UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

Computer Science Honours
Final Paper
2018

Title: Defeasible Datalog

Author: Joshua Abraham

Project Abbreviation: DDLOG

Supervisor(s): Prof. T. Meyer

Category	Min	Max	Chosen
Requirement Analysis and Design	0	20	5
Theoretical Analysis	0	25	25
Experiment Design and Execution	0	20	5
System Development and Implementation	0	15	15
Results, Findings and Conclusion	10	20	15
Aim Formulation and Background Work	10	15	15
Quality of Paper Writing and Presentation	10		
Quality of Deliverables	10		
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	0	10	
Total marks		**80**	**80**

Defeasible Datalog

Using Rational Closure to Extend the Declarative Logic Programming Language, Datalog, With Defeasible Reasoning in RDFox

Joshua J. Abraham
University of Cape Town
Department of Computer Science
joshuasl8a@gmail.com

ABSTRACT

The *Rational Closure* (RC) algorithm [1][2] is a rule based methodology that affords logical reasoners the ability to reason with incomplete or inconsistent knowledge bases. This algorithm has been proven to work for a number of reasoning formalisms and has even been extended to description logics [3][4]. However, the RC algorithm has not yet been shown to work for declarative logic programming languages. The project that this paper is based on practically proves, via implementation, that the RC algorithm can be extended to allow declarative logics the ability to reason defeasibly. Datalog was chosen as the declarative logic programming language and RDFox was chosen as the software to perform the datalog reasoning. A host of obstacles presented itself to dissuade the accomplishing of this task - apart from the issues faced during the setup of RDFox and various other complications, the fact that classical datalog has no formal way of expressing negation had to be overcome as well. However, in the end, the implementation of a defeasible datalog reasoner turned out to be a success. The reasoner performed as expected when faced with a range of unique test cases, each handcrafted to test specific characteristics of defeasible knowledge bases and how the RC algorithm deals with said knowledge bases. Furthermore, the reasoner underwent scrutiny by experts in the field and was eventually approved by both of them. Let it also be noted that this project goes hand-in-hand with a sister project, which theoretically proves the same notion by means of showing that the necessary KLM properties still hold when applied to declarative logic programming languages [5]. The source code for this paper's project can be found on the author's GitHub page [6].

CCS CONCEPTS

• **Artificial Intelligence** → *Knowledge representation*; *Reasoning*;

KEYWORDS

Defeasible Reasoning, Declarative Logic Programming Languages, Datalog, RDFox

1 INTRODUCTION

The ability for logical reasoners to handle exceptions is a well-known problem within the domain of artificial intelligence. Imagine, if you will, being told that all birds fly and that some animal, called Tweety, is a bird. From this knowledge base, it is reasonable to deduce that the animal, Tweety, can fly. However, if it is later learned that Tweety is, in fact, a penguin or an ostrich, then this would raise a clear contradiction (as penguins and ostriches are both examples of flightless birds). Furthermore, with the advancement of disciplines such as genetic engineering, there could later exist some subspecies of penguin or ostrich that indeed had this ability of flight. These exceptions and exceptions to exceptions are commonplace and almost inescapable in an ever growing knowledge base - as is the human collective of information. Further examples of these seemingly contradictory occurrences can be found in domains such as law, where contracts can be annulled in light of new evidence [7]; in medicine, where medical diagnoses can be reevaluated due to the revelation of new symptoms; in science, where scientific theories can be falsified by the ascertaining of new experimental results; and even in Chisholm's paradox of contrary-to-duty imperatives [8][9]. Thus, it is clear to see the implicational benefits that would emerge from granting programs and machines the power to employ the type of reasoning used to handle situations of this ilk. This reasoning is known as defeasible, or non-monotonic, reasoning - it is the type of reasoning which is explored in this project paper.

One method of achieving the implementation of this reasoning in programs is by using the *Rational Closure* (RC) algorithm [1][2]. This algorithm ranks defeasible statements according to exceptionality and allows a defeasible entailment check to be simplified down to a series of classical entailment checks. An adaptation of this algorithm is what is used in this paper's project to allow RDFox's datalog reasoner to reason with defeasible instances.

1.1 Structure of the Paper

The structure of this paper is given in the following order: a detailed background of essential information that needs to be understood in order to follow and appreciate the work done in and by this project; a mention of related work that has previously been done, including the work that this project builds on; a detailed description of the associated project, as well as a motivation for the project; a mention

of the key issues faced during the implementation of this project, why they occurred and how they were overcome; a discussion on the design and implementation of the defeasible datalog reasoner; a discussion on how testing and validation was carried out; a presentation of the core results and finally a conclusion to sum up the paper.

2 BACKGROUND

An explanation of the most essential topics are given in this section. It is necessary for these topics to be understood before a discussion of, and appreciation for, the details and results of this project can occur. It would be beneficial for the reader to already have a functional understanding of propositional (or sentential) and predicate logic, specifically their negation and implication operators. The topics that will be explained in this section are defeasible reasoning; the declarative logic programming language, datalog; the RDFox triple store software and, finally, the RC algorithm.

2.1 Defeasible Reasoning

Defeasibile reasoning is a form of *default reasoning* and is employed when faced with seemingly contradictory information, which opposes what is already known or has already been reasonably deduced. Defeasibile reasoning can be viewed as an extension of propositional logic and allows the ability to follow complex patterns and deductions which, as shown, cannot be followed by means of classical reasoning alone. This is done by introducing the concept of typicality. The defeasible consequence logical operator, $\rightsquigarrow$, is used to represent typicality and is syntactically analogous to the classical implication operator, $\rightarrow$. With this, we are finally able to solve the apparent inconsistencies in the Tweety-bird problem. Instead of saying that '*all birds fly*', consider the phrase: *typically birds have the ability of flight*. This entails that Tweety - which was initially classified to have the ability of fly due to the fact that it is a bird - can now be reclassified to not have the ability of flight, upon learning that Tweety is a penguin, and still maintain that Tweety is a bird. In defeasibile propositional logic syntax, this is expressed as,

$$birds \rightsquigarrow flight,$$

which should be read as '*birds typically fly*'.

Note that defeasibile logic is not the only approach to *default reasoning*. *Logical Programming without Negation as Failure* is one other approach. However, defeasibile logic outperforms the latter in both performance and expressivity. The results of the comparison of these two default reasoning approaches are explained in (Antoniou, Maher and Billington, 1999) [17] and in more depth and detail than the scope of this paper allows.

2.2 Datalog

The declarative logic programming language, datalog, is most often used as a query language for deductive databases. It is based upon predicate logic - a proven formalisation of reasoning, which is a level above propositional logic, yet below defeasible logic. Being based on predicate logic reasoning makes classical datalog sound, complete and, more importantly for the purposes of this paper, a well-established model for propositional logic reasoning.

Using the Tweety example again, the basic syntax of datalog rules are given as follows:

Bird(?tweety) :- Penguin(?tweety)
Bird(?tweety) :- Fly(?tweety)

Note that datalog rules are read from right to left. Furthermore, there is no way to represent negation in classical datalog - the designers deemed this to warrant undesirable complications. The reason for these complications is that datalog was initially designed to be a purely monotonic declarative programming language [18]. This means that expressing the fact that penguins cannot fly, even though they are birds, is not possible in classical datalog. There is a practical solution to this issue - the discussion of which can be found in the *Issues Faced* section of this paper.

Datalog is also not Turing complete and is thus mainly used in the areas of logic programming; knowledge representation and database querying, although it has recently been found useful in a number of other domains such as data integration and data extraction. Furthermore, datalog is a syntactic subset of its precursor, prolog.

2.3 RDFox

Since there is an unthinkably large amount of data flowing through the World Wide Web at any given time, it is a proportionally large benefit to have a structured form of data describing this data (metadata) to improve the discovery of and access to the data it's describing. However, this metadata requires a defined syntax and structure, in order to be machine-readable. The Resource Description Framework (RDF) was developed in collaboration by members of the World Wide Web Consortium [11], as a solution to this issue. In his article, Miller [12] gives a thorough description on what exactly RDF is and how it works. For the purposes of this paper all that needs to be known, concerning RDF, is that a RDF store (triple store) is simply a database built with the intent of storing and retrieving triples through semantic queries.

RDFox is an RDF store developed at the University of Oxford [10]. RDFox is centralised (i.e. RDF data is stored on main memory) and allows for large growth within the RDF knowledge representation store. Triples can be added to a RDFox database by means of importing them or scheduling them for incremental addition or deletion. Datalog rules are added in the same manner. This method of adding triples and rules to a database allows RDFox to compute parallel datalog reasoning, by means of incremental materialisation. This simply means that when the triple, <Tweety - is - bird>, is explicitly stored in the database and the rule, bird $\Longrightarrow$ flight is stored in the same database, then the triple, <Tweety - can - fly>, is implicitly generated (materialised) as well. This check for inference is done incrementally, for each new triple and/or rule that enters the database. RDFox also supports SPARQL query answering [13].

2.4 Explanation and Implementation of the RC Algorithm

The RC algorithm was introduced by Lehmann and Magidor [2] as a way to allow for defeasible reasoning in propositional logic. The algorithm works by ranking the defeasible statements, in a given knowledge base, according to their exceptionality and elegantly reducing defeasible implication checks to a series of classical implication checks.

The RC algorithm used in this paper's project was adapted from Britz et. al. [19], which in turn was adapted from Lehmann and Magidor [2]. The adaptation for this paper's project was necessary in order to apply the algorithm to the implementation of defeasibility in RDFox. This paper's adapted form of the algorithm will be discussed and explained in this section. The RC algorithm consists of three successive sub-procedures; their descriptions are as follows:

(1) *Exceptionality*: determining exceptions within a given knowledge base

(2) *Ranking*: computing an ordered ranking of exceptions based on their exceptionality (i.e. how typical they are)

(3) *Rational Closure*: answering queries, posed at the ranked knowledge base, that posses a defeasibile element

Furthermore, since this paper focuses on the implementation of this algorithm in RDFox, pseudo code for each sub-procedure will also be provided. The provided pseudo code is also adapted, from those presented in Casini et. al. [20]

First, the term *conditional knowledge base* needs to be defined. A conditional knowledge base, K, is a knowledge base of the form K = <T, D>. Here, T is the set of definitions and specialisations (i.e. concrete knowledge or classical formulae), also known as a TBox, and D is the set of defeasible inclusions (i.e. typical consequences or defeasibile formulae), also known as a *defeasible* TBox. With respect to the penguin example, these are expressed as T = {penguin $\rightarrow$ bird, penguin $\rightarrow \neg$ fly} and D = {birds $\rightsquigarrow$ fly}. Furthermore, the computation of the materialisations, $\overline{D}$, of the inclusions in D - which are of the form $A \rightsquigarrow B$, where A and B are propositional atoms - is simply explained in set theory notation by $\overline{D} = \{\neg A \sqcup B \mid A \rightsquigarrow B \in \text{D}\}$ (Casini and Stracia, 2010) [3].

The first function, *Exceptional*, can now be explained. A propositional atom, A, is exceptional with respect to some knowledge base K = <T, D> if and only if K $\models \neg$ A. I.e. the statements in the knowledge base, K, infer the existence of the negation of A (or *not* A). Thus, some defeasibile inclusion, $A \rightsquigarrow B \in$ D, is exceptional with respect to K if its antecedent (i.e. A) is exceptional. The *Exceptional* function takes in T and some D' $\in$ D of K and produces a subset of D', E, which contains all the exceptions in D'. The pseudo code is provided in *Algorithm 1.*

The second function, *Ranking*, deals with exceptions within exceptions by assigning each exception a rank based on their level of exceptionality. This is done by computing *Exceptional* and then repeating its execution with its previous iteration's output, until either there are no more rules to compute or the previous and current outputs are equivalent sets (i.e. further iterations will produce no change). The *Ranking* function takes in the elements of some knowledge base, K = <T, D>, and outputs the ranking R = {D_0, ..., D_n, D_∞}. Here, R is the partitioned set of D, while each D_i is the set of defeasible rules with ranking i and D_∞ is the rank of all classical and implicitly classical rules from both T and D. The *ranking* algorithm in pseudo code is as follows:

Algorithm 1 *Exceptional* (T, D')

```
let K' = T + D'
let E = []
for A ⇝ B in D' do
    if K' ⊨ ¬ A then
        extend E by [A ⇝ B]
return E
```

Algorithm 2 *Ranking* (T, D)

```
let R = []
let E0 = D
let E1 = Exceptional(T, E0)
while E1 = [] or E0 ≠ E1 do
    extend R by E1
    let E0 = E1
    let E1 = Exceptional(T, E0)
if E0 = E1 then
    extend R by [E1 + T]
else                                   ▷ i.e. E1 = []
    extend R by E0
    extend R by T
return R
```

Finally, the third function, *RationalClosure*, will be discussed here. For this step the ranking, R, of the knowledge base, K, must have already been computed. There are only two scenarios that can occur when a query is asked: either the query itself is of a defeasible nature or it is not. If the query does not contain a defeasibile element it can be resolved with classical datalog query methodologies, i.e. a simple check whether the query is stored in or inferred by only the rules in D_∞. More interesting is the scenario of a defeasible query being asked. In such a case, the function *RationalClosure* would have to compute the level of R, if any, at which the antecedent of the defeasibile query would not be considered as an exceptionality. This check would start with the entire ranked knowledge base and subsequently remove a rank, starting from rank D_0 (i.e. the least exceptional rules), until no contradictions in the knowledge base are found. In the event that the query is inherently exceptional to the knowledge base (i.e. no rank can be found that does not produce a contradiction) then rank D_∞ is the final rank that is used to check the query. This is because the rules in rank D_∞ are the most exceptional rules in the knowledge base and cannot be 'relaxed' or removed from the knowledge base without the risk of incorrectly reasoning with respect to the knowledge base. These type of extreme scenarios will be discussed in further detail in the *Testing* section of this paper. In essence, the function *RationalClosure*

takes in the ranks of R and the query, $A \rightsquigarrow B$, and computes with defeasible reasoning whether or not $K \models A \rightarrow B$. The pseudo code for this function is given as follows:

Algorithm 3 *RationalClosure* (R, $A \rightsquigarrow B$)

```
let i = 0
while R ⊨ ¬ A or length of R > 1 do
    remove R[i] from R
    let i = i +1
if R ⊨ A → B then
    return True
else
    return False
```

A proof that this algorithm does indeed work for all cases it is intended to is given in (Casini and Stracia, 2010) [3]. Furthermore, the computational complexity of the algorithm, as explained in (Casini et al., 2015) [**?**], can be proven to be EXPTIME-complete.

3 RELATED WORK

Since its conception and formalisation, defeasibility has been widely investigated. The notion of the rational closure of a positive knowledge base, K, containing typical inclusions, e.g. penguin $\rightsquigarrow$ fly, was first introduced by Lehmann [1]. This was further developed by Lehman and Magidor [2], who also presented an algorithm to compute this. More notably, however, is the research done in extending description logics with the notion of defeasibility. Britz et al. [19] propose an operator to represent defeasible subsumption operations (which is represented as $\rightsquigarrow$ in this paper) in description logics, as well as describe the workings of it. Prior to this, Casini and Stracia [3] had already extended the algorithm in (Lehman and Magidor 1992) to description logics and showed that it reduces to a sequence of classical entailment operations. Casini et al. [20], however, provide a reformulated version of this algorithm by incorporating the defeasible subsumption operator in Britz et al. [19]. Along with this, many others have endeavoured to implement defeasibility into description logics and ontologies. These include, but are not limited to: Moodley, Meyer and Varzinczak [4]; Bryant and Krause [22]; Niemela [23]; Garcia and Simari [25]; etc. In these, concepts of efficient defeasible reasoning; various defeasible reasoning implementations; defeasible reasoning in description logics ontologies; and defeasible logic programming; etc. are tackled. Furthermore, defeasible reasoning has even been extended to datalog specifically, by Martinez and Deagustini [24] - let it be noted, however, that this has only been done for a subset of datalog and not datalog in it's entirety.

By virtue, alone, of the amount of work being put into this specific area of artificial intelligence and reasoning, it is reasonable to deduce that work done in this field could institute significant effects and/or benefits. The work done in this paper not only implements defeasibility into a subsection of datalog, but into datalog in its entirety and shows that this implementation stands correct. Furthermore RDFox, the datalog RDF triple store, is extended to grant it the ability to reason with this defeasible datalog. Finally, the RC algorithm is shown to be practically extended from a description logic environment to a declarative logic one - none of which has been done before. It is believec that, in so doing, this will not only improve understanding of defeasible descriptive logics and datalog, but will have many other implementational benefits as well.

4 PROJECT DESCRIPTION AND MOTIVATION

This section of the paper provides a detailed description of the project, as well as what the motivation behind doing it was.

4.1 Project Description

The purpose of this paper's project was to show that the RC algorithm could be practically and successfully implemented in the environment of a declarative logical programming language. This was done through the development of a prototype defeasibility wrapper for the RDFox system, which uses datalog as its declarative logical programming language. In doing this the expressivity of RDFox is also extended by affording it the ability to correctly reason every and all defeasible instance that it may encounter. Furthermore, it is also shown that datalog can be extended with the defeasible reasoning that will allow for the application of defeasible reasoners in real world models, as opposed to purely theoretical formalisations.

Let the reader note that Pownall [5] does explore a specific aspect of these theoretical formalisms, by proving that the properties of propositional logic and defeasible reasoning still hold after the implementation of the defeasible datalog reasoner. This is done by showing that the KLM properties still hold.

Finally, a range of test cases were created in order to show that the extended version of the RDFox reasoner works as expected. These test cases will cover as many varying scenarios as we are able to concoct. There is, however, no way of testing whether our test cases cover every scenario possible - as is the case with most test case sets. This manual construction of test cases was a necessity due to the lack of defeasible knowledge bases or ontology databases that exist. This will be further discussed in the *Testing* section of this paper.

4.2 Project Motivation

Even though there has been a relatively extensive amount of work done in the domain of non-monotonic reasoning , some gaps still exist in the literature. Among these gaps is the lack of a practical integration of defeasible reasoning into the widely used declarative logic programming languages. More specifically, before this paper's project, the RC algorithm for computing defeasibility has never been extended to declarative logic programming languages. Furthermore, there currently exists no means of concrete proof that the classical datalog declarative logic programming language can be fully extended to allow for correct defeasible reasoning by means of computation and the current working version (as of the writing of this paper) of the RDF triple store and datalog reasoner, RDFox, has no capabilities to deal with defeasible circumstances.

There is also little to no compiled documentation detailing how to perform such a feat.

Concerning datalog, the declarative programming language has become quite popular. Even though it isn't a multi-purpose programming language, it finds uses in a wide variety of fields. These include, but are not limited to, data integration; information extraction; networking; program analysis; security; cloud computing; etc [21]. Furthermore, due to its functionally specific nature, it is easier to make full use of efficient algorithms that are developed for query resolution, when extending its computing capabilities - which is one of the motivations behind choosing datalog for the purposes of this paper.

Regarding RDFox, its extensive features make the RDFox system an immensely valuable tool when working with ontologies, data storage, and database querying. It is an essential part of many works. This is especially evident when noting that RDFox is currently being used by multinational oil and gas companies, electric utility companies, health care consortiums, etc [10]. This extension of the reasoning capability of RDFox will be beneficial, since incorporating defeasibility with its high scalability and fast retrieval rates is expected to significantly assist the research in exception handling within the domain and related domains of artificial intelligence. It is also believed that the improved functionality of RDFox will benefit those already making use of the RDF triple store.

It is a firm belief that this paper's project will have an impact in the field of artificial intelligence, regarding knowledge representation and reasoning, by further investigating and allowing large data stores to reason with exceptions and make logical decisions. In so doing, not only will this work help improve the understanding of defeasible declarative logics and datalog, but it will have many other implementational benefits as well.

5 ISSUES FACED

A significant amount of obstacles and hurdles had to be overcome in order to achieve a working implementation of the RC algorithm into RDFox and, subsequently, datalog. For the benefit of any person/s who wish to endeavour to accomplish a similar task, a few of the more significant issues will be discussed in this section, along with why they occurred and how they were overcome.

5.1 Acquisition of the RDFox Software

At the start of this paper's project (c. April 2018) the RDFox software was freely available for download under an open source licence. It was at this time that the first software downloads were made. However, due to dependency and operating system (OS) compatibility issues, the software had to be re-downloaded from the source. The issue here was that, between the time of the initial download and the need for a re-download, the RDFox system had undergone development for commercial use by the start-up company called Oxford Semantic Technologies [14]. This new development saw a change in licensing, from an open source licence to a noncommercial academic licence - which meant that the software was no longer freely available for download. This hurdle was overcome by directly contacting the developers with regards to acquiring the correct dependency files needed to run the system. The relevant patent attorney was also contacted with regards to acquiring the noncommercial academic licence for the RDFox system. More information on the current development and licensing of the RDFox system can be found on Oxford University's RDFox web page [15]. In the end, the only negative effect that this issue presented to the project was a delay in its timeline.

5.2 Lack of Documentation

Since the RDFox system was initially developed as a university project, there still lacks sufficient documentation (read 'user manuals') on the workings of RDFox, how it should be used, what dependencies the software has, how to troubleshoot common issues, etc. This may change in the near future due to the new development of the system. However, at the time of this paper's project and apart from the RDFox project paper [10] and similar writings, documentation on how to use the system proved hard to come by. This issue was especially prevalent during the initial setup of the system (more of which will be discussed in the next subsection). To combat this problem, the system's source code had to be personally scanned and understood. The available demos were also heavily scrutinised in order to gain a deeper understanding of the functionality of the system, however, these did not give a full expression of RDFox's vast functionality. For the more complex issues in setup, the developers of the system were contacted and were significantly helpful. However, this was only done on one occasion.

The lack of documentation proved only to hamper the initial learning and setup phase of the project. Once the system was running, there were little-no issues that couldn't be solved without the need for documentation and/or expert help.

5.3 RDFox Setup

The development of the defeasible datalog wrapper took place on a Linux system running Ubuntu 16.04 LTS (also known as Xenial Xerus). This is important to note since the RDFox system is OS specific. Currently, RDFox is available for Mac OS, Linux and Windows systems. Unfortunately, the initially downloaded Linux version of the source code had incorrect references to essential dynamic libraries. The dynamic libraries referenced were for the Mac OS version of RDFox. It is reasonable to assume that this was just mere a mix up of source files, since the initial development of the RDFox system took place on mainly Mac OS systems. As such, this problem was essentially an easy fix - since reference only had to be made to the correct libraries - but certainly was not as easy to figure out.

Certain issues also arose with respect to the different language implementations of RDFox, as the software is written in C++, but bridges to Python and Java have also been developed. In the end, the Python bridge of the software was chosen to be made use of, as this was the easiest version to comprehend (due to the simplicity of the Python code) in the time available and had the least dependency issues.

5.4 Lack of Test Cases

One of the most prominent issues faced in this paper's project is the current lack of standard test cases consisting of defeasible ontologies or knowledge bases. This is presumably due to the fact that most of the work done in defeasible reasoning logics have been of a conceptual or theoretical nature; there has never, yet, been much need for a standardised test case base to test a reasoners' ability to handle defeasible inferences. As such, two options presented themselves as a solution to this problem. The first option was to take an already existing, classical, knowledge base and modify it to include sufficient defeasible inferences to test the defeasible reasoners capabilities. The second option was to manually construct these defeasible knowledge bases 'from the ground up'.

For the purposes of this project, the second option was chosen. There are two main reasons behind this choice. The first is that most standardised, classical, ontologies and knowledge bases contain a relatively large amount of content (i.e. the amount of statements that they contain). This introduces the obvious risk of unknowingly affecting the knowledge base (or ontology) in an undesirable way when modifying it, since it would be difficult to comprehend the entire knowledge base in a manner that would eliminate all risk during modification. In essence, modifying a knowledge base whose TBox implications are not fully understood could potentially produce unwanted results. It would also be significantly challenging to trace the source of these results to the specific modification that took place. Furthermore, a larger test case base would not hold any significant benefit in this project, as the purpose of the project was not to create an efficient and well optimised defeasible reasoner, but rather one that is correct. As such, the second reason is that a self constructed, defeasible, knowledge base could be personalised to test this project's defeasible reasoner and, more specifically, the datalog compatible version of the RC algorithm. Indeed, this was done and so doing, more focus was placed on the correctness of the defeasible implementation, during testing, rather than the efficiency thereof. The test cases produced by this project were proven to be sufficient for the testing purposes required by this project and were also validated by second and third parties. The method of test case construction, as well as how testing was carried out, will be discussed in further detail in the *Testing* section of this paper.

5.5 Extension of Datalog to Allow for Negation

Another issue faced, which was inherent of the task that this paper's project endeavoured to complete, was the fact that classical datalog does not have any means of expressing negation in its atoms or axioms. This is simply due to the fact that classical datalog was not designed to cater for nonmonotonic inferences and "*adding negation to datalog rules permits the specification of nonmonotonic queries and hence of nonmonotonic reasoning*" [18]. Indeed, to implement the RC algorithm in datalog, the expressive power of datalog had to be extended to allow for negation.

There are many ways to overcome this barrier and a number of them, each with their own use case and drawbacks, are explained in *Foundations of Databases: The Logical Level* [18]. One of these methods is to introduce negation as failure. The idea behind this is, if given propositional atom A, to simply infer that $\neg A$ exists if and only if A cannot be proven by Selective Linear Definite (SLD) clause resolution [18] - a rudimentary, exoteric inference rule used in logic programming. However, the fundamental issue with this procedure is that a proof for A could very well exceed finite computation time. Let the reader take note that, due to the scope of this particular project, knowledge bases could be constructed that would limit the expansion and possible recursive nature of the inference checks. However, this method would still prove to be computationally complex when compared to the more practical solution that was elected to be used in this project.

Negation introduced by inflationary fixed point semantics was elected to be used in this project. There are two main reasons for this is. The first is that the materialisation process of RDFox is computationally similar to the inflationary character of the semantics - they both iteratively compute statements, simultaneously, with all valid evaluations, until no new statements can be inferred. The second reason is a much more practical - it was relatively simple to implement. The way this was implemented by creating a new class, which acted as the standard way to represent negation, thereby extending the semantics of the datalog language. This class was represented, in datalog, as:

<http://defeasibledatalog.org/hons/negation#False>

The negation class was designed to be used as the consequent in a datalog rule. The antecedent of said rule would be two objects, where one object implied the negation of the other object. As an example, the datalog rule that states that if X is a penguin, then X cannot fly, would look like the following:

<http://defeasibledatalog.org/hons/negation#False> :- Penguin(?X), Fly(?X) .

This proved to be quite effective in the representation of negation in datalog, for this purposes of this project.

6 SOFTWARE DESIGN AND IMPLEMENTATION

In this section the design process, as well as the stages of implementation of the software, will be discussed. Let it be noted that the outcome of this project was not centred around the development of a piece of software that could be rolled out for commercial use, but rather a piece of software that would act as proof that datalog can be extended to allow for defeasible reasoning, using the RC algorithm. However, in spite of this clear theoretical outcome, certain aspects of software engineering were taken into consideration and incorporated for the development of the defeasible datalog wrapper.

As a software development methodology, the waterfall method was used. This is due to the fact that the basic and essential software requirements were pretty much cemented. The main issue was understanding exactly how the rational closure algorithm worked and how to implement it in datalog. However, once this was understood, the coding aspect of the development was relatively straight forward. Apart from this, other aspects of software engineering were taken into consideration - such as modularity. Modularity

was decidedly incorporated, as the software that was developed is essentially a wrapper for a larger system, namely, RDFox. As such, it would have to be easily added into RDFox. Furthermore, as the software essentially transfigures just the datalog rules alone, only using RDFox as a reasoner, the added modularity allows it to potentially be used as a defeasible wrapper for any classical datalog reasoner.

In order to achieve a modular design, the wrapper was written as a standalone python class, which another python class could extend itself by. Each subprocedure of the RC algorithm (i.e. *Exceptional*, *Ranking* and *RationalClosure*) was implemented as a seperate function, all found in this single python class. Furthermore, the class itself makes calls to the datalog reasoner being used (which is RDFox for the thoughts and purposes of this project). After the software engineering designs were in place, the rest of the wrapper's design was reliant on the workings of the RC algorithm.

7 TESTING

In order to prove the correctness of the implementation, not only was the code reviewed by an expert in the field, but correctness testing also occurred. Let it be noted that this was the sole purpose of the testing, to prove the correctness of the implementation and not other aspects, such as the efficiency of the implementation. Furthermore, the test cases were designed and developed with this goal in mind. It was essential that these test cases be manually designed and developed as, since this area of research is still relatively new, a natural supply of reliable test case sources do not exist. An added benefit of manual creation of the test cases was that aspects of testing and predicted outcomes could be closely monitored and designed to cater for personal use.

Test cases consisted of variantly constructed TBox knowledge basis' (i.e. sets of datalog rules), which contained both classical rules and , more importantly, defeasible rules. In order to provide test cases that effectively proved the correctness of the implementation, the rational closure algorithm was closely analysed. It was decided that the test cases would have to be designed in a way that would allow the examination of the implementation's output, given distinct cases/types of defeasible knowledge base TBoxes. These test cases were required to have one of the following properties:

- During ranking, the knowledge base reaches a state where the two E levels (E0 and E1) contain the same rules
- During ranking, the knowledge base reaches a state where E1 is computed to be empty
- Ranking of the knowledge base correctly places multiple rules on the same level

Test cases were developed so that every one of these properties were accounted for and each case was tested. The results of this testing can be found in the following section, *Findings and Results Analysis*.

8 FINDINGS AND RESULTS ANALYSIS

In this section the findings after implementation was completed, as well as the results of the testing, are discussed.

Once the implementation and the testing thereof had concluded, it was found that the implementation was a success. Thus practically proving that the logical reasoning language, datalog, can be extended to cater for defeasible reasoning, with minute hazards and limited restrictions. Furthermore the testing proved that the implementation of defeasible reasoning into datalog, done by this project, was sound. A screenshot of a few examples of the output of the testing are given in Fig.1 and Fig.2, in the appendixes of this paper.

The output of the defeasible datalog wrapper shows the original knowledge base (as the imported datalog rules), as well as how the implementation has ranked the defeasible statements. It can be seen, both theoretically and practically, that these statements have been ranked correctly. This is the case for all the tests that have been done. The same experiment can be conducted by downloading the contents of this project's GitHub repository [6] and testing the wrapper against the provided test cases. All of the test cases used in and by this project, the original implementation of the defeasible datalog wrapper and instructions on how to run the wrapper can be found in the relevant folders of this repository. For the purposes of this project, this proves the correctness of the implementation.

9 LIMITATIONS AND FURTHER WORK TO BE DONE

Due to the scope of this paper's work, the efficiency in the reasoning and computation of the datalog defeasible wrapper could definately be improved. There is also a lack of optimisation that occurred when implementing the wrapper. Furthermore, even though the entirety of datalog was extended by this implementation, only one RDF class was used for said implementation: the rdf:type class [16].

Thus, obvious extensions to this paper's work would be to introduce significant optimisations to the reasoning process and computation time of the wrapper. Since the nature of the RC algorithm allows for the use of recursive procedures, the optimisations noted in (Paramá et al., 2006) [**?**] can be further investigated and implemented if proved to introduce significant improvements to defeasibility testing using the RC algorithm. Also, the implementation can be extended to include more, if not all, standard RDF classes.

10 CONCLUSION

The ability to model real world information in elaborate and simple knowledge bases is extremely powerful. It allows us to model a variety of situations and to draw logical conclusions from them using a formalised process of reasoning. One of the most useful features of logical reasoning is entailment and the ability to draw more implicit information from our knowledge base. Furthermore, it is extremely important that our reasoning does not fail when given new facts. This is often the case in real world modelling where new

facts can contradict previously entailed information. The ability to handle such contradictions makes defeasible reasoning extremely valuable for modelling real world information.

This project practically proves that the RC algorithm can be used to grant descriptive logic programming languages the ability to reason with defeasible inferences. It also shows the successful implementation of defeasible reasoning into the declarative logic programming language, datalog, using RDFox as a reasoner. It is important to note that this defeasible reasoner does still comply with properties from propositional logic, and is thus sound and complete. This is noted by this project's sister project, [5]. Furthermore, this platform for creating and performing defeasible reasoning shows promising application to real world scenarios.

A RESULTS

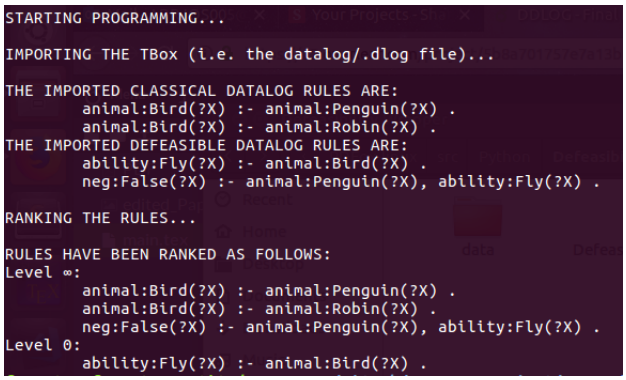

Figure 1: Defeasible Datalog Test 1

Figure 2: Defeasible Datalog Test 2

REFERENCES

[1] Lehmann, D., 1989, "What does a conditional knowledge base entail?," in Proceedings First International Conference on Principles of Knowledge Representation and Reasoning, R. Brachman and H.J. Levesque, eds., Toronto, Ontario.

[2] Lehmann, D. and Magidor, M. (1992). What does a conditional knowledge base entail? *Artificial Intelligence* 55, 1 (1992), 1–60.

[3] Casini, G. and Stracia, U. (2010). Rational Closure for Defeasible Description Logics. In: Janhunen T., Niemelä I. (eds) Logics in Artificial Intelligence. JELIA 2010. *Lecture Notes in Computer Science*, vol 6341. Springer, Berlin, Heidelberg

[4] Moodley, K., Meyer, T. and Varzinczak, I. J. (2012). A defeasible reasoning approach for description logic ontologies. In Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference (SAICSIT '12). ACM, New York, NY, USA, 69-78. DOI=http://dx.doi.org/10.1145/2389836.2389845

[5] Pownall, T. (2018). *Defeasible Datalog: Introducing Defeasible Reasoning into the Declarative Programming Language Datalog.*

[6] Abraham, J. J. (2018). *Honours Project - Defeasible Datalog*. Retrieved 2018 from https://github.com/FreddyManston/Honours-Project.git

[7] Prakken, H. and Sartor, G. (1996). A dialectical model of assessing conflicting arguments in legal reasoning. *Artificial Intelligence and Law* 4 (3-4), pp. 331-368. doi: 10.1007/BF00118496

[8] Prakken, H. and Sergot, M. (1995) *Contrary-to-duty obligations.*

[9] Arregui, A. (2017). *Chisholm's Paradox in Should-Conditionals. SALT 27*: the University of Maryland, College Park. doi: http://dx.doi.org/10.3765/salt.v18i0.2477

[10] Nenov Y., Piro R., Motik B., Horrocks I., Wu Z., Banerjee J. (2015) RDFox: A Highly-Scalable RDF Store. In: Arenas M. et al. (eds) The Semantic Web - ISWC 2015. Lecture Notes in Computer Science, vol 9367. Springer, Cham.

[11] RDF Working Group. (2014). *RDF*. Retrieved May 2018 from https://www.w3.org/RDF/

[12] Miller, E. (2005). An Introduction to the Resource Description Framework. *Bulletin of the Association for Information Science and Technology* 25(1) pp. 15-19

[13] (March 2013) *W3C: SPARQL Query Language for RDF*. Retrieved May 2018 from ://www.w3.org/TR/rdf-sparql-query/#acknowledgements

[14] (2018). *Oxford Semantic Technologies*. Retrieved August 2018 from https://www.oxfordsemantic.tech/

[15] *RDFox*. Retrieved August 2018 from http://www.cs.ox.ac.uk/isg/tools/RDFox/

[16] W3C. (25 February 2014). *RDF Schema 1.1*. Retrieved June 2018 from https://www.w3.org/TR/rdf-schema/

[17] Antoniou G., Maher M.J., Billington D. (1999). *Defeasible logic versus Logic Programming without Negation as Failure*. In The Journal of Logic Programming 42 (2000), pp. 47±57.

[18] Abiteboul S., Hull R., Vianu V. (1995). *Foundations of Databases: The Logical Level (1st ed.).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[19] Katarina Britz, Thomas Meyer, and Ivan Varzinczak. 2011. *Semantic Foundation for Preferential Description Logics*. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7106 LNAI. 491–500.

[20] Casini G., Meyer T., Moodley K., Sattler U., Varzinczak I. (2015). Introducing Defeasibility into OWL Ontologies. In The Semantic Web - ISWC 2015. Arenas, M., et al. (eds). *Lecture Notes in Computer Science*, vol 9367. Springer, Cham, 409-426.

[21] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and emerging applications: an interactive tutorial. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11). ACM, New York, NY, USA, 1213-1216. DOI: https://doi.org/10.1145/1989323.1989456

[22] Bryant, D. and Krause, P. (2004). A review of current defeasible reasoning implementations. The Knowledge Engineering Review, Vol. 00:0,pp. 1–24, Cambridge University Press. doi: 10.1017/S000000000000000. Printed in the United Kingdom.

[23] Niemela, I. (1995). Towards efficient default reasoning. In Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1 (IJCAI'95), Chris S. Mellish (Ed.), Vol. 1. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 312-318.

[24] Martinez, M. V., Deagustini, C. A. D., Falappa, M. A. and Simari, G. R. (2014). Inconsistency-Tolerant Reasoning in Datalog. *In Advances in Artificial Intelligence* – IBERAMIA 2014, Ana L.C. Bazzan and Karim Pichara (Eds.). Springer International Publishing, Cham, 15–27.

[25] García, A.J., and Simari, G.R. (2014). Defeasible logic programming: DeLP-servers, contextual queries, and explanations for answers. *Argument & Computation*, 5, 63-88.