by a prototype in the knowledge base.

Definition 2.2.4 (Prototype Knowledge Base [6])**.** *Let $PL = (\mathtt{P}_\emptyset, ID, \mathtt{PROTO})$ be the Prototype Language. Let $KB \subseteq \mathtt{PROTO}$ be a finite subset of* `PROTO`*. KB is called a* Prototype Knowledge Base *if and only if 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 $dom(KB)$.*

Because of the third rule, the example given in Figure 7 alone is not a knowledge base. It would be necessary to add prototypes for each value of the add expressions, like $(\mathtt{RoomNo1}, (\mathtt{P}_\emptyset, \emptyset, \emptyset))$.

To access a prototype in a knowledge base given its ID a syntactical resolve function is introduced.

Definition 2.2.5 (R [6])**.** *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.*

So if we had a prototype knowledge base KB for the example in Figure 7 which included prototypes for all values of add expressions as described above then an example resolve would be:

$$(KB,\mathtt{TheSmallHotel}) = \\ (\mathtt{TheSmallHotel}, (\mathtt{P}_\emptyset, \{(\mathtt{hasRoom}, \{\mathtt{RoomNo1},\ \mathtt{RoomNo2},\ \mathtt{RoomNo3}\})\}, \emptyset)$$

Prototype Semantics The semantics of the prototype system are also defined in [6]. Basically, a prototype structure is introduced that consists of a set of identifiers, a representation of the prototypes and an interpretation that maps the IDs from the syntax to the identifiers in the semantics. Then multiple interpretation functions are introduced to define the semantics of the various syntactical constructs (simple change expressions, change expressions, property values, prototype expressions, knowledge bases). The structure underlying the semantics is based on a prototype that consists of an object with an identifier and a set of properties and their values. The inheritance is computed into the properties and values of the prototype.

Definition 2.2.6 (Prototype-Structure [6]). *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.*

The IDs from the syntax are mapped to the SIDs of the Prototype Language with a Herbrand-Interpretation.

Definition 2.2.7 (Herbrand-Interpretation [6]).
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.*

Next the interpretation functions for the different syntactical expressions are defined, starting with the basic building blocks up to whole knowledge bases.

Definition 2.2.8 (I_s [6]). *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:*

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

Thus, the interpretation of the values of the simple change expressions given in Example 1 is

$$\{I_h(\texttt{RoomNo1}), I_h(\texttt{RoomNo2}), I_h(\texttt{RoomNo3})\},$$

where I_h maps these elements to distinct elements in SID.

Definition 2.2.9 (I_c [6]). *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):*

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

One change expression is the *add* from Example 2:

$$\{(\texttt{hasRoom}, \{\texttt{RoomNo1}, \texttt{RoomNo2}, \texttt{RoomNo3}\})\}$$

The interpretation of this change expression is:

$$\begin{aligned}
&\{(I_h(\texttt{hasRoom}), I_s(\{\texttt{RoomNo1}, \texttt{RoomNo2}, \texttt{RoomNo3}\}))\} \cup \\
&\quad \bigcup_{w \in (ID \setminus \{\texttt{hasRoom}\})} \{(I_h(w), \emptyset)\} \\
=&\{(I_h(\texttt{hasRoom}), \{I_h(\texttt{RoomNo1}), I_h(\texttt{RoomNo2}), I_h(\texttt{RoomNo3})\})\} \cup \\
&\quad \bigcup_{w \in (ID \setminus \{\texttt{hasRoom}\})} \{(I_h(w), \emptyset)\}
\end{aligned}$$

Notice, that for each property that is not explicitly stated, the interpretation will yield the empty set as value of these properties.

In the following, the change expressions are applied to determine the value of a property.

Definition 2.2.10 (J [6])**.** *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:*

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

As example for the computation of the property values, consider the property `hasRoom` of the prototype expression with the ID `AnotherHotel`:

$$\begin{aligned}
&J(KB, \texttt{AnotherHotel}, \texttt{hasRoom}) \\
=&(J(KB, \texttt{TheSmallHotel}, \texttt{hasRoom}) \setminus \\
&\quad I_c(KB, \{(\texttt{hasRoom}, \{\texttt{RoomNo3}\})\})(I_h(\texttt{hasRoom}))) \\
&\quad \cup I_c(KB, \{\texttt{hasRating}, \{3\}\})(I_h(\texttt{hasRoom})) \\
=&(I_c(KB, \{(\texttt{hasRoom}, \{\texttt{RoomNo1}, \texttt{RoomNo2}, \texttt{RoomNo3}\}))\})(I_h(\texttt{hasRoom})) \setminus \\
&\quad I_c(KB, \{(\texttt{hasRoom}, \{\texttt{RoomNo3}\})\})(I_h(\texttt{hasRoom}))) \\
&\quad \cup I_c(KB, \{\texttt{hasRating}, \{3\}\})(I_h(\texttt{hasRoom})) \\
=&\{I_h(\texttt{RoomNo1}), I_h(\texttt{RoomNo2}), I_h(\texttt{RoomNo3})\} \setminus \\
&\quad \{I_h(\texttt{RoomNo3})\} \\
&\quad \cup \emptyset \\
=&\{I_h(\texttt{RoomNo1}), I_h(\texttt{RoomNo2})\}
\end{aligned}$$

With the interpretation function for values of properties, the interpretation of a prototype is defined:

Definition 2.2.11 (FP [6])**.** *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.*

The prototype expression of AnotherHotel then has the semantics:

$$
\begin{aligned}
&FP(KB, (\texttt{AnotherHotel},\\
&\quad (\texttt{TheSmallHotel}, \{(\texttt{hasRating}, \{3\})\}, \{(\texttt{hasRoom}, \{\texttt{RoomNo3}\})\}))\\
=&(I_h(\texttt{AnotherHotel}),\\
&\quad \{(I_h(\texttt{hasRoom}), \{I_h(\texttt{RoomNo1}), I_h(\texttt{RoomNo2})\}), (I_h(\texttt{hasRating}), \{I_h(3)\})\})
\end{aligned}
$$

The interpretation of a knowledge base is a mapping of each prototype expression occurring in the knowledge base to the fixpoint interpretation of that prototype expression.

Definition 2.2.12 (I_{KB}:Interpretation of Knowledge Base [6])**.** *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 presentation of the syntax and semantics introduced in [6]. The semantics will later be extended to contain constraints and a specialization relation.

2.3 Frames and Description Logics

The origin of description logic stems from frames. Frame systems were based on ideas from psychology and linguistics. The first computer systems based on these ideas did not have a formal model underneath. Later on description logic with a strong formal model based on a subset of first-order logic emerged. An early description logic is KL-ONE. More recently, the Web Ontology Language (OWL) evolved, which is of special interest as it is the standard used in the Semantic Web movement.

To illustrate what can be described with description logics, consider the following general example: A mammal is an animal. A fish is also an animal and lives in the water. A whale is a mammal that lives in the water. This can be described in description logic by the following formulas:

$$
\begin{aligned}
\texttt{Mammal} &\sqsubseteq \texttt{Animal}\\
\texttt{Fish} &\sqsubseteq \texttt{Animal} \sqcap \forall \texttt{LivesIn.Water}\\
\texttt{Whale} &\sqsubseteq \texttt{Mammal} \sqcap \forall \texttt{LivesIn.Water}
\end{aligned}
$$

We distinguish between two types of object. For one, we have unary predicates like Animal or Water. These are called concepts. In addition, we can have binary predicates like LivesIn called roles. Semantically, each unary predicate is interpreted as set of objects from the domain. The roles are interpreted as a set of tuples between objects of the domain. The symbol "$\sqsubseteq$" denotes a subset relation and $\texttt{A} \sqcap \texttt{B}$ the intersection of the interpretation of A and B. Writing $\forall$LivesIn.Water denotes that all concepts which occur in the binary relation called LivesIn have the concept Water on the right side of the relation (i.e., each concept A with LivesIn(A, Water) and not LivesIn(A, C), where $\texttt{C} \neq \texttt{Water}$). In this context the concept after the dot (here: Water) is called role-filler. In the above example a whale might be a fish, because we did not specify that fishes and mammals are disjoint concepts.

Next, Frames and two representatives of description logic are described in more detail.

Frames The term *frame* was first used by Marvin Minsky in his paper 'A Framework for Representing Knowledge' [24]. There, he described a theory about how to perform visual reasoning and language processing. A *frame* describes a general framework. It has multiple properties called *slots*.

The term *Frame Knowledge Representation System* has been used to describe a family of languages that represent knowledge by means of frames and slots attached to each frame. Frames as originally proposed did not have a clear formal semantics. Only later systems that borrowed ideas from Frames introduced strong formalisms.

KL-ONE This frame system is an early example with a strong formalism. Later on, a whole family of similar languages developed [25]. In KL-ONE [26] frames are called concepts and slots are named roles. The formalism used is based on logic. It uses a subset of first-order logic.

OWL The W3C recommended OWL [4] in 2004. It is also based on a subset of first-order logic. Also a newer version called OWL 2 [5] exists. There are different subsets of OWL (and OWL 2). Most of them restrict the expressiveness in order to make reasoning decidable or even efficient. The names of the relations change again with OWL. Concepts are called classes and roles are called properties.

For the purpose of this thesis the detailed differences between these systems are not of high importance. Rather, the similarities between them are most interesting, as we are looking for the most important knowledge representation primitives.

All these systems allow to define classes by composing different restrictions. These restrictions typically include value restriction and number restrictions. Value restrictions impose what kinds of values a property might have. In the example above $\forall$`LivesIn.Water` is a value restriction. We will refer to this kind of constraint as `allValuesFrom`. The name is chosen because the values that are allowed as value of the property have to come from the class defined as value of the constraint. In the above example all values of the property `LivesIn` have to be of class `Water`. Similarly, it can be required that there is some value of a property is of some class. For example $\exists$`hasChildren.Doctor` would denote that there is at least one child who is a doctor. We will refer to this kind of constraint as `someValuesFrom`. An example for a number constraint would be ≥ 3 `hasChildren`. Only objects that have a property `hasChildren` with at least three values will satisfy this restriction. We will refer to this kind of constraint as `cardinality`. Many other kinds of restrictions can be expressed. For example, the binary predicates (roles) can also be restricted. To explore basic knowledge representation primitives we will concentrate on simpler constraints. The constraints considered in this thesis will be illustrated in more detail in Section 3.

2.3.1 Open and Closed World Assumption

When using logic to represent the world there are two major modes of reasoning, namely the open and closed world assumption. With the open world assumption it is assumed that the knowledge is not complete. Things can be left open and could be either true or false. For example, the class `MultipleParent` could be defined as:

$$\texttt{MultipleParent} \sqsubseteq \geq 2\ \texttt{hasChildren}$$

In addition a knowledge base could contain the fact that `Alice` is a `MultipleParent`. With the open world assumption (OWA) the answer to the question whether `Alice` has at least two children will be true. However, we do not know the names of the children and how many there are exactly. If asked what the name of one of `Alice` children is, the knowledge base would answer *do not know*. Notice, that statements under the open world assumption (e.g., OWL statements) are *axiomatic*. They are taken to be true, even if the exact facts, like which children `Alice` has, are not known.

Reasoning under the closed world assumption (CWA) will come to different conclusions given the same information. The assumption is that only what is known and the literals than can be interfered from that are true and that *everything else is false*. Thus, on one hand it is asserted that `Alice` is a `MultipleParent`. On the other hand, since the children of `Alice` are not known to the knowledge base, it is assumed under the closed world assumption that `Alice` has no children. This leads to an inconsistency and this knowledge base would not be useful under the closed world assumption.

However, if the knowledge base also contained the assertion of two children of `Alice`, say `John` and `Tom`, then it would be consistent both with OWA and CWA. If it is then asked whether `Alice` has three children, the answer under the OWA would be *do not know*, while under the CWA it is not possible that there are any children that are not know, so the answer would be *no*.

As another example, consider the following minimal formula which uses only propositional logic. If the formula $a \vee b$ is a knowledge base then it is possible that either only a is true or only b is true or both could be true. So under the OWA knowing $a \vee b$ does not allow to deduce that a must be true. Also it cannot be deduced from it that b must be true. It is simply not known which of the variables are true, only that at least one of them has to hold.

In contrast, with closed world reasoning knowing $a \vee b$ leads to knowing $\neg a$ and $\neg b$ because neither a nor b can be concluded from $a \vee b$. This is of course an inconsistency, similar to the example above: There is no model that can satisfy these combined conditions. Different variants of closed world assumptions exist that tackle such problems. For example the *generalized closed world assumption* only assumes a literal to be false if it is part of a conjunction and the knowledge base still entails the rest of the conjunction without the literal. In the previous example $a \vee b$ is a conjunction. Assuming that it is in the knowledge base, it is entailed. However, neither a nor b is entailed on its own, so by the generalized closed world assumption they are not considered false. If we had a knowledge base with the formulas $a \vee b \vee c$ and $b \vee c$ then the knowledge base would entail $b \vee c$ and thus with the generalized closed world assumption $\neg a$ can be interfered. Notice, that this does not lead to an inconsistency, but in an open world there might as well be models of the both formulas where a is true.

Description logics usually use the open world assumption as it is more accurate to model the real world: New information can be added without leading to an inconsistency with previously inferred knowledge. When starting with an empty knowledge base everything is possible. Adding facts to the knowledge base then restricts what is possible. A property of this kind of reasoning is *monotonicity*, i.e., if something can be interfered to be true given some knowledge it can also be interfered if more knowledge is added. Adding something that does not play a role in the interference is no problem in the OWA. Adding a contradicting fact will lead the knowledge base to be inconsistent and then everything is interfered to be true (so the rule for monotonicity holds but of course in practice an incon-

sistent knowledge base is of no use). In comparison, the CWA assumes that everything not explicitly known is false. Thus, with an empty knowledge base the answer to each query under CWA would be *no*. Adding facts to the ones interfered by the empty knowledge base will always lead to an inconsistency. This kind of reasoning is not monotone. If we look again at the basic propositional example from above then the formula $a \vee b$ can be used to interfere $\neg a$. However, if the formula a is added, then $\neg a$ can no longer be derived.

A disadvantage of the open world assumption is that there might be no clear *yes* or *no* answer to a question because the answer is simply not known. In some use-cases this behaviour is not desirable and a definite answer is required. For example, when checking integrity constraints (which will be discussed in detail in Section 2.4) we want to know whether or not the data is according to the constraints. The answer that with some additional data which might or might not be true the integrity constraint could possibly hold is not useful. Instead, a clear line which data is acceptable and which is not is needed. So the decision which assumption to make depends on the use-case.

2.4 Integrity Constraints

Many applications use data and need to ensure that the data is according to some standard or schema. In relational databases the structure of the data is clearly defined by the database schema. Still, updates in the database may violate the schema and thus the input needs to be checked. Other data representation formats do not by themselves have a strict schema. There, schema languages using integrity constraints are important to ensure that data is suitable for given applications. XML is one of these data representation formats with few restrictions. RDF is used widely in the Semantic Web community and also needs such constraint validation methods. In the following integrity constraints for these systems are shortly described with a focus on RDF.

Relational Databases Relational databases [27] typically have a strong data schema, namely the tables the data is arranged in. Deleting elements in one table may lead to problems because the element could be referenced in other tables. To detect problems like these integrity checking is done when changing data in a database. Other integrity constraints are for example regarding the primary key. Each table has to have one, it needs to be unique and not null. Many databases also feature additional schemas that can be defined by the developer.

XML The Extensible Markup Language (XML) [28] is a data format to exchange documents over the internet. It is intended to be machine-readable as well as human-readable and is widely used.

To ensure that data is of a required format for an application or in general according to some specification there are many different constraint validation systems for XML. It is possible to add a Document Type Description (DTD) to an XML file to define a grammar to which the XML file should comply. Other validation system are XML Schema, RELAX NG, or Schematron. Some formulate the constraints in XML, others use other languages for the specification. Typically, regular expressions can be used to describe value constraints. Schematron on the other hand focuses on the existence or absence of paths in the XML Tree.

RDF Integrity constraints for RDF check properties of the graph. Different systems allow for different things to be checked. In general, the existence of properties can be required and that values of the properties are of a specific datatype or according to specified values.

Some approaches use OWL to describe schemas. However, OWL usually has open world reasoning, which is not suitable for integrity checking. Consider again the example of `Alice` from Section 2.3.1. It is defined that she has at least two children but neither the exact number nor the names of the children are known. If the requirement to have at least two children is viewed as an integrity constraint then it should assert whether or not the data given about `Alice` satisfies this requirement. Since the data regarding `Alice` does not contain information about the children the data should not satisfy the constraint that she has at least two children. However, under the open world assumption this cannot be concluded. The statement about the two children is not considered a requirement but an axiom. It has to be true, otherwise it would not be in the knowledge base. Therefore, when using OWL for integrity constraints the closed world assumption is used. Furthermore, OWL usually does not have a unique name assumption. This assumption means that every name is unique and therefore if two things have a different name they cannot be the same. Using these two assumptions, OWL can be used to describe schemas [8]. However, this approach can be misleading because the semantics are different from normal OWL semantics.

There are different drafts in the W3C regarding RDF validation. One of them is the Shape Expression Schema (ShEx) [9]. It uses an extra schema language to specify graph properties and values. A shape is a requirement for a node. It can specify what links there are and how the values of these links (properties) should look like. It is possible to specify the number of values a property can have. The values of the properties can be required to be of a certain data type or match a pattern. Furthermore, the values can be required to match at least one of a set of options or required to all match a set of options. In addition, the values of properties can also be specified to match another shape and boolean connectives can be used to combine all these parts.

The Shapes Constraint Language (SHACL) is another proposal for the W3C [29]. It also uses extra schema files, however, they are also written as RDF graph and do not use an extra format as ShEx. These RDF graph schemas are called *shapes* and they differ between *Node Shapes* and *Property Shapes*. Nodes can reference to property shapes to ensure that the node has a specific link. They can also combine things with *and*, *or* and *not*. Properties can have a cardinality restriction and the values of a property can be required to be of a certain RDFS class or datatype. In addition, patterns can be used to describe the data format. As opposed to ShEx they cannot express some-of and all-of constraints.

Another approach to consistency checking is SPIN[2]. It uses the SPARQL Protocol and RDF Query Language (SPARQL). SPARQL is a query language that works on RDF Graphs. Its syntax is similar to SQL and it is commonly used to access RDF data sets. SPIN stores SPARQL Queries in RDF Syntax and can be used as constraint language. Constraints are formulated in terms of SPARQL queries. Because they are written in SPARQL they are more complicated to write than the other languages that define high level constraint languages [30]. The benefit is that they can be validated using SPARQL engines typically already running for many data sets. RDF data itself is a graph pattern and SPARQL works on these graphs. Extensions like RDFS and OWL have a semantic that allows to interfere information that is not explicitly represented in the RDF graph. So

[2] `http://spinrdf.org/`

called *entailment regimes* [31] are used to extend graph pattern matching with entailment. So, using SPARQL it is not only possible to query RDF Graphs but also take into account entailments from RDFS or OWL.

To define how data has to be formulated for an application the Dublin Core Application Profile (DCAP) [32] can be used. The term *profile* refers to the description of requirements necessary for an application to use the data provided. Description Set Profiles [33] are a formalization of the DCAP. It is defined for XML but can also be extended to RDF. It can specify which properties are used and how the values of the properties have to look like. Cardinality constraints can be also be made. It cannot express allValuesFrom constraints.

Resource Shapes (ReSh) [34] are another means to define shapes of an RDF graph. The constraints are themselves expressed in RDF. Properties and which values are allowed can be constrained. Data types are included but property cardinality can only be restricted in limited ways (zero, one, many and combinations of these). The constraints cannot be combined by boolean connectives.

3 Constraints

In this section, it is discussed which constraints to use as knowledge representation primitives for the specialization relation and how to represent these constraints.

In the following different possible types of constraints are discussed in Section 3.1. Section 3.2 then presents different ways of representing constraints and the advantages and disadvantages of these representations are evaluated.

3.1 Types of Constraints

Section 2.3 and 2.4 showed examples of systems with constraints. The prototypical system proposed in [6] does not feature a data type system. In the implementation discussed in [21] a minimal type system is introduced. While this is an interesting extension, it is not included in the scope of this thesis. However, the possibilities that would arise with a type system are discussed in Section 6.1. For now, we will focus on constraints that are independent of data types. Therefore, constraints formulated by regular expressions on strings or restricting values to be from a range of integers will not be included here.

As discussed in Section 2.3 OWL and other description logic systems have the following common constraints:

- allValuesFrom (OWL: Class, Prototypes: Set of IRIs): All values of a property have to be from the specified class (respectively, the specified set of IRIs).
- someValuesFrom (OWL: Class, Prototypes: Set of IRIs): At least one value of a property has to be from the specified class (respectively, the specified set of IRIs).
- Cardinality (OWL: Integer, Prototypes: Interval): A specified property has to fit a cardinality restriction.
- hasValue (OWL: Individual, Prototypes: IRI]): The property has a specified value.

Many integrity constraint systems for RDF discussed in Section 2.4 also use these constraints. They provide strong expressiveness to restrict values of properties and the number of properties.

A special constraint for "hasValue" is not needed in our approach, because it will be possible to express this with values (a value itself will serve as a constraint). Of course there are many more possibilities how to restrict subsumption in OWL or how integrity constraints can be formulated. In this thesis we focus on this small set of constraint types and explore how specialization can be build based on these. In Section 6 possible extensions beyond these constraints are explored.

The constraints are defined as follows:

Definition 3.1.1 (Constraint)**.** *A constraint c consists of two parts, namely a type and constraining value. We distinguish the following constraint types on properties:*

- *All values have to come from a set of values (type: allValuesFrom)*
- *Some value has to be one from a set of values (type: someValuesFrom)*
- *The number of values is within an interval (type: cardinality)*

The type of a constraint c is written as $c.type$ and the constraining value as $c.cval$.

Semantically, constraint satisfaction is defined as follows:

Definition 3.1.2 (Constraint Satisfaction)**.** *Given a Constraint c and a set of values V, we say that V satisfies c if*

$$\begin{aligned} \forall v \in V &: v \in c.cval, \text{ if } c.type = allValuesFrom \\ \exists v \in V &: v \in c.cval, \text{ if } c.type = someValuesFrom \\ &|V| \in c.cval, \text{ if } c.type = cardinality. \end{aligned}$$

In the next section, the integration of the above constraints into the prototype language is discussed.

3.2 Constraint Representation

While we defined how constraints can be written down, they need to be represented in the prototype system. Especially integrity constraint systems often use extra syntactical representations to represent constraints. We aim at a tight integration of constraints with the prototypical system, as does description logic. Therefore, constraint representation should be integrated with the prototype system.

Beside this requirement, a good representation should have certain properties. An optimal representation would be human-readable and short. It should also distinguish between values and constraints on a syntactical level. This will make it easier to assign semantics to the constraints.

As discussed above, a prototype should be allowed to have constraints and values at the same time. The values will be understood as a constraint themselves to express that a certain value has to be present. In addition, it should be easy to connect the values for one property and its constraints, i.e., they should be close to another in the representation. This will ease the connection of values and constraints which is useful if values are themselves a part of the constraining factors. If the existing syntax of the prototype system does not need to be modified then it is easier to integrate the constraints. Furthermore, if constraints are represented in such a way that they can be reused by using prototypical inheritance it

would be a handy benefit. Thereby constraints could be reused and modified for different contexts.

In the following, we will explore three possibilities to represent constraints, discuss their benefits and select one for the remainder of this thesis. The three variants will be representing constraints as

- values of a property,
- properties,
- prototypes.

Representation 1: Constraints as Values of a Property Figure 8 shows an example of values and constraints mixed as values of a property.

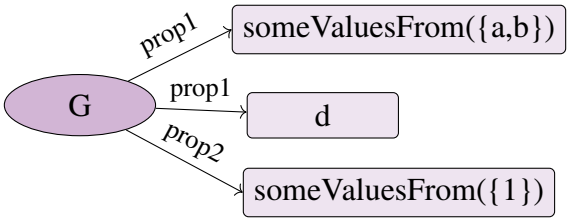


Figure 8: Example of a prototype with constraints where the constraint types and constraining values are represented as prototype value.

The prototype `G` has two properties named `prop1` and `prop2`. The property `prop1` has the value `d` and the constraint that some value has to be either `a` or `b`. The other property, `prop2`, has to have the value `1`.

This approach is simple to understand and human-readable. The values and constraints of a property are in one place and thus easy to connect. However, it is not easy to distinguish between values and constraints, which will be necessary to define the specialization relation. In addition, the combined representation of constraint type (here: someValuesFrom) and constraint values (here: {1} and {a,b}) is hard to formalize. The presented graphic would be written down as shown in Listing 1.

However, this would lead to very long and unreadable IRIs for constraint expressions and the defined sets would not be easily reusable. It would be needed to introduce parsing of the constraint IRIs and thus also to distinguish such IRIs from regular IRIs in the syntax. It would be necessary to extend the syntax each time a new constraint is introduced.

```
ex:G
base proto:P_0
add ex:prop2 ex:someValuesFrom#ex:1
add ex:prop1 ex:someValuesFrom#ex:a,ex:b
add ex:prop1 ex:d
```

Listing 1: Syntax example of Representation 1, where constraints are represented by values of properties.

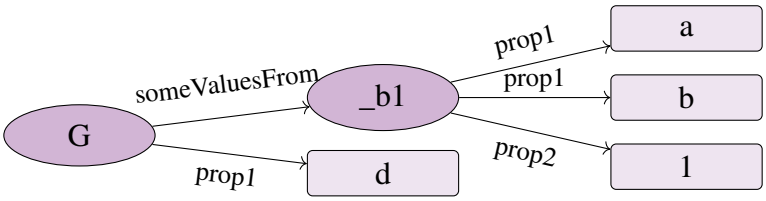


Figure 9: Example of a prototype with constraints where the constraint types are represented as prototype property and the constraining values as property value.

Representation 2: Constraints as Properties Figure 9 shows the same example but constraints are represented as properties. The representation of values and constraints is separated. The values of a property are directly linked to the prototype as in the previous representation. The constraint types are written as property names and lead to another prototype that in turn defines the constraining values for the constraint type. So, the constraint type and constraint values can be distinguished easily. Notice, that the property to which a constraining value is applied is expressed by the property name between `_b1` and the constraining values. In the example `a` and `b` are both linked to the property `prop1`. Thus, they form the constraint that some value of the property `prop1` is either `a` or `b`. The syntax of this example is shown in Listing 2.

```
ex:G
base proto:P_0
add proto:someValuesFrom :_b1
add ex:prop1 ex:d

:_b1
base proto:P_0
add ex:prop1 ex:a
add ex:prop1 ex:b
add ex:prop2 ex:1
```

Listing 2: Syntax example of Representation 2, where constraints are represented by properties.

The main difference to the previous representation is that constraint type and constraining value are stored in distinguished parts of the prototypes: The constraint types are represented as name of a property and the constraining values as values of another property. The representation is still fairly human-readable and a major advantage is that constraints and values can be clearly distinguished and that no modification of the prototype syntax is necessary. In addition, since each constraint type has its own prototype inheritance can be used to reuse parts of the constraint values. This is not optimal for reuse because there is one prototype for each type but most likely one would rather like to reuse constraints for one property only. Furthermore, property values and constraints are not in the same place. A connection between the value of e.g. `prop1` and the constraints of `prop1` needs to follow different properties of the prototype `G`.

Representation 3: Constraints as Prototypes Figure 10 shows the example from above with constraints defined as prototypes.

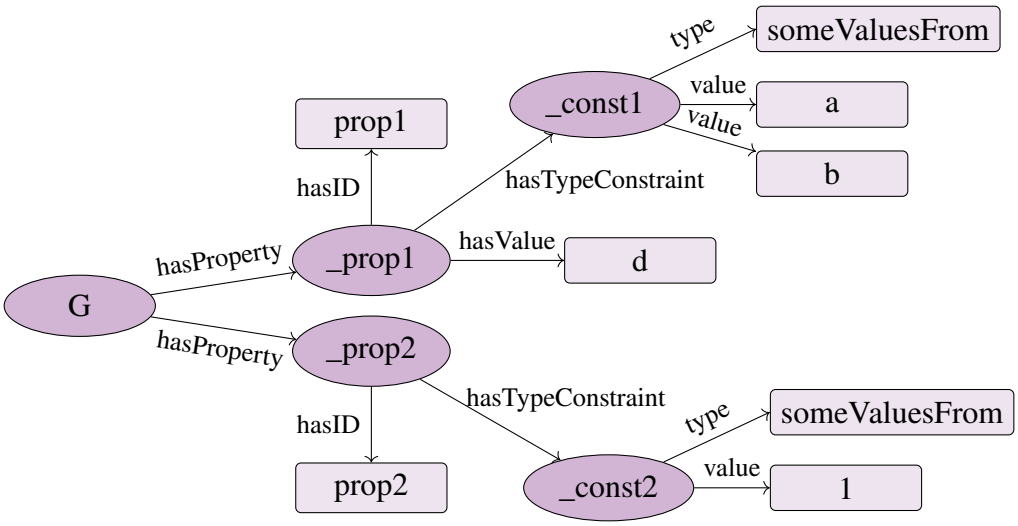


Figure 10: Example of a prototype with constraints where the properties are represented as composed prototypes. The constraints are linked to the respective property prototype and are themselves prototypes. The constraint types and values are represented by the properties `type` and `value` of the constraint prototypes.

```
ex:G
base proto:P_0
add proto:hasProperty :_prop

:_prop1
base proto:P_0
add proto:hasID prop1
add proto:hasValue ex:d
add proto:hasTypeConstraint :_const1

:_const1
base proto:P_0
add proto:type someValuesFrom
add proto:value ex:a
add proto:value ex:b

:_prop2
base proto:P_0
add proto:hasID prop2
add proto:hasTypeConstraint :_const2

:_const2
base proto:P_0
add proto:type someValuesFrom
add proto:value ex:1
```

Listing 3: Syntax example of Representation 3, where constraints are represented by composed prototypes.

Each property is defined as a prototype which has an identifier and might have values

and constraints. The constraints are also prototypes and consist of a type and values. In the prototype syntax the example would be written as shown in Listing 3.

This representation is much less readable than the two previous ones. However, it has many other advantages. As with the previous representation the syntax does not need to be modified, only the semantics of the constraints needs to be added. Furthermore, values and constraints are clearly distinguished. The values of a property and its constraints are in one place, that is, they are not mixed with values or constraints of other properties. Since each property and each constraint is an prototype by themselves it is easy to reuse properties and constraints with prototypical inheritance. In addition, if the same constraint should be applied to multiple properties (even in different prototypes), the same prototype can be referenced.

	Desired	Constraint as value	Constraint as property	Constraint as prototype
human-readable	yes	yes	a bit less	involved
distinguish values and type of constraint	yes	no	yes	yes
property value and constraint in one place	yes	yes	no	yes
syntax modification necessary	no	yes	no	no
constraints reusable (modifiable with inheritance)	yes	no	only constraint values	yes

Table 1: Comparison of the different constraint representations presented previously.

Comparison A summary of the different properties of the presented representations is shown in Table 1. The "Desired" column describes what is the desired outcome for each property as discussed at the beginning of this section. Not all binary combinations of properties where explored because some of them are not independent of each other. Most notably, there is a trade-off between being able to reuse constraints with inheritance and human readability. If a constraint should be reusable independent of the prototype it is applied to the constraint itself needs to be represented as a prototype. Representing constraints as values does not allow for this kind of reuse. However, this representation is shortly representable and human-readable. Defining constraints as properties allows to reuse types of constraints for different properties, but it does not allow to reuse a constraint independent of the property it is applied to. However, since it allows some kind of reuse it is already longer to specify and less readable. Representing constraints as prototypes needs a lengthy representation, but the constraints are fully reusable independent of their property and even whole properties can be easily reused. As discussed before, distinguishing values and constraints will make it easier to define the semantics of constraints and is therefore highly desired. Also having values and constraints of one property together will make defining the semantics easier and increases readability. Not needing to modify the syntax is also a benefit because it will ease extendability if the syntax needs not be modified for new constraints.

```
Prototype G
  property prop1
    constraints someValuesFrom(a,b)
    values d
  property prop2
    constraints someValuesFrom(1)
```

Figure 11: Block Representation of prototypes with constraints.

In the spirit of the prototypical system that should ease sharing and reusing data, representing constraints as prototypes is chosen. It is not as easy to read as other options but it will allow for full reuse and sharing of constraints and properties. Furthermore, it fulfills all other desired properties. The different presented representations are transferable from one to another. If another representation is desired for an application, it could be automatically converted.

The two other constraints that are chosen for this thesis are represented as follows: The constraint `allValuesFrom` is also a type constraint and is syntactically handled like `someValuesFrom` except assigning the type accordingly. Cardinality constraints are a different kind of constraint. Instead of the value of a property they restrict the number of values a property might have. Therefore, cardinality constraints use the property `hasCardinalityConstraint` instead of `hasTypeConstraint`. Instead of `type` and `value` they only have the properties `min` and `max` to describe the lower and upper bound of the cardinality constraint.

Since the chosen representation is not very readable we will introduce a shortened graphical representation in Section 3.2.1. This notation will abstract the different prototypes used to represent properties and constraints. In addition, syntactic abbreviations are introduced in the following section to formally define the representation of constraints.

3.2.1 Graphical Representation of Prototypes with Constraints

Since the chosen representation is by its composed nature more complex it is not very readable. Thus, we introduce a compact graphical representation for these prototypes. The representation is an extension of the Block Representation introduced in Section 2.2.4.

Consider Figure 11 which shows the running example of this section in the Block Representation. One Block describes one composed prototype with all its properties, constraints and values. The representation starts with the keyword `Prototype` followed by the name of the prototype, in this case `G`. Afterwards properties are described one after another by the keyword `property` followed by the property name. Below the property name constraints and values are listed. Constraints are represented by their type and the constraining values are listed in parentheses afterwards.

3.2.2 Syntactic Abbreviations for Constraints

In the following abbreviations are introduced to define the syntax of proper constraints and property prototype expressions.

Figure 12 shows the previously used example with only one property. In blue the names given to the different prototypes are shown. The prototype expression for a property is called *property prototype expression* and includes that prototype and its properties

and values. A prototype expression that is a type constraint is called *type constraint prototype expression* or *cardinality constraint prototype expression* if it represents a cardinality constraint.

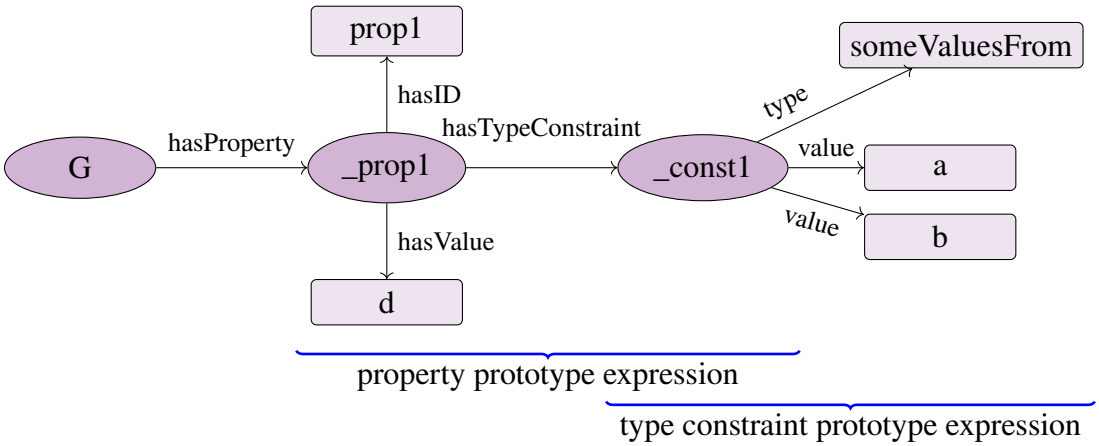


Figure 12: Illustration of syntactical structures of composed prototypes with constraints. Prototypes that represent a property are called *property prototype expression* and prototypes that represent type constraints are called *type constraint prototype expressions*.

Formally, the expressions for type constraints and cardinality constraints need to have these formats:

Definition 3.2.1 (Type Constraint Prototype Expression)**.** *Let* $PL = (\mathsf{P}_\emptyset, ID, \texttt{PROTO})$ *be a Prototype Language. Let* $p \in \texttt{PROTO}$ *be a prototype expression. We call* $p = (id, (base, add, remove))$ *a* type constraint prototype expression *if and only if*

$$
\begin{aligned}
&(\texttt{proto: type}, ids) \in add,\\
&ids = \{\texttt{proto: allValuesFrom}\} \vee \{\texttt{proto: someValuesFrom}\},\\
&(\texttt{proto: value}, vals) \in add.
\end{aligned}
$$

Definition 3.2.2 (Cardinality Constraint Prototype Expression)**.** *Let* $PL = (\mathsf{P}_\emptyset, ID, \texttt{PROTO})$ *be a Prototype Language. Let* $p \in \texttt{PROTO}$ *be a prototype expression. We call* $p = (id, (base, add, remove))$ *a* cardinality constraint prototype expression *if and only if*

$$
\begin{aligned}
&(\texttt{proto: max}, max) \in add,\\
&max \in \mathbb{N} \cup \{\texttt{proto: infty}\},\\
&(\texttt{proto: min}, min) \in add,\\
&min \in \mathbb{N}.
\end{aligned}
$$

A property prototype expression is then a prototype that defines exactly one identifier and may have values and constraints. The constraints need to be valid type or cardinality constraint prototype expressions.

Definition 3.2.3 (Property Prototype Expression)**.** *Let* $PL = (\mathsf{P}_\emptyset, ID, \texttt{PROTO})$ *be a Prototype Language. Let* $pe \in \texttt{PROTO}$ *be a prototype expression.*

We call $pe = (id, (base, add, remove))$ *a* property prototype expression *if and only if*

$$(proto: hasID, \{pname\}) \in add, \text{ and}$$
$$[(proto: hasValue, vals) \in add, \text{ or}$$
$$(proto: hasTypeConstraint, tconst) \in add, \text{ or}$$
$$(proto: hasCardinalityConstraint, cconst) \in add],$$

where the elements in tconst must be type constraint prototype expressions and the elements in cconst must be cardinality constraint prototype expressions.

To be able to easily access all the property prototype expressions of a prototype the function *property_expressions* is introduced.

Definition 3.2.4 (Property Expressions)**.** *Let* $PL = (\mathtt{P}_\emptyset, ID, \mathtt{PROTO})$ *be a Prototype Language and* FKB *be a fixpoint knowledge base for* PL*. Let* $id \in ID$ *and* $R(FKB, id) = (id, (base, add, remove))$*. We define* $property\_expressions(FKB, id)$ *to denote the set*

$$\{ppe \mid (proto: hasProperty, props) \in add \wedge prop \in props \wedge R(FKB, prop) = ppe \wedge ppe \text{ is a property prototype expression}\}.$$

4 Specialization Relation

The goal of the thesis is to extend the prototypical approach to knowledge representation on the web with knowledge representation primitives. It should allow hierarchical grouping like subsumption in description logic or inheritance in class-based systems.

Prototypical inheritance does not define a clear hierarchy itself because the inheritance link is purely focused on re-use. The children can remove properties of their parent. Thus, a hierarchical relation cannot be deduced from a prototypical inheritance link. Still, expressing hierarchies is a major part of knowledge representation and should be expressible in a prototypical system. Furthermore, applications that use a knowledge base need to ensure that the data used conforms with the expectations of the application. Many systems use integrity constraints and schemas to ensure this.

In summary, a knowledge representation system needs have two functionalities:

1. Define abstract concepts and express hierarchies.

2. Define and validate integrity constraints.

To address these demands, we will introduce a new relation called *specialization* (written: $s \preceq g$ if s is a specialization of g). It will define hierarchies as well as allow the definition and validation of integrity constraints.

To enable the first functionality, constraints can be used to define abstract concepts that should be on the top of the hierarchy. Prototypes that should be below this concept in the hierarchy can be modeled in such a way that they satisfy the constraints. Intermediate concepts that belong in the middle of a hierarchy can be expressed by imposing stricter or more constraints than the generalizations they should be sorted under. Defining hierarchies in such a way allows to discover which prototypes fit the concepts. Notice, that in contrast to class-based inheritance it is not defined which prototypes are a specialization

of another. Instead, specialization is a property between two prototypes that depends on constraint satisfaction. Class-based inheritance defines a hierarchy by passing properties to the children such that the children are like their parents. We already have a notion of inheritance with the prototypical system and do not want to further modify prototypes. Therefore, the specialization relation that will be introduced in this thesis is an observation rather than a definition: A prototype is a specialization of its generalization because it is *observed* that it satisfies the constraints of the generalization.

This also has a major benefit for using the system on the web. Given a knowledge base one can define an abstract prototype with constraints and then check which data is a specialization of it. Thus, it is possible to discover what prototypes are already defined by others that are hierarchically below a new defined concept.

Regarding the second functionality, the same mechanism can be used to define abstract prototypes with constraints to describe integrity constraints. Existing prototypes can then be checked to validate that they satisfy the constraints.

So in general, we need a relation between two prototypes that checks if either

1. the values of a specializing prototype satisfy the constraints of a generalized prototype (*satisfaction*) or

2. the constraints of a specializing prototype are stricter than the constraints of a general prototype (*matching*).

As discussed in Section 3, a prototype is allowed to have constraints and values at the same time and values are also considered to be a kind of constraint, namely, that a specialization has to have at least the same values as its generalization.

Besides the two ways to specialize a prototype (satisfaction and matching), there are two other properties that the specialization relation should possess:

1. Transitivity: If s is a specialization of t and t is a specialization of g, then s is a specialization of g, written: $s \preceq t, t \preceq g \implies s \preceq g$.

2. Multi-specialization: It is possible that s is a specialization of g_1 and g_2 at the same time, while the relation between g_1 and g_2 is not fixed (they are not necessarily a specialization of one another).
   Formally: $s \preceq g_1, s \preceq g_2 \not\Longrightarrow g_1 \preceq g_2 \vee g_2 \preceq g_1$.

For modeling ontological connections between prototypes, transitivity is useful. Is-a relations in ontologies are typically transitive and it seems that humans expect such a relation to be transitive. In addition, other knowledge representation systems model similar relations (e.g., subclasses) as transitive. Thus, the specialization relation is chosen to be transitive to allow modeling of similar relations.

Multi-specialization is an useful feature with regard to knowledge bases that are connected and shared over the internet. If it were only possible to be a specialization of one prototype (and recursively its generalizations), this would limit the reusability of prototypes for other modeling purposes. If we think of specialization as a relation that defines which prototypes can be used for a program (because they satisfy some constraints needed by the program), we might want to use one prototype for multiple programs. This would not be possible if we did not feature multi-specialization. In addition, the intended observative nature (instead of declarative) of the specialization relation suggest allowing multi-specialization.

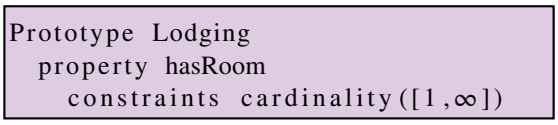


Figure 13: A lodging prototype that restricts the property `hasRoom` to have at least one value, i.e., a lodging has at least one room.

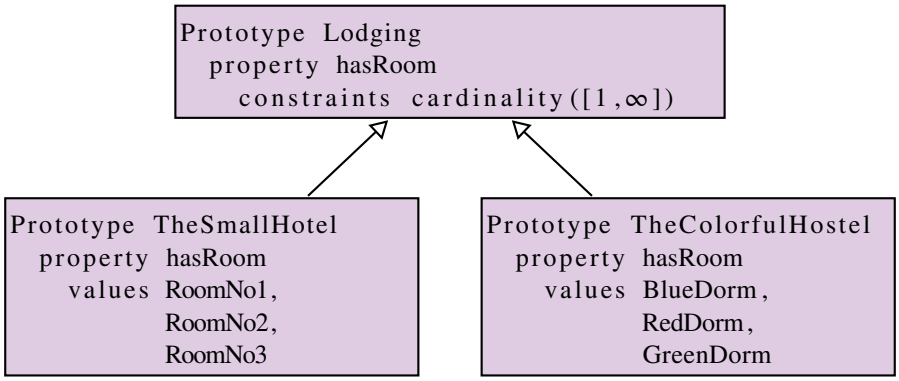


Figure 14: `TheSmallHotel` and `TheNiceHostel` are both specializations of the `Lodging` prototype because they satisfy the cardinality restriction on the hasRoom property.

The remainder of this section is structured as follows: To foster the understanding of the nature of the specialization relation, different examples are discussed in Section 4.1. Afterwards, the specialization relation and everything needed for it is defined formally in Section 4.2. In Section 4.3 properties of the specialization relation are explored.

4.1 Examples

In this section an extended example of the intended specialization relation is described. The specialization hierarchy is applied to the domain of lodgings and hotels. As a simple example, consider a lodging prototype that specifies that it has at least one room (shown in Figure 13), but does not specify which.

Then a specific lodging, like a concrete hotel or a hostel could have some specific rooms which are named in some way. Figure 14 shows `TheSmallHotel` and `TheColorfulHostel` which have only concrete values for the property `hasRoom`. Since they both satisfy the constraint imposed by the `Lodging` prototype, `TheSmallHotel` and `TheColorfulHostel` are each a specialization (symbol: $\twoheadrightarrow$) of the `Lodging` prototype.

In contrast consider Figure 15, which shows prototypical inheritance. Prototypical inheritance (symbol: $\Rightarrow$) copies all values from the parent to the child, except those that are removed. Afterwards properties that should be added are added. In contrast to inheritance, the specialization relation between the `Lodging` prototype and `TheSmallHotel` prototype in Figure 14 does not include any inheritance. The `Lodging` prototype did not specify a concrete value for the property `hasRoom`. Thus, `TheSmallHotel` and `TheColorfulHostel` prototypes do not inherit any value from the `Lodging` prototype. Instead, they are filling the specified property `hasRoom` with values that are satisfying the restriction of the `Lodging` prototype. With prototype inheritance, we could even remove the `hasRoom` property entirely, while a specialization of the prototype has to have the `hasRoom` property.

Another difference to the inheritance relation used for prototypes is, that it is possi-

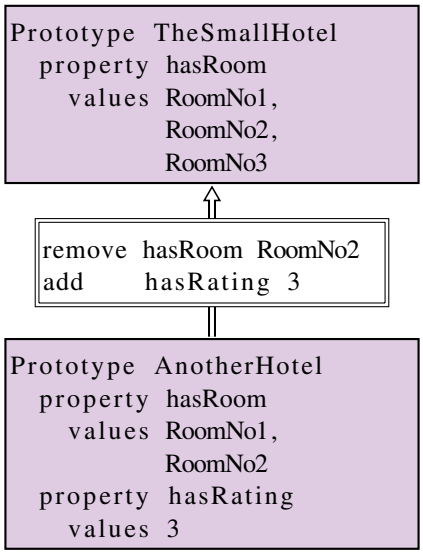


Figure 15: Example of prototypical inheritance: `AnotherHotel` inherits from `TheSmallHotel` by removing `RoomNo2` and adding a property `hasRating` with value 3.

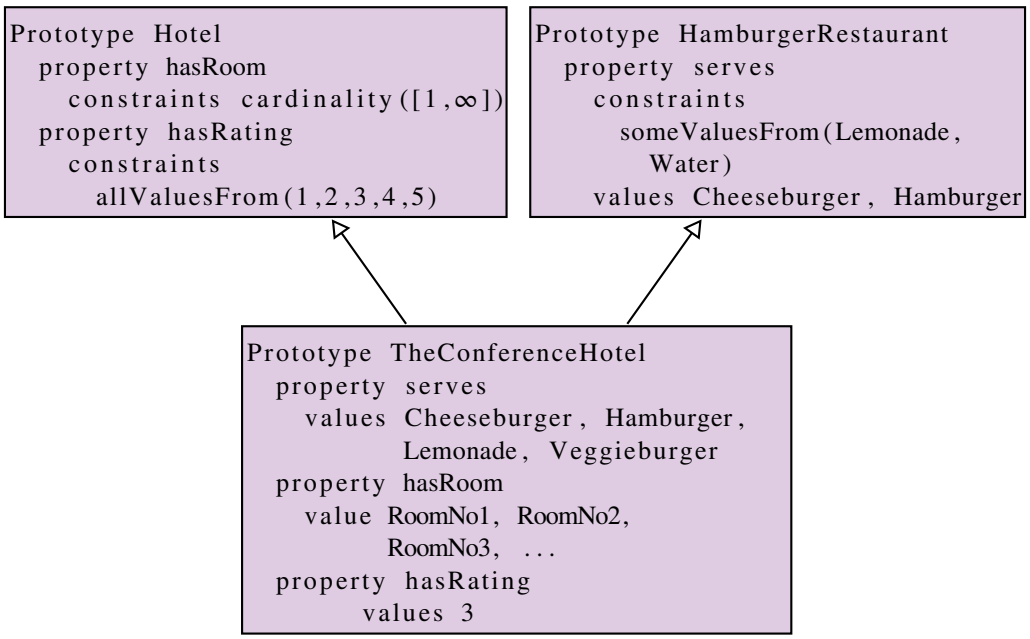


Figure 16: A `Hotel` has at least one room and a rating between one and five. A `HamburgerRestaurant` serves lemonade or water and cheeseburger and hamburgers. `TheConferenceHotel` is a specialization of both the `Hotel` and the `HamburgerRestaurant` prototype, because it satisfies the constraints imposed by these prototypes.

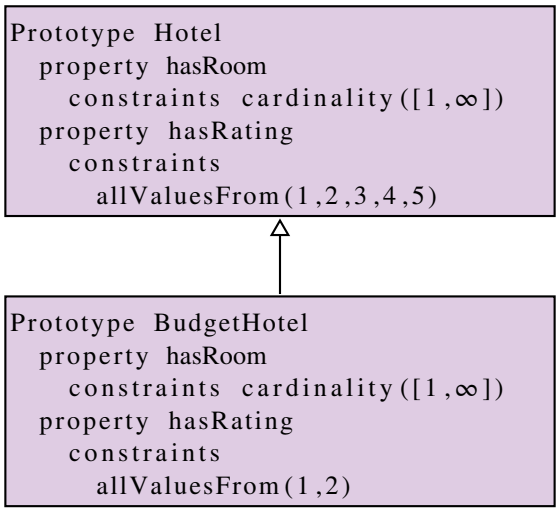


Figure 17: A prototype is a specialization of another if it makes constraints stricter and keeps the others unchanged. Here, the `BudgetHotel` may have a rating of either one or two, which is stricter that a value from one to five imposed by the `Hotel` prototype.

ble for one prototype to be a specialization of multiple prototypes while a prototype can only have one base. For example, Figure 16 shows multi-specialization. The `hotel` prototype specifies that it has at least one room and that it has a rating from one to five. The `HamburgerRestaurant` is modeled such that it serves cheeseburgers and hamburgers and has to offer water or lemonade in addition. A concrete hotel like `TheConferenceHotel` can be a specialization of both prototypes since it satisfies all constraints imposed by each of the prototypes.

In addition to filling in concrete values that satisfy the constraints it will also be possible to specialize a prototype by matching. This means making at least one of the imposed constraints stricter. For example consider the `BudgetHotel` defined in Figure 17. The `BudgetHotel` prototype restricts the property `hasRating` stronger than the `Hotel` prototype. Thus, it is a specialization of the `Hotel` prototype.

Furthermore, specializations may add additional properties without interfering with the specialization relation as long as the constraints for the specified properties are satisfied. For example, Figure 18 shows that the previously defined `TheConferenceHotel` prototype is a specialization of the `Hotel` prototype which is in turn a specialization of the `Lodging` prototype.

That `TheConferenceHotel` has the additional property `serves` does not interfere with specialization, since neither the `Lodging` nor the `Hotel` prototype imposed any constraints on that property. Notice, that the specialization relation is transitive. Therefore, `TheConferenceHotel` is also a specialization of `Lodging`.

It is also possible to mix constraints and values and to add additional values or constraints to specialize a prototype. For example one could describe that a certain kind of lodging has to have one specific kind of room as well as a fixed number of others that are not specified. This can be described by combining constraints and concrete values for one property: For example, a `BigLodging` prototype is defined to have at least 20 rooms and one of them is named `RoomNo42` (shown in Figure 19).

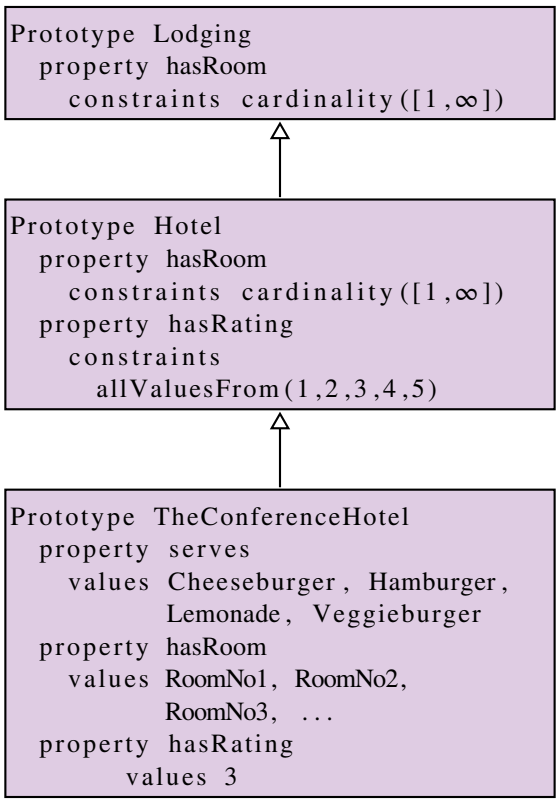


Figure 18: `TheConferenceHotel` is a specialization of the `Hotel` prototype, which is a specialization of the `Lodging` prototype. Specialization is transitive and only constraints in properties that occur in the generalization have to be satisfied in the specialization.

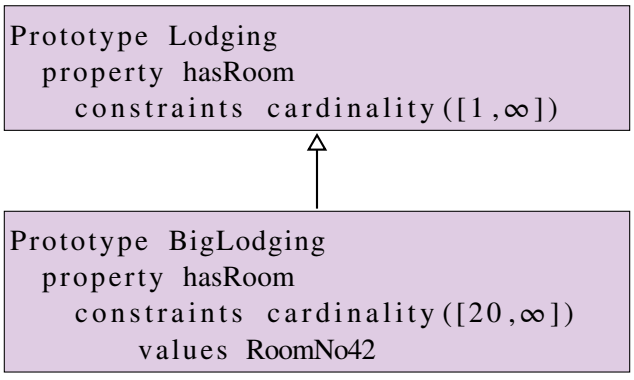


Figure 19: A `BigLodging` has a room named `RoomNo42` and at least 20 rooms overall. It is a specialization of the `Lodging` prototype because it imposes a stronger cardinality restriction and the added value restriction also makes it a stronger constraint.

4.2 Definition

In this section, the specialization relation will be defined. However, we will first take a closer look at prototypical inheritance. The specialization relation considers the results of inheritance, i.e., the properties that a prototype actually has after taking inheritance into account. In Section 4.2.1 we will describe how inheritance can be applied syntactical instead of semantically (and show that the applied operations leave the semantics unchanged).

Afterwards, in Section 4.2.2 the composed prototypes introduced in Section 3.2 are given a unifying semantic such that they can be easily accessed.

With these definitions in place the details of the specialization relation can be defined in Section 4.2.3. It is discussed, how the guiding principles defined at the beginning of this section are transformed to a definition.

4.2.1 Inheritance Free Fixpoint State of a Knowledge Base

Currently, the semantics of the prototype system define the meaning of the inheritance relation. The interpretation of a prototype expression transforms it to an object that does not contain any information about inheritance, it just has an identifier and a set of properties and values (that are the result of applying inheritance). The semantics of the specialization relation will be based on that object. Since the transformation from prototype expressions with inheritance to the prototype objects obtained after applying inheritance is in the semantics, it is not easily visible for a knowledge base designer. To spot mistakes in the design of the knowledge base regarding the specialization relation it is necessary to view the prototypes resulting from applying inheritance. Thus, a new syntactical state of knowledge bases is introduced where the inheritance has already happened. Every prototype expression will be syntactically transformed such that it inherits from $\mathtt{P}_\emptyset$ and contains all properties and values it would gain through inheritance. This different state also emphasizes one of the differences between the specialization relation and inheritance, namely, that inheritance is applied before specialization.

We will define how a knowledge base can be transformed to an inheritance free fixpoint state and prove the soundness of that new state, i.e., that the interpretation of the original and the transformed knowledge base are identical. First, we define how we can access values of properties of sets of change expressions (namely, *add* and *remove*).

Definition 4.2.1 (Access Values of Simple Change Expression Sets)**.** *Given a set of simple change expressions called* changes*. To access the values* $\{r_1, \ldots, r_m\}$ *of a simple change expression* $(p, \{r_1, \ldots, r_m\}) \in changes$ *we write* $changes[p]$.

With the capability to access property values we define a recursive function to compute the inherited property values of a property. The function is defined with regard to a prototype identified by its ID and a knowledge base.

Definition 4.2.2 (Inherited Property Values)**.** *Let* KB *be a prototype knowledge base and* $id \in ID, p \in ID$*. Let* $R(KB, id) = (id, (base, remove, add))$ *(the resolve function applied to* id*). Then the inherited value for the property* p *of the prototype* id*, i.e.,* $Values(KB, id, p)$ *is:*

$add[p]$*, if* $base = \mathtt{P}_\emptyset$

$(Values(KB, base, p) \setminus remove[p]) \cup add[p]$*, otherwise*

The fixpoint state of a knowledge base can then be defined by rewriting the inherited property values for each property of every prototype.

Definition 4.2.3 (Fixpoint State of a Knowledge Base)**.** *A knowledge base FKB is the* fixpoint state *of a knowledge base KB if and only if*

- *Each prototype in FKB has* $\mathtt{P}_\emptyset$ *as its base.*
- *It holds that* $pe = (id, (base, add, remove)) \in KB$ *if and only if* $fpe = (id, (\mathtt{P}_\emptyset, properties, \emptyset)) \in FKB$*, where* $properties = \{(p, Values(KB, id, p)) \mid p \in ID, Values(KB, id, p) \neq \emptyset\}$*. We call such a tuple* (pe, fpe) *an* inheritance pair *of* KB *and* FKB *(since* fpe *is the result of applying inheritance to* pe*).*

Note that the "if and only if" ensures that the IDs of the original knowledge base are exactly the same as the IDs of the fixpoint knowledge base.

We will now show that a knowledge base and its fixpoint state have the same interpretation. To arrive at that conclusion, it is first proven that the interpretation of inherited property values is the same as the interpretation function J (Definition 2.2.10). To prove this, a induction on the depth of inheritance is needed. The function J (Definition 2.2.10) as well as the function $Values$ (Definition 4.2.2) assume that each prototype is of the form (id, (b, r, a)). It distinguishes the cases where the base b is $\mathtt{P}_\emptyset$ and where it is not. Thus, we define the depth of inheritance as zero if the base is $\mathtt{P}_\emptyset$.

Definition 4.2.4 (Depth of Inheritance)**.** *The depth of inheritance of a prototype p (written* $depth(p)$*) is the number of inheritance steps necessary to access a prototype that has* $\mathtt{P}_\emptyset$ *as base:*

$$depth(p) = 0, \text{ if } p = (id, (\mathtt{P}_\emptyset, add, remove))$$
$$depth(p) = depth(base) + 1, \text{ if } p = (id, (base, add, remove))$$

Lemma 4.2.1. *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 knowledge base. Then* $\{I_s(v) \mid v \in Values(KB, id, p)\} = J(KB, id, p)$ *for any* id *occurring in* KB *and any property* p*.*

Proof. Given a knowledge base KB and an $id \in ID$ occurring as a prototype identifier in KB. Let $R(KB, id) = (id, (b, a, r)) = pe$. Proof by induction on the depth of inheritance of pe.

Basis. n=0 Thus, $b = \mathtt{P}_\emptyset$.

$$\begin{aligned}
&\{I_s(v) \mid v \in Values(KB, id, p))\} \\
=&\{I_s(v) \mid v \in a[p]\} \\
=&I_c(KB, a)(I_h(p)) \\
=&J(KB, id, p)
\end{aligned}$$

Inductive Hypothesis. Let $\{I_s(v) \mid v \in Values(KB, id, p)\} = J(KB, id, p)$ hold for all prototypes where the depth of inheritance is at most n.

Inductive Step.
Let $R(KB, id) = (id, (b, a, r)) = pe$, where $depth(pe) = n + 1$.

$$
\begin{aligned}
&\{I_s(v) \mid v \in Values(KB, id, p)\} \\
=&\{I_s(v) \mid v \in (Values(KB, b, p) \setminus r[p]) \cup a[p]\} \\
=&(\{I_s(v) \mid v \in Values(KB, b, p)\} \setminus \{I_s(v) \mid v \in r[p]\}) \cup \{I_s(v) \mid v \in a[p]\} \\
\overset{\text{I.H.}}{=}&(J(KB, b, p) \setminus I_c(KB, r)(I_h(p))) \cup I_c(KB, a)(I_h(p)) \\
=&J(KB, id, p)
\end{aligned}
$$

□

With this lemma it can then be shown that a fixpoint state knowledge base has the same interpretation as its origin.

Theorem 4.2.1. *Let $O = (SID, OB, I_h)$ be a Prototype-Structure for the Prototype Language $PL = (\mathbb{P}_\emptyset, ID, PROTO)$ with I_h being a Herbrand-Interpretation. Let KB be a knowledge base and FKB its fixpoint state. Then*
$I_{KB}(KB, pe) = I_{KB}(FKB, fpe)$ *for all inheritance pairs* (pe, fpe) *of* KB *and* FKB.

Proof. Let KB be a knowledge base and FKB its fixpoint state. Let (pe, fpe) be an inheritance pair of KB and FKB and the prototype expressions are $pe = (id, (base, add, remove))$ and $fpe = (id, (\mathbb{P}_\emptyset, properties, \emptyset))$.

$$
\begin{aligned}
&I_{KB}(KB, pe) \\
=&FP(KB, pe) \\
=&(I_h(id), \{(I_h(p), J(KB, id, p)) \mid p \in ID, J(KB, id, p) \neq \emptyset\}) \\
\overset{\text{L4.2.1}}{=}&(I_h(id), \{(I_h(p), \{I_s(v) \mid v \in Values(KB, id, p)\}) \mid p \in ID, Values(KB, id, p) \neq \emptyset\}) \\
=&(I_h(id), \{(I_h(p), \{I_s(v) \mid v \in properties[p]\}) \mid p \in ID, properties[p] \neq \emptyset\}) \\
=&(I_h(id), \{(I_h(p), I_c(FKB, properties)(I_h(p))) \mid p \in ID, I_c(FKB, properties)(I_h(p)) \neq \emptyset\} \\
=&(I_h(id), \{(I_h(p), J(FKB, id, p)) \mid p \in ID, J(KB, id, p) \neq \emptyset\}) \\
=&FP(FKB, fpe) \\
=&I_{KB}(FKB, fpe)
\end{aligned}
$$

□

So we have shown that prototypical inheritance can be executed on a syntactical level. In contrast to inheritance the specialization relation will work on top of the inheritance free fixpoint state of a knowledge base. Furthermore, specialization will not be assigned like the base for prototypical inheritance but it is a relation that is computed based on comparing two prototypes with each other.

4.2.2 Semantics of Composed Prototypes

Given a knowledge base in its fixpoint state which contains composed prototypes as described in Section 3.2.2 we define semantic abbreviations to access the interpretations of the different prototype expressions. In the definitions below, we always assume that O, PL, I_h and FKB are defined as follows:

Let $O = (SID, OB, I_h)$ be a Prototype-Structure for the Prototype Language $PL = (\mathtt{P}_\emptyset, ID, \mathtt{PROTO})$ with I_h being a *Herbrand-Interpretation*. Let FKB be a fixpoint knowledge base for PL.

In addition to the existing interpretation of knowledge bases that maps a knowledge base and an expression to a prototype we define an additional interpretation that maps the knowledge base to the set of prototypes that correspond to the prototype expressions in the knowledge base.

Definition 4.2.5 (Knowledge Base Interpretation as Set)**.** *Let KB be a knowledge base for the Prototype Language PL. We define an interpretation of knowledge bases that maps a knowledge base to a set of prototypes:*

$I_{KBS}(KB) = \{p | pe \in KB \wedge I_{KB}(KB, pe) = p\}$

With the composed representation of prototypes a property prototype expression is used to describe each property of a prototype. To easily use this representation, we define the set of properties of a prototype to be the interpretation of the property expressions of that prototype.

Definition 4.2.6 (Properties)**.** *Let $pe = (id, (b, a, r)) \in FKB$ with $I_{KB}(FKB, pe) = p$. The function* properties *is defined by:*

$$\text{properties}(FKB, p) = \{I_{KB}(FKB, ppe) \mid ppe \in property\_expressions(FKB, id)\}.$$

There is a Herbrand-Interpretation I_h which maps every element of ID to an element of SID. For the specialization relation and the interpretation of composed prototypes we need to reference to some elements in SID that respond to special identifiers in ID. In the following, we assume that each Herbrand-Interpretation I_h includes the following mappings:

$$\begin{aligned}
I_h(\mathtt{proto:hasConstraintValue}) &= hasConstraintValue \\
I_h(\mathtt{proto:hasTypeConstraint}) &= hasTypeConstraint \\
I_h(\mathtt{proto:max}) &= max \\
I_h(\mathtt{proto:min}) &= min \\
I_h(\mathtt{proto:infty}) &= \infty \\
I_h(\mathtt{proto:allValuesFrom}) &= allValuesFrom \\
I_h(\mathtt{proto:someValuesFrom}) &= someValuesFrom \\
I_h(\mathtt{proto:hasValue}) &= hasValue \\
I_h(\mathtt{proto:hasID}) &= id \\
I_h(\mathtt{proto:hasTypeConstraint}) &= hasTypeConstraint \\
I_h(\mathtt{proto:hasCardinalityConstraint}) &= hasCardinalityConstraint
\end{aligned}$$

We introduce shorthand notations to access type and cardinality constraints. These allow to access the constraint values (either a set of SIDs or an interval of integers) of the prototype and the type of constraint.

Definition 4.2.7 (Type Constraints)**.** *Let $pce \in FKB$ be a type constraint prototype expression with $I_{KB}(FKB, pce) = pc$ and the prototype $pc = (id, propvals)$, where*

$id \in SID$ and propvals is a Value-Space for SID as defined in Definition 2.2.6. We write ***pc.cval*** *to denote the set*

$$\{v \mid (hasConstraintValue, v) \in propvals\}$$

And we define ***pc.type***=*c, where*

$$(hasTypeConstraint, c) \in propvals.$$

For cardinality constraints the interpretation will be an integer interval. The default is the interval $[0, \infty]$ and if min and max are specified differently then they define the interval.

Definition 4.2.8 (Cardinality Constraints)**.** *Let $pce \in FKB$ be a cardinality constraint prototype expression with $I_{KB}(FKB, cpe) = cp$. We write* ***cp.cval*** *to denote the interval $[l, u]$, where $cp = (id, propvals)$ and if $(min, lv) \in propvals$ and lv is an integer then $l = lv$, otherwise $l = 0$. If $(proto : max, uv) \in propvals$ and uv is an integer then $u = uv$, otherwise $u = \infty$. And we define* ***pc.type***=*cardinality.*

To access the values and constraints of a prototype, we define the sets *val* and const.

Definition 4.2.9 (Values and Constraints of a Prototype)**.** *Let $ppe \in FKB$ be a property prototype expression with $I_{KB}(FKB, ppe) = pp$. We define*

$$\begin{aligned} \text{val}(pp) =& \{v \mid pp = (id, propvals), (hasValue, v) \in propvals\} \\ \text{const}(pp) =& \{pc \mid pp = (id, propvals), (hasTypeConstraint, pc) \in propvals\} \cup \\ & \{pc \mid pp = (id, propvals), (hasCardinalityConstraint, pc) \in propvals\} \end{aligned}$$

4.2.3 Semantics of Specialization

Now that we can easily access values and constraints and their constraint values and types we can define the specialization relation.

We need to consider how to define this relation in detail. As discussed at the beginning of this section, the specialization relation will follow three desired properties:

1. Specialization by constraint satisfaction or by constraint matching.
2. Transitivity.
3. Multi-specialization.

We will now discuss how the relation has to be defined to satisfy these properties.

Details of Specialization A prototype can have different properties with different constraints. To define specialization we have to decide under which circumstances a prototype *s* is a specialization of a prototype *g*.

The example presented in Figure 16 shows multi-specialization. There, the properties of the generalization are regarded independently. Properties that are not present in the generalization can be arbitrary in the specialization. Thereby, `TheConferenceHotel` is

	g	s	s ⪯ g possible	Rule
1	{Constraints}	{Values}	✓	each constraint satisfied by values
2	{Constraints}	{Constraints}	✓	each constraint is matched
3	{Values}	{Values}	✓	equality of values
4	{Constraints}	{Constraints, Values}	✓	{Constraints}-{Constraints}, Values ⊆ Values
5	{Values}	{Constraints}	–	–
6	{Values}	{Constraints, Values}	–	–
7	{Constraints, Values}	{Values}	✓	{Constraints}-{Values}, Values ⊆ Values
8	{Constraints, Values}	{Constraints}	–	–
9	{Constraints, Values}	{Constraints, Values}	✓	{Constraints}-{Constraints}, Values ⊆ Values

Table 2: Overview of specialization rules and allowed combinations of generalization and specialization. The first three rows describe the base cases, the other rules follow from these three.

allowed to be a specialization of both the `Hotel` and the `HamburgerRestaurant` prototype. If properties that are not present in the generalization were not allowed to be arbitrary in the specialization (e.g., properties that are not named in the generalization may not occur in the specialization), then the specialization relation would be too restricted to allow other prototypes to be also a generalization. If the specialization relation between `Hotel` and `TheConferenceHotel` would forbid all other properties to be present then the property `serves` would not be allowed in the specialization. Thus, properties that do not occur in the generalization may be arbitrary in the specialization.

Each property from the generalization will be checked independently from other properties. Each constraint is meant to restrict a property and nothing more. For example, the constraints of the property `hasRoom` have no meaning for the property `hasRating`.

Thus, we will say that a prototype s is a specialization of a prototype g if all properties of g are specialized by the corresponding property with same ID in s.

There are different cases for specialization that can be distinguished. Since each property can be viewed independently, we consider only prototypes with one property here. This property can be classified as one of the following:

1. {Values}: The property has concrete value(s) but no constraints.

2. {Constraints}: The property has constraint(s) but no values.

3. {Constraints, Values}: The property has both value(s) and constraint(s).

Table 2 lists the possible combinations of the above classification for prototypes g and s. The third column describes whether or not such a combination could be a specialization of another and the column labeled “Rule” gives an overview of the rule applied to check if

s is actually a specialization of g. In the following we will discuss the decisions described in the table.

There are three base rules and the others are combinations of these. The first row describes the base case where the generalization consists of only constraints and s has only values. In this case the specialization relation should check that each constraint of the generalization is satisfied by the values of the specialization.

In the case of both prototypes having only constraints the constraints should be *matched*, that is the constraints should be either equal or stricter (for example allValuesFrom(2,3,4) would be a stricter constraint than allValuesFrom(1,2,3,4,5)). This follows the guiding principle that constraints can be made stricter by matching.

The third base case is that both s and g have only values. The values are required to be equal, otherwise the following would be possible: Given prototypes s, g, g' with $s \preceq g \preceq g'$, where s, g have only values and g' has values and constraints. Comparing s and g does not contain any information about the constraints present in g'. Therefore, if s was not required to be equal to g, s could be chosen such that it does not satisfy the constraints of g'.

In general, we will enforce values in the generalization as a kind of constraint, too. That is, if a value occurs in the generalization it also has to be present in the specialization. So in Case 4 we require that the values of the generalization are contained in the values of the specialization. In addition the constraints are handled exactly as in Rule 1. Cases 7 and 9 are similar: The values of the generalization are required to be in a subset relation and otherwise they behave as if there were no values present in the generalization.

Since we require values of the generalization to be present in the specialization, the prototypes in Case 5 and 8 can never form a specialization relation. Case 6 is excluded because allowing it would violate transitivity: Given prototypes s, g, g' with $s \preceq g \preceq g'$, where s has constraints and values, and g has only values (as in Case 6) and g' has values and constraints. Again, as in Case 3, comparing s and g does not contain any information about the constraints present in g', therefore if s is allowed to introduce new constraint they could be chosen such that they do not match the constraints of g'.

In summary, we decide to define the specialization relation as follows: For each property present in the generalization, the specialization has to check for specialization according to the cases classified in Table 2.

Each constraint can be viewed independently. A constraint is satisfied if the values of the specialization satisfy the constraint condition. A constraint is matched if there is a constraint in the specialization with the same type that has an equal or stricter constraint value. In the case of sets this means a subset, in the case of intervals, stricter means a subinterval. We discussed, that only the other cases can be handled as a combination of the first three rules. Notice, that the cases can be clustered as follows: If g has only values then only Case 4 is possible. So in this case we can simply check if s and g are equal. If they are equal then s cannot have constraints since g has none. In the other cases we can always check that the values of s are a superset of the values of g (either this needs to be checked or the values of g are an empty set and thus the values s also are a superset of it). We need then to distinguish whether we have to additionally check Case 1 or 2 (satisfaction or matching).

Defining the specialization relation like this does allow for multi-specialization and transitivity (the latter will be proved in Section 4.2).

Specialization Definition The definition is arranged from top to bottom as follows: first prototypes are compared, then single properties are compared. If there are constraints in the specialization property, then it is checked if all constraints are matched. Otherwise, if there are no constraints in the specialization, the specializing property has to satisfy the constraints of the generalization. A constraint matches another if it is of same type and the constraining values are a subset of the other. In case of cardinality constraints the subset relation is meant to be the subset on intervals, while for type constraints it is the subset relation on sets. Satisfaction is checked according to Definition 3.1.1.

Definition 4.2.10 (Specialization Relation)**.** *Let $IKB = I_{KBS}(FKB)$ be the set of prototypes in the interpretation of the knowledge base FKB. Given prototypes $s, g \in IKB$. We say that s is a specialization of g, written:*

$$s \preceq g \text{ iff } \forall G \in \text{properties}(FKB, g) : \exists S \in \text{properties}(FKB, s) : \\ G.id = S.id \text{ and } S \preceq G$$

$$S \preceq G \text{ iff } \begin{cases} \text{val}(S) \supseteq \text{val}(G) \wedge \text{accountFor}(S, \text{const}(G)) & \text{if const}(G) \neq \emptyset \\ S = G & \text{if const}(G) = \emptyset \end{cases}$$

$$\text{accountFor}(S, GC) = \begin{cases} \forall gc \in GC : \text{isMatched}(\text{const}(S), gc) & \text{if const}(S) \neq \emptyset \\ \forall gc \in GC : \text{isSatisfied}(\text{val}(S), gc) & \text{if const}(S) = \emptyset \end{cases}$$

$$\text{isMatched}(SC, gc) = \exists sc \in SC : \text{matches}(sc, gc)$$

$$\text{matches}(sc, gc) = sc.type = gc.type \wedge sc.cval \subseteq gc.cval$$

$$\text{isSatisfied}(VS, gc) = \begin{cases} \forall v \in VS : v \in gc.cval & \text{if } gc.type = allValFrom \\ \exists v \in VS : v \in gc.cval & \text{if } gc.type = someValFrom \\ |VS| \in gc.cval & \text{if } gc.type = cardinality \end{cases}$$

We call two prototypes equivalent with regard to the specialization relation if they are a specialization as well as a generalization of the other.

Definition 4.2.11 (Equivalence w.r.t. Specialization)**.** *Two prototypes x, y are called equivalent with regard to specialization (written $x \simeq y$) iff*

$$x \preceq y \wedge x \succeq y$$

Similarly, two property prototypes X, Y are called equivalent w.r.t specialization (written $X \simeq Y$) iff

$$X \preceq Y \wedge X \succeq Y$$

4.3 Properties

In this section, properties arising from the definition of the specialization relation are discussed. First, it is shown that the specialization relation is transitive. Then it is discussed that the number of different specializations possible reduces with strict specialization.

4.3.1 Transitivity

We will show that the specialization relation is transitive. Since the relation symbol "$\preceq$" is defined on both prototypes as well as properties, we will first show that transitivity holds for property prototypes and then use this to show transitivity of the specialization relation regarding prototypes.

Lemma 4.3.1. *The specialization relation $\preceq$ is transitive with regard to property prototypes.*

The general proof idea is to assume that $X \preceq Y$ and $Y \preceq Z$ and to shown then that $X \preceq Z$ is also true. When Y has no constraints, it has to be equal to X, and is thereby a specialization of Z. Otherwise, further distinguishing whether X has constraints decides whether the constraints of Y are matched or satisfied. It then has to be shown that X also matches, respectively satisfies the constraints of Z. Which it intuitively should, because matching is based on a subset relation. Since Y has constraints it will match Z and thus in the satisfaction case the values of X also have to satisfy a superset of the constraints of Y. Formally, this is proven as follows:

Proof. Let X, Y, Z be property prototypes with $X \preceq Y$ and $Y \preceq Z$.

Case 1: $\text{const}(Y) = \emptyset$.
By $X \preceq Y$, $X = Y$. Thus, $Y \preceq Z$

Case 2: $\text{const}(Y) \neq \emptyset$.
From this it follows that $\text{const}(Z) \neq \emptyset$ because otherwise $Y = Z$ but then $\text{const}(Z) \neq \text{const}(Y)$. Therefore, by $Y \preceq Z$ it holds that both $\text{accountFor}(Y, \text{const}(Z))$ and $\text{accountFor}(Y, \text{const}(Z))$ are true because the following formula holds:
$\forall zc \in \text{const}(Z) : \text{isMatched}(\text{const}(Y), zc)$.

By $X \preceq Y$ and $Y \neq \emptyset$ it holds that $\text{accountFor}(X, \text{const}(Y))$.

Case 2.1: $\text{const}(X) = \emptyset$.
Since $\forall yc \in \text{const}(Y) : \text{isSatisfied}(\text{val}(X), yc)$ is true it follows that $\text{accountFor}(X, \text{const}(Y))$ is also true.

Let $zc \in \text{const}(Z)$. Since $\text{isMatched}(\text{const}(Y), zc)$ is true, there exists a $yc \in \text{const}(Y) : \text{matches}(yc, zc)$, that is $yc.type = zc.type$ and $yc.cval \subseteq zc.cval$. Since $\text{isSatisfied}(\text{val}(X), yc)$ is true and $yc.cval \subseteq zc.cval$, it also holds that $\text{isSatisfied}(\text{val}(X), zc)$ is true. Thus, $\text{accountFor}(X, \text{const}(Z))$ is true.

Case 2.2: $\text{const}(X) \neq \emptyset$.
Then $\text{accountFor}(X, \text{const}(Y))$ is true because it holds that $\forall yc \in \text{const}(Y) : \text{isMatched}(\text{const}(X), yc)$ is true.

Let $zc \in \text{const}(Z)$. Since $\text{isMatched}(\text{const}(Y), zc)$ is true, there exists a $yc \in \text{const}(Y) : \text{matches}(yc, zc)$, that is $yc.type = zc.type$ and $yc.cval \subseteq zc.cval$. Since $\text{isMatched}(\text{const}(X), yc)$ is true there exists a $xc \in \text{const}(X) : \text{matches}(xc, yc)$. By $yc.cval \subseteq zc.cval$, $yc.type = zc.type$ and $xc.cval \subseteq yc.cval$, $xy.type = yc.type$, it also holds that $matched(xc, zc)$ is true. Thus, $\text{isMatched}(\text{const}(X), zc)$ is true and thereby also $\text{accountFor}(X, \text{const}(Z))$.

It remains to be shown that $\text{val}(X) \supseteq \text{val}(Z)$. Since $Y \preceq Z$ it holds that $\text{val}(Y) \supseteq \text{val}(Z)$. And with $X \preceq Y$ it holds that $\text{val}(X) \supseteq \text{val}(Y)$. By the transitivity of "$\supseteq$" it follows that $\text{val}(X) \supseteq \text{val}(Z)$. Thus, $X \preceq Z$.

□

With the transitivity of the specialization for property prototypes it can be directly extended to prototypes.

Theorem 4.3.1. *The specialization relation $\preceq$ is transitive with regard to prototypes.*

Proof. Let FKB be a fixpoint knowledge base with $I_{KBS}(FKB) = IKB$. Let $x, y, z \in IKB$ be prototypes with $x \preceq y$ and $y \preceq z$.

Since $y \preceq z$ it holds that: $\forall Z \in \text{properties}(FKB, z) : \exists Y \in \text{properties}(FKB, y) : Y.id = Z.id \wedge Y \preceq Z$. By $x \preceq y$ there must exist a $X \in \text{properties}(FKB, x)$ s.t. $X.id = Y.id$ and $X \preceq Y$ for all such Y.

By transitivity of "$\preceq$" on property prototypes it follows that $X \preceq Z$. Thus: $\forall Z \in \text{properties}(FKB, z) : \exists X \in \text{properties}(FKB, x) : X.id = Z.id \wedge X \preceq Z$ and thereby $x \preceq z$. □

4.3.2 Specialization Reduces the Number of Further Specializations

We want to show that specialization reduces the number of possible further specializations. Therefore, we need to distinguish property prototypes that are in a specialization relation but that are not equal to one another with regard to the specialization relation. Thus, we define strict specialization based on equivalence of prototypes with regard to specialization as defined in Definition 4.2.11. Then it is shown that strict specialization reduces the number of possible specializations.

Definition 4.3.1 (Strict Specialization)**.** *Let s, g be prototypes. We say s is a* strict specialization *of g (written $s \prec g$) if and only if $s \preceq g$ and $s \not\approx g$. The definition for property prototypes is identical.*

It can now be shown that strict specialization reduces the number of possible specializations. Figure 20 shows the idea of the theorem: Given prototypes g_1, g_2 and g_2 is a strict specialization of g_1. Then it is to be shown that the set of all possible specializations of g_2, namely S_2, is a proper subset of S_1 (all possible specializations of g_1).

$$\begin{array}{ccc} g_1 & \succ & g_2 \\ \curlyvee & & \curlyvee \\ S_1 & \supsetneq & S_2 \end{array}$$

Figure 20: Specialization reduces the number of possible further specializations

Theorem 4.3.2. *Let g_1, g_2 be prototypes with $g_1 \succ g_2$. Let $S_1 = \{s \mid s \preceq g_1\}$, $S_2 = \{s \mid s \preceq g_2\}$ be sets of prototypes. It follows that $S_1 \supsetneq S_2$*

Notice, that the sets S_1 and S_2 are infinite, since properties not occuring in g_1 or g_2 respectively can be present in arbitrary ways.

Proof. Let $s \in S_2$. By definition of S_2, $s \preceq g_2$. By transitivity of $s \preceq g_2 \preceq g_1$ it follows that $s \preceq g_1$ and thus $s \in S_1$. Therefore $S_1 \supseteq S_2$.

To show that $S_1 \supsetneq S_2$ it remains to show that S_1 and S_2 are not equal. Thus, it suffices to show that there exists an element that is in S_1 but not in S_2. By definition $g_1 \in S_1$. We will show that $g_1 \notin S_2$: Since $g_1 \succ g_2$ it follows that $g_1 \not\approx g_2$. Thus, it has to hold that $g_1 \succeq g_2$ and $g_1 \not\preceq g_2$. By the definition of S_2, each element in S_2 has to be a specialization of g_2. Therefore $g_1 \notin S_2$. □

4.3.3 Consistent Prototypes

The definition of the specialization relation only checks if the values of the specialization satisfy the constraints once there are no constraints in the specialization. Thus, it can happen that in a chain of specializations the combination of constraints become unsatisfiable. We would like to detect this property and call a prototype *inconsistent* if its constraints cannot be satisfied.

Since partial value definitions are allowed, we cannot simply check whether a prototype satisfies its own constraints. For example, the `BigLodging` in Figure 19 specified that the property `hasRoom` has at least 20 different values. In addition, it fixes that one of these values has to be `RoomNo42`. If we were to check whether the prototype satisfies its own constraints this would result in false (nonetheless, by definition it is a specialization of itself). It is also allowed for another prototype to specialize `BigLodging` by keeping the cardinality constraint and adding another specified room. Still, that prototype would neither satisfy its own constraint nor the constraint of `BigLodging`.

While these partial definitions are useful and can be satisfied, it is also possible to define combinations of constraints, or of constraints and values that are not satisfiable. For example, it could be required that all values are from the set $\{a, b\}$ and another constraint could say that all values need to be from $\{c, d\}$. While this is easily detectable, there are also more complicated cases. Basically, the question is whether the combined constraints are satisfiable. This question can be expressed in terms of the specialization relation. Since constraint satisfaction is checked once the specialization has no constraints this can be used to answer the satisfiability question. First, we name prototypes *fully specialized* if they have no constraints. Notice, that by the definition of the specialization relation they can be further specialized by introducing new properties. However, in terms of the properties present in such a prototype, they are fixed because the property prototypes are required to be equal in this case.

Definition 4.3.2 (Fully Specialized)**.** *Given a knowledge base KB with $I_{KBS}(KB) = IKB$. A prototype $p \in IKB$ is called* fully specialized *if its properties only contain values and no constraints, i.e.:*

$$\forall pp \in \text{properties}(KB, p) : \text{const}(pp) = \emptyset$$

Consistency can then be formulated as the existence of a specialization that is a fully specialized prototype.

Definition 4.3.3 (Consistency)**.** *A prototype g is called* consistent *if there exists a prototype s with $s \preceq g$ and s is fully specialized.*

This definition of consistency is not easily reducible to conditions that can be checked only on a prototype g. It is easy to define some necessary conditions that each constraint

of g has to fulfill to be consistent. For example, a single constraint may not be defined in such a way that it is not satisfiable, e.g., a cardinality constraint with lower limit 5 and upper limit 3 is not satisfiable. However, these simple conditions are not sufficient because the combination of different constraints may lead to inconsistency.

Checking whether g is consistent is equivalent to the satisfiability of the combined constraints of each property. It would be possible to transform the constraints and values to first-order logic and check whether there exists a model for the formula. However, this problem is as hard as checking the existence of a prototype according to Definition 4.3.3.

5 Implementation

This section discusses the implementation of the specialization relation[3]. The prototype system presented in [6] has been implemented in Java[4]. The implementation details of that implementation are described in [21]. While that implementation is intended to be production ready the implementation of the specialization relation given here is exploratory in nature. The intend of the implementation is to show that the presented approach is feasible.

As language for the implementation Haskell was chosen because it is able to neatly reassemble the defined formulas of the semantics. Furthermore, it allows quick modifications and is thereby especially useful for an exploratory, extendable first implementation. In the following main excerpts of the implementation are presented.

```
1  -- Data Types for Syntax
2  data IRI = ID String
3  data Property = Prop IRI
4  data Bases = Base IRI | P0
5
6  data SimpleChangeExpression = Change Property (Set IRI)
7
8  data PrototypeExpression = Proto {
9    idIri :: IRI,
10   base :: Bases,
11   add :: Set SimpleChangeExpression,
12   remove :: Set SimpleChangeExpression,
13   remAll :: Set Property}
14
15 type KnowledgeBase = Map IRI PrototypeExpression
16
17 -- Data Types for Fixpoint Knowledge Base / Semantics
18 type PropertyMap = Map Property (Set IRI)
19 data Prototype = PT {name :: IRI, props :: PropertyMap}
20 type FixpointKB propValueType = Map IRI Prototype
```

Listing 1: Data structure to represent prototypes

[3]The source code is available at: `https://github.com/ggierse/kr-prototype`

[4]`https://github.com/miselico/knowledgebase`

Listing 1 shows the data structures used in the implementation. For readability automatic Haskell typeclass assertions have been left out. An IRI is defined to be a string preceded by `ID` to identify it as such. In a production ready version this would need to conform to RFC 3987 [23] but for exploration a string will suffice. Properties consists of IRIs. A Base is an IRI or a special token `P0` which represents the empty prototype $P_{\emptyset}$.

A `SimpleChangeExpression` consists of a property and a set of IRIs. Based on the previous definitions, a `PrototypeExpression` is defined. It has an IRI as ID, a base and sets of simple change expressions that are to be added or removed (`remAll` is used to remove everything of one property). A `KnowledgeBase` is defined as a mapping from IRIs to prototype expressions.

The data types defined below Line 17 are used to represent fixpoint knowledge bases that already computed their inheritance relations. In Definition 4.2.3 a fixpoint state of a knowledge base is defined on a syntactical level and uses prototype expressions as basis. As discussed, this is also closely related to the semantics of a prototype expression. To be able to easily distinguish the two kinds of knowledge bases and for efficiency reasons different data types are used here. Because the base will always be `P0` and `remove` and `remAll` will always be empty sets a representation which only represents the `add` set is used. Since these are the properties that a prototype actually has the data structure to represent this is called `PropertyMap`. Instead of using a set of properties and IRIs a mapping between them is used. A prototype is then represented with an IRI as name and a property map. A `FixpointKB` is a mapping between IRIs and prototypes.

Listing 2 shows the internal representation of constraints. Constraints and values are represented as discussed in Section 3.2. In terms of the semantics the constraints are internally represented by the data structure `ConstraintInfo` shown in Lines 3 to 9 from Listing 2.

```
1  data ConstraintName = AllValuesFrom | SomeValuesFrom | Cardinality
2
3  data ConstraintInfo =  TypeConst {
4    constType :: ConstraintName,
5    constValues :: Set IRI
6  } | CardinalityConst {
7    constType :: ConstraintName,
8    constInterval :: IntegerInterval
9  }
10
11 val :: Prototype -> Set IRI
12 -- ... left out for brevity
13
14 consts :: FixpointKB -> Prototype -> Set ConstraintInfo
15 -- ... left out for brevity
```

Listing 2: Data structure for internal representation of constraints

A constraint is either a `TypeConst` or a `CardinalityConst`. Both have a `constType` which is simply the name of the constraint type. Type constraints have a set of IRIs as values of the constraints while a cardinality constraint has an integer interval as constraining element. The values of a property prototype are returned by the function `val`. In addi-

tion, the constraints of a property prototype are looked up and transformed into a set of `ConstraintInfo` by the function `consts`. The details of their implementation are left out for brevity. The function simply looks up the prototypes of the constraints and transforms the there found properties and values into the `ConstraintInfo` data structure.

The specialization relation is implemented to closely reassemble the Definition 4.2.10. In the following, all parts of the definition are compared with their implementation.

To check whether two prototypes are in a specialization relation a function named `isSpecializationOf` is called. In Definition 4.2.10 this part is defined as follows:

$$s \preceq g \text{ iff } \forall G \in \text{properties}(FKB, g) : \exists S \in \text{properties}(FKB, s) : \\ G.id = S.id \text{ and } S \preceq G$$

This is exactly repeated in Haskell:

```
1 isSpecializationOf :: FixpointKB -> Prototype -> Prototype -> Bool
2 isSpecializationOf fkb special general =
3   forall existsSpecial gprops
4   where gprops = properties fkb general
5     sprops = properties fkb special
6     existsSpecial g = exists
7       (\s -> propertyIdIsEqual s g && isPropertySpecialization fkb s g)
8       sprops
```

Listing 3: Computation of $s \preceq g$, where s, g are prototypes

The functions `forall` and `exists` check whether all, respectively at least one value in a set matches a given condition. The function `isSpecializationOf` checks whether for all properties of the generalization there exists a specialization. A specialization exists if there is an `s` in `sprops` (the properties of the specialization) such that the ID of the properties are equal and they are a specialization of one another.

To compute whether something is a property specialization of another property the function `isPropertySpecialization` is used:

```
1 isPropertySpecialization :: FixpointKB -> Prototype -> Prototype ->Bool
2 isPropertySpecialization fkb s g
3   | not (Set.null gConsts) =
4     val g `Set.isSubsetOf` val s && accountFor fkb s gConsts
5   | otherwise = isPropertyProtoEqual s g
6   where gConsts = consts fkb g
```

Listing 4: Computation of $S \preceq G$, where S, G are property prototypes

Haskell allows to write case distinctions with "|" followed by the condition and what to do in that case after the equality sign. Notice again how the Haskell definition closely connects to the respective part in Definition 4.2.10:

$$S \preceq G \text{ iff } \begin{cases} \text{val}(S) \supseteq \text{val}(G) \text{ and accountFor}(S, \text{const}(G)) & \text{if const}(G) \neq \emptyset \\ S = G & \text{if const}(G) = \emptyset \end{cases}$$

Similar, the `accountFor` function reassembles the formal definition:

$$\text{accountFor}(S, GC) = \begin{cases} \forall gc \in GC : \text{isMatched}(\text{const}(S), gc) & \text{if const}(S) \neq \emptyset \\ \forall gc \in GC : \text{isSatisfied}(\text{val}(S), gc) & \text{if const}(S) = \emptyset \end{cases}$$

```
1 accountFor :: FixpointKB -> Prototype -> Set ConstraintInfo -> Bool
2 accountFor fkb s gConsts
3   | Set.null sConsts = forall (isSatisfied $ val s) gConsts
4   | otherwise = forall (isMatched sConsts) gConsts
5   where sConsts = consts fkb s
```

Listing 5: Computation of accountFor(S, GC), where S is a property prototype and GC is a set of constraints

In the source code there is again a case distinction and then the different cases are exactly applied as in the function.

The implementation of `isMatched` differs a little from the formulas:

$$\text{isMatched}(SC, gc) = \exists sc \in SC : \text{matches}(sc, gc)$$
$$\text{matches}(sc, gc) = sc.type = gc.type \wedge sc.cval \subseteq gc.cval$$

```
1  isMatched :: Foldable t => t ConstraintInfo -> ConstraintInfo -> Bool
2  isMatched sConsts gc =
3    case gc of
4      CardinalityConst _ _ ->
5        exists (\sc -> constType sc == constType gc
6          && constInterval sc `Interval.isSubsetOf` constInterval gc)
7          sConsts
8      TypeConst _ _ ->
9        exists (\sc -> constType sc == constType gc
10         && constValues sc `Set.isSubsetOf` constValues gc)
11         sConsts
```

Listing 6: Computation of isMatched(SC, gc), where SC is a set of constraints and gc is a constraint

Since computing the subset of an actual set and an integer interval are different in Haskell, it needs to be distinguished whether the constraint is a cardinality constraint or a type constraint. Since the data structures use different constructors in these cases, Haskell can do a case distinction based on which constructor is present. Except distinguishing these cases the same conditions as in the formula are checked, namely that there exists a constraint of the specialization property that has the same type and is a subset, respectively, subinterval.

Finally, the implementation of `isSatisfied` is identical to its formal definition:

$$\text{isSatisfied}(VS, gc) = \begin{cases} \forall v \in VS : v \in gc.cval & \text{if } gc.type = allValFrom \\ \exists v \in VS : v \in gc.cval & \text{if } gc.type = someValFrom \\ |VS| \in gc.cval & \text{if } gc.type = cardinality \end{cases}$$

```
1 isSatisfied :: Set IRI -> ConstraintInfo -> Bool
2 isSatisfied sVals gc =
3   case constType gc of
4     AllValuesFrom ->
5       forall (\ v -> v `Set.member` constValues gc) sVals
6     SomeValuesFrom ->
7       exists (\ v -> v `Set.member` constValues gc) sVals
8     Cardinality ->
9       toInteger (Set.size sVals) `Interval.member` constInterval gc
```

Listing 7: Computation of isSatisfied(VS, gc), where VS is a set IRIs and gc is a constraint

In conclusion, Haskell is a good choice to model the specialization relation. The close similarity between definitions and program code is a major benefit, which made it easy to adapt the implementation to modifications in the definitions during the thesis. To show that the implementation is feasible, an analysis of the time complexity is presented.

5.1 Complexity Analysis

In this section, the time complexity of the function `isSpecializationOf` (Listing 3) and the function it calls is analyzed. The complexity depends on a multitude of variables, for example prototypes need to be found in the knowledge base, thus the size of the knowledge base (i.e., the number of prototypes in the `KnowledgeBase` data type) is one constraint. In addition, the number of properties of the generalization and specialization is relevant because the function `isSpecializationOf` iterates over both. Another example would be the number of constraint values of each constraint in each property that needs to be accessed. The function `isMatched` needs to compute a subset on constraint values and the complexity of function `isSatisfied` also depends on checking membership in the set of constraint values.

While it is possible to analyze the complexity with regard to many specific different variables it is not practical. Fortunately, all these variables are bound by the size of the knowledge base: Because each property and each constraint and each value of a property or value of a constraint is a prototype by itself it has to occur in the knowledge base. Thus, it is not possible for any of these variables to grow beyond the size of the knowledge base. In the following kb will denote the size of the knowledge base `fkb` (it is a `Map` containing entries mapping form ID to prototype expression for each prototype in the knowledge base). We start with the most basic functions first and then proceed to the functions that call these.

The function `isSatisfied` shown in Listing 7 has three branches. Regarding complexity the cardinality constraint can be computed in constant time because only the upper and lower bounds need to be compared. In both other cases the membership of a set has to be computed. This can be done in $\mathcal{O}(\log n)$, where n is the size of the set. In the worst-case this needs to be done once for all elements in `sVal` (the values of a property of the specialization). Therefore, the worst-case runtime with regard to the size of the knowledge base is: $\mathcal{O}(\log kb \cdot kb)$.

The alternative function called by `accountFor` (Listing 5) is `isMatched` (Listing 6) . That functions worst-case runtime is also occurring if the constraint is a type constraint.

There, for each element in `sConsts` a subset relation has to be calculated. Determining whether two things are a subset of another can be done in $\mathcal{O}(n+m)$, where n and m are the sizes of the two sets. In our case we use the size of the knowledge base as upper bound, therefore determining the subset is in $\mathcal{O}(kb)$. Doing this for each element in `sConsts` results in the worst-case runtime $\mathcal{O}(kb^2)$ for `isMatched`.

The `accountFor` function (Listing 5) also has two cases which differ in complexity. In one case `isSatisfied` is called on all `gConsts` (Line 3 in Listing 5) and in the other it is `isMatched` (Line 4 in Listing 5). Since the runtime of `isMatched` is higher than of `isSatisfied` only this case will be considered. The constraints of a property of the generalization (`gConsts`) are at most of size kb. So, the complexity of `isMatched` gets multiplied by the size of the knowledge base because the `forall` function applies it to all elements in `gConsts`. Thus, the worst-case runtime for `accountFor` is $\mathcal{O}(kb^3)$. Notice, that other parts of the function do not play a major role. The retrieval of `sConsts` (Line 5 in Listing 5) has only a complexity of $\mathcal{O}(\log^2 kb)$ which is only computed once and thus added to the complexity.

Then the worst-case complexity of `isPropertySpecialization` (Listing 4) depends on computing the subset of the values of s and g (complexity: $\mathcal{O}(kb)$) and calculating `accountFor` (complexity: $\mathcal{O}(kb^3)$). Since these two complexities are added the overall complexity for this function is $\mathcal{O}(kb^3)$.

The computation of the `isSpecializationOf` function (Listing 3) uses the `forall` and the `exists` function. Computing `existsSpecial` computes for each property of the specialization the function `propertyIdIsEqual` and `isPropertySpecialization` (Line 6-8 in Listing 3). Computing whether the property ids are equal requires a lookup of the property in the knowledge base (complexity: $\mathcal{O}(\log kb)$). This is of lower complexity than calling the function `isPropertySpecialization`. Thus, the complexity of the function described in Line 7 of Listing 3 is the complexity of `isPropertySpecialization`, namely $\mathcal{O}(kb^3)$. This function is possibly called on each element of the knowledge base. Thus, the `existsSpecial` function has complexity $\mathcal{O}(kb^4)$. Retrieving the properties of the specialization and the generalization (Line 4 and 5 in Listing 3) is done only once and therefore only adds a term with lower exponent to the complexity, which can be ignored. The function `existsSpecial` is then called upon all properties of the generalization by the `forall` function. Therefore, the overall worst-case complexity of `isSpecializationOf` is $\mathcal{O}(kb^5)$.

In summary, determining whether one prototype is a specialization of another has polynomial runtime in the size of the knowledge base. The complexity of this algorithm is not necessarily a tight upper bound for the complexity of the problem. On the one hand, the implementation presented here is a test of feasibility and is not optimized. Thus, the runtime could be improved by optimization. On the other hand, the number of properties, constraints and values will in practice be a fragment of the size of the knowledge base. Only the lookup of properties and constraint prototypes depends on the size of the knowledge base. All other operations that increase the complexity depend on the number of properties, constraints or values. For example, the function `isSpecializationOf` iterates over all properties of the generalization and for each of these it iterates over all properties of the specialization. This accounts for the factor kb^2 in the overall complexity. In practice this part only depends on the number of properties. Furthermore, other parts depend only on the number of constraints or values.

6 Future Work

The specialization relation presented in this thesis is a first exploratory approach to knowledge representation in prototypical systems for the Semantic Web. There are many advances possible and necessary to increase the expressiveness of the constraint system in order to make it suitable for the Semantic Web movement. To integrate all these mechanisms was beyond the scope of this thesis. However, in the following, important extensions are presented and the benefits and complications arising with integrating them are discussed.

6.1 Data Types and Ranges

Currently, the prototype system does not feature a notion of data types. The implementation presented in [21] has a predefined knowledge base which contains literals to represent data types. However, it is not a full type system. Adding more semantic to data types would allow to define constraints based on data types. For example, numbers could be constrained to be within a range or strings could be required to fit a regular expression. Discussing the best possible representation of data types is left open. Range constraints can be defined like cardinality constraints. The difference is that cardinality constraints restrict the number of property values while range constraints would restrict the values that each property value can have. Many other constraint systems allow checking for regular expressions, for example [29, 9].

6.2 Recursion

In the presented system, the type constraints are not recursive. The constraint values are a set of IRIs and the constraint is satisfied if the specialization has exactly the same IRI as values.

It would advance expressiveness drastically, if one could specify that as a satisfying value everything that is a specialization of another prototype is allowed. For example, in OWL it is possible to express $\exists$`hasChildren.Doctor`. This means that there exists a child who is a doctor. Notice, how `Doctor` references to a class in OWL. Similarly, the specialization relation could specify that for the relation `hasChildren` there should be `someValuesFrom(Doctor)`, where `Doctor` is another prototype and meaning that each specialization of that prototype is an allowed filler for this constraint.

There are different possibilities to represent this. We could say that we only have the specialization of kind of values and that it is no longer possible to say that a value should be exactly the same IRI. To keep and extend expressiveness, a distinction between fixed IRIs and specializable IRIs can be made. For instance, one possible representation would include three properties of a type constraint, namely, the type, the constraint values (these would be fixed IRIs) and the specializable constraint values.

In the following we will assume this notion and define abbreviations as follows:

Definition 6.2.1. *Given a fixpoint knowledge base FKB and a type constraint prototype expression pce with $I_{KB}(FKB, pce) = pc$. We write $pc.type$ to denote the constraint type, $pc.cval$ to denote the set of fixed constraint values and $pc.cspecs$ to denote the set of specializable constraints values.*

Specializable constraint values are only relevant for type constraints, as cardinality constraints restrict only the number of values.

With this definition, we extend the definition of isSatisfied to also check for constraint values that may be specialized. Intuitively, we want to check whether a value is in the set of concrete IRIs from the fixed constraint values. If this is not the case, it is checked if in addition the value is a specialization of the specializable constraint values:

$$\text{isSatisfied}(VS, gc) = \begin{cases} \forall v \in VS : (v \in gc.cval & \text{if } gc.type = \\ \quad \vee \exists cs \in gc.cspecs : v \preceq cs) & \quad allValFrom \\ \exists v \in VS : (v \in gc.cval & \text{if } gc.type = \\ \quad \vee \exists cs \in gc.cspecs : v \preceq cs) & \quad someValFrom \\ |VS| \in gc.cval & \text{if } gc.type = \\ & \quad cardinality \end{cases}$$

In addition, isMatched should be adapted. Currently, a constraint matches another if the type is the same and the constraint values are a subset of another. As a minimum modification, it should be defined that the constraint specializations can also be matched by subset. In addition, it seems logical to allow a specialization of a specializable constraint value to replace the generalization. The definition would then look as follows:

$$\begin{aligned} \text{isMatched}(SC, gc) =& \exists sc \in SC : \text{matches}(sc, gc) \\ \text{matches}(sc, gc) =& sc.type = gc.type \wedge sc.cval \subseteq gc.cval \\ & \wedge \text{SpecsMatch}(sc.cspecs, gc.cspecs) \\ \text{SpecsMatch}(sspecs, gspecs) =& sspecs \subseteq gspecs \vee \forall s \in sspecs : \exists g \in gspecs : s \preceq g \end{aligned}$$

When integrating this feature there might be cyclic recursion in checking the specialization. This could lead to non-termination. To address this cycles would need to be detected. If satisfaction or matching is violated anywhere within a cyclic definition then the whole cycle does not form a specialization relation. In contrast, if no violation can be found within the cycle it means that everything is consistent as long as the other elements in the cycle are also consistent. In this case the relation should be accepted as specialization and the computation stopped as soon as a prototype is checked again.

Nonetheless the worst-case-runtime for computing specialization would be a lot worse. When computing isSatisfied we could have to check specialization for the whole knowledge base multiple times. Depending on the knowledge base and the amount of recursion this could be an exponential blowup. Still, in many practical cases the runtime would be feasible and the gained simplicity in expressing complex concepts would be useful.

6.3 Boolean Operators

With the current formalization, constraints are combined with an “and” if there are multiple constraints for one property. For functional completeness, negation needs to be introduced. To ease constraint definition “or” should also be added.

Introducing this would yield two challenges. On one hand the representation needs to be modified. Negation has to be introduced but more importantly the combination of constraints together needs to be represented. This will pose a further blow-up of the

complexity of the representation in terms of prototypes or the need to introduce extra syntax for formulating constraints.

On the other hand, combining negation and recursion can lead to problems. For example consider the following example modified from [30]:

$$\begin{aligned} &\text{properties}(g1) = \{L1\} && L1.id = \texttt{ex:p} \\ &\text{properties}(g2) = \{L2\} && L2.id = \texttt{ex:p} \\ &\text{const}(L1) = \{\neg L2\} && n1\ \texttt{ex:p}\ n2 \\ &\text{const}(L2) = \{\neg L1\} && n2\ \texttt{ex:p}\ n1, \end{aligned}$$

where $n1$ and $n2$ are represented shortly but are meant to be composed of property prototypes. Here we have prototypes $g1$ and $g2$, which both have only one property that has the same id (`ex:p`) and one constraint for that property. The constraint defines that specializing values of $L1$ are not a specialization of $L2$. The reverse is said about $L2$. The question is, whether $n1 \preceq g1$ or $n2 \preceq g2$. Both should not be true at the same time because they are contradictory. Trying to conclude what is true leads to a circle. This circle could be detected and the answer could be *undefined*. In their paper Boneva et al. ([30]) propose to use stratified negation to forbid these kinds of negation. With stratified negation there needs to exist an ordering of negations where two things on the same layer may not depend on the negation of another (as is the case in the above example).

Methods like this can be introduced to the system. This would further increase the expressiveness at the cost of complexity.

7 Conclusion

The Semantic Web movement defines ways to share semantic data in a machine-readable format. Semantic Web languages such as OWL feature knowledge representation primitives to model class hierarchies. In addition, different systems exist that can be used to check the integrity of data on the Semantic Web. To offer more possibilities for sharing, a prototypical approach has been presented in [6]. While the prototypical approach eases reuse and sharing it cannot model meaningful hierarchies and perform integrity checks.

This thesis enhances the prototypical approach with hierarchical modeling capabilities and the possibility to perform integrity checks by introducing an additional *specialization relation*. The relation is based on constraint satisfaction. For a first exploration, a small but expressive set of constraints on properties is chosen. Different possible representations are compared. The selected representation is fully composed and thereby follows the major design pattern of the prototypical approach: It allows to fully reuse and share properties and constraints.

The specialization relation is defined on prototypes after inheritance. In contrast to inheritance it is not declarative, but *observational*. A prototype s is a specialization of a prototype g if s either *satisfies* the constraints of g, or s *matches* the constraints of g in such a way that they are stricter than those of g. The relation allows *multi-specialization* and is *transitive*.

An exploratory implementation is presented. It shows that the relation can be computed in polynomial runtime with regard to the size of the knowledge base.

As first exploration of such a relation, this thesis opens up numerous opportunities for future research (as discussed in Section 6).

8 References

[1] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific American*, 284(5), 2001.

[2] Grigoris Antoniou and Frank Van Harmelen. *A semantic web primer*. MIT Press, 2008.

[3] Markus Lanthaler, David Wood, and Richard Cyganiak. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[4] Frank van Harmelen and Deborah McGuinness. OWL web ontology language overview. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[5] Peter Patel-Schneider, Bernardo Cuenca Grau, and Boris Motik. OWL 2 web ontology language direct semantics (second edition). W3C recommendation, W3C, December 2012. http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/.

[6] Michael Cochez, Stefan Decker, and Eric Prud'hommeaux. Knowledge Representation on the Web revisited: the Case for Prototypes. In *International Semantic Web Conference (ISWC)*. Springer, 2016.

[7] Antero Taivalsaari. Classes vs. Prototypes - Some Philosophical and Historical Observations. In *Journal of Object-Oriented Programming*. Springer, 1996.

[8] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L. McGuinness. Integrity Constraints in OWL. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. AAAI Press, 2010.

[9] Slawek Staworko, Iovka Boneva, Jose E. Labra Gayo, Samuel Hym, Eric G. Prud'hommeaux, and Harold Solbrig. Complexity and Expressiveness of ShEx for RDF. In *18th International Conference on Database Theory*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[10] Ramanathan Guha and Dan Brickley. RDF schema 1.1. W3C Recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[11] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[12] Ian Horrocks, Peter F. Patel-Schneider, and Frank Van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1), 2003.

[13] Thomas Hofweber. Logic and Ontology. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2014 edition, 2014.

[14] Eric Margolis and Stephen Laurence. Concepts. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2014 edition, 2014.

[15] Eleanor Rosch. Cognitive representations of semantic categories. *Journal of Experimental Psychology: General*, 104(3), 1975.

[16] Antero Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3), September 1996.

[17] Christophe Dony, Jacques Malenfant, and Daniel Bardou. Classifying Prototype-based Programming Languages. *Prototype-based Programming: Concepts, Languages and Applications*, 86, 1998.

[18] David Ungar and Randall B Smith. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. ACM, 1987.

[19] David Flanagan. *JavaScript: The definitive guide*. O'Reilly Media, 2006.

[20] Tom M Mitchell, John Allen, Prasad Chalasani, John Cheng, Oren Etzioni, Marc Ringuette, and Jeffrey C Schlimmer. Theo: A framework for self-improving systems. *Architectures for Intelligence*, 1991.

[21] Michael Cochez, Stefan Decker, and Eric Prud'hommeaux. Knowledge Representation on the Web revisited: Tools for Prototype Based Ontologies. *arXiv:1607.04809*, July 2016.

[22] Tim Korson and John D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9), 1990.

[23] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). RFC 3987 (Proposed Standard), January 2005. http://www.ietf.org/rfc/rfc3987.txt.

[24] Marvin Minsky. A framework for representing knowledge. Technical report, MIT, 1974.

[25] William A. Woods and James G. Schmolze. The KL-ONE family. *Computers & Mathematics with Applications*, 23(2), January 1992.

[26] Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, 9(2), April 1985.

[27] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of database systems*. Pearson, 2015.

[28] Jean Paoli, Eve Maler, Tim Bray, Michael Sperberg-McQueen, and François Yergeau. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[29] Dimitris Kontokostas and Holger Knublauch. Shapes constraint language (SHACL). W3C Recommendation, W3C, July 2017. https://www.w3.org/TR/2017/REC-shacl-20170720/.

[30] Jose Emilio Labra Gayo Iovka Boneva and Eric Prud'Hommeaux. Semantics and Validation of Shapes Schemas for RDF. In *International Semantic Web Conference (ISWC)*, 2017.

[31] Chimezie Ogbuji and Birte Glimm. SPARQL 1.1 entailment regimes. W3C Recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/.

[32] Karen Coyle, Thomas Baker, Dublin Core Metadata Initiative, and others. Guidelines for Dublin Core application profiles. 2009.

[33] Mikael Nilsson, Alistair J. Miles, Pete Johnston, and Fredrik Enoksson. Formalizing Dublin Core Application Profiles – Description Set Profiles and Graph Constraints. *Metadata and Semantics*, 2009.

[34] Arthur G. Ryman, Arnaud Le Hors, and Steve Speicher. OSLC Resource Shape: A language for defining constraints on Linked Data. *Linked Data on the Web (LDOW)*, 996, 2013.