Dmitry Kazhdan

Deduction Rules for Ontology Reasoning

Computer Science Tripos – Part II

Emmanuel College

2018

Proforma

Name:	Dmitry Kazhdan
College:	Emmanuel College
Project Title:	Deduction Rules for Ontology Reasoning
Examination:	Computer Science Tripos – Part II, 2018
Word Count:	11,839[1]
Project Originator:	Dr. Zohreh Shams
Supervisors:	Dr. Zohreh Shams, Dr. Mateja Jamnik

Original Aims of the Project

The first objective of this project was to implement the algorithm described in [24], which can be used in ontological debugging to construct proof trees of ontological entailments. The second objective was to evaluate this algorithm by measuring its coverage on a large set of entailments taken from a diverse ontology corpus. The third objective was to identify potential reasons for imperfect coverage and investigate them.

Work Completed

The above-mentioned algorithm was implemented and tested successfully. This algorithm was then evaluated by measuring its coverage on an entailment dataset extracted from a general ontology corpus. Finally, causes of imperfect coverage were identified and a quantitative evaluation of their impact was presented. Thus, all of the original goals were met successfully.

Special Difficulties

No special difficulties were encountered.

[1]This word count was computed using TEXcount

Declaration

I, Dmitry Kazhdan of Emmanuel College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed Dmitry Kazhdan

Date May 18, 2018

Contents

Chapter 1

Introduction

1.1 Motivation

An *ontology* is a formal description of entities and their relationships in a chosen domain of discourse. These descriptions are typically expressed in a formal logic-based language by introducing definitions of classes (domain concepts) and properties (relations between concepts). The need to represent information about concepts and their interactions in some domain is prevalent in many areas of computer science and research in general. A structured approach to representing such information would enable reusability of domain knowledge and make domain assumptions explicit.

Due to reasons outlined above, ontologies have been used in many domains for various purposes. Examples include building standardised vocabularies for bioinformatics systems, such as SNOMED [32], or even describing people and their social interactions on the web [10].

Building large ontologies is not a straightforward task, meaning that users often make errors when designing an ontology. Mistakes in ontology definitions lead to undesired statements being produced by the ontology. Thus it is often necessary to debug ontologies (repairing the ontology such that the undesired statements do not follow anymore) to ensure that domain knowledge has been entered correctly and to remove modelling errors. Given that typical ontology users are domain experts, not ontology experts, it is extremely important to be able to provide simple, understandable explanations of why a certain statement is entailed by the ontology.

In the rest of this dissertation, ontological axioms (which are used to represent information, as will be explained in the Preparation chapter) that have been inferred from axioms asserted by the user into the ontology will be referred to as *entailments*.

Several different approaches and algorithms for solving the problem described above have been proposed. This project will focus on one such algorithm (described in [24]), that is designed to generate understandable explanations for ontology entailments. This algorithm will be implemented and its applicability to various ontologies will be evaluated.

1.2 Related Work

Many approaches to generating explanations of entailments in ontologies have been proposed. Earlier work, such as [2], assumed that such explanations were tightly correlated with reasoning algorithms. Later work [13] revealed that this is not the case, especially with non-intuitive reasoning strategies, such as Tableau proofs. Instead, focus shifted to explanations that rely on justifications, which are sets of axioms that imply the entailment and are as small as possible (discussed further in the Preparation chapter). However, whilst many algorithms were capable of finding a justification for an entailment, they did not provide intermediate reasoning steps that would show why the entailment followed from the justification. In many cases, understanding why an entailment follows from a justification is non-obvious (as shown in [21]), meaning that extra explanation steps needed to be introduced.

This project will focus on the work described in [24], which proposed ways of addressing this issue. The work in [24] contains a description of a set of manually constructed inference rules and an algorithm capable of generating proof trees using these rules when given an entailment and its justification. These rules serve as intermediate reasoning steps between the justification and the entailment. They were designed using extensive human studies in order to ensure that every such rule is understandable for human users.

1.3 Project Goals

The main aim of this project will be to implement the algorithm and inference rule set described in Dr. Tu Anh T. Nguyen's research paper [24], that was mentioned in the previous section. This algorithm should be able to take as input an entailment together with a justification (a set of axioms that imply the entailment) and attempt to construct a proof tree from the justification to the entailment, using only the predefined inference rules. The proof tree will show why the entailment follows from the justification, using the inference rules as intermediate reasoning steps.

Due to the way in which the inference rules were designed, algorithm inference is *sound*, but not *complete*, meaning that it may not be able to generate a proof tree for certain entailment-justification pairs. Thus, the next goal will be to empirically evaluate the generality of the algorithm. This will be achieved by producing a large set of entailment-justification pairs from a diverse, general corpus of ontologies and measuring the percentage of these pairs for which the algorithm was capable of generating at least one proof tree.

Finally, this algorithm and rule set will be analysed in order to identify potential reasons for imperfect coverage. The impact of each of these reasons on rule coverage will be quantitatively evaluated.

Chapter 2

Preparation

This chapter describes the work that was undertaken before project development began. Section 2.1 introduces key ideas and concepts that will be used in the rest of this dissertation. Section 2.2 includes an overview of the project starting point. Section 2.3 lists the project requirements, and Section 2.4 describes relevant development choices that were made at the start of the project. Finally, Section 2.5 includes a summary of the entire chapter.

2.1 Background

This section includes brief descriptions of the necessary theory used by this project, including a discussion of what ontologies are and why they are used, why ontology debugging is necessary and a definition of what justifications are.

2.1.1 Ontologies

An ontology is a formal description of classes (representing concepts in some domain), individuals (objects that are instances of these concepts), object properties (binary relationships between two objects) and data properties (binary relationships between an object and a value). These definitions are used to represent information about a particular domain in a structured format. As mentioned in the Introduction chapter, ontologies are frequently used for producing formal specifications of concepts and their relations in restricted domains.

Figure 2.1 shows an example of an ontology. Here, individuals *Albert Einstein* and *Charles Darwin* are instances of the class *Scientist*. *Albert Einstein* is also an instance of the class *Physicist*, which is a subclass of *Scientist*. In this ontology, every instance of the *Physicist* class is connected by a *studiesSubject* object property to *Physics*, which is an instance of the *Subject* class.

Ontologies are created and edited using ontology editor applications, such as Protege [22] or NeOn Toolkit [23].

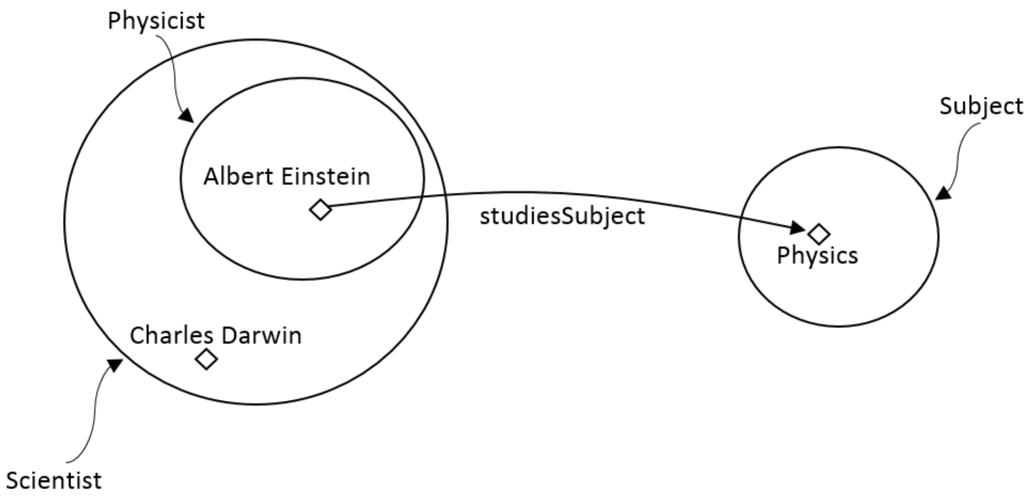


Figure 2.1: A simple ontology example.

2.1.2 OWL Ontology Representation

An ontology is represented by an *ontology language*, which is a logic-based formal language that provides an unambiguous interpretation of the ontology semantics. Ontology languages typically consist of a set of constructors that can combine classes, properties, objects etc. to form expressions and axioms.

Different ontology languages have different sets of constructors and thus provide varying levels of expressivity. This project only focuses on OWL (Web Ontology Language) [28, 29]. The latest OWL syntax is based on the description logic *SROIQ* [16]. Descriptions logics are decidable fragments of first-order logic, meaning inferences in description logics are guaranteed to be computed in finite time.

When creating an ontology, a user provides the names of classes, individuals and properties. These are referred to as *named entities*. Additionally, OWL defines two inbuilt classes: $\top$ (the class of all individuals) and $\bot$ (the empty class), which are also referred to as *owlThing* and *owlNothing*, respectively. $\top$ is a class that subsumes all class expressions, and $\bot$ is a class that is subsumed by all class expressions. Apart from named entities, description logics allow the creation of *anonymous* entities, which are created using complex expression constructors.

OWL provides a set of 30 different axiom constructors, 17 class expression constructors and 1 object property expression constructor. Several of these axiom and class expression constructors are frequently referred to in later chapters. Thus knowledge of their syntax and semantics is required to fully understand some of the later content.

Tables 2.1 and 2.2 list all axiom and class expression constructors, respectively, that will be referred to in later sections, together with brief descriptions of their semantics. These descriptions are directly quoted from [24]. In these tables, C and D represent class expressions, R_o represents an object property, n represents a cardinality expression and a and b represent individuals.

Constructor	**Semantics**
$C \sqcap D[\sqcap \ldots]$	The set of individuals that are instances of both C and D.
$C \sqcup D[\sqcup \ldots]$	The set of individuals that are instances of C or D.
$\neg C$	The set of individuals that are not instances of C.
$\exists R_o.C$	The set of individuals connected by R_o to an instance of C.
$\forall R_o.C$	The set of individuals connected by R_o to only instances of C.
$\geq nR_o.C$	The set of individuals connected by R_o to at least n instances of C
$\leq nR_o.C$	The set of individuals connected by R_o to at most n instances of C
$= nR_o.C$	The set of individuals connected by R_o to exactly n instances of C

Table 2.1: Relevant OWL class expression constructors.

Constructor	**Semantics**
$C \sqsubseteq D$	All instances of C are instances of D.
$C \equiv D[\equiv \ldots]$	All instances of C are instances of D, and vice versa.
Dis$(C, D[, \ldots])$	No instance of C is an instance of D.
Dom(R_o, C)	Only instances of C are connected by R_o to an individual.
Rng(R_o, C)	Each individual can only be connected by R_o to instances of C.
Diff$(a, b[, \ldots])$	a and b are different individuals.

Table 2.2: Relevant OWL axiom constructors.

For the remaining axiom and class expression constructors, understanding their semantics is not necessary for understanding this project, hence they will simply be referenced in relevant sections without explanations of their semantics. Detailed descriptions of all the available axiom constructors and class expression constructors can be found in [28] and [29].

2.1.3 Ontology Reasoning

Since ontologies are represented in description logic, it is possible to apply reasoning algorithms to derive new entailments from them.

For instance, if the *Physicist* class is defined to be a subclass of *Scientist*, and the *Scientist* class is defined to be a subclass of *Person*, then it is possible to infer that the *Physicist* class is a subclass of *Person*. In particular, given an instance of the *Physicist* class with a name *AlbertEinstein*, it is thus possible to infer that *AlbertEinstein* is a *Person*, as demonstrated in Figure 2.2.

Ontology editors typically contain reasoner application plugins, such as HermiT [1] or Pellet [33], that can be used to manipulate the ontology information and infer new information from it.

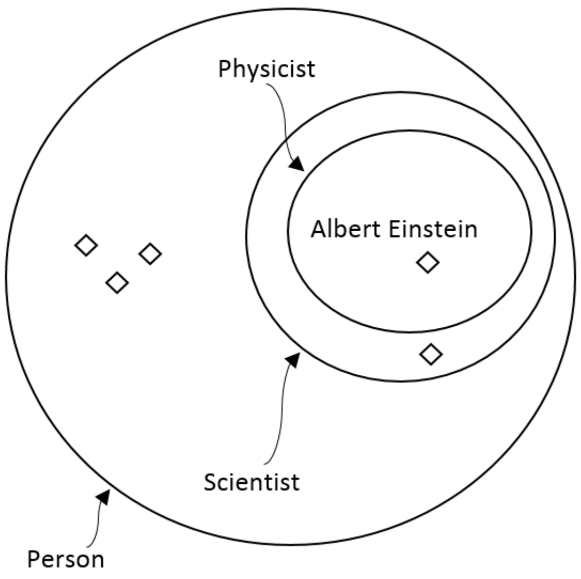


Figure 2.2: A subclass inference example.

2.1.4 Ontology Debugging

As mentioned in the Introduction chapter, making mistakes when defining an ontology can lead to undesired/unwanted entailments being produced. This creates a necessity to debug ontologies to ensure that domain information is being modelled correctly. Users who design ontologies are typically domain experts, not ontology experts. Thus it is often difficult for such users to understand what implicit assumptions cause the undesired entailment to be derived.

For instance, consider an example ontology where a user defined the following axioms:

$$GoodBook \equiv \forall hasBookReview.TopReview \tag{2.1}$$

$$GoodBook \sqsubseteq Book \tag{2.2}$$

Where (2.1) states that a good book is equivalent to something that only has top book reviews, and (2.2) states that any good book is a book. The above definition glosses over an important assumption, namely that any individual who has no book reviews at all will satisfy the $\forall hasBookReview.TopReview$ requirement.

Assume further that there is an individual *AlbertEinstein* in the ontology, that has no *hasBookReview* object properties associated with it. Then the following holds:

$$AlbertEinstein \in \forall hasBookReview.TopReview \tag{2.3}$$

As well as:

$$\forall hasBookReview.TopReview \sqsubseteq GoodBook \sqsubseteq Book \tag{2.4}$$

Where (2.3) states that *AlbertEinstein* is an instance of the $\forall hasBookReview.TopReview$ class, and (2.4) uses the fact that $A \equiv B$ is equivalent to: $(A \sqsubseteq B) \wedge (B \sqsubseteq A)$. Hence, this definition implies that: $AlbertEinstein \in Book$, i.e. that *AlbertEinstein* is an instance of a *Book* class, which is a modelling error.

One possible strategy that can be used to simplify the debugging process is to be able to produce a detailed, step-by-step derivation of an entailment that is specified by the user. Demonstrating intermediate reasoning steps for undesired entailments will help the user identify specific modelling errors (such as the one given above) and modify the ontology appropriately.

Most commonly used reasoning algorithms were designed with an emphasis on computational efficiency. Such reasoners typically produce derivations that are difficult for users to understand. For instance, many reasoners rely on Tableau-based algorithms. In order to prove a statement of the form $A \rightarrow B$, tableau-based algorithms will prove the unsatisfiability of $A \wedge \neg B$, which in practice is often difficult to follow. Similar arguments apply to Refutation-based algorithms and other commonly used approaches.

An alternative approach is to construct a set of inference rules where emphasis is placed on their understandability, not computational efficiency. The intuition is that proof trees constructed from understandable inference rules will also be understandable. This project will focus on one such approach.

2.1.5 Justifications

This section provides more precise definitions of two important concepts mentioned previously: *justifications* and *laconic justifications.*

Definition 2.1.1. A justification of an entailment is a minimal set of axioms from which the entailment can be inferred.

A simple example of a justification and a corresponding entailment is shown in (2.5):

$$\begin{aligned} &Justification = \{A \sqsubseteq C \sqcap D \sqcap E,\ C \sqsubseteq B\}, \\ &Entailment = A \sqsubseteq B \end{aligned} \tag{2.5}$$

Minimality implies that if any of the axioms of a justification are removed, the entailment will no longer follow. Axioms in an ontology may consist of a large number of expressions, thus even though a justification for an entailment is minimal, the axioms within the justification may contain irrelevant parts. For that reason, it is necessary to introduce the concept of laconic justifications.

Definition 2.1.2. A laconic justification of an entailment is a justification of the entailment, in which none of the axioms contain any superfluous parts.

In Definition 2.1.2 above, *superfluous parts* refer to subparts of an axiom that are not required for the corresponding entailment to follow. This broad definition is sufficient for this project, but a more formal, precise definition can be found in [20] and [24].

An example of a laconic justification and an entailment is shown in (2.6):

$$\begin{aligned} &Justification = \{A \sqsubseteq C,\ C \sqsubseteq B\}, \\ &Entailment = A \sqsubseteq B \end{aligned} \tag{2.6}$$

In (2.6), the first justification axiom was weakened, because intersections with D and E were not necessary for the entailment to hold.

The definitions given in this section are heavily based on descriptions that can be found in [24]. More information regarding laconic and precise justifications can be found in [13, 21, 20].

2.2 Starting Point

This section will list all of the relevant material that was covered and used during project implementation.

2.2.1 Original Work

As mentioned in the previous sections, this project relies on [24]. This is a research paper which contains a description of an algorithm that, given an entailment and a justification of that entailment, attempts to construct a proof of the entailment using the given justification axioms and a set of 57 predefined inference rules that are also described in that research paper.

Unlike other proof generation algorithms that are often used by reasoners, the primary goal of this algorithm is producing a proof that is human-readable and easily understandable. The inference rules were designed based on an intuitiveness factor and can easily serve as the basis of explanation generation in natural language. Also, their accessibility is evidenced by empirical evaluation that measures their understandability, as described in [25].

The inference rules are represented in a standard way by formulating a schemata and employing metavariables that can be instantiated to elements of the universe of discourse. For example, axiom (2.7) is a schemata with metavariables A and B standing for class expressions.

$$A \sqsubseteq B \tag{2.7}$$

During inference, these will be instantiated with class expressions, such as:

$$Physicist \sqsubseteq Scientist \tag{2.8}$$

In the rest of this dissertation, a schemata will be referred to as a *template*, and metavariables will be referred to as *free variables*. The process of assigning concrete expressions from the universe of discourse to free variables will be referred to as *free variable instantiation*.

The algorithm produces derivations in a form of proof trees. Every node of the proof tree contains an axiom and every non-leaf node and its children correspond to a conclusion and a set of premises of an inference rule, respectively. The root node is an entailment, and the leaf nodes are axioms from which the entailment follows.

An example is given in Figure 2.3, where E is an entailment, $ax1, ax2, ax3, ax4$ are justification axioms, $i1, i2, i3, i4$ are internal nodes (intermediate conclusions), and R_1, R_2, R_3, R_4, R_5 are the inference rules. Rule labels are assigned different colours for readability only.

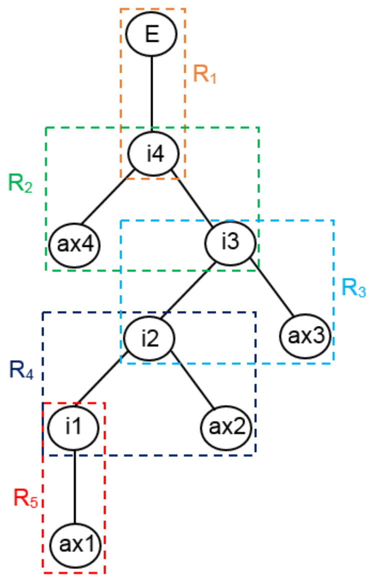


Figure 2.3: A proof tree example.

The research paper provides a high-level overview of the algorithm in a form of pseudocode, with very little detail given about how specific features were implemented. Implementation of the inference rules and inference rule matching is not provided.

Thus, the research paper was only used as a guide when making high-level design decisions. The majority of implementation choices had to be decided independently as part of the project, meaning that the overall amount of workload was suitable for a Part II project.

2.2.2 Other Relevant Material

The project was implemented in Java, for which the "Object Oriented Programming" module lectured in Part IA, as well as the Part IA and Part IB ticks were useful. Section 2.4.1 will go into more detail about the suitability of this choice of programming language.

The "Logic and Proof" module lectured in Part IB was relevant for understanding description logic syntax and reasoning algorithms.

Before starting on the project, the Protege tutorial [12], and a small tutorial on ontology development [9] were completed. These tutorials provided a basic introduction to ontologies and ontology editors.

2.3 Requirements Analysis

The project proposal gave a list of project success criteria. These were slightly expanded on to aid clarity, but were essentially unchanged otherwise. The project should be considered successful if the following conditions are satisfied:

1. The algorithm and the inference rule set described in [24] are implemented successfully.
2. The algorithm can be shown to work and construct proofs given an entailment and a justification.
3. A dataset consisting of entailment-justification pairs is extracted from a diverse, general corpus of ontologies.
4. A quantitative investigation of inference rule coverage is conducted using this dataset.

Thus by the end of the project, an implementation of the described algorithm should be given. This algorithm should be able to take as input an entailment from an ontology, together with a set of axioms that justify this entailment, and give a proof of this entailment as output.

In the original proposal, potential extension steps included extracting new inference rules either by studying the existing rule set, or by applying Machine Learning techniques to proof patterns. However, the proposal stated that these extension steps may change as more relevant information is gathered.

As discussed in the original work, all of the inference rules were produced following extensive human studies. Thus justifying the suitability of new inference rules without conducting extensive human trials would have been very problematic. Applying machine learning techniques to proof patterns of manually constructed inference rule sets is not a well-studied topic, with very few relevant resources available online.

Due to the above, the extension steps were changed. The success criteria for the modified extension steps are given below:

1. The major reasons (at least two) for why the algorithm and rule set may have failed to produce a proof for the failed cases are given.
2. A quantitative analysis is performed on the dataset to investigate the impact of the above reasons.
3. The results of the quantitative analysis are used to determine the impact each reason has on rule coverage.

Thus instead of introducing new inference rules, a decision was made to focus on analysing the existing inference rule set and identifying its most obvious pitfalls that have the largest impact on imperfect coverage.

2.4 Development Choices

This section will describe some relevant development choices that were made at the start of the project.

2.4.1 Selected Programming Language

As mentioned in Section 2.2.2, the project was entirely implemented in Java. In principle, it would have been possible to implement the project in a different language, such as Python or C++. Functional programming languages, such as OCaml would have provided convenient methods of pattern matching with rules.

However, the APIs that were used by the project, as well as the majority of ontology editors and ontology editor plugins are all implemented in Java. Thus Java was a natural choice, given that ontology-related software is primarily implemented in Java. It was also considerably simpler to setup and use the required APIs.

2.4.2 Version Control

GitHub [11] was used for version control. This provided the ability to quickly roll back any changes made to the project. More importantly, it provided simple ways of comparing different implementations of certain features in cases where multiple valid design choices were available, by developing them on separate branches.

2.4.3 Backups

Relevant project files were all stored on Dropbox [8], providing a secure, cloud-based storage. Every week, a backup of all project files was made to one of the MCS machines in the Computer Laboratory.

2.5 Summary

This chapter provided descriptions of the necessary theory relevant to the project. The Starting Point of the project was given by listing the existing materials that the project builds on, and including a description of the research paper that the project implementation is based on. The Requirements Analysis outlined the necessary project deliverables. The final section provided a list of software engineering practices that were followed during development.

Chapter 3

Implementation

This chapter focuses on the design and implementation choices taken during project development. The project ended up amounting to a considerable codebase of over 5000 lines of code. For that reason, only key parts of the algorithm and relevant data structures will be presented here in a form of pseudocode and diagrams.

Section 3.1 gives a brief overview of the necessary APIs used by the project. Sections 3.2 and 3.3 describe the implementation of the algorithm. Section 3.4 discusses inference rule implementation, including inference rule representation, rule instance matching and conclusion generation. Section 3.5 gives a description of how the implementation was tested. Finally, Section 3.6 provides a brief summary of the entire chapter.

3.1 Library Setup

Successful implementation of the algorithm relied on the use of 3 different APIs, which provided fundamental features for working with ontologies and ontology reasoners. These APIs are listed below, together with brief descriptions of the functionality they provided.

1. The *OWLAPI* [34]. This API provided a framework for working with ontologies, including the following features:
    - OWL/XML parsing and writing.
    - OWL Functional Syntax parsing and writing.
    - Reasoner interfaces for working with reasoner APIs.
2. The *HermiT Reasoner* API [1]. This API was used for deriving new entailments from ontologies. In particular, it was used for the computation of subsumption entailments from ontologies.
3. Matthew Horridge's *OWLExplanation* API [14] (referred to as the "Explanation API" in later sections), which was used for:
    - Computing a justification from an ontology for a given entailment.
    - Computing a laconic justification for an entailment, given a (potentially non-laconic) justification of that entailment.

Sections 3.2, 3.3 and 3.4 will go into more detail about the exact applications of the above APIs.

3.2 Initial Trees

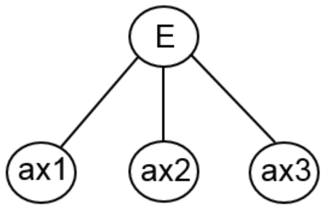
As mentioned in Section 2.1.5 of the Preparation chapter, axioms in a justification may contain parts that are not necessary for the entailment to hold. However, writing inference rules that are tolerant to redundant parts in their premises or conclusion leads to a large increase in the size of the rule set. Instead, inference rules in the rule set were written based on the assumption that axioms contain no redundant parts. Thus, a given justification had to be modified to satisfy that assumption. This was achieved by producing a set of *initial trees* from a given justification-entailment pair.

Definition 3.2.1. An Initial Tree is a tree satisfying the following constraints:

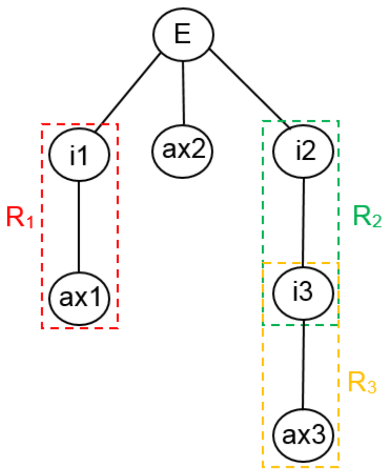
- The root node is an entailment.
- Leaf nodes are axioms of a justification that imply the entailment.
- Every internal node has exactly one child, where an internal node is defined to be any node which is not a leaf node and not a root node.
- Every subtree consisting of an internal node and its child corresponds to an instance of a single-premise inference rule and is labelled with that rule.
- The root node and its children are unlabelled.
- For every leaf node axiom containing superfluous parts, an internal node or sequence of internal nodes linking the root node to the leaf node is added.

In Definition 3.2.1 above, a node and its child correspond to an instance of an inference rule if the node axiom and child node axiom are instances of an inference rule's conclusion and premise templates, respectively. This definition is based on descriptions found in [24].

Figure 3.1 shows two examples of initial trees generated for different types of justifications: (a) shows an initial tree produced from a justification without superfluous parts, whereas (b) shows an initial tree produced from a justification in which axioms 1 and 3 contain parts unnecessary for the entailment, for which internal nodes 1, 2 and 3 have been added. Labels R_1, R_2 and R_3 refer to rules from the inference rule set.



(a)



(b)

Figure 3.1: Two different examples of initial trees.

The initial tree generation step is designed to extract only the relevant parts from the justification axioms by introducing internal nodes that serve as lemmas. A sequence of internal nodes is required in cases where this unpacking is non-obvious and has to be split into several intermediate steps.

3.2.1 Initial Tree Generation

The algorithm used for generating initial trees is presented in Algorithm 1. This algorithm is largely based on the descriptions and pseudocode provided in [24].

Algorithm 1 ComputeInitialTrees(J, E)

```
1: $ProofList_{initial} \leftarrow \{\}$
2: $LaconicJustifications \leftarrow$ getLaconicJustifications($J, E$)
3: for $LaconicJ \in LaconicJustifications$ do
4:     $SubTrees_{incomplete} \leftarrow$ matchLacToNonLacJust($J, LaconicJ$)
5:     $SubTrees \leftarrow$ findRulesForSubTrees($SubTrees_{incomplete}$)
6:     if $SubTrees\ != null$ then
7:         $P_{initial} \leftarrow$ new proof tree using $E$ as root and $SubTrees$ as its subtrees
8:         $ProofList_{initial}.add(P_{initial})$
9:     end if
10: end for
11: return $ProofList_{initial}$
```

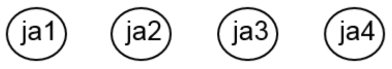
First, all laconic justifications are computed from the given justification-entailment pair *(J, E)*. This is achieved by calling the **getLaconicJustifications** method, which uses the Explanation API. Next, for every laconic justification, an attempt to construct an initial tree is made.

The **matchLacToNonLacJust** method attempts to produce a one-to-one mapping between axioms in the given justification and axioms in the laconic justification. Axioms with no redundant parts will appear in both sets and are returned directly. For the remaining axioms, the HermiT Reasoner is used to find, for every laconic axiom, an axiom in the justification which entails it. These two axioms are combined into a subtree with the laconic axiom as the root and the justification axiom as the leaf. Thus the method call returns a set of subtrees, with every subtree consisting of either one or two nodes.

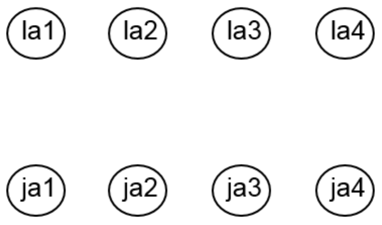
Next, the **findRulesForSubTrees** method is used to label every subtree consisting of two nodes with a corresponding inference rule. For every subtree, all single-premise inference rules in the rule set are checked. If the subtree matches any single-premise rule, it is labelled with that rule. In a few *exception cases*, the subtree may be expanded by adding intermediate, internal nodes to it. Exception cases will be discussed in more detail in Section 3.2.2.

Finally, if these rules were found successfully for all the subtrees, an initial tree is produced by attaching these subtrees to the entailment axiom root node. Otherwise, a null value is returned.

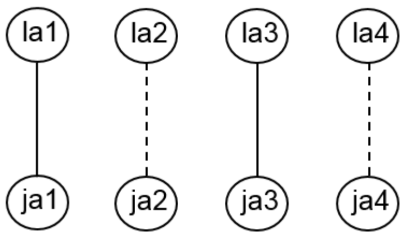
The key steps (namely, lines 4 to 9) of Algorithm 1 are summarised in Figure 3.2. In 3.2a the original justification is given (shown as a set of axioms labelled ja1 to ja4). In 3.2b, a laconic justification is computed from the given justification (shown as a set of axioms labelled la1 to la4). In practice, there may be multiple different laconic justifications returned, which is why it is necessary to iterate over all of them. For simplicity, only one such laconic justification is shown in the diagram. In 3.2c, the laconic axioms are matched to the non-laconic ones. A dashed line indicates that the axioms are identical, whereas a solid line indicates that the laconic axiom is implied by the original one. Next, 3.2d shows an attempt to find matching inference rules for the produced subtrees (shown by R_1 and R_2). Finally, 3.2e shows the initial tree.



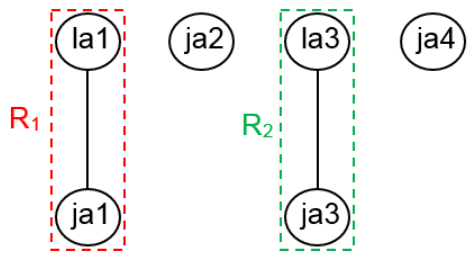
(a) Original justifications.



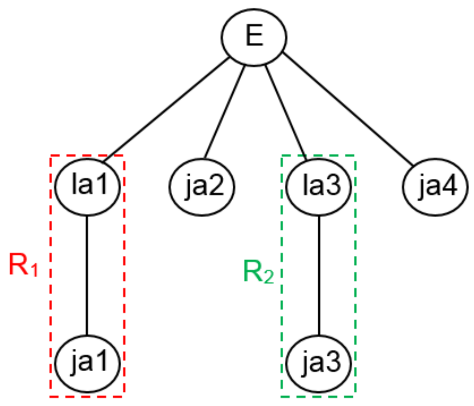
(b) Original and laconic justifications.



(c) Justification mapping.



(d) Inference rule application.



(e) Initial tree generation.

Figure 3.2: Example of an initial tree being generated.

3.2.2 Rule Exception Cases

As described in [24], several cases where unpacking produced laconic axioms that made the whole derivation more difficult to understand were identified. These would have resulted in proof trees containing steps that were difficult to follow. For instance, if we are given a justification consisting of:

$$CarOwner \sqsubseteq \exists hasCar.Car \tag{3.1}$$

$$\mathbf{Dom}(hasCar, Person) \tag{3.2}$$

As well as an entailment:

$$CarOwner \sqsubseteq Person \tag{3.3}$$

Where (3.1) means that every car owner has a car, (3.2) means that anything that has a car is a person, and (3.3) means that every car owner is a person, then the Explanation API returns the following laconic justification:

$$CarOwner \sqsubseteq \exists hasCar.\top \tag{3.4}$$

$$\mathbf{Dom}(hasCar, Person) \tag{3.5}$$

Where (3.4) states that every car owner has something as a car. Thus, in the laconic justification, the filler *Car* in the first axiom has been omitted, since it is not needed for the entailment to hold, and replaced with the universal entity $\top$. However, this is considered to be a trivial inference step which should be ignored by the algorithm. Thus, this elimination should be skipped, and the original axioms should be retained. These cases are referred to as *exception cases.*

Exception cases work by taking a subtree consisting of a justification axiom as a leaf and a laconic axiom as a root, and returning a modified subtree which either consists of the original justification axiom only, or which retains the justification axiom as the leaf, but has the laconic axiom replaced by a sequence of axiom nodes. Exception cases are defined similarly to inference rules by specifying template patterns for the input and output subtrees. In the output subtree, every non-leaf node and its child must correspond to a single-premise inference rule from the rule set.

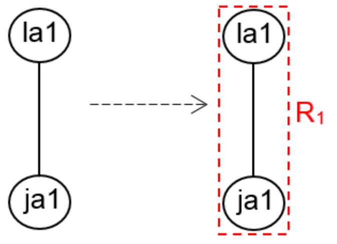
For instance, in the above example an exception case can contain the following input subtree pattern:

$$X \sqsubseteq \exists R_o.Y \ \rightarrow \ X \sqsubseteq \exists R_o.\top \tag{3.6}$$

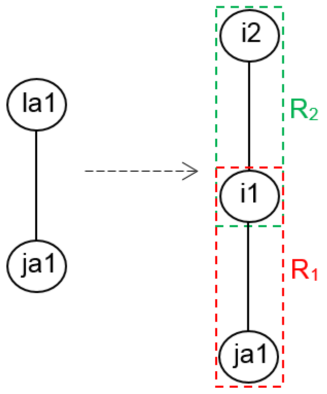
Which will match the tree consisting of (3.4) and (3.1). In this case the output subtree will consist of a single node, shown in (3.7), signifying that the justification axiom should be left unchanged:

$$X \sqsubseteq \exists R_o.Y \tag{3.7}$$

A check for exception cases is incorporated in the **findRulesForSubTrees** method call of Algorithm 1. If a given subtree matches an inference rule, it is simply labelled by that rule. If it matches an input tree of an exception case, then it is substituted with the corresponding output tree. Examples of the two cases are given in Figure 3.3.



(a) Subtree not matching an exception case.



(b) Subtree matching an exception case.

Figure 3.3: Subtree labelling examples.

A laconic axiom produced by the Explanation API contains no redundant parts (by definition). Hence at most one internal node (the laconic axiom) is added to the initial tree between the leaf (the corresponding justification axiom) and the root (the entailment). Subtrees associated with exception cases, however, may consist of multiple nodes, and so exception cases are the reason why an initial tree may contain sequences of internal nodes, as shown in Figure 3.4.

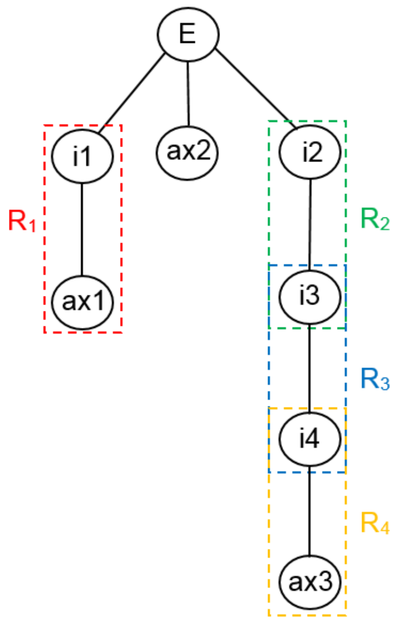


Figure 3.4: A tree containing an exception case.

In either case (exception or non-exception), the resulting initial tree has, for any internal node and its child, a single-premise inference rule label. A full list of exception cases is given in Section A.2 of the Appendix.

3.3 Complete Trees

Once a set of initial trees is obtained from the given entailment-justification pair, the next step is to compute *complete trees* from the initial trees.

Definition 3.3.1. A Complete Tree is a tree satisfying the following constraints:

- The root node is an entailment.
- Leaf nodes are axioms of a justification that implies the entailment.
- For every non-leaf node, the node and its child node(s) correspond to an instance of an inference rule and are labelled with that rule.

Definition 3.3.1 is based on descriptions found in [24]. Figure 3.5 shows an example of a complete tree.

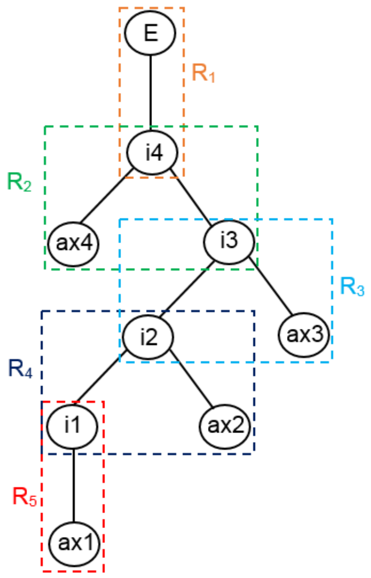


Figure 3.5: A complete tree.

A complete tree shows a step-by-step derivation of the entailment from the justification, using only the pre-defined set of inference rules for intermediate reasoning steps.

3.3.1 Complete Tree Computation

Algorithm 2 shows how complete proof trees are computed from a given initial tree. This algorithm is largely based on the descriptions and pseudocode provided in [24].

The following paragraph describes some of the key steps of Algorithm 2 in more detail. The algorithm works on *partially-complete trees* by adding intermediate nodes to them, until they become complete trees.

Algorithm 2 ComputeCompleteProofTrees($P_{initial}$)

```
1: $ProofList_{complete} \leftarrow \{\}$
2: $ProofList_{incomplete} \leftarrow \{P_{initial}\}$
3: while $ProofList_{complete}.empty$ and $!ProofList_{incomplete}.empty$ do
4:     $ProofList'_{incomplete} \leftarrow \{\}$
5:     for $P_{incomplete} \in ProofList_{incomplete}$ do
6:         $E \leftarrow P_{incomplete}.root$
7:         $Children \leftarrow P_{incomplete}.root.children$
8:         $ProofList'_{complete} \leftarrow$ matchChildrenToEnt($Children, E$)
9:         if $ProofList'_{complete}! = \{\}$ then
10:            $ProofList_{complete} \leftarrow ProofList'_{complete}$
11:            break
12:        else
13:            $PartitionList \leftarrow$ ComputeAllPartitions($Children$)
14:            for $Partition \in PartitionList$ do
15:                $P_{new} \leftarrow$ ApplyPartition($P_{incomplete}, Partition$)
16:                if $P_{new}! = \{\}$ then
17:                    $ProofList'_{incomplete}.add(P_{new})$
18:                end if
19:            end for
20:        end if
21:    end for
22:    $ProofList_{incomplete} \leftarrow ProofList'_{incomplete}$
23: end while
24: return $ProofList_{complete}$
```

Definition 3.3.2. A Partially-Complete Tree is a tree satisfying the following constraints:

- The root node is an entailment.
- Leaf nodes are axioms of a justification that implies the entailment.
- For every node except the root node and the leaf nodes, the node and its child node(s) correspond to an instance of an inference rule and are labelled with that rule.
- The root node and its children are unlabelled.

The difference between a partially-complete tree and a complete tree is that in a partially-complete tree the root node and its children may not correspond to any inference rule and are unlabelled.

The procedure can be summarised as follows: a set of partially-complete trees is maintained. The algorithm iterates over all of these trees. For every such tree: if the root node and its children already correspond to an inference rule, then they are labelled with this inference rule, thus making this partially-complete tree a complete tree. Otherwise, the partially-complete tree is extended.

Tree extension is done as follows: all possible partitions of the root child nodes are computed. For every partition, all possible applicable rules are searched for in the rule set. For every partition with at least one applicable rule, corresponding conclusions for that rule or rules are generated and added as intermediate nodes between the root and the children, with their appropriate rule labels.

Thus given a partially-complete tree, a set of new partially-complete trees is generated and added to the set. The old partially-complete tree is discarded. This is repeated until there are no more partially-complete trees (which means there are no more options to consider), or until the set of complete trees becomes non-empty, meaning that at least one complete tree has been computed.

Figure 3.6 shows an example of a single iteration of Algorithm 2, where two new partially-complete trees are produced from a given partially-complete tree by adding intermediate nodes to it.

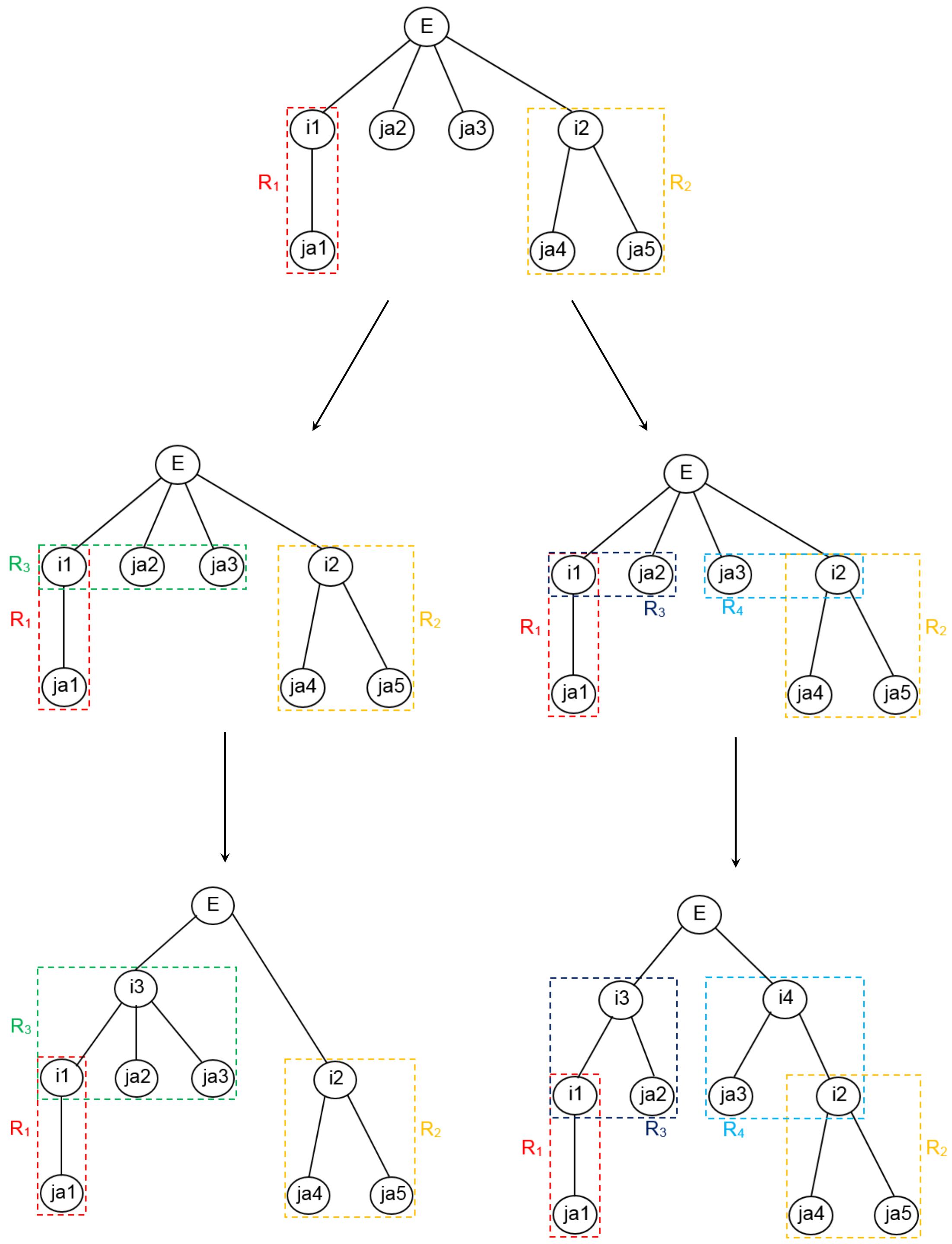


Figure 3.6: Partially-complete tree extension.

Given an entailment-justification pair, the algorithm returns a set of complete proof trees that have the entailment as the root and the justification axioms as the leaves.

An initial tree satisfies the definition of a partially-complete tree, and on every iteration new partially-complete trees are generated from old partially-complete trees. Hence for every such tree, only the root and its children may not correspond to an inference rule. All the other subtrees are guaranteed to correspond to an inference rule and be labelled by it (by definition). The invariance of this property guarantees that only complete proof trees are returned by Algorithm 2, since it terminates when the root and its children are labelled.

3.3.2 Proof Tree Computation

Algorithm 3 shows the overall proof-tree computation algorithm that takes an entailment-justification pair, and returns a set of complete proof trees. This algorithm is largely based on the descriptions and pseudocode provided in [24].

Algorithm 3 ComputeProofTrees(*Justification, Entailment*)

```
1: $ProofList_{result} \leftarrow \{\}$
2: $ProofList_{initial} \leftarrow$ ComputeInitialTrees(Justification, Entailment)
3: for $P_{initial} \in ProofList_{initial}$ do
4:     $ProofList_{complete} \leftarrow$ ComputeCompleteProofTrees($P_{initial}$)
5:     $ProofList_{result}.addAll(ProofList_{complete})$
6: end for
7: return $ProofList_{result}$
```

Every inference rule is constructed in such a way that the conclusion discards some information from the premises. Thus infinite application of rules is not possible and the algorithm is guaranteed to terminate.

3.4 Inference Rules

This section describes the implementation of inference rules and their associated features. Sections 3.4.1 and 3.4.2 discuss how rule expressions were implemented. Sections 3.4.3 and 3.4.4 describe how matching of owl axioms to rule templates and conclusion generation were implemented.

3.4.1 Rule Expressions

Every inference rule in the inference rule set consists of a conjunction of one or more premise axiom templates and a single conclusion axiom template. Out of the 30 different axiom constructors available, only 15 were used by any of the rules in the rule set. These are listed in Table 3.1.

OWL Constructor
$R_o \sqsubseteq S_o$
$\mathbf{Invs}(R_o, S_o)$
$\mathbf{Fun}(R_o)$
$\mathbf{InvFun}(R_o)$
$\mathbf{Sym}(R_o)$
$\mathbf{Tra}(R_o)$
$\mathbf{Fun}(R_d)$
$\mathbf{Rng}(R_o, C)$
$\mathbf{Dom}(R_o, C)$
$\mathbf{Rng}(R_d, D_r)$
$\mathbf{Dom}(R_d, C)$
$C \sqsubseteq D$
$\mathbf{Diff}(a, b[, \ldots])$
$C \equiv D[\equiv \ldots]$
$\mathbf{Dis}(C, D[, \ldots])$

Table 3.1: Required Axiom Constructors

In all of the inference rules used, data ranges, data properties and object properties never had internal structure, and were always represented as atomic identifiers D_r, R_d and R_o, respectively. Thus implementing constructor templates for object properties, data ranges or data properties was not necessary. Class expressions, however, did contain internal structure and thus required more complex pattern matching. Out of the 18 different class expression constructors available, 14 were used in at least one of the rules in the rule set. These are listed in Table 3.2.

Expression Constructor
$C \sqcap D[\sqcap \ldots]$
$C \sqcup D[\sqcup \ldots]$
$\neg C$
$\exists R_o.\{a\}$
$\exists R_d.\{l\}$
$\exists R_o.C$
$\forall R_o.C$
$\exists R_d.D_r$
$\geq nR_o.C$
$\leq nR_o.C$
$= nR_o.C$
$\geq nR_d.D_r$
$\leq nR_d.D_r$
$= nR_d.D_r$

Table 3.2: Required Class Expression Constructors

Examples in this section and the following sections of this chapter will write expressions in prefix form by writing them as a constructor with input arguments. For instance, "$\geq nR_o.C$" will be written as "$\geq(n, R_o, C)$". This notation emphasises an important point, namely that class expression constructors "$\sqcap$" and "$\sqcup$", as well as axiom constructors "$\equiv$", "**Diff**()" and "**Dis**()" take arbitrary-size sets of class expressions as an input argument. These constructors will be referred to as *group expression constructors*, since they take a group of expressions as their only input argument.

For example, the following expression:

$$C \equiv D \equiv E \tag{3.8}$$

In prefix notation (3.8) this will be written as:

$$\equiv(\{C, D, E\}) \tag{3.9}$$

For every axiom constructor and class expression constructor used by any of the inference rules, a corresponding template constructor taking template arguments was created. These will be distinguished from axiom constructors by appending a prime symbol to them.

Given how some inference rules that used group expression constructors were defined, implementation of these constructors required extra consideration.

For instance, rule "ObjInt-1" is given below:

$$\begin{aligned} &X \equiv Y_1 \sqcap Y_2 \sqcap \ldots \sqcap Y_n \sqcap Z_1 \sqcap \ldots \sqcap Z_n,\ n \geq 1, m \geq 1 \\ &\rightarrow X \sqsubseteq Y_1 \sqcap Y_2 \sqcap \ldots \sqcap Y_n \end{aligned} \tag{3.10}$$

As can be seen from (3.10), certain rule templates have to match expressions of arbitrary size (as n and m do not have upper bounds). Thus when defining a rule expression, it must be possible to define expression templates without size constraints.

Another important example is given below:

$$\begin{aligned} &X \sqsubseteq Y \ \wedge\ \mathbf{Dis}(X, Y[, \ldots]) \\ &\rightarrow X \sqsubseteq \top \end{aligned} \tag{3.11}$$

This example shows that an expression template may consist of class expressions explicitly referred to in other parts of the inference rule (X and Y), as well as a set of other class expressions of arbitrary size (implied by "$[, \ldots]$").

Considering the above examples, class expression constructors "$\sqcap$" and "$\sqcup$", as well as axiom constructors "$\equiv$", "**Diff**()" and "**Dis**()" were defined as constructors taking two arguments: a set of *named expressions*, which had to be manually specified, and thus limited in number, together with an *anonymous group* variable, which was a free variable that matched sets of any size. It is anonymous because expressions in this set are never referred to in any other premise of the rule, thus there is a single free variable representing the entire group of expressions, instead of free variables representing the individual expressions. The anonymous group argument was made optional, which allowed definition of patterns of a fixed size, taking a fixed set of defined class expression patterns only.

For example, in class expression (3.12), class expressions A, B and C refer to specific expressions that may be referenced in other parts of the rule, whereas $D_1, \ldots, D_n$ refers to a disjunction of unspecified size.

$$A \sqcup B \sqcup C \sqcup D_1 \sqcup \ldots \sqcup D_n, \quad n \geq 1 \tag{3.12}$$

According to our definition, expression (3.12) will be represented as follows:

$$\sqcup'(\{A,\ B,\ C\}, D) \tag{3.13}$$

Thus a group expression constructor takes a set of named expressions (A, B, and C), as well as a single anonymous group free variable (D) that will match to a set of class expressions.

The design choices described in this section placed certain constraints on how inference rules can be defined. Fundamentally, this implementation does not allow definition of arbitrarily complicated inference rule patterns, which limits the extensibility of the rule set.

However developing a universal rule framework capable of representing any axiom expression pattern was outside the scope of this project. The chosen implementation satisfies all of the requirements of the rules in the rule set, whilst providing a degree of extensibility and allowing efficient matching and generation.

A list of all the inference rules in the rule set can be found in Section A.1 of the Appendix.

3.4.2 Rule Restrictions

Some of the inference rules contained rule restrictions as part of their definition, designed to restrict the possible values that free variables can take. These restrictions were specific to the rule set, and the OWLAPI did not provide sufficient functionality to implement them using existing features. Hence these restrictions had to be implemented manually as part of the rule definition.

Rule restrictions used by the rule set are listed below:

- Absolute Cardinality Restriction: restricts the value of a quantifier variable. Given in terms of an upper or lower bound, with the bound being expressed as an integer.

- Relative Cardinality Restriction: restricts the value of a quantifier variable. Given in terms of an upper or lower bound, with the bound being expressed as another free variable.
- Subset Restriction: restricts the value of a group expression, stating that the group expression has to be a proper subset of another specified group expression.
- Unequal Datatypes Restriction: restricts the value of two datatypes, stating that they must not be equal.
- Unequal Literals Restriction: restricts the value of two literals (data values), stating that they must not be equal.

Examples of rules using these restrictions are given in Table 3.3.

Restriction	Example
Absolute cardinality restriction	$X \sqsubseteq \geq nR_o.Y, \quad n > 0$ $\wedge\ \exists R_o.Y \sqsubseteq Z$ $\rightarrow X \sqsubseteq Z$
Relative cardinality restriction	$X \sqsubseteq \geq n_1 R_o.Y, \quad n_1 \geq n_2$ $\rightarrow X \sqsubseteq \geq n_2 R_o.Y$
Subset Restriction	$X \equiv Y_1 \sqcap \ldots \sqcap Y_n \sqcap Z_1 \sqcap \ldots \sqcap Z_n$ $\rightarrow X \sqsubseteq Y_1 \sqcap \ldots \sqcap Y_n, \ n \geq 1, m \geq 1$
Unequal Datatypes Restriction	$X \sqsubseteq \exists R_d.D_{t0}$ $\wedge\ \mathbf{Rng}(R_d, D_{t1}), \ D_{t0} \neq D_{t1}$ $\rightarrow\ X \sqsubseteq \bot$
Unequal Literals Restriction	$X \sqsubseteq \exists R_d.\{l_0\}$ $\wedge\ X \sqsubseteq \exists R_d.\{l_1\}, \quad l_0 \neq l_1$ $\wedge\ \mathbf{Fun}(R_d)$ $\rightarrow\ X \sqsubseteq \bot$

Table 3.3: Rule restriction examples

Thus, an inference rule was defined by providing a set of premise axiom templates, a conclusion axiom template and a (possibly empty) set of restrictions for that rule.

3.4.3 Rule Instance Matching

A key feature used by the algorithm is determining whether a set of axioms is an instance of one of the inference rules.

Due to the way certain inference rules were defined, the matching and generating algorithms had to satisfy several very specific requirements, such as the ability to enforce rule restrictions that were described in Section 3.4.2. Thus, it was decided to implement the matching algorithm and generating algorithm (discussed in the next section) manually, instead of relying on existing resources.

Matching of an axiom to an axiom template was done using a recursive strategy. The top-level constructors of the axiom and the template were matched. If the match was successful, the sub-expressions (constructor arguments) were matched recursively. The recursion bottomed out when a free variable was reached in the template. In that case, the free variable had to be instantiated to the corresponding expression.

As discussed in Section 3.4.1, matching could be done unambiguously for all axiom constructors except "**Diff**()", "$\equiv$" and "**Dis**()", and for all class expression constructors except "$\sqcap$" and "$\sqcup$". Here "unambiguously" means that there exists a unique way of producing a matching. Figure 3.7 shows an example of an non-ambiguous matching, which produces a unique free variable instantiation.

$$\begin{tikzcd}
match(\sqsubseteq'(X,\ \geq'(n, R_o, Y)),\ \sqsubseteq(Car,\ \geq(4, hasPart, Wheel))) \arrow[d] \\
\begin{array}{c} match(X,\ Car), \\ match(\geq'(n, R_o, Y),\ \geq(4, hasPart, Wheel)) \end{array} \arrow[d] \\
\begin{array}{c} match(n,\ 4), \\ match(R_o,\ hasPart), \\ match(Y,\ Wheel), \\ X = Car \end{array} \arrow[d] \\
\textbf{return: } X = Car,\ n = 4,\ R_o = hasPart,\ Y = Wheel
\end{tikzcd}$$

Figure 3.7: Unambiguous matching example.

For the constructors mentioned in the previous paragraph, multiple instantiations needed to be considered due to multiple possible orderings of the group expression argument.

Axiom constructor **Diff**() was only ever used with 2 inner expressions in any of the rules. For simplicity, matching for this constructor was restricted to the case of 2 inner expressions only. Hence two different matches corresponding to the two different argument orderings were considered, as shown in Figure 3.8.

For the remaining 4 constructors ("$\equiv$", "**Dis**()", "$\sqcap$" and "$\sqcup$"), their set of named expressions and their anonymous group free variable had to be matched to a corresponding set of class expressions.

First, named expressions were considered. All possible orderings of the given expression set were produced and an attempt to match each one against the named

$$\begin{tikzcd}
& match(\mathbf{Diff}'(X,Y),\ \mathbf{Diff}(Car,Truck)) \arrow[dl] \arrow[dr] & \\
\begin{array}{c} match(X,Car), \\ match(Y,Truck) \end{array} \arrow[d] & & \begin{array}{c} match(X,Truck), \\ match(Y,Car) \end{array} \arrow[d] \\
\begin{array}{c} \textbf{return: } X = Car, \\ Y = Truck \end{array} & & \begin{array}{c} \textbf{return: } X = Truck, \\ Y = Car \end{array}
\end{tikzcd}$$

Figure 3.8: Matching with the **Diff** constructor.

expressions was made. For every successful match, the remaining expressions of the set were matched with the group expression variable. This step could potentially generate many instantiations from a given one.

Figure 3.9 shows an example of such a matching. In this example, the template expression group consists of named expressions only. Failed matchings are indicated with a red cross.

$$\begin{tikzcd}
& \begin{array}{c} match(\ \equiv'(\{\ \sqcup'(\{A, B\}), C\}), \\ \equiv(\{\ \sqcup(\{Cat, Dog\}),\ \ Pet\})) \end{array} \arrow[dl] \arrow[dr] & \\
\begin{array}{c} match(\sqcup'(\{A, B\}),\ \sqcup(\{Cat, Dog)\}), \\ match(C,\ \ Pet) \end{array} \arrow[d] \arrow[drr] & & \xcancel{\begin{array}{c} match(\sqcup'(\{A, B\}), Pet), \\ match(C,\ \sqcup(\{Cat, Dog\})) \end{array}} \\
\begin{array}{c} match(A, Cat), \\ match(B, Dog), \\ C\ =\ Pet \end{array} \arrow[d] & & \begin{array}{c} match(A, Dog), \\ match(B, Cat), \\ C\ =\ Pet \end{array} \arrow[d] \\
\begin{array}{c} \textbf{return:} \\ A\ =\ Cat, \\ B\ =\ Dog, \\ C\ =\ Pet \end{array} & & \begin{array}{c} \textbf{return:} \\ A\ =\ Dog, \\ B\ =\ Cat, \\ C\ =\ Pet \end{array}
\end{tikzcd}$$

Figure 3.9: Ambiguous instantiation.

In practice, a free variable could appear in multiple axioms within an inference rule, for instance, the free variable Y appears in both premises of the rule shown in (3.14).

$$X \sqsubseteq Y \wedge\ Y \sqsubseteq Z\ \rightarrow X \sqsubseteq Z \tag{3.14}$$

For that reason, a global set of instantiations (mappings of free variables to concrete expressions) was maintained when performing a matching. When matching a free variable to an expression, this set of instantiations was updated as follows:

- All instantiations that contained a mapping of this free variable to a different expression were removed, since these instantiations were inconsistent.
- All instantiations that contained a mapping of this free variable to that same expression were left unchanged.
- All instantiations that did not contain this free variable were updated with the new mapping.

Matching of a set of axioms to a set of templates is shown in Algorithm 4.

Algorithm 4 RuleMatching(*axioms, templates, restrictions*)

```
1: $Instantiations \leftarrow \{\{T \rightarrow owlThing,\ F \rightarrow owlNothing\}\}$
2: $Instantiations' \leftarrow \{\}$
3: for $i = 0$ to $axioms.size$ do
4:     $Axiom \leftarrow axioms.get(i)$
5:     $Template \leftarrow templates.get(i)$
6:     for $Instantiation \in Instantiations$ do
7:         $AxiomInstantiations \leftarrow findInstantiations($
8:                                     $Axiom, Template, Instantiation)$
9:         $Instantiations'.addAll(AxiomInstantiations)$
10:     end for
11:     $Instantiations \leftarrow Instantiations'$
12:     $Instantiations' \leftarrow \{\}$
13: end for
14: $Instantiations \leftarrow enforceRestrictions(Instantiations, restrictions)$
15: return $Instantiations$
```

For every rule, matching proceeded through generation of all possible instantiations of the free variables of the rule.

The set of instantiations is initialised with the default instantiation that matches predefined template symbols T and F to the predefined class expressions *owlThing* and *owlNothing*, respectively. After initialisation, a simple iterative strategy was adopted that updated the previous set of instantiations by matching the next axiom and the next template. Once all possible instantiations were produced, rule restriction checking was performed that removed any instantiations that did not conform to the rule restrictions.

The next section will discuss how conclusion axioms were generated for a given rule.

3.4.4 Conclusion Generation

Another key feature of the algorithm is generating a conclusion axiom from an inference rule, given a set of axioms that match the rule premises.

Similarly to matching, conclusion generation was done recursively, by generating the top-level constructor of an expression, and then filling in its argument values with recursively generated sub-expressions. The recursion bottomed out when the template pattern was a free variable.

For the majority of inference rules, every free variable in the conclusion was used in at least one of the premises. Hence, in order to generate a conclusion for these rules, the matching algorithm was run on the premises first. Then, for every instantiation in the computed instantiation set, a conclusion was generated unambiguously, since all of its free variables were already given values in the instantiation.

Figure 3.10 shows a simple example of unambiguous conclusion generation for the template $X \sqsubseteq {=}nR_o.Z$, from the instantiation $\{X \rightarrow Car,\ n \rightarrow 4,\ S_o \rightarrow hasPart,\ Z \rightarrow Wheel\}$.

$$\begin{tikzcd}
generate(\sqsubseteq'(X,\ ='(n,\ S_o,\ Z))) \arrow[d] \\
\sqsubseteq(generate(X),\ generate(='(n,\ S_o,\ Z))) \arrow[d] \\
\sqsubseteq(Car,\ =(generate(n),\ generate(S_o),\ generate(Z))) \arrow[d] \\
\sqsubseteq(Car,\ =(4,\ hasPart,\ Wheel)) \arrow[d] \\
\textbf{return: } \sqsubseteq(Car,\ =(4,\ hasPart,\ Wheel))
\end{tikzcd}$$

Figure 3.10: Unambiguous generation example.

A few inference rules contained free variables in their conclusions that were not present in any of their premises. These cases are discussed below.

Rule "ObjExt", given in (3.15), had an uninstantiated cardinality variable (variable n_2).

$$\begin{aligned} &X \sqsubseteq\ \geq n_1 R_o.Y, \quad n_1 \geq n_2 \geq 0 \\ &\rightarrow X \sqsubseteq\ \geq n_2 R_o.Y \end{aligned} \tag{3.15}$$

To generate conclusions with uninstantiated cardinalities, cardinality restrictions were used. If a variable had an upper bound restriction UB and a lower bound restriction LB, then values in the range $[LB,\ min(LB+5, UB)]$ were generated. Otherwise, no generation was done. Restricting the range to $min(LB+5, UB)$, instead of generating all values from LB to UB ensured that the amount of generated results was computationally feasible.

An example is shown in Figure 3.11, where the conclusion $X \sqsubseteq\ =nS_o.Z$ is generated, given an instantiation $\{X \rightarrow Car,\ S_o \rightarrow hasPart,\ Z \rightarrow Wheel\}$, as well as a restriction $12 \geq n \geq 4$.

$$\begin{tikzcd}
 & generate(\sqsubseteq'(X,\ ='(n,\ S_o,\ Z))) \arrow[d] & \\
 & \sqsubseteq(generate(X),\ generate(='(n,\ S_o,\ Z))) \arrow[d] & \\
 & \sqsubseteq(Car,\ =(generate(n),\ generate(S_o),\ generate(Z))) \arrow[dl] \arrow[dr] & \\
\sqsubseteq(Car,\ =(4,\ hasPart,\ Wheel)) \arrow[d] & \cdots & \sqsubseteq(Car,\ =(9,\ hasPart,\ Wheel)) \arrow[d] \\
\begin{matrix}\textbf{return:} \\ \sqsubseteq(Car,\ =(4,\ hasPart,\ Wheel))\end{matrix} & \cdots & \begin{matrix}\textbf{return:} \\ \sqsubseteq(Car,\ =(9,\ hasPart,\ Wheel))\end{matrix}
\end{tikzcd}$$

Figure 3.11: Uninstantiated cardinality expression generation.

Rules “Top” and “Bot”, given in (3.16) and (3.17), respectively, had uninstantiated class expression identifiers (the variable Y).

$$\top \sqsubseteq X \ \rightarrow \ Y \sqsubseteq X \tag{3.16}$$

$$X \sqsubseteq \bot \ \rightarrow \ X \sqsubseteq Y \tag{3.17}$$

An uninstantiated class expression free variable could in principle be instantiated with any class expression at all, making the amount of possible instantiations infinite. For simplicity, these two rules were only used for matching and not generation.

Some rules had uninstantiated anonymous group expressions in their conclusion, such as rule “ObjUni-2”, given in (3.18). Here, $\{Y_1, \ldots, Y_n\}$ is an uninstantiated anonymous group expression.

$$\begin{aligned}&Y_1 \sqcup \ldots \sqcup Y_n \sqcup Z_1 \sqcup \ldots \sqcup Z_n \sqsubseteq X,\ n \geq 1, m \geq 1\\ &\rightarrow Y_1 \sqcup \ldots \sqcup Y_n \sqsubseteq X\end{aligned} \tag{3.18}$$

For rules with uninstantiated anonymous group expressions, subset restrictions were used. If the anonymous group was a subset of an instantiated group, all possible non-empty, proper subsets of that group were generated as possible instantiations. Otherwise no generation was done.

An example is given in Figure 3.12, where the conclusion $X \equiv Y \equiv Z_1 \equiv \ldots \equiv Z_n$ is generated, given an instantiation $\{X \rightarrow Car,\ Y \rightarrow Bike,\ Q \rightarrow \{Green,\ Red\}\}$, as well as a restriction $\{Z_1, \ldots, Z_n\} \subset Q$. Note that the anonymous group $\{Z_1, \ldots, Z_n\}$ is represented by a single anonymous group variable Z.

$$\begin{tikzcd}
 & generate(\equiv'(\{X,\ Y\},\ Z)) \arrow[d] & \\
 & \equiv(\{generate(X),\ generate(Y)\} \cup generate(Z)) \arrow[dl] \arrow[dr] & \\
\equiv(\{Car,\ Bike\} \cup \{Red\}) \arrow[d] & & \equiv(\{Car,\ Bike\} \cup \{Green\}) \arrow[d] \\
\textbf{return:}\ \equiv(\{Car,\ Bike,\ Red\}) & & \textbf{return:}\ \equiv(\{Car,\ Bike,\ Green\})
\end{tikzcd}$$

Figure 3.12: Uninstantiated cardinality expression generation.

This section described how conclusion generation was done and concludes the discussion of the algorithm and inference rule implementation. The next section will focus on how these components were tested.

3.5 Testing

The project was composed of a large number of interacting modules, all of which were tested to ensure that all components were working as required. The following sections briefly describe the testing procedures.

3.5.1 Unit Testing

A modular implementation was used for project development, which allowed a convenient way of testing different parts of the system separately. Table 3.4 lists the individual modules that were tested, together with short descriptions of the tests.

Module Name	**Test Description**
Initial Tree Generation	Testing correct generation of initial trees, including correct application of exception cases.
Rule Application	Testing correct application of inference rules to expressions. Every inference rule was tested individually.
Rule Generation	Testing correct generation of expressions from inference rules. Every inference rule was tested individually.
Rule Restriction	Testing correct application of rule restrictions during matching and generation.
Intermediate Node Generation	Testing correct generation of intermediate nodes during computation of complete proof trees.

Table 3.4: Tested Modules

The OWLAPI provided simple methods for generating expressions and axioms, which allowed creation of arbitrarily complicated test cases for all of the modules.

3.5.2 Integration Testing

Apart from testing every module in isolation using unit tests, an integration test suite was developed to test the entire system end-to-end. Test cases consisted of an input entailment-justification pair, and a corresponding output set of complete proof trees that should have been generated from that input pair.

The OWLAPI and the Explanation API provided simple ways of generating arbitrarily complicated test cases.

3.6 Summary

Overall, this chapter described the main implementation details of the project. Implementation of the algorithm and the inference rules was discussed, and justifications for various design choices were provided. The next section will focus on how this algorithm was evaluated and will demonstrate different extensions that were implemented to further improve its performance.

Chapter 4

Evaluation

This chapter describes how the project was evaluated, and the extensions that were implemented to investigate potential performance improvements.

The algorithm and inference rule set were designed to work with any ontology of any domain and structure. Thus a key performance measure used was the percentage of justification-entailment pairs for which the algorithm was capable of producing at least one proof tree, out of a set of justification-entailment pairs computed from a diverse corpus of ontologies. This result would be indicative of how generally-applicable the algorithm and rule set are.

Section 4.1 describes the selection of the corpus and the extraction of the dataset. Section 4.2 presents the rule coverage results. Section 4.3 describes the implemented extensions that were designed to increase the rule coverage even further. Section 4.4 discusses how successful the implemented project was at meeting the original success criteria. Finally, Section 4.5 concludes with a brief summary of the entire chapter.

4.1 Dataset Construction

This section discusses how the dataset used for evaluation of the inference rule set was produced. Section 4.1.1 describes how the ontology corpus was selected, and Section 4.1.2 describes how the dataset was extracted from this corpus.

4.1.1 Corpus Selection

Constructing a corpus that is diverse and not specific to any domain is a time-consuming, laborious process with no commonly agreed metrics for assessing corpus generality (a thorough discussion is given in [18]). For that reason, manual corpus construction was outside the scope of this project and an existing corpus was selected instead.

The *Manchester OWL Repository* [5] was selected as the corpus source. This repository provided a choice of many ontology corpora of varying sizes and domains. It also provided additional metrics and measures for every corpus, such as counts of different types of axioms, class expressions etc. which are useful for analysing the corpus in more detail. Other alternative repositories were available, such as [15, 37, 27], though they were less suitable for this project due to them being outdated, domain-specific, poorly documented or a combination thereof.

The *ORE 2015 Reasoner Competition Corpus* [6] was selected from the chosen repository. This corpus was originally used as a benchmarking corpus in the evaluation of ontology reasoners. Benchmarking corpora such as this one are handcrafted to be diverse, complex and rich in context. This corpus contains a total of 1920 ontologies, ranging from 5 to over 3 million axioms in size and varying in domain, expressivity etc. Most of these ontologies are taken from other existing corpora, such as BioPortal [38], Oxford Ontology Library [30], or MOWLCorp [19, 35]. It was thus a suitable choice for evaluating the rule set coverage. More details about how the corpus was constructed can be found in [31].

4.1.2 Data Extraction

Once the corpus was selected, a set of entailment-justification pairs were extracted from it. The HermiT Reasoner was used to compute all non-trivial subsumption entailments for every ontology. A non-trivial subsumption was defined to be a subsumption between any two classes in the ontology, except for: $X \sqsubseteq \top$, $\bot \sqsubseteq X$ and $X \sqsubseteq X$, where X is any class in the ontology. These three cases hold by definition for any class, and were thus ignored.

Once all such subsumptions were computed for an ontology, 150 of them were selected at random. This was done to keep the dataset size manageable, since some larger ontologies generated hundreds of thousands of such subsumptions. Randomised selection ensured that the chosen subset was representative.

Finally, for every such subsumption, the Explanation API was used to compute a justification of the subsumption, which (as described in the Preparation chapter), is a minimal set of axioms from which the subsumption can be inferred. All justification-entailment pairs where the number of axioms in the justification was greater than 10 were removed from consideration, since larger justifications were considered too difficult to be understood by users, as described in [24]. All the remaining entailment-justification pairs constituted the extracted dataset.

In order to compute the dataset in reasonable time, a timeout of 10 minutes was set for every ontology during the computation of its subsumption entailments, and a timeout of 30 seconds was set for every computation of a justification from a subsumption. The above process was run using Amazon Web Services (AWS) [3] to speed up computation, on a single *m5.12xlarge* instance (detailed specifications of this instance can be found at [4]). The final dataset consisted of 175,018 such subsumption-entailment pairs.

4.2 Rule Coverage Analysis

This section describes the results obtained from the extracted dataset. Section 4.2.1 describes the initial results obtained, while Section 4.2.2 talks about results obtained from a more refined dataset.

4.2.1 Initial Results

Once the dataset was generated, the rule coverage was measured by running the algorithm on every entailment-justification pair in the dataset. A timeout of 60 seconds was set for every entailment-justification pair. This process was run on a Macbook computer with 16GB of RAM and an Intel Core i7 2.5GHz processor. The results are presented in Table 4.1.

A similar study was conducted and described in the original research paper [24]. A total of 153,808 justification-entailment pairs were extracted from 179 ontologies taken from the TONES [36], ODP [26] and Swoogle [7] repositories (all of which are now out of date), using techniques similar to those described in Section 4.1.2. The coverage results obtained in that study were 75.6%, 2.9% and 21.5% for the computed, timed out and failed cases, respectively.

Case Type	**Case Count**	**Percentage**
Computed	144,078	82.3%
Timed out	25,307	14.5%
Failed	5633	3.2%
Total	175,018	100%

Table 4.1: Initial coverage results

After the initial analysis, a more fine-grained analysis of individual rule usages was produced by counting how many times each rule in the rule set was used in all of the computed trees. Since the algorithm often returned more than one tree for a given justification-entailment pair, the amount of computed trees was considerably greater than the size of the dataset. The results are presented in Figure 4.1. A total of 533,598 trees have been computed from 144,078 justification-entailment pairs.

As Figure 4.1 shows, there was an order of magnitude difference in usage counts between the rules.

Observation of the computed cases, as well as the rule usage counts, revealed an interesting property of the dataset, which will be discussed further in the next section.

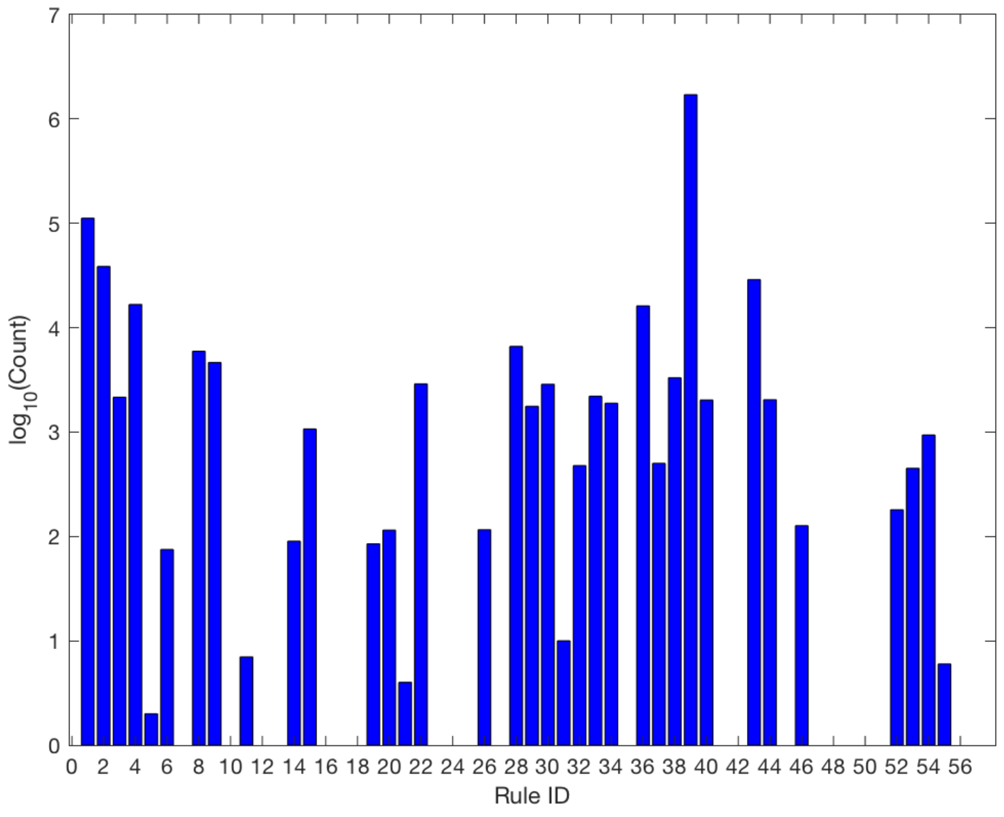


Figure 4.1: Initial rule usage count.

4.2.2 Refined Dataset Results

As shown in Figure 4.1, rule 39 had a larger count than any of the other rules. This rule is given in (4.1).

$$\begin{aligned} &X \sqsubseteq Y \ \wedge \ Y \sqsubseteq Z \\ &\rightarrow \ X \sqsubseteq Z \end{aligned} \tag{4.1}$$

Observation of the dataset and of the computed proof trees revealed that a large number of entailment-justification pairs were of the following form: $X_1 \sqsubseteq X_2 \wedge X_2 \sqsubseteq X_3 \ \wedge \ldots \wedge \ X_{n-1} \sqsubseteq X_n \ \rightarrow X_1 \sqsubseteq X_n$, which is a generalisation of rule 39.

The above observation was used to derive a useful result, which is summarised in Lemma 4.2.1:

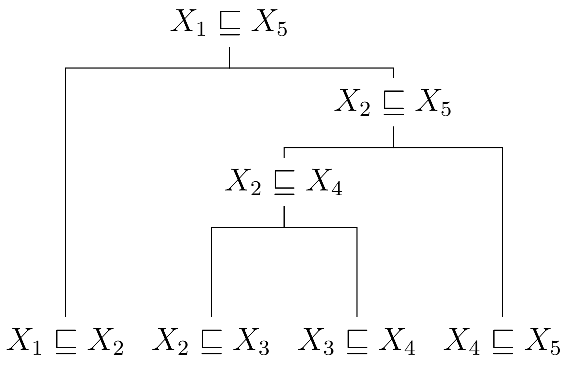
Lemma 4.2.1. *A complete proof tree only consists of applications of rule 39 iff the complete proof tree was created from a justification and an entailment that can be written in the following form:*
$X_1 \sqsubseteq X_2 \ \wedge X_2 \sqsubseteq X_3 \ \wedge \ldots \wedge \ X_{n-1} \sqsubseteq X_n \ \rightarrow \ X_1 \sqsubseteq X_n$

These cases are referred to as *transitive cases* in the rest of this chapter.

Figure 4.2 gives an example demonstrating this equivalence. Subfigure 4.2a shows the justifications and the entailment, and subfigure 4.2b shows a possible corresponding complete proof tree. Here, the complete proof tree is drawn in a way that emphasises the equivalence described in Lemma 4.2.1. Proof labels are omitted since only rule 39 is applied.

$$X_1 \sqsubseteq X_2 \ \wedge\ X_2 \sqsubseteq X_3 \ \wedge\ X_3 \sqsubseteq X_4 \ \wedge\ X_4 \sqsubseteq X_5 \ \rightarrow\ X_1 \sqsubseteq X_5$$

(a) An entailment and justification axioms.



(b) Proof tree representation.

Figure 4.2: Transitive case equivalence.

Lemma 4.2.1 was used to develop an algorithm that could check whether a given justification-entailment pair was an instance of the transitive case. This algorithm worked in $O(n^2)$ time, where n was the number of axioms in the justification. Due to the restriction on justification size, this check could be done rapidly for any of the justification-entailment pairs in the dataset, including timed out cases.

This algorithm was used to determine that over 70% of test cases were transitive cases. This meant that transitive cases would have had a considerable impact on coverage results. Thus, a second coverage analysis was performed on a modified dataset that was obtained by using the above algorithm to remove all transitive cases from the original dataset. Table 4.2 shows the updated case counts.

Case Type	**Case Count**	**Percentage**
Computed	34,507	72.3%
Timed out	7618	16.0%
Failed	5633	11.7%
Total	47,758	100%

Table 4.2: Coverage results without transitive cases.

Table 4.2 also shows that there was a high percentage of timed out cases (16%). In order to reduce the amount of timed out cases, AWS were used once again to provide a source of larger computing power. The coverage analysis and rule count analysis were re-run on an m5.12xlarge instance, using multi-instancing by running 10 processes in parallel.

Table 4.3 and Figure 4.3 show the coverage and rule count analyses, respectively, obtained from this *refined* dataset. For the rule usage count, a total of 130,794 trees have been computed from the 36,618 justification-entailment pairs.

Case Type	**Case Count**	**Percentage**
Computed	36,618	76.7%
Timed out	4627	9.7%
Failed	6513	13.6%
Total	47,758	100%

Table 4.3: Coverage results without transitive cases using AWS.

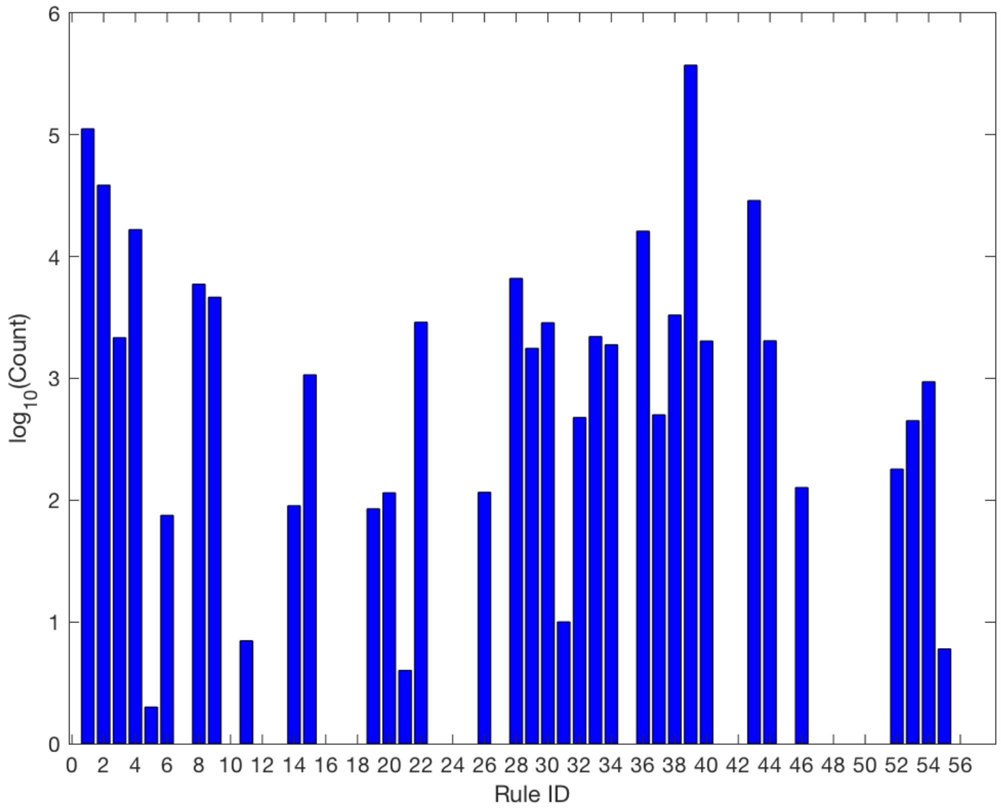


Figure 4.3: Refined rule usage count.

As Table 4.3 shows, the percentage of timeout cases was reduced from 16% to 9.7%. The algorithm performed reasonably well even on the refined dataset, obtaining a coverage of 76.7%.

4.3 Extensions

After examining the rules and the techniques used to construct them, several potential reasons for imperfect coverage were identified. The impact from three main reasons was quantitatively evaluated and will be discussed below in Sections 4.3.1, 4.3.2 and 4.3.3.

4.3.1 Unused Axiom Count

As mentioned in Section 3.4.1 of the Implementation chapter, 15 out of 30 OWL axiom constructors were not used by any of the rules in the rule set. This means that any justification which used any of these constructors would have produced a failed case, since a justification has no redundant axioms (by definition), and thus it is impossible to construct a derivation of the entailment without using every single axiom at least once.

The extent to which this observation affected rule coverage was investigated by counting the percentage of failed cases containing at least one justification axiom that used an unimplemented constructor. The results are given in Table 4.4.

Axiom Constructor	**Failed Cases**
$\mathbf{DisUni}(A, C, D[, \ldots])$	278
$R_o(a, b)$	242
$C(a)$	88
$\mathbf{Ref}(R_o)$	87
$R_d(a, l)$	4
$R_d \sqsubseteq S_d$	1
$R_o \equiv S_o[\equiv \ldots]$	1
$R_d \equiv S_d[\equiv \ldots]$	0
$\neg R_o(a, b)$	0
$\mathbf{Dis}(R_d, S_d[, \ldots])$	0
$\mathbf{Asym}(R_o)$	0
$\mathbf{Sam}\ (a, b, [, \ldots])$	0
$\mathbf{Irr}(R_o)$	0
$\neg R_d(a, l)$	0
$\mathbf{Dis}(R_o, S_o[, \ldots])$	0

Table 4.4: Unused axiom constructors.

As Table 4.4 shows, the total percentage of failed cases containing axioms using unimplemented constructors was only 10.8%. Hence this issue does not have a large impact on the number of failed cases.

4.3.2 Expression Depth

This section investigates the impact of only considering class expressions of limited depth.

The depth of a class expression is defined recursively as follows: the depth of a class name (an atomic class expression) is 1. The depth of a complex class expression consisting of a constructor and its arguments is the largest depth among all of the class expression constructor arguments plus 1. All other types of expressions (e.g. object properties) are not counted.

Figure 4.4 shows an example of depth calculation (written in prefix notation for clarity), where C and D are class names, R_o and S_o are object property names, and n is an integer.

$$\begin{tikzcd}
\mathbf{Depth}(\sqcap(\{\geq(n, R_o, \exists(S_o, C)),\ D\})) \arrow[d] \\
\max(\mathbf{Depth}(\geq(n, R_o, \exists(S_o, C))),\ \mathbf{Depth}(D)) + 1 \arrow[d] \\
\max(\mathbf{Depth}(\exists(S_o, C)) + 1,\ 1) + 1 \arrow[d] \\
\max(\mathbf{Depth}(C) + 1 + 1,\ 1) + 1 \arrow[d] \\
\max(1 + 1 + 1,\ 1) + 1 \arrow[d] \\
3 + 1 \arrow[d] \\
4
\end{tikzcd}$$

Figure 4.4: Class expression depth calculation.

OWL syntax places no restrictions on class expression depth, meaning that an ontology may contain class expressions of any depth. The inference rule set, however, only consists of class expression templates of depths 1, 2 and 3. This is based on the assumption that in practice, ontologies rarely contain class expressions of a higher depth.

The refined dataset was used to investigate the validity of these assumptions. The definition of class expression depth given above was used to develop an algorithm that, given a justification, returned the maximum class expression depth of any class expression in any of its axioms.

The expression depth counts for the failed cases are given in Table 4.5.

Class Expression Depth	**Case Count**
1	1514
2	3364
3	1338
4	183
5	29
6	8
7	5
8	12

Table 4.5: Failed case depths.

Expression depth counts for the cases computed successfully are given in Table 4.6.

Class Expression Depth	**Case Count**
1	17630
2	6445
3	11638
4	494
5	268
6	57
7	68
8	8
9	1
10	3
11	6

Table 4.6: Computed case depths.

Table 4.5 shows that over 96% of failed cases contained class expressions of depth less than 4.

Table 4.6 further shows that a large portion of justifications containing class expressions of depth greater than 3 were still computed successfully. This implies that in practice it is often not necessary to use axiom class expressions at all different depths when computing an explanation.

Thus, these results show that the assumption was warranted, and limited class expression depth does not have a serious impact on rule coverage.

4.3.3 Edge Cases

As mentioned in Section 2.1.2 of the Preparation chapter, every ontology has an inbuilt class $\top$, which subsumes any other class or class expression. Thus for any class expression C in any ontology, $C \sqsubseteq \top$ by definition.

Many of the inference rules contain a premise of the form $X \sqsubseteq Y$. However, if a justification contains $\top$ as a class expression in one of its axioms, then $X \sqsubseteq \top$ is an implicit axiom that is not included in the justification, but will hold for any class expression X. Since this axiom is implicit, rule premises will not be instantiated properly and the algorithm will not generate a proof tree. A similar argument can be constructed for the inbuilt class $\bot$, which is subsumed by any class expression.

For example, (4.2) shows the "ObjSom-SubCls" rule.

$$\begin{aligned} &X \sqsubseteq \exists R_o.Y \wedge\ Y \sqsubseteq Z \\ &\rightarrow\ X \sqsubseteq \exists R_o.Z \end{aligned} \tag{4.2}$$

Given a concrete premise axiom: $CatOwner \sqsubseteq \exists hasPet.Cat$ (every cat owner has a cat as a pet) and the premise axiom: $Cat \sqsubseteq Animal$ (every cat is an animal), this rule can be used to infer: $CatOwner \sqsubseteq \exists hasPet.Animal$ (every cat owner has an animal as a pet).

However, given the same premise axiom: $CatOwner \sqsubseteq \exists hasPet.Cat$, this rule cannot be used to infer: $CatOwner \sqsubseteq \exists hasPet.\top$ (every cat owner has something as a pet), because the second required premise axiom: $CatOwner \sqsubseteq \top$ will be implicit.

In practice, there are many ways of incorporating this implicit assumption into the rule set. However, selecting a method that maintains useful properties of the algorithm, such as termination, while avoiding a considerable increase in computation time is a non-obvious task. For instance, introducing a single rule $X \sqsubseteq Y\ \rightarrow\ X \sqsubseteq \top$ will result in infinite looping if Y is ever instantiated to $\top$, unless extra restrictions (such as $Y \neq \top$) are implemented.

More importantly, all inference rule were constructed using extensive human trials, as described in the Preparation chapter, thus any such change must be re-evaluated using a similar approach.

For reasons outlined above, the chosen methodology only focused on demonstrating whether the above mentioned edge cases significantly affect the percentage of failed cases. A new set of rules was temporarily added to the rule set and the failed case coverage was re-computed. Developing changes that could be permanently incorporated into the algorithm or the rule set was outside of the scope of this project.

All inference rules containing a premise of the form $X \sqsubseteq Y$ are given in Table 4.7.

For every such rule, a new rule was added, where the premise of the form $X \sqsubseteq Y$ was removed, and any occurrence of Y was substituted with $\top$. These new rules are

Rule ID	Inference Rule
15	$X \sqsubseteq Y \wedge$ **Dis** $(X, Y, \ldots)$ $\rightarrow X \sqsubseteq \bot$
28	$X \sqsubseteq Y \wedge$ **Dom** (R_o, X) $\rightarrow$ **Dom** (R_o, Y)
32	$X \sqsubseteq Y \wedge$ **Rng** (R_o, X) $\rightarrow$ **Rng** (R_o, Y)
39	$X \sqsubseteq Y \ \wedge \ Y \sqsubseteq Z$ $\rightarrow X \sqsubseteq Z$
40	$X \sqsubseteq Y \ \wedge \ X \sqsubseteq Z$ $\rightarrow X \sqsubseteq (Y \sqcap Z)$
43.1	$Y \sqsubseteq Z \ \wedge \ X \sqsubseteq \exists R_o.Y$ $\rightarrow X \sqsubseteq \exists R_o.Z$
43.2	$Y \sqsubseteq Z \ \wedge \ X \sqsubseteq \geq nR_o.Y,\ n \geq 0$ $\rightarrow X \sqsubseteq \geq nR_o.Z$
43.3	$Y \sqsubseteq Z \ \wedge \ X \sqsubseteq =nR_o.Y,\ n \geq 0$ $\rightarrow X \sqsubseteq =nR_o.Z$
45	$Y \sqsubseteq Z \ \wedge \ X \sqsubseteq (Y \sqcup Z)$ $\rightarrow X \sqsubseteq Z$
52	$U \sqsubseteq X \ \wedge \ V \sqsubseteq Y \wedge$ **Dis** (X, Y) $\rightarrow$ **Dis** (U, V)
53	$X \sqsubseteq Y \ \wedge \ X \sqsubseteq Z \wedge$ **Dis** (Y, Z) $\rightarrow X \sqsubseteq \bot$
54	$U \sqsubseteq Z \ \wedge \ V \sqsubseteq Z \ \wedge \ X \sqsubseteq (U \sqcup V)$ $\rightarrow X \sqsubseteq Z$

Table 4.7: Rules with edge cases.

given in Table 4.8.

Rule ID	Inference Rule
15.top	**Dis** $(X, \top, \ldots) \rightarrow X \sqsubseteq \bot$
28.top	**Dom** $(R_o, X) \rightarrow$ **Dom** $(R_o, \top)$
32.top	**Rng** $(R_o, X) \rightarrow$ **Rng** $(R_o, \top)$
43.1.top	$X \sqsubseteq \exists R_o.Y \rightarrow X \sqsubseteq \exists R_o.\top$
43.2.top	$X \sqsubseteq \geq nR_o.Y, \ \ n \geq 0$ $\rightarrow X \sqsubseteq \geq nR_o.\top$
43.3.top	$X \sqsubseteq =nR_o.Y, \ \ n \geq 0$ $\rightarrow X \sqsubseteq =nR_o.\top$

Table 4.8: $\top$ edge case rules.

Rule 39 had a corresponding ".top" rule in the rule set already. New rules corresponding to rules 40, 45, 53 and 54 would have been redundant rules that do not provide any new information. A ".top" rule corresponding to rule 52 would have had a free variable in its conclusion. These rules were thus not taken into consideration.

The failed cases were then re-computed using the extended rule set. Some of these new rules could have caused infinite looping of the algorithm, thus a timeout of 30 seconds was set for every re-computation of a failed case, to ensure termination. This approach resulted in 20.3% of the failed cases being computed successfully, which is a considerable improvement.

Overall, the percentage of cases containing $\top$ in at least one of their axioms was 43.4% for failed cases and only 17.6% for computed cases in the refined dataset. These results, together with the new coverage results presented in this section, suggest that these edge cases may have a significant impact on rule coverage. A more thorough investigation could potentially improve the coverage even further.

Unfortunately, using the above approach to produce new rules for the "$\perp$" edge case resulted in rules that were either redundant (conveyed no new information), or rules with free variables in their conclusion. Consideration of these cases was thus outside of the scope of this project.

4.4 Success Criteria

This section discusses the extent to which the project met the original requirements.

The success criteria for the project, as stated in the Preparation chapter, are given below:

1. The algorithm and the inference rule set described in [24] are implemented successfully.
2. The algorithm can be shown to work and construct proofs given an entailment and a justification.
3. A dataset consisting of entailment-justification pairs is extracted from a diverse, general corpus of ontologies.
4. A quantitative investigation of inference rule coverage is conducted using this dataset.

A working algorithm capable of producing a proof tree given an entailment and a justification using the predefined inference rule set was successfully developed, as described in Sections 3.3 and 3.4 of the Implementation chapter, which addresses points 1 and 2 of the success criteria.

Section 4.1 provides a discussion of how a corpus and dataset were obtained, as well as the calculation of inference rule coverage, which addresses points 3 and 4. Furthermore, these sections explore an interesting property of the dataset (the high amount of transitive cases) and the impact this property has on the coverage results.

The success criteria for the extension steps, as stated in the Preparation chapter, are given below:

1. The major reasons (at least two) for why the algorithm and rule set may have failed to produce a proof for the failed cases are given.
2. A quantitative analysis is performed on the dataset to investigate the impact of the above reasons.
3. The results of the quantitative analysis are used to determine the impact each reason has on rule coverage.

Three reasons that affect rule coverage are presented in Section 4.3. For every reason, a quantitative analysis designed to evaluate its impact on rule coverage is also given. Thus, the success criteria of both the core and the extension steps were met successfully.

Furthermore, the obtained rule coverage results are similar to those shown in the original research paper [24] (as shown in Section 4.2.1), despite them being computed using different datasets. Therefore, results presented in this chapter can be used to support the claims made in the research paper regarding general applicability of the inference rule set to any ontology corpus.

4.5 Summary

This chapter discussed how a dataset used for evaluating the algorithm was computed and the results obtained after running the rule coverage analysis on this dataset. Reasons for imperfect coverage were also presented with quantitative measurements of their impact. The final section demonstrated that all of the original success criteria have been met successfully.

The next chapter will provide overall conclusions of the work completed for this project.

Chapter 5

Conclusion

This chapter provides a brief summary of the work done during project development by listing the relevant accomplishments. A discussion of possible future work that was not explored due to time constraints is also provided. The final section includes some closing remarks.

5.1 Accomplishments

Overall the project was a success. The tree-generating algorithm and inference rule set described in [24] were implemented successfully. A large dataset of entailment-justification pairs was extracted from a diverse corpus of ontologies, and two analyses of rule coverage were provided. Both of these attained a coverage score of over 75%, which suggests that this algorithm generalises well.

Potential reasons for the coverage score being imperfect were also identified and analysed, with one of these reasons being responsible for over 20% of the failed cases. These reasons can be used to determine the potential next steps that could be taken to improve rule coverage even further.

5.2 Useful Modifications

In hindsight, parallelisation of some of the components would have been advantageous.

Dataset extraction and rule coverage analysis steps, described in Sections 4.1.2 and 4.2 of the Evaluation chapter, both consist of many independent steps, thus making these modules suitable for parallelisation. The tree generation algorithm also includes steps that could be run in parallel, such as inference rule search and application, or parallel handling of incomplete trees. These improvements would offer a considerable computation speed up, which would allow extraction and analysis of larger datasets, potentially offering more insightful solutions, as well as a reduction in timed out cases.

Developing a solution that used parallelisation would have taken more time to implement and test, and was thus not included in the original objectives, given that it was difficult to accurately predict how long algorithm development would take. Nevertheless, parallelisation was partially exploited by running several instances of coverage analysis in parallel on a partitioned dataset, as described in the Evaluation chapter.

5.3 Future Work

As mentioned in the research paper [24], this algorithm was designed to handle subsumption entailments only. Consideration of other types of entailments is thus one possible next step that can be explored to make the project even more generally applicable.

The research paper also introduced a *Facility Index* metric, which can be used to assign a numerical understandability score to a proof tree. By generating a large set of proof trees (e.g. using similar techniques to the ones mentioned the Evaluation chapter) and using the Facility Index, it may be possible to identify commonly-occurring reasoning patterns with a high understandability score. These results would be useful for work associated with analysis of user-friendly reasoning tactics in reasoning systems, such as [17].

Finally, rule usage counts computed in the Evaluation chapter showed that some inference rules were used an order of magnitude times more often than others. Due to time constraints, further investigation of this result was outside the scope of this project. Nevertheless, exploring the reasons for why this is the case and how these usage counts change depending on the dataset may lead to interesting new insights and optimisations of the rule set.

5.4 Closing Remarks

In conclusion, this was an interesting and rather challenging project to work on. The completed work provided useful insights into the domain of ontology reasoning by analysing and expanding on existing work related to generating understandable explanations of entailments.

Bibliography

[1] Boris Motik et al. *HermiT OWL Reasoner*. 2014. URL: `http://www.hermit-reasoner.com/download.html`.

[2] Enrico Franconi Alex Borgida and Ian Horrocks. "Explaining ALC Subsumption". In: *European Conference on Artificial Intelligence (ECAI 2000)* (2000), pp. 209–213.

[3] *Amazon Web Services*. URL: `https://aws.amazon.com/`.

[4] *Amazon Web Services*. URL: `https://aws.amazon.com/ec2/instance-types/m5/`.

[5] Nicolas Matentzoglu Bijan Parisa. *Manchester OWLRepository*. 2017. URL: `http://mowlrepo.cs.manchester.ac.uk/`.

[6] Nicolas Matentzoglu Bijan Parsia. *ORE 2015 Reasoner Competition Dataset*. 2015. URL: `https://zenodo.org/record/18578#.Wstvi9NuaqA`.

[7] Li Ding et al. "Swoogle: a search and metadata engine for the semantic web". In: *International Conference on Information and Knowledge Management (CIKM)* (2004), pp. 652–659.

[8] *Dropbox*. URL: `https://www.dropbox.com`.

[9] N F. Noy and Deborah Mcguinness. "Ontology Development 101: A Guide to Creating Your First Ontology". In: *Knowledge Systems Laboratory* 32 (Jan. 2001).

[10] *Friend of a Friend (FOAF): an experimental linked information system*. URL: `http://www.foaf-project.org/`.

[11] *GitHub*. URL: `https://github.com/`.

[12] Matthew Horridge. *A Practical Guide To Building OWL Ontologies Using Protege 4 and CO-ODE Tools, Edition 1.3*. 2011. URL: `http://mowl-power.cs.man.ac.uk/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_3.pdf`.

[13] Matthew Horridge. "Justification Based Explanation in Ontologies". PhD thesis. University of Manchester, 2011.

[14] Matthew Horridge. *OWL Explanation API*. 2015. URL: `https://github.com/matthewhorridge/owlexplanation`.

[15] Matthew Horridge. *TONES Ontology Repository (corpus)*. Oct. 2015. DOI: `10.5281/zenodo.32717`. URL: `https://doi.org/10.5281/zenodo.32717`.

[16] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. "The Even More Irresistible SROIQ". In: *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning.* KR'06. Lake District, UK: AAAI Press, 2006, pp. 57–67. ISBN: 978-1-57735-271-6. URL: `http://dl.acm.org/citation.cfm?id=3029947.3029959`.

[17] Sven Linker, Jim Burton, and Mateja Jamnik. "Tactical Diagrammatic Reasoning". In: *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016.* Ed. by Serge Autexier and Pedro Quaresma. Vol. 239. EPTCS. 2016, pp. 29–42. DOI: `10.4204/EPTCS.239.3`. URL: `https://doi.org/10.4204/EPTCS.239.3`.

[18] Nicolas Matentzoglu, Samantha Bail, and Bijan Parsia. "A Corpus of OWL DL Ontologies". In: *Description Logics.* 2013.

[19] Nicolas Matentzoglu, Samantha Bail, and Bijan Parsia. "A Snapshot of the OWL Web". In: *The Semantic Web – ISWC 2013.* Ed. by Harith Alani et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 331–346.

[20] Bijan Parsia Matthew Horridge and Ulrike Sattler. "Laconic and Precise Justications in OWL". In: *International Semantic Web Conference (ISWC 2008)* (2008), pp. 323–338.

[21] Bijan Parsia Matthew Horridge and Ulrike Sattler. *Lemmas for Justifications in OWL.* URL: `http://www.cs.ox.ac.uk/DL2009/proceedings/oral/Horridge_Parsia_Sattler.pdf`.

[22] M.A Musen. *The Protege project: A look back and a look forward.* 2015. URL: `https://protege.stanford.edu/`.

[23] *NeOn Toolkit.* URL: `http://neon-toolkit.org/wiki/Main_Page.html`.

[24] Tu Anh T. Nguyen. "Generating Natural Language Explanations For Entailments In Ontologies". PhD thesis. Open University, 2013.

[25] Tu Anh T. Nguyen et al. "Measuring the understandability of deduction rules for OWL". In: *First International Workshop on Debugging Ontologies and Ontology Mappings.* Oct. 2012. URL: `http://oro.open.ac.uk/34591/`.

[26] *Ontology Design Patterns Repository.* URL: `http://ontologydesignpatterns.org/ont/`.

[27] *Open Ontology Repository.* URL: `http://www.oor.net/`.

[28] *OWL 2 Web Ontology Language Document Overview (Second Edition).* 2012. URL: `http://www.w3.org/TR/owl2-overview/`.

[29] *OWL Web Ontology Language - Abstract Syntax.* 2004. URL: `https://www.w3.org/TR/owl-semantics/syntax.html`.

[30] *Oxford Ontology Library.* URL: `http://www.cs.ox.ac.uk/isg/ontologies/`.

[31] Bijan Parsia et al. "The OWL Reasoner Evaluation (ORE) 2015 Competition Report". In: *Journal of Automated Reasoning* 59.4 (Dec. 2017), pp. 455–482. ISSN: 1573-0670. DOI: `10.1007/s10817-017-9406-8`. URL: `https://doi.org/10.1007/s10817-017-9406-8`.

[32] Price and Spackman. "SNOMED clinical terms". In: *BJHC & IM-British Journal of Healthcare Computing & Information Management* 17(3) (2000), pp. 27–31.

[33] Evren Sirin et al. "Pellet: A practical OWL-DL reasoner". In: *Web Semantics: Science, Services and Agents on the World Wide Web* 5.2 (2007). Software Engineering and the Semantic Web, pp. 51–53. ISSN: 1570-8268. DOI: `https://doi.org/10.1016/j.websem.2007.03.004`. URL: `http://www.sciencedirect.com/science/article/pii/S1570826807000169`.

[34] OWLCS Research Team. *OWL API*. 2014. URL: `https://github.com/owlcs/owlapi`.

[35] *The Manchester OWL Corpus (MOWLCorp)*. URL: `http://mowlrepo.cs.manchester.ac.uk/datasets/mowlcorp/`.

[36] *The TONES Ontology Repository*. URL: `http://www.inf.unibz.it/tones/index93b7.html?option=com_content&task=view&id=37&Itemid=86`.

[37] *VIVO Ontology Repository*. URL: `http://swl.slis.indiana.edu/repository/index.html`.

[38] Noy NF Whetzel PL et al. "BioPortal: enhanced functionality via new Web services from the National Center for Biomedical Ontology to access and use ontologies in software applications". In: *Nucleic Acids Res. 2011 Jul;39(Web Server issue):W541-5*. 2011.

Appendix A

Implemented Rules

A.1 Inference Rules

Table A.1 contains a list of all the implemented inference rules. The majority of them are written exactly as in [24]. A few of them have been re-written to aid clarity and readability, but their meaning was left unchanged.

Sets of rules that are slight variations of each other are written with a common name and with a common prefix ID.

Size	**ID**	**Name**	**Deduction Rule**
1	1	EquCls	$X \equiv Y[\equiv ...]$ $\rightarrow X \sqsubseteq Y$
	2.1	ObjInt-1	$X \equiv Y_1 \sqcap \ldots \sqcap Y_n \sqcap Z_1 \sqcap \ldots \sqcap Z_m$, $n \geq 1, m \geq 1$ $\rightarrow X \sqsubseteq Y_1 \sqcap \ldots \sqcap Y_n$
	2.2	ObjInt-1	$X \equiv Y_1 \sqcap \ldots \sqcap Y_n$, $n \geq 1$ $\rightarrow X \sqsubseteq Y_i$, $n \geq i \geq 1$
	2.3	ObjInt-1	$X \equiv \exists R_o.(Y_1 \sqcap \ldots \sqcap Y_n \sqcap Z_1 \sqcap \ldots \sqcap Z_m)$ $\rightarrow X \sqsubseteq \exists R_o.(Y_1 \sqcap \ldots \sqcap Y_n)$, $n \geq 1, m \geq 1$
	2.4	ObjInt-1	$X \equiv \exists R_o.(Y_1 \sqcap \ldots \sqcap Y_n)$, $n \geq 1$ $\rightarrow X \sqsubseteq \exists R_o.(Y_i)$, $n \geq i \geq 1$
	3.1	ObjInt-2	$X \sqsubseteq Y_1 \sqcap \ldots \sqcap Y_n \sqcap Z_1 \sqcap \ldots \sqcap Z_m$, $n \geq 1, m \geq 1$ $\rightarrow X \sqsubseteq Y_1 \sqcap \ldots \sqcap Y_n$
	3.2	ObjInt-2	$X \sqsubseteq Y_1 \sqcap \ldots \sqcap Y_n$, $n \geq 1$ $\rightarrow X \sqsubseteq Y_i$, $n \geq i \geq 1$
	3.3	ObjInt-2	$X \sqsubseteq \exists R_o.(Y_1 \sqcap \ldots \sqcap Y_n \sqcap Z_1 \sqcap \ldots \sqcap Z_m)$ $\rightarrow X \sqsubseteq \exists R_o.(Y_1 \sqcap \ldots \sqcap Y_n)$, $n \geq 1, m \geq 1$
	3.4	ObjInt-2	$X \sqsubseteq \exists R_o.(Y_1 \sqcap \ldots \sqcap Y_n)$, $n \geq 1$ $\rightarrow X \sqsubseteq \exists R_o.(Y_i)$, $n \geq i \geq 1$
	4.1	ObjUni-1	$X \equiv Y_1 \sqcup \ldots \sqcup Y_n \sqcup Z_1 \sqcup \ldots \sqcup Z_m$, $n \geq 1, m \geq 1$ $\rightarrow Y_1 \sqcup \ldots \sqcup Y_n \sqsubseteq X$
	4.2	ObjUni-1	$X \equiv Y_1 \sqcup \ldots \sqcup Y_n$, $n \geq 1$ $\rightarrow Y_i \sqsubseteq X$, $n \geq i \geq 1$
	4.3	ObjUni-1	$X \equiv \exists R_o.(Y_1 \sqcup \ldots \sqcup Y_n \sqcup Z_1 \sqcup \ldots \sqcup Z_m)$ $\rightarrow \exists R_o.(Y_1 \sqcup \ldots \sqcup Y_n) \sqsubseteq X$, $n \geq 1, m \geq 1$
	4.4	ObjUni-	$X \equiv \exists R_o.(Y_1 \sqcup \ldots \sqcup Y_n)$, $n \geq 1$

		1	$\rightarrow \exists R_o.(Y_i) \sqsubseteq X,\ n \geq i \geq 1$
	5.1	ObjUni-2	$Y_1 \sqcup \ldots \sqcup Y_n \sqcup Z_1 \sqcup \ldots \sqcup Z_m \sqsubseteq X,\ n \geq 1, m \geq 1$ $\rightarrow\ Y_1 \sqcup \ldots \sqcup Y_n \sqsubseteq X$
	5.2	ObjUni-2	$Y_1 \sqcup \ldots \sqcup Y_n \sqsubseteq X,\ n \geq 1$ $\rightarrow\ Y_i \sqsubseteq X,\ n \geq i \geq 1$
	5.3	ObjUni-2	$\exists R_o.(Y_1 \sqcup \ldots \sqcup Y_n \sqcup Z_1 \sqcup \ldots \sqcup Z_m) \sqsubseteq X$ $\rightarrow\ \exists R_o.(Y_1 \sqcup \ldots \sqcup Y_n) \sqsubseteq X,\ n \geq 1, m \geq 1$
	5.4	ObjUni-2	$\exists R_o.(Y_1 \sqcup \ldots \sqcup Y_n) \sqsubseteq X,\ n \geq 1$ $\rightarrow \exists R_o.(Y_i) \sqsubseteq X,\ n \geq i \geq 1$
	6.1	ObjExt	$X \sqsubseteq\ = n_1 R_o.Y,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ X \sqsubseteq\ \geq n_2 R_o.Y$
	6.2	ObjExt	$X \sqsubseteq\ = n R_o.Y,\ n \geq 0$ $\rightarrow\ X \sqsubseteq\ \leq n R_o.Y$
	6.3	ObjExt	$X \sqsubseteq\ \geq n_1 R_o.Y,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ X \sqsubseteq\ \geq n_2 R_o.Y$
	7	ObjAll	$X \equiv \forall R_o.Y$ $\rightarrow\ \forall R_o.\bot \sqsubseteq X$
	8	Top	$\top \sqsubseteq X$ $\rightarrow\ Y \sqsubseteq X$
	9	Bot	$X \sqsubseteq \bot$ $\rightarrow\ X \sqsubseteq Y$
	10	ObjCom -1	$X \sqsubseteq \neg X$ $\rightarrow\ X \sqsubseteq \bot$
	11	ObjCom -2	$\neg X \sqsubseteq Y$ $\rightarrow\ \top \sqsubseteq X \sqcup Y$
	12.1	DatSom -DatRng	$X \sqsubseteq \exists R_d.D_{t0}$ $\rightarrow X \sqsubseteq \bot$
	12.2	DatSom -DatRng	$X \sqsubseteq \exists R_o.(\exists R_d.D_{t0}) \wedge$ $\mathbf{Rng}(R_d, D_{t1}),\ D_{t0} \neq D_{t1}$ $\rightarrow X \sqsubseteq \bot$
2	13.1	DatMin -DatRng	$X \sqsubseteq\ \geq n R_d.D_{t0},\ n > 0$ $\wedge\ \mathbf{Rng}(R_d, D_{t1}),\ D_{t0} \neq D_{t1}$ $\rightarrow X \sqsubseteq \bot$
	13.2	DatMin -DatRng	$X \sqsubseteq \exists R_o.(\geq n R_d.D_{t0}),\ n > 0$ $\wedge\ \mathbf{Rng}(R_d, D_{t1}),\ D_{t0} \neq D_{t1}$ $\rightarrow X \sqsubseteq \bot$
	14.1	DatVal -DatRng	$X \sqsubseteq \exists R_d.\{l0 \star D_{t0}\}$ $\wedge\ \mathbf{Rng}(R_d, D_{t1}),\ D_{t0} \neq D_{t1}$ $\rightarrow X \sqsubseteq \bot$
	14.2	DatVal -DatRng	$X \sqsubseteq \exists R_o.(\exists R_d.\{l0 \star D_{t0}\})$ $\wedge\ \mathbf{Rng}(R_d, D_{t1}),\ D_{t0} \neq D_{t1}$ $\rightarrow X \sqsubseteq \bot$
	15	SubCls-DisCls	$X \sqsubseteq Y \wedge\ \mathbf{Dis}(X, Y[, \ldots])$ $\rightarrow X \sqsubseteq \bot$
	16	Top-DisCls	$\top \sqsubseteq Y \wedge\ \mathbf{Dis}(X, Y[, \ldots])$ $\rightarrow X \sqsubseteq \bot$
	17.1	ObjMin -ObjMax	$X \sqsubseteq\ \geq n_1 R_o.Y$ $\wedge\ X \sqsubseteq\ \leq n_2 R_o.Y,\ 0 \leq n_2 < n_1$ $\rightarrow X \sqsubseteq \bot$

17.2	ObjMin -ObjMax	$X \sqsubseteq = n_1 R_o.Y$ $\wedge\ X \sqsubseteq\ \leq n_2 R_o.Y,\ 0 \leq n_2 < n_1$ $\rightarrow X \sqsubseteq \bot$
18.1	ObjMin -ObjFun	$X \sqsubseteq\ \geq nR_o.Y,\ n > 1$ $\wedge\ \mathbf{Fun}(R_o)$ $\rightarrow X \sqsubseteq \bot$
18.2	ObjMin -ObjFun	$X \sqsubseteq = nR_o.Y,\ n > 1$ $\wedge\ \mathbf{Fun}(R_o)$ $\rightarrow X \sqsubseteq \bot$
19.1	DatMin -DatFun	$X \sqsubseteq\ \geq nR_d.D_r,\ n > 1$ $\wedge\ \mathbf{Fun}(R_d)$ $\rightarrow X \sqsubseteq \bot$
19.2	DatMin -DatFun	$X \sqsubseteq = nR_d.D_r,\ n > 1$ $\wedge\ \mathbf{Fun}(R_d)$ $\rightarrow X \sqsubseteq \bot$
20.1	ObjSom -Bot-1	$X \sqsubseteq \exists R_o.Y\ \wedge\ Y \sqsubseteq \bot$ $\rightarrow X \sqsubseteq \bot$
20.2	ObjSom -Bot-1	$X \sqsubseteq\ \geq nR_o.Y,\ n > 0$ $\wedge\ Y \sqsubseteq \bot$ $\rightarrow X \sqsubseteq \bot$
20.3	ObjSom -Bot-1	$X \sqsubseteq = nR_o.Y,\ n > 0$ $\wedge\ Y \sqsubseteq \bot$ $\rightarrow X \sqsubseteq \bot$
21.1	ObjSom -Bot-2	$X \sqsubseteq \exists R_o.(Y \sqcap Z[\sqcap \ldots])\ \wedge\ Y \sqsubseteq \bot$ $\rightarrow X \sqsubseteq \bot$
21.2	ObjSom -Bot-2	$X \sqsubseteq\ \geq nR_o.(Y \sqcap Z[\sqcap \ldots]),\ n > 0$ $\wedge\ Y \sqsubseteq \bot$ $\rightarrow X \sqsubseteq \bot$
21.3	ObjSom -Bot-2	$X \sqsubseteq = nR_o.(Y \sqcap Z[\sqcap \ldots]),\ n > 0$ $\wedge\ Y \sqsubseteq \bot$ $\rightarrow X \sqsubseteq \bot$
22.1	ObjInt -DisCls	$X \sqsubseteq \exists R_o.(Y_1 \sqcap \ldots \sqcap Y_m),\ m \geq 2$ $\wedge\ \mathbf{Dis}(Y_1 \sqcap \ldots \sqcap Y_m[, \ldots])$ $\rightarrow X \sqsubseteq \bot$
22.2	ObjInt -DisCls	$X \sqsubseteq\ \geq nR_o.(Y_1 \sqcap \ldots \sqcap Y_m),\ m \geq 2,\ n > 0$ $\wedge\ \mathbf{Dis}(Y_1 \sqcap \ldots \sqcap Y_m[, \ldots])$ $\rightarrow X \sqsubseteq \bot$
22.3	ObjInt -DisCls	$X \sqsubseteq = nR_o.(Y_1 \sqcap \ldots \sqcap Y_m),\ m \geq 2,\ n > 0$ $\wedge\ \mathbf{Dis}(Y_1 \sqcap \ldots \sqcap Y_m[, \ldots])$ $\rightarrow X \sqsubseteq \bot$
23	SubCls-ObjCom-1	$X \sqsubseteq Y\ \wedge\ X \sqsubseteq \neg Y$ $\rightarrow X \sqsubseteq \bot$
24	SubCls-ObjCom-2	$X \sqsubseteq Y\ \wedge\ \neg X \sqsubseteq Y$ $\rightarrow \top \sqsubseteq Y$
25.1	ObjDom -ObjAll	$\mathbf{Dom}(R_o, X)\ \wedge \forall R_o.\bot \sqsubseteq X$ $\rightarrow \top \sqsubseteq X$
25.2	ObjDom -ObjAll	$\exists R_o.\top \sqsubseteq X\ \wedge\ \forall R_o.\bot \sqsubseteq X$ $\rightarrow \top \sqsubseteq X$

26	SubObj -SubObj	$R_o \sqsubseteq S_o \ \wedge\ S_o \sqsubseteq T_o$ $\rightarrow R_o \sqsubseteq T_o$
27	ObjTra -ObjInv	$\mathbf{Tra}(R_o) \ \wedge\ \mathbf{Invs}(R_o, S_o)$ $\rightarrow \mathbf{Tra}(S_o)$
28	ObjDom -SubCls	$\mathbf{Dom}(R_o, X) \ \wedge X \sqsubseteq Y$ $\rightarrow \mathbf{Dom}(R_o, Y)$
29	ObjDom -SubObj	$\mathbf{Dom}(R_o, X) \ \wedge S_o \sqsubseteq R_o$ $\rightarrow \mathbf{Dom}(S_o, X)$
30	ObjRng -ObjInv	$\mathbf{Rng}(R_o, X) \ \wedge\ \mathbf{Invs}(R_o, S_o)$ $\rightarrow \mathbf{Dom}(S_o, X)$
31	ObjRng -ObjSym	$\mathbf{Rng}(R_o, X) \ \wedge\ \mathbf{Sym}(R_o)$ $\rightarrow \mathbf{Dom}(R_o, X)$
32	ObjRng -SubCls	$\mathbf{Rng}(R_o, X) \ \wedge X \sqsubseteq Y$ $\rightarrow \mathbf{Rng}(R_o, Y)$
33	ObjRng -SubObj	$\mathbf{Rng}(R_o, X) \ \wedge S_o \sqsubseteq R_o$ $\rightarrow \mathbf{Rng}(S_o, Y)$
34	ObjDom -ObjInv	$\mathbf{Rng}(R_o, X) \ \wedge\ \mathbf{Invs}(R_o, S_o)$ $\rightarrow \mathbf{Rng}(S_o, X)$
35	ObjDom -ObjSym	$\mathbf{Rng}(R_o, X) \ \wedge\ \mathbf{Sym}(R_o)$ $\rightarrow \mathbf{Rng}(R_o, X)$
36.1	ObjSom -ObjDom	$X \sqsubseteq \exists R_o.Z \ \wedge\ \mathbf{Dom}(R_o, Y)$ $\rightarrow X \sqsubseteq Y$
36.2	ObjSom -ObjDom	$X \sqsubseteq \geq nR_o.Z,\ n > 0$ $\wedge\ \mathbf{Dom}(R_o, Y)$ $\rightarrow X \sqsubseteq Y$
36.3	ObjSom -ObjDom	$X \sqsubseteq = nR_o.Z,\ n > 0$ $\wedge\ \mathbf{Dom}(R_o, Y)$ $\rightarrow X \sqsubseteq Y$
37.1	DatSom -DatDom	$X \sqsubseteq \exists R_d.D_r \ \wedge\ \mathbf{Dom}(R_d, Y)$ $\rightarrow X \sqsubseteq Y$
37.2	DatSom -DatDom	$X \sqsubseteq \geq nR_d.D_r,\ n > 0$ $\wedge\ \mathbf{Dom}(R_d, Y)$ $\rightarrow X \sqsubseteq Y$
37.3	DatSom -DatDom	$X \sqsubseteq = nR_d.D_r,\ n > 0$ $\wedge\ \mathbf{Dom}(R_d, Y)$ $\rightarrow X \sqsubseteq Y$
38.1	ObjSom -ObjRng	$X \sqsubseteq \exists R_o.Y \ \wedge\ \mathbf{Rng}(R_o, Z)$ $\rightarrow X \sqsubseteq \exists R_o.(Y \sqcap Z)$
38.2	ObjSom -ObjRng	$X \sqsubseteq \geq nR_o.Y,\ n > 0$ $\wedge\ \mathbf{Rng}(R_o, Z)$ $\rightarrow X \sqsubseteq \geq nR_o.(Y \sqcap Z)$
38.3	ObjSom -ObjRng	$X \sqsubseteq = nR_o.Y,\ n > 0$ $\wedge\ \mathbf{Rng}(R_o, Z)$ $\rightarrow X \sqsubseteq = nR_o.(Y \sqcap Z)$
39	SubCls- SubCls-1	$X \sqsubseteq Y \ \wedge\ Y \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq Z$
40	SubCls- SubCls-2	$X \sqsubseteq Y \ \wedge\ X \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq (Y \sqcap Z)$

41.1	ObjSom -ObjMin	$X \sqsubseteq \exists R_o.Y$ $\wedge\ \geq nR_o.Y \sqsubseteq Z,\ n=1$ $\rightarrow\ X \sqsubseteq Z$
41.2	ObjSom -ObjMin	$X \sqsubseteq\ \geq nR_o.Y,\ n>0$ $\wedge\ \exists R_o.Y \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq Z$
41.3	ObjSom -ObjMin	$X \sqsubseteq\ = nR_o.Y,\ n>0$ $\wedge\ \exists R_o.Y \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq Z$
42.1	DatSom -DatMin	$X \sqsubseteq \exists R_d.D_r$ $\wedge\ \geq nR_d.D_r \sqsubseteq Z,\ n=1$ $\rightarrow\ X \sqsubseteq Z$
42.2	DatSom -DatMin	$X \sqsubseteq\ \geq nR_d.D_r,\ n>0$ $\wedge\ \exists R_d.D_r \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq Z$
42.3	DatSom -DatMin	$X \sqsubseteq\ = nR_d.D_r,\ n>0$ $\wedge\ \exists R_d.D_r \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq Z$
43.1	ObjSom -SubCls	$X \sqsubseteq \exists R_o.Y\ \wedge\ Y \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq \exists R_o.Z$
43.2	ObjSom -SubCls	$X \sqsubseteq\ \geq nR_o.Y,\ n \geq 0$ $\wedge\ Y \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq \geq nR_o.Z$
43.3	ObjSom -SubCls	$X \sqsubseteq = nR_o.Y,\ n \geq 0$ $\wedge\ Y \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq = nR_o.Z$
44.1	ObjSom -SubObj	$X \sqsubseteq \exists R_o.Y$ $\wedge\ R_o \sqsubseteq S_o$ $\rightarrow\ X \sqsubseteq \exists S_o.Y$
44.2	ObjSom -SubObj	$X \sqsubseteq\ \geq nR_o.Y,\ n \geq 0$ $\wedge\ R_o \sqsubseteq S_o$ $\rightarrow\ X \sqsubseteq\ \geq nS_o.Y$
44.3	ObjSom -SubObj	$X \sqsubseteq\ = nR_o.Y,\ n \geq 0$ $\wedge\ R_o \sqsubseteq S_o$ $\rightarrow\ X \sqsubseteq\ = nS_o.Y$
45	ObjUni -SubCls	$X \sqsubseteq (Y \sqcup Z)\ \wedge\ Y \sqsubseteq Z$ $\rightarrow\ X \sqsubseteq Z$
46	ObjAll -ObjInv	$X \sqsubseteq \forall R_o.Y\ \wedge\ \mathbf{Invs}(R_o, S_o)$ $\rightarrow\ \exists S_o.X \sqsubseteq Y$
47	ObjSom-ObjAll-1	$\exists R_o.Y \sqsubseteq X\ \wedge\ \forall R_o.\bot \sqsubseteq X$ $\rightarrow\ \forall R_o.Y \sqsubseteq X$
48	ObjSom-ObjAll-2	$X \sqsubseteq \exists R_o.T\ \wedge\ X \sqsubseteq \forall R_o.Y$ $\rightarrow\ X \sqsubseteq \exists R_o.Y$
49.1	ObjSom -ObjTra	$X \sqsubseteq \exists R_o.(\exists R_o.Y)\ \wedge\ \mathbf{Tra}(R_o)$ $\rightarrow\ X \sqsubseteq \exists R_o.Y$
49.2	ObjSom -ObjTra	$X \sqsubseteq\ \geq nR_o.(\geq nR_o.Y),\ n>0$ $\wedge\ \mathbf{Tra}(R_o)$ $\rightarrow\ X \sqsubseteq\ \geq nR_o.Y$

	50	ObjDom -Bot	$\mathbf{Dom}(R_o, X) \ \wedge X \sqsubseteq \bot$ $\rightarrow \ \top \sqsubseteq \forall R_o.\bot$
	51	ObjRng -Bot	$\mathbf{Rng}(R_o, X) \ \wedge X \sqsubseteq \bot$ $\rightarrow \ \top \sqsubseteq \forall R_o.\bot$
3	52	DisCls -SubCls -SubCls	$\mathbf{Dis}(X, Y) \ \wedge U \sqsubseteq X$ $\wedge V \sqsubseteq Y$ $\rightarrow \ \mathbf{Dis}(U, V)$
	53	SubCls -SubCls -DisCls	$X \sqsubseteq Y \ \wedge \ X \sqsubseteq Z$ $\wedge \ \mathbf{Dis}(Y, Z)$ $\rightarrow \ X \sqsubseteq \bot$
	54	ObjUni -SubCls -SubCls	$X \sqsubseteq (U \sqcup V) \ \wedge \ U \sqsubseteq Z$ $\wedge V \sqsubseteq Z$ $\rightarrow \ X \sqsubseteq Z$
	55.1	ObjSom -ObjSom -ObjTra	$X \sqsubseteq \exists R_o.Y \ \wedge \ Y \sqsubseteq \exists R_o.Z$ $\wedge \ \mathbf{Tra}(R_o)$ $\rightarrow \ X \sqsubseteq \exists R_o.Z$
	55.2	ObjSom -ObjSom -ObjTra	$X \sqsubseteq \ \geq nR_o.Y, \ n > 0$ $\wedge \ Y \sqsubseteq \ \geq nR_o.Z \ \wedge \ \mathbf{Tra}(R_o)$ $\rightarrow \ X \sqsubseteq \ \geq nR_o.Z$
	56	DatVal -DatVal -DatFun	$X \sqsubseteq \exists R_d.\{l_0 \star D_{t0}\} \ \wedge \ \mathbf{Fun}(R_d)$ $\wedge \ X \sqsubseteq \exists R_d.\{l_1 \star D_{t1}\}, \ D_{t0} \neq D_{t1}$ or $l_0 \neq l_1$ $\rightarrow \ X \sqsubseteq \bot$
4	57	ObjVal -ObjVal -DifInd- ObjFun	$X \sqsubseteq \exists R_o.\{i\} \ \wedge \ X \sqsubseteq \exists R_o.\{j\}$ $\wedge \ \mathbf{Diff}(i, j) \ \wedge \ \mathbf{Fun}(R_o)$ $\rightarrow \ X \sqsubseteq \bot$

Table A.1: Implemented inference rules.

A.2 Rule Exceptions

Table A.2 contains a list of all the exception cases, written exactly as in [24].

Every input subtree and output subtree consists of nodes that have at most one child (i.e. a chain of nodes). They are thus written as a list of axiom templates, starting from the leaf node and progressing up the tree until the root is reached.

For instance, if the input subtree template is a tree consisting of two nodes, with the root node template being: $C \sqsubseteq \exists R_o.\top$, and the child node template being: $C \sqsubseteq \exists R_o.D$, this would be written as shown in (A.1):

$$\begin{aligned} &C \sqsubseteq \exists R_o.D \\ &\rightarrow \ C \sqsubseteq \exists R_o.\top \end{aligned} \tag{A.1}$$

ID	Input Subtree	Output Subtree
1.1	$C \sqsubseteq \exists R_o.D$ $\rightarrow\ C \sqsubseteq \exists R_o.\top$	$C \sqsubseteq \exists R_o.D$
1.2	$C \equiv \exists R_o.D$ $\rightarrow\ C \sqsubseteq \exists R_o.\top$	$C \equiv \exists R_o.D$ $\rightarrow\ C \sqsubseteq \exists R_o.D$
1.3	$C \sqsubseteq C_1 \sqcap \exists R_o.D$ $\rightarrow\ C \sqsubseteq \exists R_o.\top$	$C \sqsubseteq C_1 \sqcap \exists R_o.D$ $\rightarrow\ C \sqsubseteq \exists R_o.D$
1.4	$C \equiv C_1 \sqcap \exists R_o.D$ $\rightarrow\ C \sqsubseteq \exists R_o.\top$	$C \equiv C_1 \sqcap \exists R_o.D$ $\rightarrow\ C \sqsubseteq \exists R_o.D$
2.1	$C \sqsubseteq \exists R_o.\{i\}$ $\rightarrow\ C \sqsubseteq \exists R_o.\top$	$C \sqsubseteq \exists R_o.\{i\}$
2.2	$C \equiv \exists R_o.\{i\}$ $\rightarrow\ C \sqsubseteq \exists R_o.\top$	$C \equiv \exists R_o.\{i\}$ $\rightarrow\ C \sqsubseteq \exists R_o.\{i\}$
2.3	$C \sqsubseteq C_1 \sqcap \exists R_o.\{i\}$ $\rightarrow\ C \sqsubseteq \exists R_o.\top$	$C \sqsubseteq C_1 \sqcap \exists R_o.\{i\}$ $\rightarrow\ C \sqsubseteq \exists R_o.\{i\}$
2.4	$C \equiv C_1 \sqcap \exists R_o.\{i\}$ $\rightarrow\ C \sqsubseteq \exists R_o.\top$	$C \equiv C_1 \sqcap \exists R_o.\{i\}$ $\rightarrow\ C \sqsubseteq \exists R_o.\{i\}$
3.1	$C \sqsubseteq\ \geq nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq\ \geq nR_o.\top$	$C \sqsubseteq\ \geq nR_o.D,\ n \geq 0$
3.2	$C \sqsubseteq\ = n_1R_o.D,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ C \sqsubseteq\ \geq n_2R_o.\top$	$C \sqsubseteq\ = n_1R_o.D,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ C \sqsubseteq\ \geq n_2R_o.D$
3.3	$C \equiv\ = n_1R_o.D,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ C \sqsubseteq\ \geq n_2R_o.\top$	$C \equiv\ = n_1R_o.D,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ C \sqsubseteq\ = n_1R_o.D$ $\rightarrow\ C \sqsubseteq\ \geq n_2R_o.D$
3.4	$C \sqsubseteq C_1 \sqcap = n_1R_o.D,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ C \sqsubseteq\ \geq n_2R_o.\top$	$C \sqsubseteq C_1 \sqcap = n_1R_o.D,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ C \sqsubseteq = n_1R_o.D$ $\rightarrow\ C \sqsubseteq\ \geq n_2R_o.D$
3.5	$C \equiv C_1 \sqcap = n_1R_o.D,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ C \sqsubseteq\ \geq n_2R_o.\top$	$C \equiv C_1 \sqcap = n_1R_o.D,\ n_1 \geq n_2 \geq 0$ $\rightarrow\ C \sqsubseteq = n_1R_o.D$ $\rightarrow\ C \sqsubseteq\ \geq n_2R_o.D$
4.1	$C \sqsubseteq\ \leq nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.\top$	$C \sqsubseteq\ \leq nR_o.D,\ n \geq 0$
4.2	$C \sqsubseteq\ = nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.\top$	$C \sqsubseteq\ = nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.D$
4.3	$C \equiv\ = nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.\top$	$C \equiv\ = nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq\ = nR_o.D$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.D$
4.4	$C \sqsubseteq C_1 \sqcap = nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.\top$	$C \sqsubseteq C_1 \sqcap = nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq = nR_o.D$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.D$
4.5	$C \equiv C_1 \sqcap = nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.\top$	$C \equiv C_1 \sqcap = nR_o.D,\ n \geq 0$ $\rightarrow\ C \sqsubseteq = nR_o.D$ $\rightarrow\ C \sqsubseteq\ \leq nR_o.D$
5	$C \sqsubseteq \exists R_d.\{l\}$ $\rightarrow\ C \sqsubseteq \exists R_d.Literal$	$C \sqsubseteq \exists R_d.\{l\}$
6	$\mathbf{Invs}(R_o, S_o)$ $\rightarrow\ R_o \sqsubseteq \mathbf{Inv}(S_o)$	$\mathbf{Invs}(R_o, S_o)$

Table A.2: Exception cases.

Appendix B

Project Proposal

The original project proposal is attached to this section of the appendix.

Dmitry Kazhdan

Emmanuel College

dk525

Deduction Rules for Ontology Reasoning

Project Originator: Dr Zohreh Shams

Project Supervisors: Dr Zohreh Shams, Dr Mateja Jamnik

Director of Studies: Dr Thomas Sauerwald

Overseers: Dr Markus Kuhn, Dr Peter Sewell

Introduction and Description of Work

A common way of representing information related to a particular domain in a structured format is by expressing it as an ontology. An ontology is a formal description of classes (which represent concepts in a domain), class properties (describing various features and attributes of the class), and restrictions on those properties. An ontological definition taken together with a set of instances of the defined classes is referred to as a knowledge base of the domain.

Creating an ontology typically includes the following steps:

- Defining classes (describing concepts in the domain).
- Defining properties and describing their permitted values.
- Defining the subclass–superclass hierarchy of classes (subclasses represent concepts that are more specific than their superclass).
- Declaring instances of classes.

Ontologies have typically been used for the following reasons: representing information using a common, shared, well-defined structure (which makes certain domain assumptions explicit), reusing domain knowledge and analyzing domain knowledge.

Originally ontologies were used in the field of Artificial Intelligence, however, recently ontologies have been used in a variety of other fields, such as the World-Wide-Web. Examples of these ontologies include Yahoo! (for large classifications categorizing Web sites) and Amazon (for categorizing products and product features). Other uses of ontologies include: medicine (for producing large standardized, structured vocabularies such as SNOMED) and expert systems.

Ontologies are encoded and expressed using symbolic languages that are usually based on either first-order logic or description logic. These languages are used for encoding knowledge about domains.

Ontology editors are applications used for creating and manipulating ontologies. They often express ontologies in one of the ontology languages described above. One of the primary capabilities offered by such editors is the ability to use reasoners to output all of the facts (referred to as “entailments”) that can be inferred from the ontology definition. This is done by performing inference on the logical representation of the ontology used by the language. This provides users of the ontology with new useful information about ways in which concepts in the ontology are related. Some examples of ontologies with descriptions can be found at [1].

Despite the numerous benefits ontologies offer, there is still room for improvement when it comes to using them. Making mistakes when defining an ontology can lead to the ontology reasoner producing undesired/unwanted entailments. Thus it is often necessary to debug ontologies (repairing the ontology such that the undesired entailment does not follow anymore) to ensure that domain knowledge has been entered correctly and to remove modelling errors. Debugging ontologies is a difficult task, due to the fact that debugging messages given by ontology reasoners are typically unhelpful, and because users are usually domain experts, not computer scientists. Reasoners often simply return an error when the ontology is inconsistent, without any further information.

This creates a need of generating more helpful, user-friendly explanations of the entailments. One way of achieving this is by constructing a proof for the entailment using the set of justification axioms defined in the ontology and a set of inference rules. This project aims to investigate the suitability of one such approach by implementing a given inferential algorithm that is designed to produce human-readable proofs and seeing how well it applies to a set of ontologies.

Starting Point

This project will heavily rely on [2]. This is a thesis which describes an algorithm that, given a set of axioms and an entailment, attempts to construct a proof of the entailment using the given axioms and a set of 57 predefined inference rules that are also described in the thesis. Unlike other proof generation algorithms that are often used by reasoners, the primary goal of this particular algorithm is producing a proof that is human-readable and easily understandable. The inference rules used are designed based on an intuitiveness factor and can easily serve as the basis of explanation generation in natural language. Also, their accessibility is evidenced by empirical evaluation that measures their understandability.

The algorithm will be implemented in Java, for which the "Object Oriented Programming" module lectured in Part IA, as well as the Part IA and Part IB ticks are useful.

A preexisting, representative set of ontologies will be selected as the source of entailments from which proofs will be generated. Part of the project will involve deciding exactly which set of ontologies should be selected and justifying this choice. One of the most likely choices will be selecting numerous ontologies from ontology repositories listed at [3]. This resource provides a source of ontologies that are commonly used and are diverse in topic.

Substance and Structure of the Project

The core aim of this project will be to implement the algorithm described in the thesis. This algorithm can then be used to investigate the claim made in the thesis, which states that the algorithm and the predefined set of inference rules it uses is sufficient to produce proofs (and thus explanations) for over 50% of entailments in any ontology, thus making this algorithm a universal solution that can be applied to any ontology.

This aim will be achieved by implementing the algorithm described above, and testing its coverage on a large set of entailments taken from a representative set of ontologies. The coverage and frequency of inference rules will be used to quantitatively evaluate whether this algorithm and set of inference rules can indeed be applied to arbitrary ontologies for producing proofs of entailments.

Possible extensions will involve extracting new inference rules from the ones used by the algorithm and using them to replace other inference rules in order to reduce the overall rule number and/or to improve their coverage. These extension steps may change as progress is made and more information is accumulated regarding the project.

The project will include the following steps:

1. A representative set of ontologies will be selected.
2. An off-the-shelf ontology description logic reasoner will be used to derive all entailments of the ontologies, of which a representative subset will be selected.
3. The algorithm described in the thesis will be implemented (using Java).
4. Statistical analysis will be used to measure the coverage and frequency of the inference rules by investigating how likely the inference rules are to construct a proof for the selected entailments.
5. A conclusion will be drawn, assessing how well the algorithm deals with the selected ontology entailments.

Possible extension steps include:

1. The inference rules can be inspected in order to extract a smaller, core set of inference rules. These rules can be used to produce a modified version of the original algorithm, with improved coverage.
2. Machine learning techniques can be used in order to extract common sequences of inference rules in the proofs produced in order to produce new inference rules. These rules can be used to produce a modified version of the original algorithm that potentially produces shorter proofs.

Success Criteria

The project should be considered successful if the following conditions are satisfied:

1. The algorithm described in the thesis is implemented to a high standard (e.g. readability, coding style).
2. The algorithm can be shown to work and construct proofs for a representative set of entailments (a demonstration is given).
3. A detailed, quantitative investigation of inference rule coverage is conducted and presented.

Thus by the end of the project, the algorithm described in the thesis should be implemented successfully. This algorithm should be able to take as input an entailment from an ontology and a set of axioms that justify this entailment, and give a proof of this entailment as output. The ontologies that will be used to test this algorithm include (but are not limited to): “Breast Cancer Grading Ontology” and “Environmental Ontology (ENVO)” from the NCBO BioPortal ontology repository, and the “Basic Formal Ontology”. These results can be used to produce quantitative evidence that can be used to show how general the results in [2] are.

Success criteria for extension steps include:

- 1.1 A smaller set of new inference rules is extracted from the ones given.
- 1.2 A new version of the algorithm is produced, with the new inference rules replacing some of the old ones.
- 1.3 The rule coverage is shown to be greater than that of the original ones.
- 2.1 Machine Learning techniques are used to analyze and extract proof patterns from the generated proofs. These patterns are used to produce new inference rules.
- 2.2 A new version of the algorithm is produced, with the new inference rules replacing some of the old ones.
- 2.3 The proofs produced by the new algorithm are shown to be shorter than the ones of the original one. This can be done by comparing proofs produced by the new and old algorithms for a selected set of entailments.

Timetable

Weeks 1 – 2 (19th October – 1st November)

The first objective will be to begin reading up on relevant resources. These mostly include the mentioned thesis, as well as any other resources that give more details about ontology reasoning.

The second objective is to select a representative set of ontologies and to justify this choice.

Milestones: Discussion with supervisors regarding suitability of the selected ontologies.

Weeks 3 – 4 (2nd November – 15th November)

The main objective is to finish reading up on the relevant resources (mainly the thesis).

Milestones: Discussion with supervisors to ensure that I understood the ideas behind the thesis correctly.

Weeks 5 – 6 (16th November – 29th November)

The first and foremost objective will be to begin developing the algorithm and implementing the first few inference rules (of which there are 57 in total).

The second objective will be to choose an off-the-shelf reasoner, use it to derive entailments of the selected ontologies and select a representative subset of them.

Milestones: Present the initial algorithm source code to supervisors. Present supervisors with the set of entailments. Agree with supervisors on which set of entailments to choose as the set to be tested in the investigation.

Weeks 7 – 8 (30th November – 13th December)

The main objective will be to continue developing the algorithm, implementing the next set of inference rules.

Milestone: Present the progress made on the algorithm to the supervisors.

Weeks 9 – 10 (14th December – 27th December)

The main objective will be to ensure that significant progress is made on the algorithm, and that the majority of its functionality is implemented.

Milestone: Present the progress made on the algorithm to the supervisors. Ensure that the algorithm is close to completion (minor implementations left only).

Weeks 11 – 12 (28th December – 10th January)

The main objective will be to successfully finish implementing the algorithm and all of its 57 inference rules. This will also involve testing the algorithm using different entailments.

Milestones: Present the finished algorithm source code to the supervisors and demonstrate that it works.

Weeks 13 – 14 (11th January – 24th January)

The main objective will be to begin work on one of the extensions.

The secondary objective will be to begin work on the progress report and presentation.

This time will also be used as a buffer period to ensure that the algorithm is completed and that progress is on track.

Milestones: Begin work on one of the extensions. Begin work on the progress report and presentation.

Weeks 15 – 16 (25th January – 7th February)

The main objective will be to measure the coverage of the inference rules based on generated proofs for selected entailments.

The second objective will be to finish the progress report and presentation.

Milestones: Present coverage statistics of rules to supervisors. Submit the progress report.

Weeks 17 – 18 (8th February – 21st February)

The main objective will be to finish the coverage analysis of the inference rules.

The second objective will be to make progress on one of the extensions.

Milestones: Present coverage statistics of rules to supervisors. Present supervisors with work made on the extension step.

Weeks 19 – 20 (22nd February – 7th March)

The first objective will be to write an initial draft of the Introduction chapter and of the Preparation chapter.

The second objective is to make progress on one of the extension steps.

Milestones: Initial drafts of the Introduction chapter and the Preparation chapter. Progress on one of the extension steps shown to supervisors.

Weeks 21 – 22 (8th March – 21st March)

The first objective will be to start writing the Implementation chapter.

The second objective is to complete the extension.

Milestones: Implementation chapter started on. At least one of the extension goals completed.

Weeks 23 – 24 (22th March – 4th April)

The main objective is to finish writing the Introduction and Preparation chapters, and to make progress on the Implementation chapter.

Milestones: Introduction chapter and Preparation chapter written. Implementation chapter close to completion.

Weeks 25 – 26 (5th April – 18th April)

The main objectives are to finish writing the Implementation chapter and to make a start on the Evaluation and Conclusion chapters.

Milestones: Implementation chapter finished. Evaluation and Conclusion chapters started on.

Weeks 27 – 28 (19th April – 2nd May)

The first objective will be to finish writing the Evaluation and Conclusions chapters.

The second objective will be to review the algorithm and improve it, if possible. If time permits, the extension work can be reviewed and improved as well.

Milestones: Improving and eliminating bugs from the algorithm. Producing a dissertation report describing the project.

Weeks 29 – 30 (3rd May – 16th May)

The main objective will be reviewing the project and dissertation. This will involve patching up anything that is incomplete.

Milestone: Complete the project and prepare to submit it.

Week 31 (17th May – 18th May)

Milestone: Submit dissertation.

<u>References</u>

[1] – http://mowl-power.cs.man.ac.uk/2009/07/sssw

[2] – Nguyen, Tu (2013). *Generating Natural Language Explanations For Entailments In Ontologies*. PhD thesis The Open University.

[3] – http://owl.cs.manchester.ac.uk/tools/repositories/

Resources Declaration

Most of the work will be undertaken on a privately-owned laptop.

The laptop has the following specifications:

- OS: OS X El Capitan
- Processor: Intel Core i7 2.5 GHz
- RAM: 16 GB

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

In the unlikely case that extra computing power will be needed, either Microsoft Azure machines or the University Supercomputer may be used. In both of these cases the project supervisors will be able to provide access to these resources.

Regular (weekly) backups will be made to the MCS machine in the laboratory, as well as to an external storage device. Dropbox will also be used for file storage.

When developing the algorithm, a revision-control system will be used (GitHub).

The ontology reasoner and other off-the-shelf packages to be used are all open source, and thus do not require any special permissions.