Notation3 as the Unifying Logic for the Semantic Web

'Notation3' als de verenigende logica voor het semantisch web

Dörthe Arndt

Promotoren: prof. dr. ing. E. Mannens, prof. dr. ir. R. Verborgh, prof. dr. ir. T. Schrijvers
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

UNIVERSITEIT
GENT

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2019 - 2020

ISBN 978-94-6355-292-9
NUR 984, 983
Wettelijk depot: D/2019/10.500/100

Examination board

Chair

prof. dr. Patrick De Baets *Ghent University, Belgium*

Secretary

prof. dr. Femke Ongenae *Ghent University, Belgium*

Reading committee

ir. Jos De Roo *Agfa Healthcare, Belgium*

prof. dr. Harold Boley *University of New Brunswick, Canada*

prof. dr. Christophe Scholliers *Ghent University, Belgium*

prof. dr. Axel Polleres *WU Vienna, Austria*

Supervisors

prof. dr. Erik Mannens *Ghent University, Belgium*

prof. dr. Ruben Verborgh *Ghent University, Belgium*

prof. dr. Tom Schrijvers *KU Leuven, Belgium*

Preface

Mit Mathematikern ist kein heiteres Verhältnis zu gewinnen.
Johann Wolfgang von Goethe

Before I started this journey, I always thought that computer scientists and mathematicians are somehow similar: the former apply what the latter studied before in theory. Now, after I did my research, participated in different projects, read many papers and talked to different people at conferences, I can tell: I was wrong! In mathematics many things are only done on a theoretical level and the validity of the results needs to be formally proven to be sure that they are correct. As a consequence, mathematicians focus a lot on details. They need to be able to exclude all possible problems. In (applied) computer science on the other hand, new ideas can be implemented. The fact that the resulting program works properly is then a first indication that the initial idea was not too bad. For many people this first indication is enough. They want to deliver working tools as fast as possible and therefore accept the risk that problems can occur later.

Being a mathematician who stepped over to computer science in order to learn more about this fascinating world, I often struggled with this *cultural difference* (as many of the persons mentioned below can confirm). It was not always easy to accept that some people consider theory a burden, it was even harder to understand that they do not enjoy discussing details even if these details directly impact their implementations, but the hardest thing for me was that even by showing them the beauty of logic I could not change their mind. On the other hand I also learned a lot by facing this new point of view: I learned to always take practical aspects into account if it comes to computer programs, and I learned that sometimes (but just sometimes) it is good to start implementing before you think a problem through, just to better understand it.

Having the hope that everyone who did not enjoy discussing the scoping of variables as much as I do or who sometimes did not want to solve pos-

sible problems before even having written a single line of code also sees *some* positive aspects in having worked with me, I chose the above quote to let you know that you are not alone: There are many people who consider mathematicians *difficult* people among them the famous Johann Wolfgang von Goethe. Taking up that quote from above, I thank everyone who tried to get "ein heiteres Verhältnis" with me despite the difficulties mentioned above.

In particular I thank Miel for guiding me through the writing process of this thesis, Femke for listening and helping with all kinds of problems, and the Knowledge on Web-Scale (KNoWS) team for sharing their knowledge with me. As a team is always just as good as its members, I want to thank everyone individually (taking the risk of forgetting someone, sorry if this is the case), especially I thank my former colleagues Tom D.N., Sam, Hajar, Cristian, Laurens, Gayane, Dieter D.P., Dieter D.W. and all current members of the team, in particular Joachim, Pieter C., Pieter H., Ben, Sven, Gerald, Martin, Julian, Harm, Brecht, Anastasia, and Ruben T. I also thank all current and former members of IBCN, I had the pleasure to work with, in particular Pieter B., Mathias, Stijn and Alexander.

The research conducted in this thesis has mostly been done in the context of projects involving industry partners. I thank all these partners. In this context I want to give a special mention to Agfa healthcare. I thank everyone at Agfa I worked with. Visiting you was always the highlight of my week. In particular I thank Giovanni for always sharing the programmer's perspective with me, I thank Hong for his challenging questions, Boris for helping me with decisions, and Els for always trying to keep us focussed – I know that this is not an easy task given that Jos and I really enjoy discussing logical details. A very special thanks goes to Jos for exactly those discussions. I think there is no one with whom talking about logics is more enjoyable. It was you who reminded me how exciting research can be.

I furthermore thank all members of the jury for thoroughly reviewing my thesis. I am convinced that your detailed reports as well as your challenging questions during my defence helped me to improve this book. I would also like to thank you for all occasions on conferences, meetings and courses in which you – knowingly or unknowingly – helped me to perform the research presented in this thesis. A very special thanks goes in this context to Harold for making me feel welcome in the RuleML community, especially at my very first RuleML conference (2015 in Berlin).

I also want to thank my supervisors. I thank Tom S. for encouraging me to aim for *good* papers instead of papers *good enough to be accepted*. I thank Erik for always backing me up. I thank Ruben for sharing his enthusiasm

with me and for pointing me to my research topic: I am very happy that you immediately anticipated my passion for N3Logic.

I also thank all my friends who supported me during the writing of my thesis or even before. A special thanks goes to Stefanie, my former colleague, who encouraged (or forced?) me to start this journey, without her I would never have applied for my PhD position. I thank my family, my parents, my brothers and, especially, my sister Sabine who helped me with the Dutch translation of my summary (it was a very last minute help, so please do not judge her language skills based on that summary).

Last, but not least, I thank Nicolas who gave me a lot practical and moral support during this whole process. He agreed to move to Belgium with me, he accepted that I spent many weekends with N3 instead of him (I agree that this is a questionable choice) and he counselled me whenever I came home complaining about these *"weird computer scientists"* who seemed not to care about the correctness of their implementations. During the writing process he furthermore took care of our daughter Paula and thereby gave me the time to finish this book. Without you this thesis would not have been possible. Thank you!

Dörthe Arndt
Gent, October 25, 2019

Contents

Summary

To fully realise the vision of the Semantic Web – a machine understandable version of our current Web – we need to make use of logic. Only a clear logical language with properly defined semantics enables us to share the knowledge we have about the world with computers. The Semantic Web architecture foresees the use of logic for three different kinds of applications: (1) logic should enable querying, (2) it should make it possible to express and make sense of ontologies[1] and taxonomies, and (3) it should support rule-based reasoning. While these kinds of applications have been realised so far by using different technologies – mainly SPARQL, RDFS/OWL and RIF – it is part of the vision to cover these functionalities and connect these technologies by using one single framework: the *Unifying Logic*. This logic needs to furthermore support the layer of proofs: it needs to be possible to express and exchange proofs performed by applying *Unifying Logic*.

In this thesis we investigate this idea of a *Unifying Logic* and test the suitability of one possible candidate to fulfil that role: Notation3 Logic (N3). N3 is a rule-based logic which has been introduced about a decade ago. It extends the Resource Description Framework (RDF) – the most used format of the Semantic Web – by universal quantification, by the possibility to cite formulas and to express rules, and by a number of different built-in predicates. N3 makes it possible to reason about any input which can be syntactically represented in RDF, and most N3 reasoners already produce proofs to explain their derivations. This makes N3 a promising candidate to become the *Unifying Logic*, but there are also obstacles: the semantics of N3 is only informally defined which not only makes it difficult to study its formal properties, it has also practical consequences: in many cases the interpretations of the same formula differ between reasoning engines.

We understand the *Unifying Logic* as a practical tool which needs to support the common tasks associated with querying, reasoning over ontologies and

[1] Here an ontology should be understood as a "formal, explicit specification of a shared conceptualization" [1].

taxonomies, and rule-based inference. This tool itself needs to be part of the Semantic Web – for example by understanding its language RDF – and it should support the layer of proofs. This view is different to many other approaches which mainly try to define a logic which combines description logic and rule-based reasoning and still remains decidable. While we agree that this is an interesting goal, we are not convinced that this approach brings the Semantic Web further, especially if we consider that the combination of RDF and OWL DL, OWL Full, is already not decidable. The semantics of the *Unifying Logic* should furthermore be well defined and it should be compatible with RDF.

Having identified these requirements – (1) clear semantics, (2) compatibility with RDF, (3) support for querying, reasoning over ontologies/taxonomies and rule-based inferencing, and (4) support of proofs – we next investigate in how far these can be met by N3.

As the Semantics of N3 is not clearly defined and implementations even differ in their understanding of certain formulas, we start by performing tests to better understand the problem: We detect that the reasoners EYE and Cwm disagree in their interpretation of implicitly quantified variables. In N3 it is possible to use universally or existentially quantified variables for which the quantifiers are not explicitly stated but implicitly assumed. The documents introducing N3Logic state that implicitly universally quantified variables are quantified on their parent formula. But since this term is not further explained, Cwm and EYE understand it differently.

In order to better understand this difference and its consequences, we then define a logic which is similar to N3 but only supports explicit quantification: N3 Core Logic. We provide a model theory for N3 Core Logic which is mostly compatible with RDF. We then define an attribute grammar which maps formulas expressed in N3 syntax to their two N3 Core Logic interpretations, one following Cwm and the other following EYE. By doing so, we defined two possible semantics for N3. We furthermore created a mechanism to decide for every N3 formula whether or not it leads to disagreements between the reasoners.

Next, we implemented the attribute grammar we defined in Haskell. In order to be able to measure the practical impact of the problem we detected, we ran our implementation on different N3 files which have been used in industry-related research projects. For 31% of these files we could detect discrepancies between the N3 Core Logic translations. We further analysed these discrepancies and identified three different kinds of sources: proofs, built-ins and nested formulas which do not contain proof-predicates or built-in functions. As proofs are often computer-generated and built-ins can be

reasoner-dependent, we argue that the last group of constructs, nested formulas which do not involve built-ins or proof-predicates, is the most dangerous: users writing such kinds of rules normally expect interoperability. 13% of our critical files fall under that group. We therefore come to the conclusion that this problem needs to be solved by the community. We discuss the different options of which we favour to simply follow the interpretation of EYE, since it is easier to implement and – at least in our opinion – easier to understand.

Having discussed the Semantics of N3 and its relation to RDF, we next focus on the tasks which can be performed by using N3. In order to know whether N3 can solve the same practical problems as OWL DL reasoning and SPARQL querying with a comparable performance, we considered two use cases: a semantic nurse call system and a system to perform RDF-validation. Both use cases have been implemented previously by applying OWL DL reasoning and/or SPARQL querying. We approached both use cases by applying N3 instead. Our resulting systems were able to provide the same functionality as the original ones. The performance of our nurse call system was faster in all cases. The RDF-validation system was faster for datasets below 100,000 triples. We can thus conclude that the implementations deliver comparable results.

After that investigation on N3's ability to support ontology reasoning and querying, we turn to the next requirement: proofs. As proofs in the Semantic Web vision are designed to be exchanged among different parties in the Web, they should not contain formulas whose meaning is ambiguous. We therefore introduce the notion of *simple formulas* which are formulas in which the universal variables are interpreted equally by the reasoners Cwm and EYE. For such formulas we define the direct semantics. We then formally define the different proof steps which can be expressed using the N3 proof vocabulary and prove their correctness.

In addition to that formal consideration of proofs, we also study how N3 proofs can be used in practice. Here we consider a use-case which goes beyond the simple establishment of trust: In the context of automatic API composition, we use proofs as plans. If we describe possible operations which can be performed by calling a hypermedia API by means of rules clearly stating under which circumstances the operation can be performed and which situation it will produce, these rules can be combined in a proof towards a desired goal. The proof can take context knowledge into account, and each operation contributing to the goal is listed in the proof and can then be executed. In case the context changes, the proof can be easily adapted. We define the concrete format to describe Web APIs, RESTdesc, and an algorithm to use and update such plans. We provide a proof for the termination of our

algorithm. We furthermore discuss its limits and extend the idea of using proofs as plans to another use case.

In conclusion, the research conducted for this thesis shows that N3 Logic is indeed a very promising candidate to become the *Unifying Logic* of the Semantic Web. N3 is an extension of RDF, it supports reasoning over ontologies/taxonomies, querying and rule-based inferencing, and it supports the layer of proof. N3 reasoners do not only provide proofs, N3 is even expressive enough to allow the reasoner to represent such proofs in the logic itself. The research has shown that it is possible to formalise the semantics of N3 is a way that it is compatible with RDF. For a standardised formalisation the Semantic Web community needs to come to an agreement on how to interpret implicit universal quantification. We hope that this thesis provides a valuable step towards this agreement.

Samenvatting

Om de visie van het Semantisch Web - een machine verstaanbare versie van ons huidige web - volledig te realiseren, moeten we gebruik maken van logica. Alleen een zuivere logische taal met een goed gedefinieerde semantiek stelt ons in staat om de kennis die we over de wereld hebben met computers te delen. De architectuur van het Semantisch Web voorziet het gebruik van logica voor drie verschillende soorten toepassingen: (1) logica moet queries mogelijk maken, (2) zij moet het mogelijk maken om ontologieën en taxonomieën uit te drukken en te begrijpen, en (3) zij moet regelgebaseerde redenering ondersteunen. Hoewel deze toepassingen tot nu toe met behulp van verschillende technologieën gerealiseerd zijn - voornamelijk SPARQL, RDFS/OWL en RIF – is de visie om deze functionele eigenschappen te ondersteunen en deze technologieën te verbinden door gebruik te maken van één enkel framework: de verenigende logica. Deze logica moet bovendien bewijzen ondersteunen: het moet mogelijk zijn om de uitgevoerde bewijzen uit te drukken en uit te wisselen door de verenigende logica toe te passen.

In dit proefschrift onderzoeken wij dit idee van een verenigende logica en testen we de geschiktheid van een mogelijke kandidaat om die rol te vervullen: Notation3 Logic (N3). N3 is een regelgebaseerde logica die ongeveer tien jaar geleden is geïntroduceerd. Het vult het Resource Description Framework (RDF) - het meest gebruikte formaat van het Semantisch Web – aan met universele kwantificatie, met de mogelijkheid om formules aan te halen en regels uit te drukken, en met een aantal verschillende ingebouwde predicaten. N3 maakt het mogelijk te redeneren over elke input die syntactisch kan worden weergegeven in RDF, en de meeste N3-reasoners produceren al bewijzen om hun afleidingen te toe te lichten. Dit maakt N3 een veelbelovende kandidaat om de verenigende logica te worden, maar er zijn ook obstakels: de semantiek van N3 is alleen informeel gedefinieerd. Dit maakt het niet alleen moeilijk om de formele eigenschappen te bestuderen, het heeft ook praktische gevolgen: vaak interpreteren verschillende reasoning engines één en dezelfde formule op verschillende manieren.

We begrijpen de verenigende logica als een praktisch hulpmiddel dat de alge-

mene taken moet ondersteunen die verbonden zijn aan querying, redeneren over ontologieën en taxonomieën, en regelgebaseerde gevolgtrekking. Dit hulpmiddel moet zelf deel uitmaken van het Semantisch Web - bijvoorbeeld doordat het diens taal, RDF, begrijpt - en het moet bewijzen ondersteunen. Deze aanpak is anders dan vele andere benaderingen die vooral proberen om een logica te definiëren die beschrijvingslogica en regelgebaseerde redenering combineert en nog steeds beslisbaar blijft. Hoewel we het erover eens zijn dat dit een interessant doel is, zijn we er niet van overtuigd dat deze benadering het semantisch web verder brengt, vooral als we bedenken dat de combinatie van RDF en OWL DL, OWL Full, al niet beslisbaar is. De semantiek van de verenigende logica moet bovendien goed gedefinieerd zijn en compatibel zijn met RDF.

Na identificatie van deze vereisten: (1) duidelijke semantiek, (2) compatibiliteit met RDF, (3) ondersteuning van querying, redeneren over ontologieën/taxonomieën en regelgebaseerde afleiding, en (4) ondersteuning van bewijzen - richten we onze aandacht op N3 om te onderzoeken in hoeverre deze logica aan die vereisten kan voldoen.

Aangezien de semantiek van N3 niet duidelijk is gedefinieerd en implementaties zelfs verschillen in hun interpretatie van bepaalde formules, beginnen we met het uitvoeren van tests om het probleem beter te begrijpen: we stellen vast dat de reasoners EYE en Cwm het oneens zijn in hun interpretatie van impliciet gekwantificeerde variabelen. In N3 is het mogelijk om universeel of existentieel gekwantificeerde variabelen te gebruiken waarvoor de kwantoren niet expliciet worden vermeld maar impliciet worden verondersteld. De documenten die de N3Logic introduceren, stellen dat impliciet universeel gekwantificeerde variabelen zijn gekwantificeerd op basis van hun bovenliggende formule. Maar omdat deze term niet verder wordt uitgelegd, begrijpen Cwm en EYE hem op verschillende manieren.

Om dit verschil en de gevolgen ervan beter te begrijpen, definiëren we vervolgens een logica die vergelijkbaar is met N3, maar die alleen expliciete kwantificering ondersteunt: N3 Core Logic. We bieden een modeltheorie voor N3 Core Logic aan die grotendeels compatibel is met RDF. Vervolgens definiëren we een attribuutgrammatica die formules in N3-syntaxis toewijst aan hun twee N3 Core Logic-interpretaties, de ene volgens Cwm en de andere volgens EYE. Daardoor hebben we twee mogelijke semantieken voor N3 gedefinieerd. Bovendien hebben we een mechanisme ontwikkeld om voor elke N3-formule te beslissen of het al dan niet leidt tot onenigheid tussen de reasoners.

Vervolgens hebben we de attributengrammatica in Haskell geïmplementeerd. Om de praktische impact van het probleem dat we hebben uitgemaakt te kunnen meten, hebben we onze implementatie toegepast op verschillende

N3-bestanden die zijn gebruikt in sectorgerelateerde onderzoeksprojecten. Voor 31% van deze bestanden konden we discrepanties uitmaken tussen de N3 Core Logic-vertalingen. Vervolgens hebben we deze discrepanties verder geanalyseerd en drie verschillende soorten bronnen geïdentificeerd: bewijzen, built-ins en geneste formules die geen bewijs-predicaten of ingebouwde functies bevatten. Omdat bewijzen meestal door de computer gegenereerd en ingebouwd kunnen worden, zijn wij van mening dat de laatste groep constructies, geneste formules die geen built-ins of proof-predikaten bevatten, het gevaarlijkst is: gebruikers die dergelijke regels schrijven, verwachten normaal gesproken interoperabiliteit. 13% van onze kritieke bestanden vallen onder die groep. We komen daarom tot de conclusie dat dit probleem door de community moet worden opgelost. We bespreken de verschillende opties, waarvan we er de voorkeur aan geven om gewoon de interpretatie van EYE te volgen, omdat het gemakkelijker te implementeren is en - althans wat ons betreft - gemakkelijker te begrijpen is.

Nadat we de semantiek van N3 en zijn relatie tot RDF hebben besproken, concentreren we ons daarna op de taken die kunnen worden uitgevoerd met behulp van N3. Om te weten of N3 dezelfde praktische problemen als OWL DL-reasoning en SPARQL-query's met een vergelijkbare prestatie kan oplossen, hebben we twee use-cases overwogen: een semantisch oproepsysteem voor verpleegkundigen en een systeem voor het uitvoeren van RDF-validatie. Beide use-cases zijn eerder geïmplementeerd door OWL DL-reasoning en/of SPARQL-query's toe te passen. We hebben beide use-cases benaderd door in plaats daarvan rule-based reasoning toe te passen. De resulterende systemen waren in staat om dezelfde functionaliteit te bieden als de originele. De prestaties van ons verpleegoproepsysteem waren in alle gevallen sneller. Het RDF-validatiesysteem was sneller voor datasets van minder dan 100.000 triples. We kunnen dus concluderen dat de implementaties vergelijkbare resultaten opleveren.

Na het onderzoek naar het vermogen van N3 om ontologie-reasoning en -query's te ondersteunen, gaan we over op het volgende vereiste: bewijzen. Omdat bewijzen in de visie van het Semantisch Web uitwisselbaar dienen te zijn tussen verschillende partijen op het web, mogen ze geen formules bevatten waarvan de betekenis dubbelzinnig is. We introduceren daarom het begrip simpele formules, dat zijn formules waarin de universele variabelen gelijk worden geïnterpreteerd door de reasoners Cwm en EYE. Voor dergelijke formules definiëren we de directe semantiek. Vervolgens formuleren we formeel de verschillende bewijsstappen die kunnen worden uitgedrukt met behulp van het N3-bewijsvocabulaire en we bewijzen hun juistheid.

Naast die formele beschouwing van bewijzen, bestuderen we ook hoe N3-bewijzen in de praktijk kunnen worden gebruikt. We beschouwen hier een

use-case die verder gaat dan het eenvoudig vaststellen van vertrouwen: in de context van automatische API-compositie gebruiken we bewijzen als plannen. Als we mogelijke operaties beschrijven die kunnen worden uitgevoerd door gebruik van een hypermedia-API aan te roepen door middel van regels die duidelijk aangeven onder welke omstandigheden de bewerking kan worden uitgevoerd en welke situatie dat zal produceren, kunnen deze regels worden gecombineerd in een bewijs van een gewenst doel. Het bewijs kan rekening houden met kennis van de context, en elke bewerking die bijdraagt aan het doel wordt vermeld in het bewijs en kan vervolgens worden uitgevoerd. Als de context verandert, kan het bewijs eenvoudig worden aangepast. We definiëren het concrete formaat om Web API's, RESTdesc en een algoritme te beschrijven om dergelijke plannen te gebruiken en bij te werken. We leveren een bewijs voor het stoppen van ons algoritme. We bespreken bovendien de limitaties en breiden het idee uit om bewijzen als plannen voor een andere use-case te gebruiken.

Concluderend laat het onderzoek in dit proefschrift zien dat N3 Logic inderdaad een veelbelovende kandidaat is om de verenigende logica van het Semantisch Web te worden. N3 is een uitbreiding van RDF, het ondersteunt redeneren over ontologieën/taxonomieën, querying en regelgebaseerd afleiden en het ondersteunt de bewijslaag. N3-reasoners leveren niet alleen bewijzen, N3 is zelfs expressief genoeg om de reasoner in staat te stellen om dergelijke bewijzen in de logica zelf weer te geven. Het onderzoek heeft aangetoond dat het mogelijk is om de semantiek van N3 zodanig te formaliseren dat het compatibel is met RDF. Voor een gestandaardiseerde formalisering moet de semantisch web gemeenschap overeenstemming bereiken over de interpretatie van impliciete universele kwantificatie. We hopen dat dit proefschrift een waardevolle stap is naar deze overeenkomst.

Chapter 1

Introduction

With the idea of the Semantic Web [2], a new vision on how to use the Internet was born. The World Wide Web [3] and its numerous functionalities should no longer only be used by humans – who can read websites or follow links to gain knowledge or use applications – but also by computers. To enable the computer to use the Web – just as humans do – it must *understand* the Web. But how does a computer think? How can we teach it content? And which language does it understand?

When a human user reads a text in the Web, he has an idea of the concepts behind it. He *knows* for example that the cockatoo is a kind of bird, that most birds can fly and that he can browse the Web to find further information. To provide the computer with similar knowledge about concrete objects (the cockatoo) and bigger concepts (birds, flying), and to enable it to reason and draw conclusions (cockatoos can fly) a representation of these facts and a clear definition of their meaning is needed: a logic.

To represent basic facts, the Semantic Web follows a common standard, the Resource Description Framework (RDF) [4]. The semantics of RDF is clearly defined and other standards are designed on top of it, but RDF itself does only include very basic entailment regimes, in other words: there is no internal mechanism designed how the computer is supposed to draw complex conclusions. To either query a graph, or to derive new knowledge applying description logics or rule based reasoning, there are other standards defined. The envisioned architecture of the Semantic Web then foresees a *Unifying Logic* on top of those. Of which nature this logic needs to be and what the possible candidates are is still subject to discussion.

In this thesis we take a closer look at the *Unifying Logic* and discuss one of the candidates to become such a logic: Notation3. Introduced by Tim Berners-Lee et al. [5], Notation3 Logic is a rule-based logic which extends

RDF by different features like an inference mechanism, the option to cite graphs and the possibility to express proofs. We discuss how the logic itself can be formalised, how it interacts with other formalisms and how proofs are useful for applications.

Next to this introduction, this thesis consists of the following chapters:

Chapter 2 – Unifying Logic includes an introduction to the envisioned architecture of the Semantic Web with a special focus on the concept of a *Unifying Logic*. In this context we also introduce N3Logic as a promising candidate to fulfil this role. This then leads to the research questions we are going to explore in this thesis.

Chapter 3 – Implicit Quantification discusses a problem of N3Logic: In N3 it is possible to use bound variables for which the universal or existential quantifiers are not explicitly stated, but implicitly assumed. The meaning of these variables is not clearly defined and their interpretations differ. We put this observation in relation to RDF and to other frameworks which support some kind of implicit quantification.

Chapter 4 – Defining the Semantics of N3Logic aims to further clarify the problem. We define a core logic for N3Logic which covers all relevant features of N3 but only supports explicit quantification. We specify an attribute grammar which maps N3 formulas to that logic according to the interpretations of the reasoners Cwm and EYE and thereby define two possible semantics of N3.

Chapter 5 – The impact of having different N3 interpretations treats the practical consequences of the differences we found. We implemented the attribute grammar and tested for files used in practical applications whether their interpretations differ. This is the case for 31% of all files. We analyse the differences found and discuss possible solutions for the problem.

Chapter 6 – Querying and ontology reasoning with Notation3 Logic focusses on the applications which can be realised using N3Logic. In particular, we discuss two use cases: a semantic nurse call system and a system for data validation. Both have previously been implemented by applying SPARQL querying and OWL-DL reasoning. We show that both use cases can also be tackled with N3 and that the implementations using N3 provide comparable results.

Chapter 7 – Proofs in N3 discusses proofs in N3. For N3, there is a proof vocabulary defined. We explain this vocabulary and formally define

the proof steps it expresses. As proofs in the Web are meant to be exchanged, we define a subset of N3 which is not ambiguous. For that subset we define the direct semantics and then proof the correctness of the proof calculus.

Chapter 8 – Applications of proofs: Adaptive API composition introduces use cases for proofs which go beyond simple verification and assessment of trustworthiness. If we express possible API operations or possible sensor queries we can perform as rules – including an explanation when they can be applied and what they mean for our context – the applications of these rules can appear in proofs verifying goals we want to achieve. These proofs can then be understood as plans. We can execute and – in case the context changes – update them.

Chapter 9 – Conclusions concludes this thesis. We answer the research questions and discuss the different lessons learned while performing the research which lead to this thesis. We then give an outlook to future work and discuss how the Semantic Web community could move forward based on the findings of this thesis.

These chapters are partly based on peer-reviewed publications. We list these publications together with the chapters there were used in *(in brackets)* below:

- D. Arndt, T. Schrijvers, J. De Roo, R. Verborgh, Implicit quantification made explicit: How to interpret blank nodes and universal variables in Notation3 Logic, Journal of Web Semantics 58 (2019) 100501. doi:10.1016/j.websem.2019.04.001.
  URL https://authors.elsevier.com/c/1ZWg55bAYUaMkH *(Chapters 3–5)*

- D. Arndt, R. Verborgh, J. De Roo, H. Sun, E. Mannens, R. Van de Walle, Semantics of Notation3 logic: A solution for implicit quantification, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 127–143.
  URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_9 *(Chapters 3–5 and 7)*

- D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle, E. Mannens, Ontology reasoning using rules in an eHealth context, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.),

Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 465–472.
URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_31 *(Chapter 6)*

- D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle, E. Mannens, Improving OWL RL reasoning in N3 by using specialized rules, in: V. Tamma, M. Dragoni, R. Gonçalves, A. Ławrynowicz (Eds.), Ontology Engineering: 12th International Experiences and Directions Workshop on OWL, Vol. 9557 of Lecture Notes in Computer Science, Springer, 2016, pp. 93–104. doi:10.1007/978-3-319-33245-1_10.
URL http://dx.doi.org/10.1007/978-3-319-33245-1_10 *(Chapter 6)*

- D. Arndt, B. De Meester, A. Dimou, R. Verborgh, E. Mannens, Using rule-based reasoning for RDF validation, in: S. Costantini, E. Franconi, W. Van Woensel, R. Kontchakov, F. Sadri, D. Roman (Eds.), Proceedings of the International Joint Conference on Rules and Reasoning, Vol. 10364 of Lecture Notes in Computer Science, Springer, 2017, pp. 22–36. doi:10.1007/978-3-319-61252-2_3 *(Chapter 6)*

- B. De Meester, P. Heyvaert, D. Arndt, A. Dimou, R. Verborgh, RDF graph validation using rule-based reasoning, submitted to: Semantic Web Journal *(Chapter 6)*

- R. Verborgh, D. Arndt, S. Van Hoecke, J. De Roo, G. Mels, T. Steiner, J. Gabarró Vallés, The pragmatic proof: Hypermedia API composition and execution, Theory and Practice of Logic Programming 17 (1) (2017) 1–48. doi:10.1017/S1471068416000016.
URL http://arxiv.org/pdf/1512.07780v1.pdf *(Chapter 7 and 8)*

- D. Arndt, P. Bonte, A. Dejonghe, R. Verborgh, F. De Turck, F. Ongenae, SENSdesc: connect sensor queries and context, in: R. Zwiggelaar, H. Gamboa, S. Fred, Ana Bermúdez i Badia (Eds.), Proceedings of the 11th International Joint Conference on Biomedical Engineering Systems and Technologies, SCITEPRESS, 2018, pp. 671–679. doi:10.5220/0006733106710679.
URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006733106710679

  (Chapter 8)

Chapter 2

Unifying Logic

In this chapter we discuss the idea of the *Unifying Logic* for the Semantic Web. The term itself stems form the so-called *Semantic Web Stack*. This schema represents the envisioned architecture for the Semantic Web and the *Unifying Logic* is one of its pieces. Therefore we start with a discussion of that stack in general to then focus on the parts relevant for this thesis. These considerations then lead to the research questions.

2.1 The architecture of the Semantic Web

To realise the dream of a machine understandable Web as it was envisioned by Tim Berners-Lee et al. [2], an architecture has been proposed, the *Semantic Web Stack* (aka *Semantic Web Layer Cake*). This stack provides a structural overview of the different concepts, technologies and standards needed for the Semantic Web to become reality. Its shape has changed over the years [14] due to the progress made – several languages such as RDF [4] and SPARQL [15] have been standardised – but also as a consequence of controversial discussion [16, 17]. Knowing that this version can – and most probably will – evolve and change in the coming years, we focus on the latest official variant and discuss its different parts. This variant is displayed in Figure 2.1.[1] Each layer placed on top of other layers depends on these other layers but not on those above them. Neighbouring layers can, but not necessarily have to make use of each other. We go from the bottom of the stack to its top to explain all its elements:

[1] Colours and shapes are changed in this version to make it fit better into this book. The original picture is available at: https://www.w3.org/2007/03/layerCake.svg. A newer (unofficial) version is for example provided by Hogan [18].

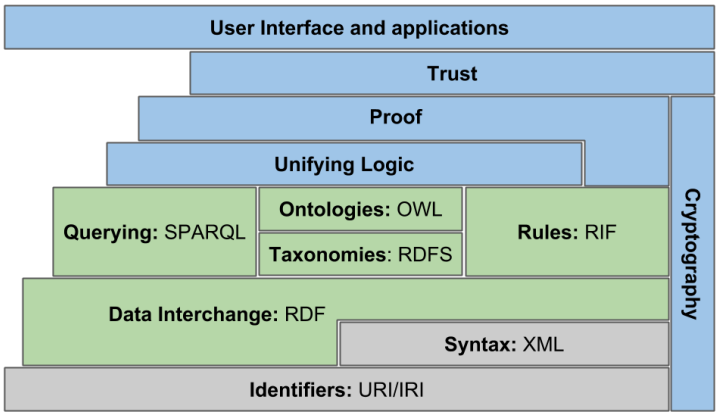


Figure 2.1: Semantic Web Stack.

Identifiers The Semantic Web is a global network which relies on interoperability. In such a big setting, it is crucial that the names of individuals and concepts are unique. For their representation Uniform Resource Identifiers (URIs) [19] and Internationalized Resource Identifiers (IRIs) [20] are used.

Syntax There are several syntax formats frequently used in the Semantic Web. Standards of the World Wide Web are employed – such as XML [21] or JSON [22] – but also formats only created for the Semantic Web such as for example the Terse RDF Triple Language (Turtle) [23].

Data Interchange In order to exchange data it must be agreed on a common structure to express knowledge. This structure should not be too complex for a machine to parse but still powerful enough to make simple statements and connections. In the Semantic Web the Resource Description Framework (RDF) [4] was developed with that goal in mind. Knowledge is expressed using simple triples consisting of subject, predicate and object. By using unique names (IRIs and URIs) connections between different triples are made. Ideally, one resulting graph then forms the Semantic Web.

Taxonomies For simpler statements about classes, instances and properties the Semantic Web uses the standard RDF Schema (RDFS) [24]. RDFS extends the basic representation format RDF by several predicates with a predefined meaning such as for example `rdfs:subclassOf` to denote that one class is a subclass of another[2].

[2]Here and later in this section we use the prefix "`rdfs`" which stands for `http://www.w3.org/2000/01/rdf-schema#`. In Section 2.5.1 we further explain prefixes.

Ontologies The taxonomy already gives some basic information about concepts and individuals in the web. To express more complex things in a fixed way, ontologies are used. Ontologies can be understood as a collection of statements about concepts, classes and individuals using a broad variety of logical predicates which have a fixed meaning for the computer. Even though, in that sense a collection of rules over RDF data can (and should) also be seen as an ontology, the Semantic Web stack only lists the Web Ontology Language (OWL) [25] as a standard to express ontologies. This language is based on description logics [26] and a reasoner can use these statements to draw conclusions.

Rules Alternatively to OWL, but also in combination with this standard, there is another way of stating knowledge about concepts and data in the Semantic Web: rules. Having their origin in classical logic programming, rules are used to directly state which triples or patterns of triples can be concluded from given information. To combine rule-based reasoning with OWL reasoning the Semantic Web Rule Language (SWRL) [27] can be used. As a general format to exchange different kinds of rules, the Rule Interchange Format (RIF) [28] was created. There are also many other rule-based logics used in the Semantic Web which are not standardised (yet), one of them is Notation3 Logic (N3Logic) [5] which is the subject of this thesis.

Querying Independently of whether reasoning is performed on RDF data or not, there are many situations in which a user or an application wants to retrieve information by searching for triples of a certain kind. Therefore, querying is an important part of the Semantic Web. To query data from RDF the SPARQL Protocol And RDF Query Language (SPARQL) [15] is used.

The different layers described so far have – at least up to some point – already been realised – for all building blocks there exist standards. The higher layers we discuss now are either not realised yet or the community has not yet agreed on a solution (or even on a concrete definition of the problem).

Unifying Logic Above these concepts of querying, ontologies and rule-based reasoning, there is the layer of the *Unifying Logic*: a logical framework which connects the other concepts and makes it possible interoperate between them. The logic should thus support the inference mechanisms of these three underlying concepts and the formats below, in particular RDF.

Proof Once this *Unifying Logic* is found, it should be possible to provide formal proofs for the derivations done using this logic (and thereby also for all derivations of the underlying standards). These proofs should be exchangeable by different parties and it should be possible to automatically check their correctness. Additional information like for example the source of knowledge used could also form part of these proofs.

Cryptography The cryptography layer lies aside of the layers discussed so far. For all the resources and at all layers of the Semantic Web cryptographic techniques should be used to for example verify the identity of an agent or to implement access control mechanisms. Here, existing Web technologies and protocols like for example RSA [29] or HTTPS [30], but also newer techniques, like blockchain, could be used.

Trust The layers proof and cryptography together form the base for the trust layer: only if reasoning steps, sources of information and the trustworthiness of all parties involved can be verified and manipulation can be excluded, people and machines can trust the information they get from the Semantic Web.

User interface and applications This highest layer of the stack is also the broadest: a Semantic Web only makes sense if it is more than a scientific construct. It needs to be used by humans but also by different machines which, following the original vision, should be able to interact between each other and make use of the resources available. Of course there are already applications making use of the Semantic Web implemented, but together with the progress of the Semantic Web as a whole, we expect a lot more to come.

One detail which often leads to discussion[3] is that the current stack explicitly names XML [21] as a base for the syntax. In practice many applications rather use formats like the JSON-based JSON-LD [31] or Turtle [23], which is also easy to read for humans, to represent RDF[4]. This is certainly a valid criticism. However, in this thesis we see the formats occurring in the stack as open lists of formalisms which have already been standardized to serve the purpose mentioned rather than a simple declaration of the decision in favour of a format. To emphasize this, we added to every layer which is labelled with a standard also the general purpose this layer fulfils. These names are not always present in the original stack our figure is based on.

[3] See for example: https://lists.w3.org/Archives/Public/semantic-web/2016Feb/0121.html

[4] Even though in the case of Turtle this point is somehow addressed as the RDF layer directly touches the layer of identifiers which are directly used by Turtle.

2.2 RDF as the Base of the Logical Building Blocks

While the standards proposed for the other building blocks of the Semantic Web stack are subject to discussion – some more than others as the example of XML shows – RDF as the standard used for data interchange is widely accepted. The Linked Open Data Cloud (LOD Cloud) [32] – at the time of writing (November 2018) consisting of 1,229 datasets each containing at least 1,000 RDF triples – is one indicator for that, the various papers using RDF papers published at different conferences, e.g. [33, 34], are another. Given this strong position of RDF as the proposed standard for data interchange, the logical building blocks *Querying*, *Ontologies/Taxonomies* and *Rules* and – as a consequence – also the *Unifying Logic* need to connect to RDF. This connection can be on syntactic level – other standards can use RDF's syntax of triples – but to really realise a *Semantic* Web such a connection needs to be stronger: RDF has its own formally defined semantics [35] and other logical concepts relying on it need to take that fact into account. We now discuss the relation of RDF and the different standards listed in the Semantic Web Stack in detail.

2.2.1 RDF and SPARQL

The query language SPARQL [15] treats RDF as a data structure in which can be searched for particular patterns. In that sense, SPARQL only makes use of the syntax of RDF and is agnostic about its specific semantics. However, the query semantics of SPARQL is compatible with RDF and the specification even states, how SPARQL can be aligned with different entailment regimes including simple RDF entailment. The syntax of SPARQL is defined on top of RDF and uses RDF's triple structure to express the graph pattern to search for. There exists an approach to fully express SPARQL queries in RDF, the SPARQL Inferencing Notation (SPIN) [36]. While this notation is syntactically compatible with RDF, there are several constructs which make the integration of SPIN on a semantic level problematic: SPIN represents SPARQL queries as blank nodes whose different properties are then specified in RDF triples. The problem with this notation is that these blank nodes do not simply represent anonymous objects in RDF but implicitly existentially quantified variables. This means that, by using SPIN notation, a user just states that a query *exists*. This is something different than writing down a

query which one wants to execute.[5] Similar problems arise from the use of blank nodes to represent other concepts like for example the search variables of a query.

2.2.2 RDF and RDFS

RDF Schema (RDFS) [24] was developed together with RDF and extends this rather basic logic by several predicates having a fixed meaning such as for example `rdfs:range` and `rdfs:domain` to specify the domain and range of a predicate or `rdfs:subClassOf` to express hierarchies between different classes. The semantics of RDFS is defined in the same official document [35] RDF's is and therefore it is not surprising that these two standards are fully compatible on both levels, syntax and semantics. RDFS thus truly extends RDF.

2.2.3 RDF and OWL

For the ontology layer the Semantic Web Stack proposes the Web Ontology Language (OWL) [37] which has evolved further to OWL 2 [25]. On the syntactic level OWL 2 is compatible with RDF. There are different syntaxes defined for OWL but the RDF/XML syntax is mandatory for all systems supporting OWL 2 and other syntaxes can be mapped to that [38]. For the semantics the situation is more complicated: There are two different semantics for OWL 2. The *direct semantics* [39] for the structural syntax of OWL 2 [40] is purely based on Description Logics [26] a subset of first order logic which is known to be decidable. This flavour of OWL 2 is also called *OWL DL* and there are several reasoners for it and its subprofiles OWL 2 RL, OWL 2 QL and OWL 2 EL [41] like for example Pellet [42], HermiT [43] or RDFox [44]. Even though it can be represented using RDF *syntax*, OWL DL is not *semantically compatible* with RDF and does for example not interpret blank nodes as existentially quantified variables. To provide a semantic connection between OWL and RDF, the *RDF-based semantics* [45] for the RDF-representation of OWL 2 was created. The logic supported by this semantics is also called *OWL Full*. OWL Full is not decidable and there are no reasoners implemented to fully support this logic.

[5] To understand this difference, consider the two sentences: (1) *"The moon is made of cheese."* and (2) *"There exists the statement that the moon is made of cheese."* The person using sentence 1 expresses his believe that the moon is made of cheese while by stating sentence 2 nothing is said about the believe of the speaker concerning the moon.

2.2.4 RDF and RIF

For the rule layer, the Semantic Web Stack lists the Rule Interchange Format (RIF) [28]. While RDFS, OWL and SPARQL are concrete logics disposing over their own semantics, RIF [28] is a format to *interchange* rules. Being designed for that purpose, RIF supports different paradigms with sometimes conflicting model-theoretic semantics (e.g. well-founded vs stable semantics) or even with no model-theoretic semantics at all: For the RIF Production Rule Dialect (RIF-PRD) [46] only operational semantics is defined. As a connection between these formats, RIF offers RIF Core [47], a minimal rule language supporting common features of rule languages which can be extended to define the semantics of concrete languages. The syntax of RIF is based directly on XML but can be mapped to an RDF representation [48]. As it was the case with SPIN, this representation can be used for data exchange but does not really combine both logics: blank nodes are used to express RIF formulas and variables. RDF interprets these as existentially quantified. The semantics of RIF Core and its different extensions is defined independently of RDF. The standard also specifies a way to semantically integrate RDF to RIF [49] but to apply rules on RDF data, this data needs to be translated and there are only very few reasoners which perform that translation.[6]

2.3 Connecting the Logical Building Blocks

On top of the logical building blocks *Querying*, *Ontology/Taxonomy* and *Rules* the Semantic Web Stack foresees a connecting layer: the *Unifying Logic*. Such a connection can, for example, only support the exchange of the results obtained by the different blocks [17] or semantically combine Description Logics and rule-based reasoning in one logic [50–54]. Especially this last approach normally deals with two major challenges: the support of a *closed vs open world assumption* and *decidability*. We discuss both topics in detail.

2.3.1 Open vs. closed world

Classical database languages (e.g. SQL) but also many rule languages (e.g. Prolog) support the *closed world* assumption which basically says that if a statement cannot be derived, it is considered to be false (negation as failure, NAF). While such an assumption comes in handy when we have one single

[6] From the different engines listed at https://www.w3.org/2005/rules/wiki/Implementations only FuXi, riftr and EYE claim to support RDF, but for the latter it is clearly stated that the RIF syntax first needs to be translated into the native syntax of EYE by an external tool.

database which can be completely searched by applying a query, it is problematic in the environment of the Web: the Web is decentralised and it is impossible to know whether or not something is stated to be true *somewhere* in the Web. For that reason RDF, RDFS and OWL are defined with the *open world assumption*. No conclusions can be taken by the fact that something cannot be derived. In contrast to that, the query language SPARQL supports a form of negation as failure. The filter `NOT EXISTS` retrieves a positive result if no match can be found for a given pattern [15, Section 8]. For rule languages, it is more complicated: RIF-Core [47] does not cover negation and it depends on the designer of a RIF dialect whether and how this concept is handled. From the dialects specified in the W3C recommendation, only RIF-PRD [46] supports the closed world assumption.

But even these cases of negation can be covered by open world concepts: As a SPARQL query is always executed on one or more specific databases and rule languages rely on facts and formulas which are explicitly stated somewhere, we can use the concept of *scoped negation as failure* (SNAF) [17, 55, 56]. SNAF makes use of the sources (the scope) which where taken into account when executing a query or performing rule-based reasoning. Instead of simply stating that something cannot be proven such as it is done when applying NAF, in SNAF we draw conclusions from the fact that something cannot be derived from a *fixed dataset* which we explicitly name. So, even if we find out later that somewhere in the Semantic Web a fact x is declared to be true, the statement that we cannot derive that fact x from a fixed knowledge base K keeps being valid.

2.3.2 Decidability

Description Logics [26] in general, and with them OWL DL [39] in particular, are decidable: For every statement expressed in the language of these logics there exists an effective method to decide whether or not this statement is consequence of a set of valid formulas. This is not the case for classical First Order Logic (FOL) and one of the main goals driving the development of Description Logics was to cover as much of the features of FOL as possible while still guaranteeing decidability. Therefore it is not surprising that many approaches of finding a *Unifying Logic* mainly focus on decidability (e.g. [52, 57]). While this focus forms a very interesting research problem – especially when considering the fact that simply combining decidable rule-based reasoning with description logics does not lead to decidability [58] – it is of rather theoretical nature when searching for a *Unifying Logic*: OWL Full, the semantic combination of RDF and OWL DL is already undecidable which is the reason why no decidable combination of rule-based reasoning

and description logics can cover these two important building blocks of the Semantic Web. The quest for a *Unifying Logic* should therefore rather focus on ways how information specified in OWL, RDF and rules can be used to draw conclusions and accept that we cannot know everything (which is not only a problem for Semantic Web technologies but also something we all deal with in our daily life).

2.3.3 Different Approaches to find a Unifying Logic

Having discussed the two most important aspects previous research on the *Unifying Logic* has focussed on, we now give a more detailed overview of that work and related approaches which aim to combine (some of) the building blocks of the logical layer.

The relationship of querying and rules has been investigated in several works. By defining mappings between the frameworks, Angles and Gutierrez [59] show that non-recursive safe Datalog with negation has the same expressive power as SPARQL. Similarly, Polleres and Wallner show that SPARQL querying can be implemented with answer set programming [60]. This work shows that the building blocks rules and querying of the Semantic Web stack are closely related and that when finding a combination of description logic-based reasoning and a rule language which covers one of the two approaches this combination also covers querying. We therefore next discuss these approaches.

An early approach to combine rule-based reasoning with OWL DL-based reasoning has been performed by Grosof et al. [50] who introduced Description Logic Programs (DLP). Description Logic Programs can be understood as the intersection of Description Logic and Horn Logic Programs. As such, this logic is decidable and does not support NAF. DLP can be expressed by means of description logics as well as by using Horn rules. The paper furthermore details, how a mapping from the former to the latter can be performed while the opposite direction is only briefly explained. In that sense it provides a way of performing OWL DL reasoning by means of rules and can be seen as a predecessor the OWL 2 RL profile [41]. The paper mainly provides a theoretical underpinning of the logic which later has been applied in different implementation among which KAON2 [61] and OWLim [62]. A criticism which was often raised (for example by Parsia et al. in [63]) is that DLP are very limited in expressiveness. In practice DLP are mostly implemented in rule-based frameworks (KAON2 is for example based on Datalog [64]) which makes it harder to extend the coverage towards description logics than towards rule logics. However, newer research has shown that by extending the

expressiveness of rules, for example by allowing existential quantification, bigger parts of OWL DL can be covered [65, 66].

The Semantic Web Rule Language (SWRL) [27] was developed as the union of rules and description logic. The semantics of SWRL is defined as an extension of OWL DL. In that sense, SWRL can be seen as an approach towards defining the *Unifying Logic* starting at the description logic point of view. As a very important aspect of description logics is its decidability, most SWRL implementations only support dl-safe rules [67], i.e. the share of SWRL rules which can be combined with OWL DL without loosing decidability. From a practical point of view the close relation of SWRL and OWL DL can form a burden: SWRL inherits for example the difference between datatype properties and object properties from OWL DL which can be confusing for the user; if SWRL is integrated in OWL DL reasoners, the execution of rules is not always optimised. This makes it difficult to use SWRL for rule-based reasoning.

Next to the approaches mentioned above which mainly combine Datalog reasoning with OWL DL, there are also approaches which go further in terms of the expressivity of the Rule language. Different aspects such as the above mentioned support of negation as failure or also whether or not the rule language follows the unique name assumption need to be taken into account [68]. One approach of combining non-monotonic rule-based reasoning with OWL DL reasoning has been performed by Knorr et al. [51]. In their work, the authors show how so called nominal schemas (introduced in [52]) can be extended to support different approaches for closed world reasoning.

Before we finish this section we also want to have a look at a rather theoretical work. De Bruijn et al. [69] define how RDFS, OWL DL and SPARQL can be jointly represented in first order logic. With that approach, the authors facilitate the research on the properties of these frameworks and possibly other logics. In this context, first order logic can be seen as a *Unifying Logic*. While the use of FOL makes it easy to compare and understand the frameworks maps to – FOL and its properties are well studied – this approach cannot be used on practical level. Even though there exist many reasoners for FOL (see for example the theorem provers participating in CADE[7]) these are known to not be very performant. FOL furthermore supports many constructs which are not present in any of the considered standards and its syntax does not natively support linked data.

Most of the work we described in this section primary focusses on the theoretical aspects of the *Unifying Logic* (which does not exclude that there are implementations), but to the best of our knowledge there is not much work done which focusses on the connection of the possible candidates to the

[7] http://www.tptp.org/CASC/

upper layers of the Semantic Web Stack: Can the (possible) *Unifying Logic* be used in practical applications? How well does it support the layer of proof? Do the proofs connect to the layer of applications as well? As we will further explain in the following section, we consider these aspects as very relevant in order to fully realise the vision of the Semantic Web and make the content of the Web understandable for computers.

2.4 Requirements on a Unifying Logic

Before we continue and discuss the research questions this thesis aims to answer, we briefly discuss the aspects of a *Unifying Logic* we want to focus on. The *Unifying Logic* for the Semantic Web should fulfil the following requirements:

1. **Clear semantics** The semantics of the *Unifying Logic* should be formally defined to avoid ambiguities.
2. **Compatibility with RDF** The *Unifying Logic* should be *syntactically* and *semantically* compatible with RDF.
3. **Connection of the logical building blocks:** The *Unifying Logic* should facilitate the practical tasks the building blocks *Querying*, *Taxonomies/Ontologies* and *Rules* aim to solve and it should support reasoning on the RDF representation of the standards listed in the current stack (i.e. RIF, OWL and SPARQL).
4. **Support of proofs:** It should be possible to formally express proofs for derivations done in the *Unifying Logic*. The proof format should be based on RDF, it should be exchangeable and it should be usable in practical applications which go beyond their primary purpose of establishing trust.

We further explain these points below starting with the two aspects discussed in the previous sections: A *Unifying Logic* must be compatible with RDF and it should connect the layers it directly relies on. We have already seen that the concept of *connection* can be understood in different ways. One of these is the creation of a decidable logic which supports DL reasoning and rule-based inferencing at the same time. Since this solution excludes OWL Full, and with it also RDF, we do not follow this approach here and focus on practical aspects instead: we search for a logic which supports the different practical applications the layers *Querying*, *Ontologies/Taxonomies* and *Rules* aim to facilitate and combines them in one framework. This logic

should be semantically and syntactically compatible with RDF and thereby also allow the user to directly operate on the syntactic RDF-representations of the standards listed in the Semantic Web stack. It should furthermore have a clearly defined semantics. While this is a basic requirement every logic needs to fulfil regardless of the role it is designed for, this property has a special importance for a Semantics Web logic: One key concept of the Semantic Web is interoperability. This means that different systems exchanging information and applying the logic need to have a common agreement about the meaning of formulas.

Another important aspect of a *Unifying Logic* is its connection to the layers on top of it: By emphasizing the usability for practical applications, we already made a connection to the layer *User Interface and Applications* but the connection to the layers in between, in particular to the next higher layer *Proofs*, has not been discussed so far. It needs to be possible to formally express how conclusions are drawn applying the *Unifying Logic*. For this, a calculus must be defined and its correctness needs to be proven. The proofs composed of the different inference steps of the calculus need to be exchangeable between different parties and these need to be able to verify them. With RDF being the most established standard of the Semantic Web, the proof layer should be compatible with it. Since the implementation of the Semantic Web stack with its different building blocks is not a scientific goal in itself but aims to realise the vision of a Semantic Web, the connection of all the building blocks occurring in the stack with the upper layer, the layer of applications, is crucial: only if the different standards support real life applications they push the realisation of the Semantic Web forward. We therefore also expect from the proof format based on the *Unifying Logic* that it can be used in different scenarios which preferably go beyond the establishment of trust.

2.5 Notation3 Logic

Having discussed the architecture of the Semantic Web and the role the *Unifying Logic* plays in it, we now introduce a possible candidate to fulfil this role, Notation3 Logic (N3Logic) [5]. The aim of this section is to make the reader familiar with the different features of this logic and to illustrate its connection to RDF. A more formal specification of N3Logic is given later in this thesis.

2.5.1 Simple triples and conjunctions

N3Logic is an extension of RDF. All RDF turtle triples [23] are also valid in N3. Simple statements can be expressed in triples of the form *subject*, *predicate* and *object*, such as:

```
:Kurt :knows :Albert.
```
(2.1)

This triple means "*Kurt knows Albert*". Each ground component in N3 is represented by either an Internationalized Resource Identifier (IRI) [20], as done here, or by a literal [4, Section 3.3]. In N3Logic, literals can occur in all positions of a triple. In this aspect N3 differs from RDF which only allows literals in object position. In the example the IRI is abbreviated [23] with the empty prefix which here and in the remainder of this thesis refers to an example namespace:

```
@prefix :  <http://example.org/ex#>.
```

For instance, `:Kurt` as it occurs in triple 2.1 stands for the full IRI

```
<http://example.org/ex#Kurt>
```

If two triples occur together as in:

```
:Kurt :knows :Albert. :Albert :knows :Kurt.
```
(2.2)

This is understood as the conjunction of the formulas: *"Kurt knows Albert* and *Albert knows Kurt."* If two triples share the same subject, the conjunction can be abbreviated using a semicolon "`;`".

```
:Kurt :knows :Albert; :knows John.
```

means: *"Kurt knows Albert* and *Kurt knows John."*

If two triples share the same subject and predicate, the comma "`,`" can be used to express a conjunction. The formula

```
:Kurt :knows :Albert, John.
```

has the exact same meaning as the previous one.

The prefix notation, the treatment of triples occurring together as their conjunction and the two ways of abbreviating such a conjunction are equally supported by RDF.

2.5.2 Citations and Implications

The formulas shown above, but also more complex formulas, can be cited using curly brackets. The statement

```
:John :says {:Kurt :knows :Albert.}.
```
(2.3)

means "*John says that Kurt knows Albert.*" The same notation of curly brackets is used, if one formula implies another, the implication symbol is expressed by "=>". The formula

```
{:Kurt :knows :Albert.} =>
                        {:Albert :knows :Kurt.}.
```
(2.4)

represents the rule "*If Kurt knows Albert then Albert knows Kurt.*"

Citations and implications as presented above are not supported by RDF.

2.5.3 Variables

Next to literals and IRIs, N3 supports the usage of quantified variables. The exact quantifier for these variables is not stated explicitly but it is implicitly assumed. Therefore, we distinguish between two kinds of variables, *existential variables* (short: *existentials*), also called *blank nodes*, and *universal variables* (short: *universals*). We give examples for the usage of both of them.

Existential variables or blank nodes are present in N3 as well as in RDF[8] and stand for an existentially quantified variable. They are either represented by a string starting with "`_:`" or they are expressed using square brackets "`[ ]`". The formula

```
_:x :knows :Albert.
```
(2.5)

means "*There exists someone who knows Albert.*" In the formula

```
:John :knows [ :knows :Albert ].
```
(2.6)

the bracket stands for a "fresh" existentially quantified variable. The formula means "*John knows someone who knows Albert.*" While RDF allows the usage of blank nodes only in subject or object position, they can be subject, predicate or object in N3.

[8] Here and later on in this thesis we always represent RDF using Turtle-syntax [23]. Other syntaxes for RDF like JSON-LD [31] or XML syntax [70] represent blank nodes differently.

Universal variables are an extension N3 makes to RDF. They start with a question mark `?` and are understood as implicitly universally quantified. The formula

```
{:Kurt :knows ?x.} => {?x :knows :Kurt.}.
```
(2.7)

means "*Everyone Kurt knows also knows Kurt.*"

Both kinds of variables can also be combined: When universals and existentials occur together in the same formula the scope of the universals is always outside the scope of the existentials [71].

```
_:x :thinks {?y :is :pretty}.
```
(2.8)

is interpreted as

$$\forall y \exists x : \text{thinks}(x, \text{is}(y, \text{pretty})) \tag{2.8a}$$

"*For everyone there is someone who thinks that he/she is pretty*"

and not as

$$\exists x \forall y : \text{thinks}(x, \text{is}(y, \text{pretty})) \tag{2.8b}$$

"*There is someone who thinks that everyone is pretty.*"

2.5.4 Proofs

By providing the option to cite formulas, N3 also makes it possible to express proofs referring to the different rules which – applied to the statements of the dataset at hand – produce new knowledge. For that purpose the N3 proof vocabulary[9] was created in the context of the Semantic Web Application Platform (SWAP) [72].

To illustrate how proofs can be expressed using this vocabulary, we first take a closer look to a possible proof step. Getting back to Formulas 2.1 and 2.4 and their meaning as discussed above, we can apply the the rule from Formula 2.4 on the fact in Formula 2.1 and derive the conclusion:

```
:Albert :knows :Kurt.
```
(2.9)

This step is a classical modus ponens as it can be found in most text books about mathematical logic [73–75]. This derivation is also valid in N3.

[9] https://www.w3.org/2000/10/swap/reason#

```
1 PREFIX : <http://example.org/ex#>
2 PREFIX r:  <http://www.w3.org/2000/10/swap/reason#>
3
4 <#lemma3> a r:Inference;
5  r:gives {:Albert :knows :Kurt.};
6  r:evidence (<#lemma2>);
7  r:rule <#lemma1>.
```

Listing 1: Example inference step. Formula 2.9 gets derived by applying the rule in Formula 2.4 (lemma 1) on Formula 2.1 (lemma 2).

In Listing 1 we display how this step can be expressed in the proof vocabulary. The vocabulary refers to a generalised version of the modus ponens by the name `r:Inference`. For every proof step, the vocabulary provides the option to refer to its result using the predicate `r:gives`. In the example, the instance `<#Lemma3>` of the proof step `r:Inference` (Line 4) yields Formula 2.9 (Line 5). The vocabulary also makes it possible to refer to the different formulas the proof step was applied to. For the inference step, the predicate `r:evidence` refers to the facts taken into account and the predicate `r:rule` to the rule that was used. In our example these formulas are themselves consequences of the proof steps `<#Lemma2>` (Line 6) and `<#Lemma1>` (Line 7). These two steps here have been the simple parsing of Formula 2.1 and Formula 2.4, respectively.

A combination of different proof steps form a proof. Proof steps covered by the vocabulary are generalised modus ponens (`r:Inference`), the parsing of an axiom (`r:Parsing`), conjunction introduction (`r:Conjunction`), conjunction elimination (`r:Extraction`) and the step `r:Proof` which indicates what exactly has been verified by a proof. As the example already indicates, all these proof steps for N3 derivations can be stated using the logic itself.

2.5.5 Notation3 in practice

Having introduced the different features of N3 as a logic, we now briefly discuss how N3 can be and is used in practise.

There are various reasoners which support N3 reasoning, like Cwm [76], FuXi [77], swish [78] or EYE [79]. They follow different reasoning strategies – Cwm for example supports only forward-chaining, while with EYE the user can decide for each rule whether it should be applied in a forward or backward manner – and are implemented in different programming languages – swish for example in Haskell and FuXi in Python. But since all reasoners support the same logic, they are all suited for a broad variety of practical

implementations. N3 rules are used in healthcare [80, 81], in the construction domain [82] and in the context of the internet of things [83] to only name a few areas.

Provided with different facts and rules written in N3, reasoners can either produce the deductive closure of this input knowledge – this is similar to how OWL DL reasoners like for example Pellet [42] or Hermit [43] work – or they can provide the answer to a specific query – this is in some aspects similar to the idea of ontology-based data access (OBDA) [84] where queries are rewritten and only the ontology knowledge relevant for a certain query is taken into account. In the case of N3 reasoning, the query is a simple N3 rule which only differs from other rules by the simple details that it is marked as a query. We illustrate that by getting back to our previous example. Let us assume, that the reasoner it provided with the two formulas we already used in our proof step, namely Formula 2.1 and Formula 2.4. A meaningful query for this very small knowledge base is:

```
{?x :knows ?y}=>{?x :knows ?y}
```
(2.10)

At first sight this rule might look confusing, given that its antecedent equals its consequent. But as it is given to the reasoner as a query, the reasoner will only output all ground instances of the formula

```
?x :knows ?y.
```

which can be concluded from the given knowledge. In our example these are the Formulas 2.1 and 2.9.

2.5.6 Open challenges

Many of the different properties we discussed so far make N3 a promising candidate to become the *Unifying Logic* of the Semantic Web: N3 truly extends RDF on both the semantic and the syntactic level and it contains a construct to cite formulas. This makes it possible to easily express proofs in N3 and there is already a specific vocabulary defined to do so. The fact that there are several reasoners implemented for N3 and that these reasoners are already used in practical implementations furthermore shows that N3Logic is more then just a theoretical construct: N3 actually connects to the highest level of the Semantic Web stack, the level of application. But there are also open challenges related to N3 which we briefly discuss in this section.

The first open problem we discuss here is that the semantics of N3 lacks a formal definition. Instead, there are only websites [71, 85] and a paper [5] explaining in an informal way what the different constructs present in N3

are supposed to mean. Next to the obvious problems resulting from this lack of formalisation – without a proper definition of N3's semantics the concept of correctness is not even defined and formal properties like its expressivity compared to other frameworks cannot be studied – there are also more practical consequences: Since some informal explanations are rather vague, the different reasoners mentioned above assume different interpretations. As a consequence, the reasoning result of these reasoners can differ when provided with the same input. While a situation like this would be a problem for any logic, it is in particular a problem for a logic in the Semantic Web, where interoperability is key.

Related to the problem we just described, there is another one: it is not clear yet, how N3 logic behaves in comparison to other logics. We already mentioned the expressivity which is related to the semantics. Apart from this we want to know for which kinds of tasks N3Logic is well suited and when it is better to use another formalism. Having the idea of a *Unifying Logic* in mind, it would in particular be interesting to know whether common SPARQL querying tasks and the reasoning tasks which are classically solved using OWL DL can also be addressed by applying N3 reasoning.

2.6 Research questions

In this thesis we want to investigate to what extend N3Logic can fulfil the role of the *Unifying Logic* for the Semantic Web. In Section 2.4 we identified the four requirements on the *Unifying Logic* we want to focus on: (1) clear semantics, (2) compatibility with RDF, (3) connection of the logical layers and (4) support of proofs. For each of these requirements we pose research questions.

We start with the semantics of N3 (Requirement 1). As discussed above, there is no clear formal definition of N3's semantics and existing N3 reasoners differ in their understanding of the informal explanation provided by the W3C team submission [71] and the journal paper introducing this logic [5]. This situation cannot be solved by simply formalising our own understanding of N3Logic since in the context of the Semantic Web we aim for interoperability and to reach that we need a common agreement. It therefore makes sense to first understand the existing problems and provide a way to discuss them. This leads us to our first research question:

Research Question 1: *How can the differences between the individual interpretations of N3 assumed by different reasoners following the W3C team submission [71] and the journal paper [5] be formally expressed?*

Most disagreements in the interpretations of N3 formulas are related to the concept of implicit quantification. In N3 quantified formulas can be written without explicitly stating the position of the quantifier as exemplified in Formula 2.8. In that example one of the two possible interpretations we discussed could be identified as the correct one by consulting the W3C team submission. If implicit quantification is used in a more complex graph structure, the sources are less explicit about the intended meaning of a formula, we therefore test the hypothesis:

Hypothesis 1: *Existing interpretations differ in their understanding of implicit quantification and this difference can be expressed by mapping formulas containing such constructs to a core logic which only supports explicit quantification.*

Being aware of this difference, we next want to know whether we deal with a theoretical problem which only can be observed by thoroughly testing the reasoners – then the theory needed to be fixed to avoid future problems – or whether we can observe differences in the interpretations of N3 rules used in practical applications. In that case, the problem already caused harm. We therefore ask our second research question:

Research Question 2: *How big is the impact of having different interpretations for N3 formulas in practice?*

To be able to answer this question, we take a closer look at such N3 formulas used in practical applications. We take a set consisting of formulas used in different industry-related research projects and of example formulas from the EYE website which were created to illustrate such use cases and test the following hypothesis:

Hypothesis 2: *For at least one quarter of all N3 formulas contained in our dataset of practical examples, the interpretations by the reasoners EYE and Cwm differ.*

Once the differences in the interpretations can be clearly identified, the community needs to agree on *one* interpretation for N3Logic which then needs to be formalised. In this context we need to remember that we postulated that the *Unifying Logic* needs to be compatible with RDF (Requirement 2). For the syntactic level, we already know that N3 is compatible with RDF – N3 only adds symbols and constructs to the existing RDF syntax – for the semantic level we need to answer the following research question:

Research Question 3 *How can we define the semantics of N3Logic in a way which is compatible to RDF?*

To answer this research question, we need to take a closer look to the semantics of RDF [35]. This semantics is defined following a model theoretic

approach. In order to stay consistent with these definitions, we need to follow the same approach when defining the semantics of N3 according to the interpretations of Cwm and EYE. Here, we test the following hypothesis:

Hypothesis 3: *The model theory for N3Logic can be specified in such a way that it is compatible with the model theory of RDF with the only exception that RDF blank nodes need to refer to named instances in the domain of discourse.*

Next to a clear semantics and compatibility with RDF, the *Unifying Logic* needs to connect the logical layers of the Semantic Web stack (Requirement 3). In this context we discussed that this connection can be established in different ways including the definition of a decidable logic which covers as much of the three building blocks as possible – knowing that a full coverage can only be established by giving up decidability – or by focussing on the different applications connected to the building blocks. Having the different options in mind, we ask our fourth research question:

Research Question 4: *In which aspects can Notation3 Logic cover and connect the building blocks* Querying*,* Ontologies/Taxonomies *and* Rules *of the Semantic Web stack?*

Being a rule-based logic, we already know that N3Logic fully covers the building block *Rules*. We furthermore know that if N3 can cover the other two building blocks, it also covers their combinations. We therefore test:

Hypothesis 4: *With N3Logic we can tackle the same use cases as with OWL DL reasoning and/or SPARQL querying without suffering a loss of performance in terms of execution times or correctness of the result.*

The last important property we identified for the *Unifying Logic* was that it should support the layer of *Proof* (Requirement 4). There exists a vocabulary in N3 which was created to express proofs, the SWAP-proof vocabulary [86]. This vocabulary offers the option to exchange proofs between different reasoning engines, check them for correctness and thereby establish the layer of *Trust*. But without a definition of the semantics of N3 this concept of correctness is also not well-defined. We therefore ask:

Research Question 5: *How can we verify that the proof steps defined by the SWAP vocabulary are correct?*

In order to answer this question we need a model theory for N3 as it was already required to answer the previous research questions. Once we have that we can verify the hypothesis:

Hypothesis 5: *The proof steps included in the SWAP vocabulary can be formally defined on top of the model theory for N3. That formalisation allows us to prove that the calculus is correct.*

With such a correct proof calculus which can be used within the language – proofs written with SWAP are themselves again N3 formulas – we can not only support the layer of *Trust* but also the layer of *Applications*. We illustrate that on the idea of using N3 proofs for adaptive planning.

Chapter 3

Implicit Quantification

The semantics of N3Logic is only defined in an informal way by the W3C team submission [71] and a journal paper [5]. This leaves room for interpretation and it actually leads to contradicting implementations. In this chapter, we take a closer look at this problem and its possible solutions. We first collect evidence by giving concrete example formulas which, when given to the reasoners Cwm [76] and EYE [79], lead to different results. The discrepancies we encounter are mainly related to implicit quantification: N3 allows the use of quantified variables whose universal or existential quantifier is not explicitly stated but implicitly assumed. The problem can be solved by providing a formalisation which clearly indicates how implicit quantification needs to be interpreted. To better understand how such a formalisation could look like we discuss different logics supporting some kind of implicit quantification, in particular RDF and other formats related to the Semantic Web, and discuss in which aspects they differ from N3.

3.1 Quantification in N3 Logic

The example formulas we discussed in Section 2.5 were rather simple and understanding their intended meaning was not difficult (maybe with the exception of cited formulas which can lead to discussions [87]). For the only case where two different interpretations were plausible – Formula 2.8 which had universals and existentials occurring together – the W3C team submission [71] contains a clear statement which interpretation needs to be chosen:

> *"If both universal and existential quantification are specified for the same formula, then the scope of the universal quantification is outside the scope of the existentials".* (I)

Unfortunately, not all cases are that clear. When implicitly quantified variables occur in deeply nested formulas, their intended meaning is not always obvious and the interpretations of such formulas sometimes differ between reasoning engines. In this section we want to better understand these differences. With this goal, we perform several tests on the reasoners Cwm [76] and EYE [88] and compare their results. Cwm and EYE were chosen because they cover most constructs specified in N3's W3C team submission. In contrast to for example FuXi [77], they both support rather complex constructs like nested rules.

3.1.1 Existentials

We start our considerations by taking a closer look at implicit existential quantification in nested formulas, i.e. formulas where the blank node occurs within curly brackets { }. Such constructs are typically used to cite formulas or to state rules. We take the following example for the latter:

```
_:x :says {_:x :knows :Albert.}.
```
(3.1)

This triple is interesting because it contains the blank node `_:x` at two positions: as a subject of the triple and inside a nested formula. We already know that a blank node represents an implicitly existentially quantified variable, but what we do not know yet is: what is the exact position of its quantifier and what is the scope of the variable? Depending on the answer, the two occurrences of `_:x` could either refer to the same instance of the domain of discourse, then the formula means

$$\exists x : \text{says}(x, \text{knows}(x, \text{Albert})) \tag{3.1a}$$

"There exists someone who says about himself that he knows Albert."

or the `_:x` is always quantified in the direct formula it occurs in (i.e. the brackets { }), then the two existentials can refer to different objects and we get[1]

$$\exists x_1 : \text{says}(x_1, (\exists x_2 : \text{knows}(x_2, \text{Albert}))) \tag{3.1b}$$

"There exists someone who says that there exists someone (possibly someone else) who knows Albert."

```
1 @prefix :  <http://example.org/ex#>.
2
3 [ :says { [ :knows :Albert ]. } ].
```

Listing 2: Reasoning result of cwm for Formula 3.1. The two occurrences of the variable `_:x` get translated to two blank nodes in square bracket notation.

```
1 PREFIX : <http://example.org/ex#>
2
3 _:t_0 :says {_:t_1 :knows :Albert}.
```

Listing 3: Reasoning result of EYE for Formula 3.1. The two occurrences of the variable `_:x` get translated to two different blank nodes.

As explained earlier in Section 2.5.5, N3 reasoners can be invoked to output the deductive closure of their input files. If we provide Cwm and EYE with Formula 3.1 and let them derive the deductive closure we get the result displayed in Listings 2 and 3, respectively.[2] We see, that Cwm uses the bracket notation `[ ]` for blank nodes where, as explained in Section 2.5.3, every new bracket refers to a fresh new blank node; EYE gives them different names (`_:t_1` and `_:t_2`). This behaviour shows, that both reasoners understand the two occurrences of `_:x` as different blank nodes which results in Interpretation 3.1b.[3]

This example shows what is meant by the following quote from the W3C team submission [71]:

> *"When formulae are nested, _: blank nodes syntax* [is] *used to only identify blank node in the formula it occurs directly in. It is an arbitrary temporary name for a symbol which is existentially quantified within the current formula (not the whole file). They can only be used within a single formula, and not within nested formulae."* (II)

In other words: Existential variables are always quantified in the direct

[1] Note that Formula 3.1a implies 3.1b but not the other way around.

[2] Unless indicated otherwise we use in this thesis the following versions of the reasoners: Cwm v 1.197 2007/12/13 15:38:39 syosi and EYE v18.0312.0936

[3] Strictly speaking, this example still does not make sure that the reasoners really assume Interpretation 3.1b since both blank nodes could be quantified on top level: $\exists x_1 : \exists x_2 : \text{says}(x_1, (\text{knows}(x_2, \text{Albert})))$. To exclude this possibility the reasoners can be tested with Triple 2.1 and the rule `{_:x :knows :Albert.} => {:Albert :is :known.}.`: only if the quantifier for the blank node is in the antecedent of the rule, this input results in `:Albert :is :known.` This is the case for both reasoners.

formula – marked by curly brackets `{ }` – they occur in. This quantifier does always only count for that formula and not for the nested formulas depending on it.

3.1.2 Universals

The case of implicit existential quantification in N3 was still clear and the reasoners we tested did not disagree on the meaning of implicitly existentially quantified variables. This is different for universal quantification. A typical example for a formula containing universal quantification was given before in Formula 2.7:

```
{:Kurt :knows ?x.} => {?x :knows :Kurt.}.   (2.7)
```

It would not be very useful to handle the quantification in this equally as existential quantification. If universal quantifiers where also always quantified on their direct formula we would get the interpretation:

$$(\forall x_1 : \text{knows}(\text{Kurt}, x_1)) \rightarrow (\forall x_2 : \text{knows}(x_2, \text{Kurt})) \tag{2.7a}$$

"If everyone knows Kurt then Kurt knows everyone"

Here, we would never be able to write any rules which reason on RDF triples since this rule only gets invoked for a universal statement – "*Everyone knows Kurt.*" – but it is not possible to make universal statements in plain RDF. Therefore, the interpretation we already discussed earlier in Section 2.5.3 is correct in this case:

$$\forall x : (\text{knows}(\text{Kurt}, x)) \rightarrow (\text{knows}(x, \text{Kurt})) \tag{2.7b}$$

"Everyone Kurt knows also knows Kurt."

The quantifier for both occurrences of `?x` is in front of the whole formula. But what happens if the universal variable occurs in a deeper nested formula? Here, the W3C team submission states the following:

> *"There is a also a shorthand syntax ?x which [...] implies that x is universally quantified not in the formula but in its* ***parent formula****."* (III)

We learn that universal variables are quantified on their *parent formula*. Unfortunately neither the W3C team submission nor the journal paper about N3Logic provide us with a definition of that concept. We therefore perform some tests to better understand which formula is the parent.

Consider the following formula:

```
{{?x :p :a.} => {?x :q :b.}.} =>
                {{?x :r :c.} => {?x :s :d.}.}.   (3.2)
```

Are all `?x` the same? If not, which ones do we have to understand as equal and where are they quantified? Of the several options to interpret this formula, two seem to be most likely:

$$\forall x : ((p(x,a) \rightarrow q(x,b)) \rightarrow (r(x,c) \rightarrow s(x,d))) \tag{3.2a}$$

or

$$(\forall x_1 : p(x_1,a) \rightarrow q(x_1,b)) \rightarrow (\forall x_2 : r(x_2,c) \rightarrow s(x_2,d)) \tag{3.2b}$$

Interpretation 3.2a understands the top formula as the parent of all other formulas, the quantifier for all occurrences of `?x` is on that formula. For Interpretation 3.2b the parent of the first two occurrences of `?x` is the formula

```
{?x :p :a.} => {?x :q :b.}.   (3.3)
```

and the parent of the third and fourth occurrence of `?x` is the formula

```
{?x :r :c.} => {?x :s :d.}.   (3.4)
```

In this interpretation each occurrence of curly brackets `{ }` marks a level. The direct formula is what we encounter inside the same brackets a variable occurs in, the parent formula is then the next formula outside of these brackets. This explains, why the two sub formulas above carry universal quantifiers.

If a reasoner follows Interpretation 3.2a, Formula 3.2 applied on the following rule

```
{:e :p :a.} => {:e :q :b.}.   (3.5)
```

results in

```
{:e :r :c.} => {:e :s :d.}.   (3.6)
```

While this result cannot be derived by following Interpretation 3.2b.

If we test the reasoners Cwm and EYE with Formulas 3.2 and 3.5 as input we get the results displayed in Listing 4[4] and 5, respectively. We clearly see that Formula 3.6 is derived by EYE but not by Cwm. We furthermore see that the output of Cwm contains the key word `@forAll` at the beginning of the

[4]The predicate `log:implies` present in the original output is replaced here by implication arrow =>. The two symbols have the same meaning.

```
1 @prefix :  <http://example.org/ex#>.
2 @prefix unive:  <#> .
3 {:e :p :a.}  => {:e :q :b .}.
4
5 {@forAll unive:x . {unive:x :p :a .}=>{unive:x :q :b.}.}
6 =>
7 {@forAll unive:x . {unive:x :r :c .}=>{unive:x :s :d .}.}.
```

Listing 4: Output of cwm for for Formulas 3.2 and 3.5. The reasoner does not derive Formula 3.6.

```
1 PREFIX : <http://example.org/ex#>
2
3 {{?U_0 :p :a}=>{?U_0 :q :b}}=>{{?U_0 :r :c}=>{?U_0 :s :d}}.
4 {:e :p :a} => {:e :q :b}.
5 {:e :r :c} => {:e :s :d}.
```

Listing 5: Output of EYE for Formulas 3.2 and 3.5. The reasoner derives Formula 3.6.

antecedent and of the consequent of the main rule in Lines 5 and 7. Here, this keyword has the same meaning as the first order symbol $\forall$. Cwm follows Interpretation 3.2b. We see that Cwm and EYE understand the concept of a *parent* differently. For the remainder of this thesis we write $parent_c$ when we refer to the parent formula as implemented in Cwm and $parent_e$ for the concept of a parent realised in EYE.

While the concept of parent_e is rather simple – all variables have the same parent formula – we need to perform more tests to better understand the concept parent_c: What happens if variables occur on different levels of a formula? Take as an example:

```
{?x :q :b.} =>{ :a :b {?x :s :d.}.}.
```
(3.7)

According to the explanation above, the parent of the second occurrence of `?x` seems to be the formula `:a :b {?x :s :d.}.`, but is that also the formula which carries the quantifier for the variable? Listing 6 shows the output Cwm produces for Formula 3.7. We clearly see that there is only one universal quantifier (`@forAll`, Line 4) for both occurrences of `?x`. As a general rule we can derive from that behaviour that if the parent formula of a universal variable is already in scope of a universal quantifier this scoping also counts for its subformula.

The previous examples reveal a general problem: We needed to explain how the W3C team submission is *interpreted* by the reasoners EYE and Cwm. There are many different possibilities how the term *parent formula* could

```
1 @prefix :  <http://example.org/ex#>.
2 @prefix unive:  <#> .
3
4 @forAll unive:x.
5  {unive:x :q :b.} =>{:a :b {unive:x :s :d.}.}.
```

Listing 6: Output of Cwm for Formula 3.7 in which the variable `?x` occurs on different levels. The output formula only contains one universal quantifier for both occurences.

be understood and specific details like the interpretation of formulas with variables nested on different levels are not even mentioned in the W3C team submission. Only specific testing made us come to our conclusion that the above are EYE's and Cwm's interpretations. To be sure that the reasoners behave as intended we furthermore contacted their creators.[5] Not all users are aware of the differences in interpretations and the lack of a formalism renders it difficult to discuss or even just express them. To come to a clear and unambiguous definition of the logic, N3 needs to be formalised and there needs to be a formalism to express the differences of existing interpretations.

3.1.3 Explicit Quantification

We already saw above that apart from implicit quantification, N3Logic also provides the possibility to explicitly quantify over variables. To do so, the quantifiers `@forSome` and `@forAll` are used. With explicit quantification, Interpretation 2.8a of Formula 2.8 can be expressed as follows:

```
@forAll :y.  @forSome :x.
                      :x :thinks {:y :is :pretty}.
```

Seeing this example, the reader might think that the misunderstandings described above could be avoided by only using explicit quantification and that this notation could even be used to explain the differences. Unfortunately, that is not the case. Independently of the order they appear in the formula, universal quantifiers are always understood to be outside of existential quantifiers in N3 [71]. The formula

```
@forSome :x.  @forAll :y.
```

[5] We got clarification about Cwm's reasoning by asking on the public mailing list https://lists.w3.org/Archives/Public/public-cwm-talk/. We furthermore stood in close contact with Tim Berners-Lee with whom one of the supervisors of this thesis, Ruben Verborgh, collaborates. We thoroughly discussed the working of EYE with with Jos De Roo.

```
:x :thinks {:y :is :pretty}.
```

has the exact same meaning as the previous one, namely Interpretation 2.8a:[6] $\forall y \exists x : \text{thinks}(x, \text{is}(y, \text{pretty}))$. To express Interpretation 2.8b, $\exists x \forall y : \text{thinks}(x, \text{is}(y, \text{pretty}))$, a more complicated construction is needed. As the empty graph refers to true in N3, we can nest the universal quantifier and thereby force a different order of quantifiers:

```
@forSome :x.
 {}=>{@forAll :y.  :x :thinks {:y :is :pretty} }.
```

This last formula can be interpreted as:

$$\exists x : \text{true} \rightarrow (\forall y : \text{thinks}(x, \text{is}(y, \text{pretty})))$$

This last formula is equivalent to interpretation 2.8b. Even though, this last example shows that the described dominance of universal quantifiers over existentials does not make the logic less expressive, the construct we used looks rather artificial and not very intuitive.[7]

The peculiarity described above leads furthermore to open questions. For example, what does the following valid N3 formula mean?

```
@forSome :x.  :x :knows :y.
                    @forAll :y.  :y :knows :x.   (3.8)
```

The scope of the `@forAll` is outside of the scope of the `@forSome`, but what about the first occurrence of `:y` (underlined in the example)? Is `:y` a universally quantified variable or a constant? This formula is not supported by Cwm and its intended meaning is not specified in any source we are aware of. This and many similar examples make explicit quantification in N3 a complex topic on its own, which is outside the scope of this dissertation. Here we focus on implicit quantification.

3.2 Blank nodes in RDF

The previous section showed that the meaning of implicit quantification in N3 is not always obvious. The problems we encountered arise from the

[6] We suppose that this has historic reasons: When N3 was created, everything, including quantifiers, was represented in triples. The order of triples should not matter in any context.

[7] For that reason, the W3C community group on Notation3 is currently discussing whether this feature should be changed. See also: https://github.com/w3c/N3/issues/6. This is also in line with our own opinion.

fact that is is not easy to informally express where exactly universals and existentials are quantified. We therefore want to provide a formalisation. To better understand how the specific details of N3 and in particular implicit quantification can be formalised, we take a closer look at the logic it is extending: RDF. Here, we do not find implicit universal quantification – this is a concept that N3Logic adds to the RDF model – but implicit existential quantification is also present: as in N3 blank nodes are understood as implicitly existentially quantified variables.

3.2.1 RDF semantics

We start by discussing RDF's formal semantics. All definitions we display here are taken from the W3C recommendation [35].

As mentioned earlier, RDF and N3 (see Section 2.5.1) distinguish between three kinds of symbols:

Definition 1 (RDF alphabet). *The alphabet of RDF consists of the following disjoint sets of symbols: the set* IRI *of IRI symbols, the set* L *of literals and the set* B *of blank node symbols.*

IRIs are all Unicode strings that conform to the syntax defined in RFC 3987 [20]. Literals express concrete values such as numbers, strings or dates and consist of a lexical form combined with a datatype IRI and – depending on the concrete datatype – possibly an additional language tag.[8] The only syntactic condition the RDF specification puts on blank nodes is that they need to be disjoint from IRIs and literals. In this thesis we follow the turtle syntax [23] which means that we either express blank nodes using square brackets "`[ ]`" or strings starting with underscore and column "`_:`". These symbols can now be combined into triples and graphs:

Definition 2 (Triples and graphs). *For the RDF alphabet consisting of IRI symbols* IRI*, literals* L *and blank nodes* B *we define:*

- *An* RDF triple $s\ p\ o.$ *consists of three components: the* subject $s \in \mathrm{IRI} \cup \mathrm{B}$*, the* predicate $p \in \mathrm{IRI}$*, and the* object $o \in \mathrm{IRI} \cup \mathrm{B} \cup \mathrm{L}$*. We denote the set of all RDF triples by* $\mathcal{T}$*.*
- *An* RDF graph *is a set of RDF triples. We denote the set of all RDF graphs by* $\mathcal{G} = 2^{\mathcal{T}}$*.*

[8] Our examples often contain literals without a specific datatype. In these cases the datatype `http://www.w3.org/2001/XMLSchema#string` is assumed. `"example"` stands for `"example"^^<http://www.w3.org/2001/XMLSchema#string>`.

For the reader familiar with first order logic there is one detail of RDF syntax – and thereby also of N3 – which is very interesting: RDF does not strictly distinguish between constants and predicates. The definition allows all IRIs in all positions. It is possible to state for example

```
:Kurt :knows :Albert.  :knows a :Predicate.
```
(3.9)

This also influences how the interpretation of RDF triples and graphs is defined:

Definition 3 (Simple interpretation). *A* simple interpretation I *is a structure consisting of the following elements:*

1. *A non-empty set* IR *of resources, called the domain or universe of* I.
2. *A set* IP*, called the set of properties of* I.
3. *A mapping* $\mathrm{IEXT} : \mathrm{IP} \to 2^{\mathrm{IR}\times\mathrm{IR}}$
4. *A mapping* $\mathrm{IS} : \mathrm{IRI} \to \mathrm{IR} \cup \mathrm{IP}$
5. *A partial mapping* $\mathrm{IL} : \mathrm{L} \nrightarrow \mathrm{IR}$

Next to the domain of discourse (the set of resources IR) as we know it from first order logic, we additionally have a set of properties IP. The interpretation function IS maps IRIs to the union of IR and IP. IRIs can stand for properties, resources or – as it is the case in Formula 3.9 – both. For properties there is an additional function to determine their meaning, the *extension* IEXT. This function maps each property to the set of pairs of resources (which can again also be properties at the same time) for which the property holds. For ground graphs, i.e. graphs which do not contain blank nodes, we get the following definition:

Definition 4 (Semantic conditions for ground graphs). *A simple interpretation* I *can be applied to an element* $E \in \mathrm{IRI} \cup \mathrm{L} \cup \mathcal{T} \cup \mathcal{G}$ *as follows:*

1. *If* E *is a literal then* $\mathrm{I}(E) = \mathrm{IL}(E)$
2. *If* E *is a IRI then* $\mathrm{I}(E) = \mathrm{IS}(E)$
3. *If* E *is a ground triple* `s p o.` *then* $\mathrm{I}(E) =$ *true if* $(\mathrm{I}(s), \mathrm{I}(o)) \in \mathrm{IEXT}(\mathrm{I}(p))$*, otherwise* $\mathrm{I}(E) =$ *false*
4. *If* E *is a ground graph then* $\mathrm{I}(E) =$ *false if there exists a triple* E' *in* E *for which* $\mathrm{I}(E') =$ *false, otherwise* $\mathrm{I}(E) =$ *true*

Note, that the mapping IL from Definition 3 is only a partial function, i.e. it is not defined for all literal values which are syntactically possible. The reason for that is that RDF also covers the semantics of datatypes[9] and according to that it is possible that a literal does have a referent. Triples whose objects are not in the domain of the function IL are according to point 3 in the above interpretation false.

Having a notation for the meaning of ground terms we now take a closer look at blank nodes:

Definition 5 (Semantic condition for blank nodes). *If $E \in \mathcal{G}$ is an RDF graph then $\mathrm{I}(E) =$ true if $[\mathrm{I}+\mathrm{A}](E) =$ true for some mapping $\mathrm{A} : \mathrm{B} \to \mathrm{IR}$, otherwise $\mathrm{I}(E) =$ false.*

RDF treats blank nodes as implicitly existentially quantified variables. A graph is true under an interpretation if there exists one single mapping which assigns resources to all blank nodes occurring in it such that the interpretation of the ground graph is true. If we had to make the implicit quantification in RDF – or at least of the part which is covered by RDF semantics – explicit, we would need to always put the quantifier on the top formula. In that sense RDF is simpler than N3: RDF semantics does not cover such nested constructs as we explained earlier when discussing Formula 3.1.

We finish the subsection by defining the concepts of satisfaction and simple entailment:

Definition 6 (Satisfaction and simple entailment). *Let $E, F, G \in \mathcal{G}$ be RDF graphs*

- *An interpretation* I (simply) satisfies *E, written as $\mathrm{I} \models E$, if $\mathrm{I}(E) =$ true.*
- *We call E (simply) satisfiable if there exists a simple interpretation I such that $\mathrm{I} \models E$, otherwise we call it* (simply) unsatisfiable.
- *G simply entails E, written as $G \models E$, if for every interpretation I which satisfies G ($I \models G$), also satisfies E ($I \models E$).*
- *If $E \models F$ and $F \models E$ we call F and G* equivalent.

The semantics of RDF covers two other kinds of interpretations: D-interpretations which handle literals and their datatypes and RDF interpretations which add a few properties and resources with a predefined

[9] We do not cover datatype semantics here since it is not relevant for this thesis. We refer the reader interested in that topic to https://www.w3.org/TR/rdf11-mt/#literals-and-datatypes.

meaning to the logic. These concepts are captured by the RDF vocabulary and contain among others the predicate `rdf:type`[10] which we used above and `rdf:Property`, the class of all properties. The definitions done in this context can directly be added to a possible definition of the semantics of N3, entailments defined in this context can be covered by adding rules for that. We therefore omit the discussion of D-interpretations and RDF interpretations in this thesis.

3.2.2 Structural blank nodes

Next to the definition of RDF's semantics covered by the concept of simple, D-, and RDF interpretations, the RDF specification also includes an informal part explaining the intended use of several elements of the RDF vocabulary. Especially the concepts explained in this informal part often require constructs using blank nodes which are not really meant to express the existence of a resource in the domain of discourse but serve a technical purpose: All knowledge in RDF needs to be expressed in triples. In some cases these triples are rather artificial and need to make use of auxiliary blank nodes. In this thesis we call these blank nodes *structural blank nodes*. We further explain how such blank nodes are used on the example of *RDF collections*.

Collections are used to describe list structures in RDF. RDF follows a similar idea as for example Prolog [89, 90] or LISP [91], which both represent lists using a binary function. Starting with the empty list, the binary function can be used to recursively construct new lists from existing lists by simply adding new elements at their beginning. To illustrate the idea let us assume that `f` is the binary function we use in this context and `nil` is the empty list. If we want to represent a list consisting of *Kurt*, *Albert* and *John* we write:

$$\texttt{f}(\textit{Kurt}, \texttt{f}(\textit{Albert}, \texttt{f}(\textit{John}, \texttt{nil}))) \tag{3.10}$$

As RDF does not allow the use of compound terms in subject or object position of a triple – we can only use simple IRIs, literals or blank nodes – the above notation cannot be directly translated to RDF.

To express lists RDF provides the terms `rdf:first`, `rdf:rest` and `rdf:nil`. The term `rdf:nil` corresponds to the term `nil` from the example above and represents the empty list. The first and the second argument from the above function `f` are expressed by the predicates `rdf:first` and `rdf:rest`, respectively. In order to connect these two arguments, blank

[10] Here and later on in this thesis the prefix `rdf:` stands for http://www.w3.org/1999/02/22-rdf-syntax-ns#.

nodes are used. The list from Formula 3.10 can be represented in RDF as follows:

```
_:c1 rdf:first :Kurt.
_:c1 rdf:rest _:c2.
_:c2 rdf:first :Albert.
_:c2 rdf:rest _:c3.
_:c3 rdf:first :John.
_:c3 rdf:rest rdf:nil.
```
(3.11)

Strictly following the definitions in the previous section the above expression means that *there exist* the lists `c1`, `c2` and `c3` and that the first element of list `c1` is `:Kurt` and the rest of the list is `c2`, that the list `c2` has `:Albert` as first element and list `c3` as rest and that `c3` has `:John` as first element and the empty list as rest. Using `_:c1` in a triple is therefore not the same as directly using a list as for example in the formats mentioned above.

For lists as the one above, RDF provides an extra notation. The collection given in Formula 3.11 can also be expressed by simply listing its elements in brackets "`( )`":

```
(:Kurt :Albert :John)
```
(3.12)

This notation is syntactic sugar for the above and even when brackets are used instead of first-rest-ladders in RDF we still have implicit blank nodes and thereby there is also implicit existential quantification present.

3.2.3 Referencing formulas in RDF

In Section 3.2.1 above we learned that RDF semantics understands blank nodes as implicitly existentially quantified on top level, i.e. if a blank node is occurring in a graph its existential quantifier is always assumed to be in front of this graph. In this context we also mentioned that this interpretation makes sense because the part of RDF which is covered by RDF semantics does not support the concept of referring to other graphs or formulas. Therefore, constructs as we have seen in Example 3.1 where we had a blank nodes nested in a cited formula cannot occur here.

Next to the part of RDF which is covered by RDF semantics there are concepts in RDF which are syntactically valid but whose meaning is only informally defined. Some of these concepts were created to serve a similar purpose as the citation of graphs in N3 does: in some way they enable the user to reference triples or graphs. For our considerations, it is interesting how exactly

blank nodes occurring in these referenced graphs or triples are quantified: Are they quantified on top level or on a lower level?

Before we further discuss the concepts for referencing formulas and answer the question above for each of it, we discuss an example in N3 to clarify that the exact position of the existential quantifier matters. Consider the following formula:

```
:Kurt :denies {_:x rdf:type :Unicorn}.
```
(3.13)

An interpretation which puts the existential quantifier on top level would be:

$$\exists x : \text{denies}(\text{Kurt}, (\text{type}(x, \text{Unicorn}))) \tag{3.13a}$$

There exists something of which Kurt denies that it is a unicorn.

So, if Kurt for example knows for sure that his friend Albert is not a unicorn then this statement is true. There is no statement made whether or not Kurt beliefs in the existence of unicorns in general.

This is different for the interpretation which assumes the quantifier to be nested as the W3C team submission for N3 [71] imposes. Here we get:

$$\text{denies}(\text{Kurt}, (\exists x : \text{type}(x, \text{Unicorn}))) \tag{3.13b}$$

Kurt denies that unicorns exist.

In this case the statement means that Kurt does not believe in the existence of unicorns. The two interpretations are fundamentally different and it is important where we put the quantifier for the existentially quantified variable.

Below, we explain the concepts RDF reification, named graphs, RDF* and singleton properties, and discuss how they understand blank nodes in referenced triples and formulas.

RDF reification

As discussed above, RDF follows a simple triple structure which does not support the use of compound constructs in any position of the triple and it is not possible to directly refer to triples or graphs. To overcome this problem in the case of triples and syntactically stay in RDF, the concept of *reification* was introduced. The idea behind this concept is to represent triples by concrete IRIs or blank nodes and thereby make them part of the domain of discourse.

For this purpose, RDF contains the reification vocabulary consisting of the terms `rdf:subject`, `rdf:predicate`, `rdf:object` and `rdf:Statement`. If we want to represent a triple like for example

```
_:x rdf:type :Unicorn.
```
(3.14)

we can give it a concrete name like for example `:triple1` and state which *subject*, *predicate* and *object* it has. For our example triple we state:

```
:triple1 a rdf:Statement;
         rdf:subject _:x;
         rdf:predicate rdf:type;
         rdf:object :Unicorn.
```
(3.15)

This enables us to talk about our `:triple1` and use it further. We can now for example state:

```
:Kurt :denies :triple1.
```
(3.16)

The exact meaning of Formula 3.15 in combination with Formula 3.16 is not formally defined[11] and we do not want to get deeper into a probable formalisation here. But even without such a formal model we can say something about the blank node contained in this example: As the reference to the triple is done in simple RDF syntax without any syntactic extension, Definition 5 holds here which states that all blank nodes occurring in simple RDF triples are quantified at graph level. An interpretation of the above formulas would thus be similar[12] to Interpretation 3.13a rather than to Interpretation 3.13b. In this aspect and in the fact that reification is only meant to reference triples but not their conjunctions RDF reification differs from citation in N3.

Named graphs

Next to reification which can be used to refer to triples in RDF, RDF syntax also provides the option to reference graphs. For this purpose the RDF Dataset Language TriG [92] was defined. With TriG it is possible to list a graph – a very similar construct to N3's nested formula – next to a name and

[11] For the informal definition of RDF reification see: https://www.w3.org/TR/rdf11-mt/#reification

[12] We want to emphasize here that so far Interpretations 3.13a and 3.13b have no fixed meaning. The first order like structure we use here is only meant to clarify the exact position of the quantifier implicitly expressed by the blank node `_:x`.

then use the name in other triples. A possible expression using TriG would be:

```
:graph { _:x rdf:type :Unicorn}
:Kurt :denies :graph.
```
(3.17)

The problem with that construct is now, that not only the semantics of named graphs is not formally defined, there is not even a consensus within the Semantic Web community how such named graphs need to be interpreted: Does `:graph` denote a graph? If yes, is that graph open or closed in the above example? If Kurt denies `:graph` does he then only deny the statement we see above or does `:graph` contain more triples? Where exactly is the scope of the blank node? A list of possible interpretations of named graphs is gathered on a Web site [87]. Here, it is also recommended to indicate the intended meaning of a named graph when using it. As long as the semantics of TriG is not fixed, it is not possible to say how the scoping of blank nodes in TriG relates to scoping in N3. Depending on how we formalise the meaning of cited formulas in the latter, it is possible to use N3 to express the relationship between a graph name (`:graph`) and the graph pattern which stands next to it (`{ _:x rdf:type :Unicorn}`)[93].

RDF*

Another way of referring to RDF triples in an RDF graph is RDF*[94, 95]. RDF* is not covered by the W3C recommendation of RDF, but it recently gained popularity in the community and it is therefore very likely that it will be part of the next version of RDF. We therefore include it in this discussion. To incorporate triples into triples, RDF* uses angle brackets `<< >>` in a similar way N3 uses curly brackets `{ }` for graphs. An example for such a construct is:

```
:Kurt :denies <<_:x rdf:type :Unicorn>>.
```
(3.18)

As RDF* is currently still at proposal stage, its semantics is not fixed yet. Currently two possible ways of interpreting expressions like the one above are discussed: The `<< >>`-notation could either be understood as syntactic sugar for reification – which leaves us, as discussed above, without a formal definition – or it can be a proper extension of RDF. To establish the latter solution, the creators of RDF* introduce the concept of *redundancy* [94]: If Formula 3.18 is stated together with the RDF triple

```
_:x rdf:type :Unicorn.
```
(3.19)

then this last triple is redundant. This definition is motivated by the idea that triples referencing triples are mainly used to add metadata to valid triples. Following that logic, stating Formula 3.18 already implies that Formula 3.19 is true.

In this last aspect, RDF* differs from N3: The journal paper introducing N3 clearly states that quoted formulas in N3 are not necessarily true [5, p 11]. We furthermore encounter differences in the interpretation of blank nodes occurring in referenced triples: While N3 assumes such blank nodes to be existentially quantified on the nested formula both possible interpretations of referenced triples in RDF* assume them to be quantified on the top graph. Reification does not support the concept of nesting at all and from the fact that under the second possible formalisation Formula 3.18 implies Formula 3.19 we can conclude that is is not possible to express nested existential quantification with RDF*.

Singelton Properties

As a last option to refer to triples in RDF we discuss singleton properties [96]. Similar to RDF* this approach of referring to triples was introduced as an alternative to RDF reification and does not form part of the standard. The idea behind singleton properties is to use unique URIs for the predicates of triples to be referred to. Instead of using the property `rdf:type` in Triple 3.19 we would use a predicate `rdf:type#123`:[13]

```
_:x rdf:type#123 :Unicorn.          (3.20)
```

This so-called *singleton property* then can be understood as a special instance of the predicate `rdf:type`. This can be indicated by adding an extra triple:

```
rdf:type#123 rdf:singletonPropertyOf rdf:type.  (3.21)
```

The semantics of singleton properties is clearly defined. Singleton properties can only be used once in predicate position and in this position they have the same meaning as the general predicate they belong to. Triple 3.20 stated together with Triple 3.21 implies Triple 3.19. Additionally, singleton properties can be used in subject or object position. Following our example, we can state:

```
:Kurt :denies rdf:type#123.          (3.22)
```

[13] In practical cases we would use a more complex identifier than `123` to insure that we have indeed a property which is only used once in the Web. The triples indicated here should only be understood as an illustrative example.

The singleton property `rdf:type#123` is now understood as a resource in the domain of discourse. As we know that this property occurs only once and is therefore only connected to one subject and one object, we can understand it as a reference to the triple it forms together with these two elements. In the semantics, it is not clearly defined how we need to understand blank nodes in such referred triples but since in this approach we can only refer to triples which are stated on top level where blank nodes have global scoping, it also makes sense to assume the same scoping for the reference to these triples. Here this means that there exists an x for which Kurt denies that it is a unicorn. Given that Triple 3.20 clearly states that x is indeed a unicorn, we also know that Kurt is wrong with his denial.

The last example shows a clear difference between referring to formulas by using singleton properties and doing so by means of N3: while with the singleton property approach it is not possible to talk about a triple which is not true, we can easily do so with N3. Blank nodes are not quantified in the referred triple but globally. As reification and RDF*, singleton ptoperties can furthermore only be used to refer to triples and not – as it is possible with named graphs and N3 – to graphs.

3.3 Logics with implicit quantification

Having discussed how implicit quantification is handled in RDF – RDF only allows implicit existential quantification and in most cases this quantification is not nested[14] – we now take a broader look to other frameworks related to RDF or rule-based reasoning which support some kind of implicit quantification.

Implicit quantification is widely used in different contexts. Most frameworks around RDF support the use of *blank nodes*. Hogan et al. [97] provide an overview of their meaning and use in these different contexts. Being a direct extension of RDF, the meaning of blank nodes in RDFS is the same as in RDF, they are assumed as existentially quantified and RDFS does not support the concept of nesting. Hogan et al. furthermore point out that, despite the clear definition, users do not always understand and use blank nodes as existentially quantified variables in RDF: often blank nodes are rather used to refer a concrete object whose IRI is unknown. A possible reason for that is that SPARQL [15] interprets blank nodes occurring in the queried RDF graph exactly in that manner.

[14] The only possible exception is TriG where the semantics is not fixed.

Implicit universal quantification can be found in different programming languages and RDF related frameworks. In Prolog [89] variables are understood to be universally quantified. The scope of this quantification is the clause in which the variable occurs. This is similar to the universal quantification on top level as implemented in EYE. But as Prolog does not allow the construction of nested rules, which form the biggest challenge for the determination of scoping in N3, Prolog's quantification is only partly comparable to N3's. The RDF-query language SPARQL [15] supports *query variables* which, if a query is understood as some kind of filter rule, can be seen as universally quantified. SPARQL allows nesting of graphs and queries. A SPARQL query consists of two parts, an outer part starting with one of the keywords SELECT, DESCRIBE, ASK, or CONSTRUCT which can contain search variables and a WHERE-part which specifies the search pattern. If a query is nested in another one, i.e. a new query occurs in the WHERE-part of another query, only the universal variables which occur in the SELECT-part of the sub-query share their variables with the WHERE-part of the top query.[15] The aspect that variables in nested queries are clearly separated in these cases is slightly similar to the separation of different deeply nested graphs in N3. However, when we translate such nested queries to core logic rules, we only need to rename identical variables occurring separately on different levels, the universal quantifier for all variables is still on top level.

3.4 Conclusion

In this chapter we discussed the concept of implicit quantification. N3 assumes blank nodes to be quantified on their *direct formula* and universals to be quantified on their *parent formula*. While we could clearly identify the direct formula as the formula in brackets { } surrounding a blank node, the identification of the *parent* was more difficult. Our tests showed that the reasoners EYE and Cwm understand this concept differently and that neither the W3C team submission nor the journal paper introducing N3 clarify this term.

In order to agree on *one* interpretation we need to be able to express the difference and to understand its impact for practical use cases. We therefore define a core logic for N3 which supports the same features as the original logic but only supports explicit quantification. Having such a logic at hand, we can map the syntax of N3 to its different interpretations and compare the results. This will be the topic of the next chapter.

[15] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#subqueries

This chapter was partly based on the publications:

D. Arndt, T. Schrijvers, J. De Roo, R. Verborgh, Implicit quantification made explicit: How to interpret blank nodes and universal variables in Notation3 Logic, Journal of Web Semantics 58 (2019) 100501. doi:10.1016/j.websem.2019.04.001.
URL https://authors.elsevier.com/c/1ZWg55bAYUaMkH

D. Arndt, R. Verborgh, J. De Roo, H. Sun, E. Mannens, R. Van de Walle, Semantics of Notation3 logic: A solution for implicit quantification, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 127–143.
URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_9

Chapter 4

Defining the semantics of N3Logic

The examples in Section 3.1 from the previous chapter explain how N3 formulas are interpreted differently by different reasoners. To point out these variations, we used natural language together with a first-order-logic-like structure which allowed to cite formulas. However, both the natural language and logical structure do not have a fixed definition of their semantics and can thus still be understood in different ways. In order to dispose of this ambiguity when comparing interpretations, we now define *N3 Core Logic*. This logic supports all important features of N3 such as the possibility to refer to formulas or to use quantified variables in predicate position, but only allows *explicit* quantification. To provide the semantics for N3Logic we then make use of an attribute grammar which maps the syntax of N3 to our *N3 Core Logic* following the interpretations assumed by Cwm and EYE. Thereby we make N3's implicit quantification explicit and provide two possible formal models for N3. As our mechanism can be expanded to other possible interpretations we thereby furthermore provide a tool to discuss about and to compare different possible interpretations of N3Logic. This is a first, but very important step to enable the Semantic Web community to come to an agreement.

4.1 N3 Core Logic

To find a proper way to formalise N3Logic we already took a closer look at the logic it extends, RDF. Some concepts of N3 are also present in RDF, like for example blank nodes representing implicitly existentially quantified

f ::=		formulas:
	t t t	atomic formula
	e → e	implication
	f f	conjunction
	∀v.f	universal formula
	∃v.f	existential formula
t ::=		terms:
	v	variables
	c	constants
	e	expressions
	(k)	lists
	()	empty list
k ::=		list content
	t	term
	t k	term tail
e ::=		expressions:
	<f>	formula expression
	<>	true
	false	false

Figure 4.1: Syntax of the N3 Core Language $\mathcal{L}$ over $\mathcal{V} \cup \mathcal{C}$.

variables or the idea of not strictly separating constants and relations. Our definition *N3 Core Logic* takes the formalisation of these concepts as well as the well-known theory of First order Logic into account.

4.1.1 Syntax

Given disjoint countable sets of variables $\mathcal{V}$ and constants $\mathcal{C}$ we define the N3 Core Language $\mathcal{L}$ of N3 over $\mathcal{C} \cup \mathcal{V}$ as displayed in Figure 4.1. Before we discuss the details of this definition, we first give an example. If we go back to Formula 3.1 above:

```
_:x :says {_:x :knows :Albert.}.
```

We can express Interpretation 3.1a as:

```
∃x.  x says <x knows Albert>
```

And Interpretation 3.1b as:

```
∃x1.  x1 says <∃x2.  x2 knows Albert>
```

Note that this notation is close to the original N3 notation. To make a clear distinction between N3 Core Logic and N3Logic, we use angle brackets

instead of curly brackets and a different kind of arrow. For the same reason, we do not represent constants and variables using IRIs (i.e. we write `x` instead of `:x`) in our examples.[1] The main difference between N3Logic and N3 Core Logic is the symbol used for explicit quantification in the latter, which is taken from first order logic to emphasize that the quantifiers here are interpreted in the order they occur.

Variables in a formula can either occur free or bound:

Definition 7 (Free variables)**.** *The set of free variables of a language element $l \in \mathcal{L}$, written $FV(l)$ is defined as follows:*

$$
\begin{aligned}
FV(v) &= \{v\} \\
FV(c) &= \emptyset \\
FV(\texttt{<}f\texttt{>}) &= FV(f) \\
FV(\texttt{<>}) &= \emptyset \\
FV(false) &= \emptyset \\
FV(\texttt{(}t_1 \ldots t_n\texttt{)}) &= FV(t_1) \cup \ldots \cup FV(t_n) \\
FV(\texttt{()}) &= \emptyset \\
FV(t_1 t_2 t_3) &= FV(t_1) \cup FV(t_2) \cup FV(t_3) \\
FV(e_1 \rightarrow e_2) &= FV(e_1) \cup FV(e_2) \\
FV(f_1 f_2) &= FV(f_1) \cup FV(f_2) \\
FV(\forall v.f) &= FV(f) \setminus \{v\} \\
FV(\exists v.f) &= FV(f) \setminus \{v\}
\end{aligned}
$$

We call every language element $l \in \mathcal{L}$ with $FV(l) = \emptyset$ ground*. We denote the set of ground language elements by $\mathcal{L}_g$, for the set $\mathcal{T}$ of terms as defined in Figure 4.1, we define $\mathcal{T}_g := \mathcal{L}_g \cap \mathcal{T}$ as the set of ground terms.*

For example, in the formula

```
∀x.  x knows y
```

the variable `x` occurs bound, while `y` is free.

[1] Note that the representation of the constants and variables only depends on the choice of $\mathcal{C}$ and $\mathcal{V}$.

4.1.2 Semantics

To define the semantics of the core language $\mathcal{L}$ over $\mathcal{V} \cup \mathcal{C}$ we first introduce the notion of structure.

Definition 8 (Structure). *A structure over a language $\mathcal{L}$ is a quadruple $\mathfrak{A} = (\mathcal{D}, \mathcal{P}, \mathfrak{a}, \mathfrak{p})$ containing:*

- *A non empty set $\mathcal{D}$ called the* domain.
- *A non empty set $\mathcal{P} \subset \mathcal{D}$ called the* set of properties.
- *A mapping $\mathfrak{a} : \mathcal{T}_g \to \mathcal{D}$ called the* object mapping.
- *A mapping $\mathfrak{p} : \mathcal{P} \to 2^{\mathcal{D} \times \mathcal{D}}$ called the* predicate mapping.

Similarly to a structure in the classical first order sense, a core logic structure consists of a domain of discourse and maps from the terms of the language into that domain. There are two main differences:

Firstly, the interpretation function for relations does not directly act on symbols of the language but on a subset of the domain of discourse. The reason for that is the same why in RDF we have the distinction between resources (IR) and properties (IP) and the notion of an extension (IEXT) (see Definition 3): RDF and N3 – and thereby also its core logic – do not distinguish between constants and relations. If a term is used in predicate position and in subject or object position at the same time, an interpretation needs to take the connection between the two occurrences of the term into account.

To better understand that, consider Formula 3.9, which we discussed in Section 3.2.1. In N3 Core Logic the formula looks as follows:

```
knows a predicate.  Albert knows Kurt.
```

Now, imagine that we have another concept `knows2` which has the exact same meaning as our original predicate `knows`.[2] If we want to formally define this equality we need to be able to state that `knows2` *used as subject or object* denotes the same resource in the domain of discourse as `knows` *and* that `knows2` *used in predicate position* of a triple denotes the same relation as `knows` does. If the interpretation function for relations acts on the domain of discourse we can formalise the equality of `knows` and `knows2` by simply saying that they denote the same element from the domain of discourse. By

[2] In the context of the Semantic Web it can easily happen that two ontologies refer to the exact same concept using different URIs.

defining interpretation function $\mathfrak{p}$ on $\mathcal{P} \subset \mathcal{D}$ we furthermore ensure that our definition is compatible with RDF semantics.[3]

The second difference is that our structure is only defined for ground terms and not, as it is in other logics, for terms containing variables. To define the semantics for these we make use of a grounding function:

Definition 9 (Grounding Function). *A* grounding function γ_g *over a language* $\mathcal{L}$ *is a mapping from the set of variables* $\mathcal{V}$ *into the set of ground terms* $\mathcal{T}_g$*.*

This function can be extended to all elements of the language:

Definition 10 (Extended Grounding Function). *For a language* $\mathcal{L}$ *and a* grounding function $\gamma_g : \mathcal{V} \to \mathcal{T}_g$ *we define its extension* $\gamma : \mathcal{L} \to \mathcal{L}_g$ *as follows:*

$$
\begin{aligned}
\gamma(v) &= \gamma_g(v) \\
\gamma(c) &= c \\
\gamma(\texttt{<}f\texttt{>}) &= \texttt{<}\gamma(f)\texttt{>} \\
\gamma(\texttt{<>}) &= \texttt{<>} \\
\gamma(\texttt{false}) &= \texttt{false} \\
\gamma(\texttt{(}t_1 \dots t_n\texttt{)}) &= \texttt{(}\gamma(t_1) \dots \gamma(t_n)\texttt{)} \\
\gamma(\texttt{()}) &= \texttt{()} \\
\gamma(t_1 t_2 t_3) &= \gamma(t_1)\gamma(t_2)\gamma(t_3) \\
\gamma(e_1 \to e_2) &= \gamma(e_1) \to \gamma(e_2) \\
\gamma(f_1 f_2) &= \gamma(f_1)\gamma(f_2) \\
\gamma(\forall v.f) &= \forall v.\gamma[v \mapsto v](f) \\
\gamma(\exists v.f) &= \exists v.\gamma[v \mapsto v](f)
\end{aligned}
$$

Where $\gamma[x \mapsto t]$ *denotes the extended grounding function which is identical to* γ *except for* $x \mapsto t$*.*

For the definition of an interpretation and meaning we now use the grounding function instead of a classical valuation function which maps the set of variables into the domain of discourse. We do that to be able to make a distinction between terms occurring in a cited formula and subjects, predicates and objects of formulas directly occurring in a graph. The informal specification of N3's semantics claims that nested formulas are not "*referentially*

[3] In RDF semantics the set of resources is not necessarily a subset of the domain of discourse and the interpretation function for IRIs maps to the union of these two sets. We need to understand our domain of discourse as this union to do the alignment.

transparent" [5, p.7]. This means that even if two terms in a cited formula denote the same object in the domain of discourse two cited formulas which only differ in the use of these two terms should be considered as different. As a concrete example consider the following:[4]

```
LoisLane believes <Superman can fly.>.
```

Even if we know that the term "`Superman`" denotes the same object as "`ClarkKent`", this knowledge combined with the above formula should not imply the following formula:

```
LoisLane believes <ClarkKent can fly.>.
```

In the concrete example `LoisLane` might not know that `ClarkKent` denotes the same resource as `Superman`.

The need of a grounding function now becomes clear if we slightly change the above formulas and use them together with quantification. Consider the following example:

```
∃x.  LoisLane believes <x can fly.>.
```

To know whether or not this last statement is true, it needs to be possible to map an existentially quantified variable to its representation instead of directly mapping it to the resource it represents.

An interpretation for core formulas therefore makes use of a grounding function:

Definition 11 (Core logic interpretation). *A core logic interpretation $\mathfrak{I}$ is a pair $(\mathfrak{A}, \gamma)$ consisting of a structure $\mathfrak{A}$ and a grounding function γ.*

With the present definitions we can now define the meaning of formulas.

Definition 12 (Meaning of formulas). *Let $\mathfrak{I} = (\mathfrak{A}, \gamma)$ be a core logic interpretation of a language $\mathfrak{L}$. Then:*

1. $\mathfrak{I} \models t_1 t_2 t_3$ *iff* $(\mathfrak{a}(\gamma(t_1)), \mathfrak{a}(\gamma(t_3))) \in \mathfrak{p}(\mathfrak{a}(\gamma(t_2)))$
2. $\mathfrak{I} \models \texttt{<}f_1\texttt{>} \rightarrow \texttt{<}f_2\texttt{>}$ *iff* $\mathfrak{I} \models f_2$ *if* $\mathfrak{I} \models f_1$.
3. $\mathfrak{I} \models \texttt{false} \rightarrow \texttt{<}f\texttt{>}$
4. $\mathfrak{I} \models \texttt{<}f\texttt{>} \rightarrow \texttt{<>}$

[4] This example is often called the "superman problem" and has been broadly discussed in the context of RDF reification. See for example: https://www.w3.org/2001/12/attributions/#superman.

5. $\mathfrak{I} \models$ `<f>` $\rightarrow$ `false` *iff* $\mathfrak{I} \not\models$ `f`.
6. $\mathfrak{I} \models$ `<>` $\rightarrow$ `<f>` *iff* $\mathfrak{I} \models$ `f`.
7. $\mathfrak{I} \models$ `f`$_1$ `f`$_2$ *iff* $\mathfrak{I} \models$ `f`$_1$ *and* $\mathfrak{I} \models$ `f`$_2$.
8. $\mathfrak{I} \models \forall$ `v . f` *iff* $(\mathfrak{A}, \gamma[\mathtt{v} \mapsto t]) \models$ `f` *for all* $t \in \mathcal{T}_g$.
9. $\mathfrak{I} \models \exists$ `v . f` *iff* $(\mathfrak{A}, \gamma[\mathtt{v} \mapsto t]) \models$ `f` *for some* $t \in \mathcal{T}_g$.

We call a formula `f` true *in* $\mathfrak{I}$ *iff* $\mathfrak{I} \models$ `f`.

Note that in N3 lists are treated as *first-class citizens*, this means that N3 does not use RDF's `rdf:first-rdf:rest` notation (explained in Section 3.2.2) to construct lists, but allows the treatment of lists as a proper datatype (see also [5, p.6]). This is reflected in our core logic and its semantics, where we directly map ground lists into the domain of discourse. We do the same for cited formulas since such a treatment allows later refinement by specifying the mapping function further.

Another remark we want to make here is that by using grounding for quantified predicates and mapping them first to a single element of the domain before assigning them to the set of pairs of domain elements for which the relation they denote holds ensures that we stay in first order logic.[5] We do not quantify over $2^{\mathcal{D}\times\mathcal{D}}$ but over the countable subset of relations which have a name in $\mathcal{L}$ and can therefore follow the idea of Henkin Semantics [98] to map core logic to first order logic.

The definition of a model is similar to first order logic:

Definition 13 (Model). *Let* Φ *be a set of formulas. We call a core logic interpretation* $\mathfrak{I}$ *a* model *of* Φ *iff every formula in* Φ *is true in* $\mathfrak{I}$.

We finish this section by defining the classical concepts of logical consequence and equivalence:

Definition 14 (Logical consequence). *Let* Φ *be a set of ground formulas. A ground formula* `f` *is called a* logical consequence *of* Φ *(written:* $\Phi \models$ `f`*) iff* `f` *is true in every model of* Φ.

Instead of $\{\mathtt{f}_1\} \models \mathtt{f}_2$ we sometimes write $\mathtt{f}_1 \models \mathtt{f}_2$.

Definition 15 (Logical equivalence). *Two formulas* ϕ *and* ψ *are called logically equivalent (*$\phi \equiv \psi$*) iff* $\phi \models \psi$ *and* $\psi \models \phi$.

[5] Strictly speaking one of the two mechanisms would already be enough to ensure that we stay in first order logic.

`s ::=`		start:
	`f`	formula
`f ::=`		formulas:
	`t t t.`	atomic formula
	`e => e.`	implication
	`f f`	conjunction
`t ::=`		terms:
	`uv`	universal variable
	`ex`	existential variable
	`c`	constant
	`e`	expression
	`(k)`	list
	`()`	empty list
`k ::=`		list content:
	`t`	term
	`t k`	term tail
`e ::=`		expressions:
	`{f}`	formula expression
	`{}`	true
	`false`	false

Figure 4.2: Overview N3 Syntax

4.2 N3 Syntax

N3 Core Logic as defined above enables us to give more formal descriptions of the N3 interpretations explained in Section 3.1. Next, we also want to formalise the connection between N3 syntax and its possible interpretations. In order to do so, we make use of syntax trees and define attribute grammars [99, 100] – a formalism allowing us to pass information through the syntax tree of a formula. Before we discuss this in detail, we first provide a context-free grammar for N3 and explain how the interpretations of N3 illustrated so far are related to the syntax trees which can be constructed using that grammar.

We start with the definition of the alphabet. This contains all symbols which can appear in the concrete representation of an N3 formula:

Definition 16 (N3 Alphabet). *Let C be a set of constants, U a set of universal variables and E a set of existential variables. Let these sets be mutually disjoint and disjoint with {`{`, `}`, `(`, `)`, `=>`, `false`, `.`}. Then we call*

$$\mathcal{A} := C \cup U \cup E \cup \{\texttt{\{}, \texttt{\}}, \texttt{(}, \texttt{)}, \texttt{=>}, \texttt{false}, \texttt{.}\}$$

an N3 alphabet.

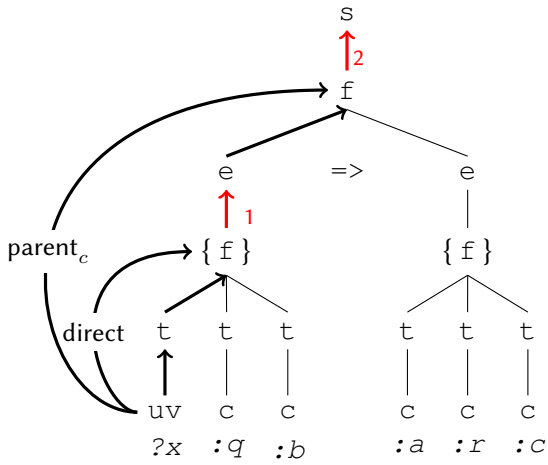


Figure 4.3: Direct and parent$_c$ formula.

In concrete representations constants can either be literal values or IRIs. Universal variables start with the symbol `?`, existential variables with `_:`.

Given such an N3 alphabet $\mathcal{A}$, we define an N3 grammar over $\mathcal{A}$ as displayed in Figure 4.2 where the node `ex` stands for an existential variable, i.e. an element of E, `uv` for a universal variable, i.e. an element of U, and `c` for a constant, i.e. an element of C.

As discussed earlier, N3Logic mainly differs from N3 Core Logic in the symbols used – the curly brackets { } can be translated into angle brackets < >, the N3 arrow => into a simple $\rightarrow$ – and in the way quantification is expressed: N3Logic only supports implicit quantification[6] while in N3 Core logic only explicit quantification is allowed. To translate the implicitly quantified variables of N3 into the explicit ones of N3 Core Logic we use the structure of the syntax tree.

To illustrate the idea behind that, we take a closer look at one of the concepts used in the informal definitions of N3's quantification: According to the W3C team submission [71] existential variables are quantified on their *direct formula* (Section 3.1.1) and universal variables are quantified on their *parent formula* (Section 3.1.2). Informally such a *direct formula* f of a term t is the next formula surrounded by curly brackets { } that contains t, or, if such a formula does not exist, the top formula. In EYE's interpretation, the *parent formula* is now the top formula (concept *parent*$_e$). For Cwm, the *parent formula* g of f is the next higher formula surrounded by curly brackets,

[6] As explained in Section 3.1.3 we exclude the discussion of N3's explicit quantification from this formalisation. It is therefore also not reflected in the grammar.

Grammar		Synthesized attribute *ds*		
s ::=	n	s.*ds*	←	n.*ds*
n ::=	d	n.*ds*	←	d
	$n_1 n_2$		←	$n_1.ds + n_2.ds$

Figure 4.4: Context-free grammar producing integers of arbitrary length (left) and definition of the attribute *ds* which composes the digit sum (right).

i.e. the direct formula of $\{f\}$ (*parent*$_c$). Being the top formula, the *parent*$_e$ *formula* is easy to find. For the *direct formula* and the *parent*$_c$ *formula* we use the syntax tree: The difference between a simple formula f and a formula in brackets $\{f\}$ is in the grammar point of view the difference between a *formula* f and a *formula expression* e. If we want to find the direct formula of a component and its parent$_c$, we can go through the syntax tree from the bottom to the top. The *direct formula* is then the first formula on that path which is a direct child of either an expression e or the start symbol s. The *parent*$_c$ *formula* is the second such formula.[7] We illustrate this in Figure 4.3 on the syntax tree of the formula:

```
{?x :q :b} => {:a :r :c}.
```
(4.1)

The direct formula of the terminal node ?x is the formula ?x :q :b., the parent$_c$ formula is the top formula. This parent$_c$ formula carries the universal quantifier for the variable. The formula means according to Cwm:

$$\forall \texttt{x}.\quad \texttt{<x q b>} \rightarrow \texttt{<a r c>}. \tag{4.1'}$$

The above process of *going through the syntax tree* can be described in a more precise way by using attribute grammars. This concept will be introduced in the following section.

4.3 Attribute Grammars

Attribute grammars [99, 100] are defined on top of context-free grammars like the one given above and extend this concept by so-called *attributes*. Such attributes are defined on the nodes of the syntax tree and can take values. These values can depend on other attribute values of the node itself and either its descendent nodes in the syntax tree – in this case the attribute is called

[7] Note that universals on top level like for example in the formula ?x :p :o. do not have a parent$_c$. We assume here, that these formulas are also quantified on top level. This differs from Cwm which does not support universal quantification on top level.

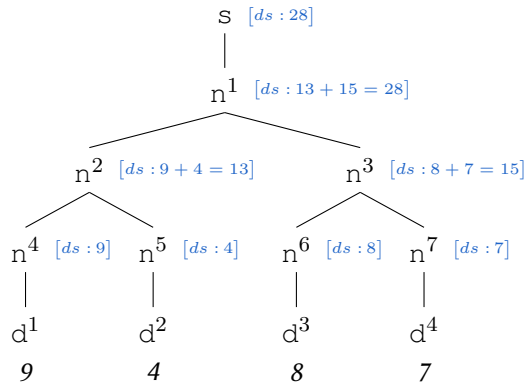


Figure 4.5: Syntax tree for 9487 produced by the grammar in Figure 4.4 with *attribute values* for *ds*.

synthesized – or on the attribute values of the parent node – in this case it is called *inherited*. The definitions used follow the notation of Paakki [101].

4.3.1 Example

Before providing a formal definition, we illustrate the idea of attribute grammars on a simpler example than the grammar provided above. Consider the context-free grammar displayed on the left side of Figure 4.4. If $\mathtt{d}$ can take any value of the alphabet $\mathcal{A} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$, this grammar produces integers of arbitrary length. A possible[8] syntax tree for 9487 is displayed in Figure 4.5. To ease the following discussion, the tree nodes of the same kind are numbered.

If we now want to get the digit sum of an integer produced by that grammar, we can go through the syntax tree from the bottom to the top: from the nodes resulting in a digit, we store that digit. For every node higher in the syntax tree, we take the values of its children and sum them up to a new value. Then the values of the start node is the digit sum of the integer.

To formalise this process we define the synthesized attribute *ds*. This attribute takes values on each node and these values depend on the production rules of the grammar. For each production rule, we define an *attribute rule*. These rules are displayed at the right side of Figure 4.4. We denote the attribute value for a node $\mathtt{n}$ by $\mathtt{n}.ds$. When we use a node resulting in an alphabet

[8] Note that there are several options to form a tree for 9487, attribute *ds* also computes the digit sum if one of these is chosen.

symbol of the grammar (in our example $\mathtt{d}$) in an attribute rule, that symbol refers its alphabet symbol. Going through the syntax tree from the bottom to the top, we get for node $\mathtt{n}^4$ by attribute rule $\mathtt{n}.ds \leftarrow \mathtt{d}$ the value $\mathtt{n}^4.ds = 9$. The same rule is used to determine $\mathtt{n}^5.ds$ to $\mathtt{n}^7.ds$ whose values are displayed in Figure 4.5. On the next higher level, the attribute value of each node is composed by taking the sum of the attribute values of the direct descendants. This is done by the rule $\mathtt{n}.ds \leftarrow \mathtt{n}_1.ds + \mathtt{n}_2.ds$. We get $\mathtt{n}^2.ds = 13$ and $\mathtt{n}^3.ds = 15$. The same rule computes one level higher for $\mathtt{n}^1$ the value $\mathtt{n}^1.ds = 28$ which is then passed to the start node $\mathtt{s}$ by the rule $\mathtt{s}.ds \leftarrow \mathtt{n}.ds$. And 28 is indeed the digit sum of 9487.

4.3.2 Terminology

After our example, we now provide a more formal introduction to attribute grammars and fix the terminology we use. Attribute grammars are defined on top of context free grammars $G = \langle N, \mathcal{A}, P, S \rangle$, with N the set of non-terminal symbols, $\mathcal{A}$ the alphabet or set of terminal symbols, P the set of production rules and $S \in N$ a start symbol. The set P for the context-free grammar above is displayed in Figure 4.4. This figure also contains all non terminal symbols N. The start symbols is $\mathtt{s}$.

For each non-terminal symbol $X \in N$ of the grammar we now define a set of attributes $A(X)$. Each of these attributes either belongs to the set of *inherited* attributes $I(X)$, or to the set of *synthesized* attributes $S(X)$. We assume these two sets of attributes to be disjoint. In our example above we have for each node X: $I(X) = \emptyset$, $S(X) = \{ds\}$, and $A(X) = \emptyset \cup \{ds\} = \{ds\}$.

To take information, the attributes have assigned values which depend on the production rules they occur in.

Definition 17 (Attribute Occurrence). *Let $G = \langle N, \mathcal{A}, P, S \rangle$ be a context-free grammar and $A := \bigcup_{X \in N} A(X)$ a set of attributes defined on N. Let $p \in P$ be a production rule of the form*

$$X_0 ::= X_1 \cdots X_n \text{ with } n \geq 1,$$

- *We say that p has an* attribute occurrence $X_i.a$ *for the attribute a if $a \in A(X_i)$ for some i with $0 \leq i \leq n$.*
- *We say that p has a* left-side occurrence $X_0.a$ *of the attribute a, if $a \in A(X_0)$.*
- *We say that p has a* right-side occurrence $X_k.a$ *of the attribute a, if $a \in A(X_k)$ for some k with $1 \leq k \leq n$.*

If we consider another attribute for the grammar in Figure 4.4, which is defined only for the node d, the attribute *at*, then the rule

$$\texttt{n ::= d}$$

has a *right-side occurrence* d.*at* of the attribute *at* while the rule

$$\texttt{n ::= } n_1 n_2$$

does not have any occurrence of *at*.

These occurrences $X_i.a$ now take values, so-called *attribute values*, which are defined by *attribute rules*. These have the form

$$X_i.a \leftarrow f(y_1 \dots y_n)$$

where f is a function and $y_1 \dots y_n$ are other attribute values. An example from above for such an attribute rule is

$$\texttt{n}.ds \leftarrow n_1.ds + n_2.ds$$

which determines the first occurrence n.*ds* of the attribute *ds* in the production rule

$$\texttt{n ::= } n_1 n_2$$

We denote the set of all attribute rules for a production rule $p \in P$ by $R(p)$. Which exact attribute values can be taken into account when calculating a new attribute value depends on the kind of attribute:

Definition 18 (Attribute Grammar). *Let $G = \langle N, \mathcal{A}, P, S \rangle$ be a context-free grammar. For every element $X \in N$ let $A(X) = I(X) \cup S(X)$ be a finite set of attributes with $I(X) \cap S(X) = \emptyset$. Let $A = \bigcup_{X \in \mathcal{A} \cup N} A(X)$ be the set of all these attributes. Let $R = \bigcup_{p \in P} R(p)$ be a finite set of attribute rules. We call*

$$AG = \langle G, A, R \rangle$$

an Attribute Grammar *if for each production rule $p \in P$ of the form X_0 ::= $X_1 \dots X_n$ the following holds:*

- *for each left-side occurrence $X_0.a$ of a synthesised attribute $a \in S(X_0)$ there exists* exactly one *attribute rule $(X_0.a \leftarrow f(y_1, \dots, y_k)) \in R(p)$ where f is a function and $y_i \in A(X_0) \cup \dots \cup A(X_n)$ for all $1 \leq i \leq k$.*
- *for each right-side occurrence $X_i.a$, $1 \leq i \leq n$, of an inherited attribute $a \in I(X_i)$ there exists* exactly one *attribute rule $(X_i.a \leftarrow f(y_1, \dots, y_k)) \in R(p)$ where f is a function and $y_j \in A(X_0) \cup \dots \cup A(X_n)$ for all $1 \leq j \leq k$.*

name	type	defined for	purpose
eq	syn	N	existentials
v_1	syn	N	universals in Cwm
v_2	syn	N	universals in Cwm
s	inh	$\{\mathtt{f}, \mathtt{t}, \mathtt{e}, \mathtt{k}\}$	universals in Cwm
q	inh	$\{\mathtt{f}\}$	universals in Cwm
u	syn	N	universals in EYE
m_c	syn	N	translation Cwm
m_e	syn	N	translation EYE

Figure 4.6: Overview of all attributes defined in the N3-attribute grammar. The type can be *syn* for synthesized or *inh* for inherited.

In terms of a syntax tree generated by the grammar synthesized and inherited attributes have exactly the properties mentioned above: inherited attributes pass information *down* in the syntax tree, synthesized attributes pass information *upwards*. We assume the values of synthesized attributes for terminal symbols to be defined externally. The start symbol cannot take an inherited attribute.[9]

4.4 From N3 syntax to N3 Core Logic

After the definition of attribute grammars in the previous section, we now apply this concept to formalise the different interpretations of implicit quantification for N3Logic. The context-free grammar of N3 is given in Figure 4.2 which also contains all symbols of N. The alphabet is given in Definition 16. The start symbol is $\mathtt{s}$.

On top of this grammar we now define an attribute grammar in order to be able to produce the two different translations of an N3 formula in N3 Core Logic, one according to Cwm and the other according to EYE. As a first step we provide an overview of the attributes we define, state whether they are inherited or synthesized and list for which nodes $X \in N$ they are defined. This information is displayed in Figure 4.6. For each of these attributes we also indicate the purpose or context they are used for. This context can be

[9]The attentive reader might have noticed another difference in the structure of the context-free grammar of N3 Core Logic (Figure 4.1) and of N3 (Figure 4.2): the latter has an extra start symbol. This has a technical reason: the attribute grammar we define in the following sections makes use of inherited attributes defined on the symbol $\mathtt{f}$. Such attributes cannot be defined on a start symbol.

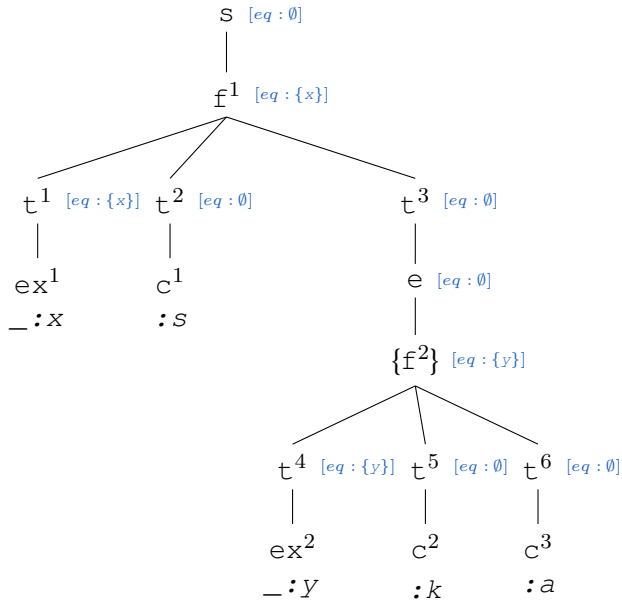


Figure 4.7: Syntax tree for Formula 4.2 with *values* for the attribute eq.

grouped in four parts: the collection of the variables which are existentially quantified under a (sub-)formula (1), the collection of the universal variables quantified under a (sub-)formula – once for Cwm (2) and once for EYE (3) – and attributes to construct the concrete N3 Core Logic expressions reflecting the reasoners' interpretations (4). We go through these four groups one by one, explain the need for the different attributes and provide their definition. We start with attribute eq, which is used for existential scoping.

4.4.1 Existentials

As we have seen in Section 3.1.1 the scope on an implicitly existentially quantified variable is the *direct* formula it occurs in. The concept of a *direct formula* has been further explained above: the direct formula is either the next formula in curly brackets surrounding the existential variable, or – in case such a formula does not exist – the formula as a whole. To recall the idea, consider the following formula:[10]

```
_:x :s {_:y :k :a.}.
```
(4.2)

[10] Note that structure-wise this formula follows the same pattern as Formula 3.1. The names of constants are only shortened to make the presentation of the formula easier.

production rule		attribute rule
s ::=	f	s.eq $\leftarrow \emptyset$
f ::=	$t_1 t_2 t_3$.	f.eq $\leftarrow t_1$.eq $\cup\, t_2$.eq $\cup\, t_3$.eq
	e_1=>e_2.	f.eq $\leftarrow e_1$.eq $\cup\, e_2$.eq
	$f_1 f_2$	f.eq $\leftarrow f_1$.eq $\cup\, f_2$.eq
t ::=	uv	t.eq $\leftarrow \emptyset$
	ex	t.eq $\leftarrow \{$ex$\}$
	c	t.eq $\leftarrow \emptyset$
	e	t.eq $\leftarrow$ e.eq
	(k)	t.eq $\leftarrow$ k.eq
	()	t.eq $\leftarrow \emptyset$
k ::=	t	k.eq $\leftarrow$ t.eq
	t k_1	k.eq $\leftarrow$ t.eq $\cup\, k_1$.eq
e ::=	{f}	e.eq $\leftarrow \emptyset$
	{}	e.eq $\leftarrow \emptyset$
	false	e.eq $\leftarrow \emptyset$

Figure 4.8: Attribute rules for the synthesized attribute eq (right) and their corresponding production rules (left) from the N3 grammar (Figure 4.2).

The direct formula of variable _:y is the sub-formula _:y :k :a. The direct formula of _:x is the formula as a whole. Translated into N3 Core Logic the formula means:

$$\exists x.x\ s\ <\exists y.y\ k\ a>. \tag{4.2'}$$

In Section 4.2 we discussed the idea of using the syntax tree to find the direct formula. If we go upwards in the syntax tree from the occurrence of an existential variable to the next higher node e or s, the child formula f of that node is the existential's direct formula which thus carries its quantifier. The syntax tree for our example, Formula 4.2, is displayed in Figure 4.7. Here node f^2 is such a node for _:y and node f^1 for _:x.

Attribute eq is now used in this context to keep track of all existential variables which need to be quantified. If we do a direct translation of the symbols of the alphabet as mentioned at the end of Section 4.2 and add existential quantifiers whenever we encounter a direct formula, the attribute carries for each node the set of existential variables which are free (see Definition 7) under that node and need to be bound when new existential quantifiers are added.

Following that idea, we define the attribute rules for eq. As eq is synthesised we need to state one attribute rule for each left-side occurrence of an attribute on a production rule of Figure 4.2. As the attribute is defined for all nodes, we need one attribute rule for each production rule. These attribute rules are

displayed in Figure 4.8 (right) next to their corresponding production rules (left). To illustrate how this attribute works, we added the attribute values for each node to the syntax tree in Figure 4.7. We explain these rules by going through the tree beginning at the bottom.

The production rule `t ::= ex` results in an existential variable and this variable is not quantified under that node. Therefore, the corresponding attribute rule `t`.eq $\leftarrow$ {`ex`} collects this variable in a singleton set (values $\texttt{t}^1$.eq and $\texttt{t}^4$.eq). The attribute rules for other production rules directly resulting in only symbols of the alphabet do not pass any values since there are no free existential variables occurring under them. For the applications of `t ::= c` the attribute rule `t` $\leftarrow \emptyset$ assigns the empty set to the nodes $\texttt{t}^2$, $\texttt{t}^5$ and $\texttt{t}^6$. Most other rules now pass the variables from the descendant nodes up to the parents. The attribute value of a formula node `f` is the union of its descendants' values. For $\texttt{f}^2$ that is the set {`_:y`}. The only exceptions for this behaviour of passing the variables upwards can be found on the attribute rules for the production rules `s ::= f` and `e ::= {f}`. As discussed above, the child formulas `f` of `e` or `s` are *direct formulas*. All free existential variables occurring under such direct formulas get bound on these formulas. The attribute rules for `s ::= f` and `e ::= {f}` thus do not pass any variables upwards. The attribute value for node `e` in our syntax tree is the empty set. This value is again passed upwards via the attribute rule `t`.eq $\leftarrow$ `e`.eq and can the be used to determine the attribute value for $\texttt{f}_1$ which is again the union of all its direct descendants' values ($\texttt{f}^2$.eq = {`_:x`}). This node is again a direct formula, it is the child node of the node `s`. All existential variables are thus bound on $\texttt{f}^2$, the attribute rule does not pass existential variables upwards.

The attribute value of eq is always the set of free existential variables occurring under a node. For the formulas $\texttt{f}^1$ and $\texttt{f}^2$ these are the sets {`_:x`} and {`_:y`}, respectively. These are exactly the variables existentially quantified on these formulas as can be seen in Formula 4.2'.

4.4.2 Universals in Cwm

The interpretations of N3's implicit universal quantification differs between reasoners. With a few exceptions[11], which we also discuss in Appendix A, Cwm implements interpretation *parent*$_c$ of the concept *parent formula* from the W3C team submission while for EYE the *parent formula* is the top formula (*parent*$_e$). In this section we explain the concept of *parent*$_c$ in more detail

[11]The developer of Cwm, Tim Berners-Lee, confirmed that these exceptions were not intended and rather need to be seen as mistakes in the implementation.

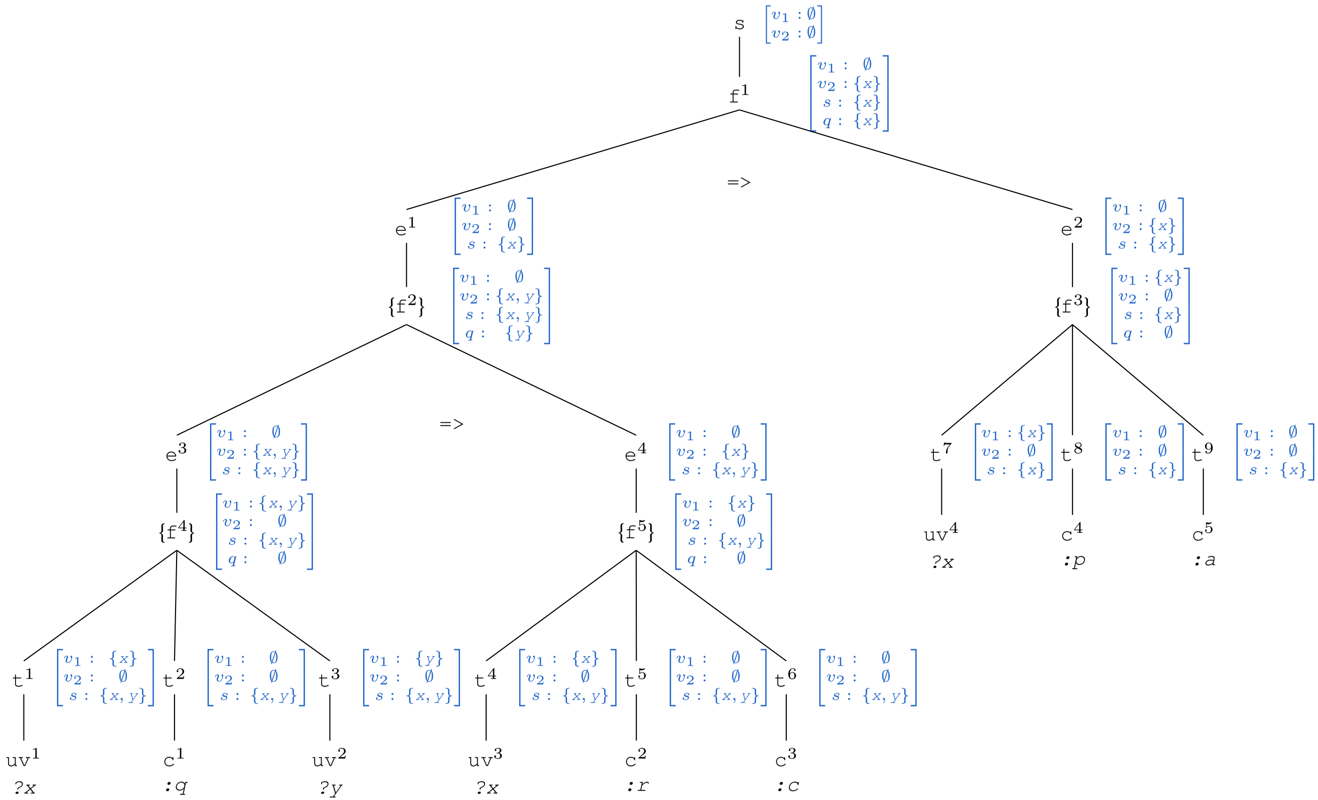


Figure 4.9: Syntax tree of Formula 4.3 with *attribute values* for the attributes v_1, v_2, s and q.

CFG production rules		Synthesized attributes: rules for v_1	Synthesized attributes: rules for v_2	Inherited attributes: rules for s	Inherited attributes: rules for q
s ::=	f	$\mathtt{s}.v_1 \leftarrow \emptyset$	$\mathtt{s}.v_2 \leftarrow \mathtt{f}.v_1$	$\mathtt{f}.s \leftarrow \mathtt{f}.v_1 \cup \mathtt{f}.v_2$	$\mathtt{f}.q \leftarrow \mathtt{f}.v_1 \cup \mathtt{f}.v_2$
f ::=	$\mathtt{t}_1\mathtt{t}_2\mathtt{t}_3$.	$\mathtt{f}.v_1 \leftarrow \mathtt{t}_1.v_1 \cup \mathtt{t}_2.v_1 \cup \mathtt{t}_3.v_1$	$\mathtt{f}.v_2 \leftarrow \mathtt{t}_1.v_2 \cup \mathtt{t}_2.v_2 \cup \mathtt{t}_3.v_2$	$\mathtt{t}_i.s \leftarrow \mathtt{f}.s$	
	$\mathtt{e}_1$=>$\mathtt{e}_2$.	$\mathtt{f}.v_1 \leftarrow \mathtt{e}_1.v_1 \cup \mathtt{e}_2.v_1$	$\mathtt{f}.v_2 \leftarrow \mathtt{e}_1.v_2 \cup \mathtt{e}_2.v_2$	$\mathtt{e}_i.s \leftarrow \mathtt{f}.s$	
	$\mathtt{f}_1\mathtt{f}_2$	$\mathtt{f}.v_1 \leftarrow \mathtt{f}_1.v_1 \cup \mathtt{f}_2.v_1$	$\mathtt{f}.v_2 \leftarrow \mathtt{f}_1.v_2 \cup \mathtt{f}_2.v_2$	$\mathtt{f}_i.s \leftarrow \mathtt{f}.s$	$\mathtt{f}_i.q \leftarrow \emptyset$
t ::=	uv	$\mathtt{t}.v_1 \leftarrow \{\mathtt{uv}\}$	$\mathtt{t}.v_2 \leftarrow \emptyset$		
	ex	$\mathtt{t}.v_1 \leftarrow \emptyset$	$\mathtt{t}.v_2 \leftarrow \emptyset$		
	c	$\mathtt{t}.v_1 \leftarrow \emptyset$	$\mathtt{t}.v_2 \leftarrow \emptyset$		
	e	$\mathtt{t}.v_1 \leftarrow \mathtt{e}.v_1$	$\mathtt{t}.v_2 \leftarrow \mathtt{e}.v_2$	$\mathtt{e}.s \leftarrow \mathtt{t}.s$	
	(k)	$\mathtt{t}.v_1 \leftarrow \mathtt{k}.v_1$	$\mathtt{t}.v_2 \leftarrow \mathtt{k}.v_2$	$\mathtt{k}.s \leftarrow \mathtt{t}.s$	
	()	$\mathtt{t}.v_1 \leftarrow \emptyset$	$\mathtt{t}.v_2 \leftarrow \emptyset$		
k ::=	t	$\mathtt{k}.v_1 \leftarrow \mathtt{t}.v_1$	$\mathtt{k}.v_2 \leftarrow \mathtt{t}.v_2$	$\mathtt{t}.s \leftarrow \mathtt{k}.s$	
	t $\mathtt{k}_1$	$\mathtt{k}.v_1 \leftarrow \mathtt{t}.v_1 \cup \mathtt{k}_1.v_1$	$\mathtt{k}.v_2 \leftarrow \mathtt{t}.v_2 \cup \mathtt{k}_1.v_2$	$\mathtt{t}.s \leftarrow \mathtt{k}.s$ $\mathtt{k}_1.s \leftarrow \mathtt{k}.s$	
e ::=	{f}	$\mathtt{e}.v_1 \leftarrow \emptyset$	$\mathtt{e}.v_2 \leftarrow \mathtt{f}.v_1$	$\mathtt{f}.s \leftarrow \mathtt{e}.s \cup \mathtt{f}.v_2$	$\mathtt{f}.q \leftarrow \mathtt{f}.v_2 \setminus \mathtt{e}.s$
	{}	$\mathtt{e}.v_1 \leftarrow \emptyset$	$\mathtt{e}.v_2 \leftarrow \emptyset$		
	false	$\mathtt{e}.v_1 \leftarrow \emptyset$	$\mathtt{e}.v_2 \leftarrow \emptyset$		

Figure 4.10: Attribute rules for universal quantification in Cwm defined on the context-free grammar of N3 (Figure 4.2).

and define attributes to place universal quantifiers according to this concept. The details on the interpretation following to EYE are discussed in the next section.

In the previous sections we have seen that in contrast to implicitly existentially quantified variables, universals are – according to the W3C team submission – not quantified on the *direct formula* but on the *parent*. This concept has (at least) two conflicting interpretations which have been further explained in Section 4.2. One of them is parent$_c$: The parent$_c$ is the direct formula of the direct formula or – in terms of the syntax tree – the second descendant `f` of a node `s` or `e` we find when going up the syntax tree from the bottom to the top. In the following formula, for example,

```
{{?x :q ?y.} => {?x :r :c.}.} =>{?x :p :a.}.
```
(4.3)

the parent$_c$ of the last occurrence of `?x` is the formula as a whole, the parent$_c$ of `?y` is the subformula

```
{?x :q ?y.} => {?x :r :c.}.
```

or, if we take a look into the formula's syntax tree displayed in Figure 4.9, the parent$_c$ of the last `?x` (uv^4) is f^1 and the parent$_c$ of `?y` (uv^2) is f^2. Therefore formula f^1 carries a universal quantifier for `?x` and f^2 carries a universal quantifier for `?y`. The first universal quantifier also covers the other occurrences of `?x` since the parent$_c$ formula of these other occurrences, namely f^2, is already in scope of this quantifier. Formula 4.3 means:

$$\forall \texttt{x}.\quad \texttt{<}\forall \texttt{y}.\quad \texttt{<x q y>} \rightarrow \texttt{<x r c>} \texttt{ >} \rightarrow \texttt{<x p a>} \qquad (4.3')$$

Similarly to the example of existential quantification, we define attributes which pass universal variables occurring under a parent$_c$ formula up to that parent$_c$ formula. For this purpose, we use two attributes whose values are as follows:

v_1 the set of universal variables occurring as direct components under a node.

v_2 the set of universal variables occurring as direct components of direct components (parent level).

The first attribute v_1 is used to pass universal attributes up to their direct formulas, the second attribute v_2 is used to further pass these variables from the direct formulas to the parent$_c$ formulas. The attributes are again synthesized and we have one attribute rule for each production rule. These

attribute rules are displayed in the second and third column of the table in Figure 4.10. The first column of the table shows the corresponding production rules. To better explain the attributes, we apply them to the syntax tree of Formula 4.3. The tree and the corresponding attribute values are displayed in Figure 4.9.

Passing Variables to their Direct Formula

Attribute v_1 works the exact same way as attribute eq with the only difference that at term level the universals get captured in a singleton set ($\texttt{t}.v_1 \leftarrow \{\texttt{uv}\}$) instead of the existentials. These universals are then passed upwards to their parent$_c$ formulas. In our example, the formulas $\texttt{f}^1$ and $\texttt{f}^2$ do not have universal variables as direct components, $\texttt{f}^3$ has `?x`, $\texttt{f}^4$ has `?x` and `?y`, and formula $\texttt{f}^5$ has `?x`.

Passing Variables to their Parent Formula

Attribute v_2 now passes the universal variables, which are collected under their direct formula using v_1 to their parent$_c$ formula. This attribute works in a similar way as the previous one: at the level where the universal variables of a direct formula `f` under a node `e` are found, the rule for the synthesized attribute takes these values gathered by the attribute v_1 and passes them upwards till the next formula being direct descendant of a node `s` or `e` is encountered. This formula is then the parent$_c$.

We display the rules for attribute v_2 in the third column of Figure 4.10. We explain the rules by going through the syntax tree in Figure 4.9 from the bottom to the top. The rules only resulting in symbols of the alphabet cannot have direct components. The attribute rule $\texttt{t}.v_2 \leftarrow \emptyset$ for the production rules `t ::= uv` and `t ::= c` assign the empty set to the term nodes $\texttt{t}^1$ to $\texttt{t}^9$. These values are passed upwards via the attribute rule $\texttt{f} \leftarrow \texttt{t}_1.v_2 \cup \texttt{t}_2.v_2 \cup \texttt{t}_3.v_2$ on production rule `f ::=` $\texttt{t}_t\texttt{t}_2\texttt{t}_3$, the attribute values for $\texttt{f}^3$, $\texttt{f}^4$ and $\texttt{f}^5$ are again the empty set. The attribute rule for the next higher level, the production `e ::= {f}` is more interesting. The variables which occur as direct components on the formula node `f` and are captured by the attribute v_1 are passed as children of the next parent$_c$ formula to the node `e` via the attribute rule $\texttt{e}.v_2 \leftarrow \texttt{f}.v_1$. We get: $\texttt{e}^2.v_2 = \{\texttt{?x}\}$, $\texttt{e}^3.v_2 = \{\texttt{?x}, \texttt{?y}\}$ and $\texttt{e}^4.v_2 = \{\texttt{?x}\}$. For $\texttt{f}^2$ these values are passed upwards via the attribute rule $\texttt{f} \leftarrow \texttt{e}_1 \cup \texttt{e}_2$. As $\texttt{f}^2$ is a direct descendant of the node $\texttt{e}^1$, it is the parent$_c$ formula of the variables mentioned above. These variables are not passed further through. Instead, the attribute value for $\texttt{e}^1$ is again taken from the attribute v_1 (via $\texttt{e}.v_2 \leftarrow \texttt{f}.v_1$), which in this case is the empty set. With these

values we can determine the attribute value for $\mathtt{f}^1$ (via $\mathtt{f} \leftarrow \mathtt{e}_1 \cup \mathtt{e}_2$) which in this case is the singleton set only containing the variable `?x`. The attribute value for `s` is the empty set.

For formulas which are direct descendants of a node `s` or `e` the value is now the set of universal variables for which the formula is a parent$_c$: for $\mathtt{f}^1$ that is $\{\mathtt{?x}\}$, for $\mathtt{f}^2$ it is $\{\mathtt{?x}, \mathtt{?y}\}$, and for $\mathtt{f}^3$, $\mathtt{f}^4$ and $\mathtt{f}^5$ the value is the empty set since these formulas are not parent$_c$ formulas.

Passing Scoped Variables to the Descendants

When we introduced the concept of a parent formula according to Cwm in Section 3.1, we already discussed an important exception from the rule that the quantifier for every universal variable is always the parent$_c$ formula: If the parent$_c$ formula of a universal variable is already in scope of a quantifier for a variable with the same name, then this quantifier also counts for the variable of the child formula (this was explained on Formula 3.7). Such a behaviour can also be observed on our example formula: Even though formula $\mathtt{f}^2$ is the parent$_c$ of the first and second occurrence of `?x`, Interpretation 4.3' only contains one universal quantifier for all occurrences of `?x` in Formula 4.3 and that is on formula $\mathtt{f}^1$ which is the parent$_c$ formula of the last occurrence of `?x`.

Before we define attribute s below which takes care of this behaviour, we consider one additional example to fully understand the consequences of the exception. If we add a conjunct

```
:s :p {:a :b ?y.}.
```

to our Formula 4.3 this addition changes the scoping. The formula

```
{{?x :q ?y.} => {?x :r :c.}.}=>{?x :p :a.}.
                                :s :p {:a :b ?y.}.   (4.4)
```

means

```
∀x.∀y.(< <x q y> → <x r c> > → <x p a>.
                                 s p <a b y>.  )   (4.4')
```

Note that the quantifier for `y` which is nested in the interpretation of Formula 4.3 is on top level in the interpretation of Formula 4.4. This is the case because the conjunction, i.e. the formula as a whole, is the parent$_c$ formula of the second occurrence of `?y` and thus carries a quantifier which also covers the nested occurrence of `?y`.

Attribute s keeps track of this behaviour. The value of s is for each node the set of variables which are universally quantified on the node, either by a quantifier on the node itself, or by a quantifier on a higher level. This kind of information needs to be passed downwards in the syntax tree therefore attribute s is inherited. As the value of s is only relevant for potential parent$_c$ formulas, we only define the attribute for the node `f` and all nodes which can occur above `f` in the syntax tree. These are the nodes `f`, `t`, `e` and `k`. As s is inherited, this time we need to define an attribute rule for each *right-side* occurrence of s on a production rule. These rules are displayed in the fourth column of Figure 4.10. We explain them by going through the syntax tree in Figure 4.9.

For the occurrence of `f` in the rule `s ::= f` we take all the variables of which `f` is the parent$_c$ formula since these are quantified on that highest level. Via the attribute rule $\texttt{f}.c \leftarrow \texttt{f}.v_1 \cup \texttt{f}.v_2$[12] we get: $\texttt{f}^1.s = \{\texttt{?x}\}$. For the nodes $\texttt{e}^1$ and $\texttt{e}^2$ this information is passed downwards, via $\texttt{e}_1.s \leftarrow \texttt{f}.s$ and $\texttt{e}_2.s \leftarrow \texttt{f}.s$ we get the value $\{\texttt{?x}\}$ for both nodes. On the production rule `e ::= {f}` the nodes `e` can now again be the direct ancestor of a parent$_c$ formula `f`. The set of variables $\texttt{f}.v_2$ the formula `f` is parent$_c$ of are either quantified on that same formula or they are already quantified beforehand in which case they are already present in $\texttt{e}.s$. In both cases the union of these values covers all variables quantified at that point, we have as attribute rule $\texttt{f}.s \leftarrow \texttt{e}.s \cup \texttt{f}.v_2$. We get $\texttt{f}^2.s = \{\texttt{?x}, \texttt{?y}\}$ and $\texttt{f}^3.s = \{\texttt{?x}\}$. This information is passed further downwards via the attribute rules $\texttt{t}_{1,2,3}.s \leftarrow \texttt{e}.s$. for the production rule $\texttt{f ::= t}_1\texttt{t}_2\texttt{t}_3$ and via the rules $\texttt{e}_{1,2}.s \leftarrow \texttt{f}.s$ for $\texttt{f ::= e}_1\texttt{e}_2$, we get $\texttt{t}^{7,8}.s = \{\texttt{?x}\}$ and $\texttt{e}^{3,4}.s = \{\texttt{?x}, \texttt{?y}\}$. The descendants of these last two expressions are not parent$_c$ formulas, the attribute thus simply passes their values down to $\texttt{f}^4$ and $\texttt{f}^5$ which then get passed further to the nodes $\texttt{t}^1$ to $\texttt{t}^6$. For all nodes in our syntax tree for which the attribute s is defined we now capture the set of variables which are scoped under that node.

Determining the Universally Quantified Variables for a Formula

In order to use the information captured by the previous attributes one step is missing: we still need to determine the exact set of universal variables quantified on a specific formula. For this we define attribute q: for each

[12] Note that here attribute s does not only collect all universal variables `f` is parent$_c$ of but also the universals occurring as direct components. The reason for that is that these variables cannot be passed further upwards. In the original N3 specification, implicitly universally quantified variables on top level are allowed according to the grammar, but their meaning is not covered in the semantics Cwm applies. We handle this problem by adding quantifiers for them to the top level.

formula node f the value of q is the set of universal variables for which f carries a quantifier in the translation. We define q as an inherited attribute on f. The rules for q are listed in the last column of Figure 4.10.

If f occurs under the starting node, all variables it is parent$_c$ of are quantified on that formula. For the rule s ::= f the value of the attribute is the same as the value of v_2: $\mathtt{f}.q = \mathtt{f}.v_1 \cup \mathtt{f}.v_2$.[13] In our syntax tree in Figure 4.9 we get $\mathtt{f}^1.q = \{\mathtt{?x}\}$. If a formula is only a conjunct of a conjunction, it does not carry any universal quantifier (remember for example Formula 4.4). Therefore the attribute value assigned to $\mathtt{f}_1$ and $\mathtt{f}_2$ on the production rule f ::= $\mathtt{f}_1\mathtt{f}_2$ is the empty set: $\mathtt{f}_{1,2} \leftarrow \emptyset$. For the third rule with a right-side occurrence of f, e ::= {f}, we take the values of the attributes v_2 and q into account: the value of s on the node e contains all the universal variables which are already quantified on that or a higher node. The value of v_2 on f contains all variables which need to be quantified above or at f since f is their parent node. The set of universal variables for which f needs to carry a quantifier is the difference of these two sets: $\mathtt{f}.q \leftarrow \mathtt{f}.v_2 \setminus \mathtt{e}.s$. For the node $\mathtt{f}^2$ in our syntax tree we get $\mathtt{f}^2.q = \{\mathtt{?x}, \mathtt{?y}\} \setminus \{\mathtt{?x}\} = \{\mathtt{?y}\}$. For the remaining nodes $\mathtt{f}^3$, $\mathtt{f}^4$ and $\mathtt{f}^5$ the attribute value is the empty set. And it is indeed the case that the formula $\mathtt{f}^1$ carries a universal quantifier for ?x and formula $\mathtt{f}^3$ for ?y. The only thing which still needs to be done to obtain Cwm's interpretation of the formula is to construct the concrete translation. The attribute to perform this task is defined below in Section 4.4.4.

4.4.3 Universals in EYE

After having defined several attributes to handle implicit universal quantification according to Cwm in the previous section, we do the same for the reasoner EYE in this section. EYE understands the term *parent formula* from the W3C team submission as the top formula. Universal variables are thus for EYE always quantified on the top level. Formula 4.3 means according to EYE:

```
∀x.∀y. <<x q y>>   →   <x r c>>   →   <x p a>   (4.3")
```

To deal with universal quantification we therefore only need one attribute that passes all universal variables occurring in a formula to the top level. We define the synthesized attribute u for all nodes of the grammar. The value of u for a node is always the set of universal variables occurring anywhere under that node. The attribute rules for u are displayed in Figure 4.11. As the

[13] As above, this attribute does not only carry the universal variables f is parent$_c$ of but also those which occur directly in the formula.

production rule		attribute rule
s ::=	f	$\mathtt{s}.u \leftarrow \mathtt{f}.u$
f ::=	$\mathtt{t}_1\mathtt{t}_2\mathtt{t}_3.$	$\mathtt{f}.u \leftarrow \mathtt{t}_1.u \cup \mathtt{t}_2.u \cup \mathtt{t}_3.u$
	$\mathtt{e}_1\mathtt{=>}\mathtt{e}_2.$	$\mathtt{f}.u \leftarrow \mathtt{e}_1.u \cup \mathtt{e}_2.u$
	$\mathtt{f}_1\mathtt{f}_2$	$\mathtt{f}.u \leftarrow \mathtt{f}_1.u \cup \mathtt{f}_2.u$
t ::=	uv	$\mathtt{t}.u \leftarrow \{\mathtt{uv}\}$
	ex	$\mathtt{t}.u \leftarrow \emptyset$
	c	$\mathtt{t}.u \leftarrow \emptyset$
	e	$\mathtt{t}.u \leftarrow \mathtt{e}.u$
	(k)	$\mathtt{t}.u \leftarrow \mathtt{k}.u$
	()	$\mathtt{t}.u \leftarrow \emptyset$
k ::=	t	$\mathtt{k}.u \leftarrow \mathtt{t}.u$
	t $\mathtt{k}_1$	$\mathtt{k}.u \leftarrow \mathtt{t}.u \cup \mathtt{k}_1.u$
e ::=	{f}	$\mathtt{e}.u \leftarrow \mathtt{f}.u$
	{}	$\mathtt{e}.u \leftarrow \emptyset$
	false	$\mathtt{e}.u \leftarrow \emptyset$

Figure 4.11: Attribute rules for the synthesized attribute u (right) and their corresponding production rules (left) from the N3 grammar (Figure 4.2).

calculation of u's attribute values is rather simple, we do not discuss these rules in detail and only capture for further considerations the value of the top formula $\mathtt{f}^1.u = \{\mathtt{?x}, \mathtt{?y}\}$.

4.4.4 Generation of the translation

In order to obtain the different translations from N3 to N3 Core Logic, one last step is needed: the N3 Core Logic formulas need to be generated. We use synthesized attributes to perform this task. To clarify the difference between the signs used in the core formula produced and the logical symbols we use to describe this production, we underline all terminal symbols belonging to N3 Core Logic.

We start by defining the following auxiliary functions which add quantifiers to any set of symbols of the alphabet:

Let $ex : 2^{\mathcal{A}} \rightarrow 2^{\mathcal{A}^*}$ be defined as:

$$ex(V) := \{\underline{\exists v_1.} \ldots \underline{\exists v_n.} | v_i \in V; 1 \leq i \leq n = |V|; i \neq j \Rightarrow v_1 \neq v_j\}.$$

Let $uv : 2^{\mathcal{A}} \rightarrow 2^{\mathcal{A}^*}$ be defined as:

$$uv(V) := \{\underline{\forall v_1.} \ldots \underline{\forall v_n.} | v_i \in V; 1 \leq i \leq n = |V|; i \neq j \Rightarrow v_1 \neq v_j\}.$$

CFG production rules		Cwm rules for m_c	EYE rules for m_e
s ::=	f	$\mathtt{s}.m_c \leftarrow \dot{uv}(\mathtt{f}.q)\ \dot{ex}(\mathtt{f}.\mathrm{eq})\ \mathtt{f}.m_c$	$\mathtt{s}.m_e \leftarrow \dot{uv}(\mathtt{f}.u)\ \dot{ex}(\mathtt{f}.\mathrm{eq})\ \mathtt{f}.m_e$
f ::=	$\mathtt{t}_1\mathtt{t}_2\mathtt{t}_3$.	$\mathtt{f}.m_c \leftarrow \mathtt{t}_1.m_c\ \mathtt{t}_2.m_c\ \mathtt{t}_3.m_c$	$\mathtt{f}.m_e \leftarrow \mathtt{t}_1.m_e\ \mathtt{t}_2.m_e\ \mathtt{t}_3.m_e$
	$\mathtt{e}_1$=>$\mathtt{e}_2$.	$\mathtt{f}.m_c \leftarrow \mathtt{e}_1.m_c \underline{\rightarrow} \mathtt{e}_2.m_c$	$\mathtt{f}.m_e \leftarrow \mathtt{e}_1.m_e \underline{\rightarrow} \mathtt{e}_2.m_e$
	$\mathtt{f}_1\mathtt{f}_2$	$\mathtt{f}.m_c \leftarrow \mathtt{f}_1.m_c\ \mathtt{f}_2.m_c$	$\mathtt{f}.m_e \leftarrow \mathtt{f}_1.m_e\ \mathtt{f}_2.m_e$
t ::=	uv	$\mathtt{t}.m_c \leftarrow \underline{\mathtt{uv}}$	$\mathtt{t}.m_e \leftarrow \underline{\mathtt{uv}}$
	ex	$\mathtt{t}.m_c \leftarrow \underline{\mathtt{ex}}$	$\mathtt{t}.m_e \leftarrow \underline{\mathtt{ex}}$
	c	$\mathtt{t}.m_c \leftarrow \underline{\mathtt{c}}$	$\mathtt{t}.m_e \leftarrow \underline{\mathtt{c}}$
	e	$\mathtt{t}.m_c \leftarrow \mathtt{e}.m_c$	$\mathtt{t}.m_e \leftarrow \mathtt{e}.m_e$
	(k)	$\mathtt{t}.m_c \leftarrow \underline{(}\mathtt{k}.m_c\underline{)}$	$\mathtt{t}.m_e \leftarrow \underline{(}\mathtt{k}.m_e\underline{)}$
	()	$\mathtt{t}.m_c \leftarrow \underline{<>}$	$\mathtt{t}.m_e \leftarrow \underline{<>}$
k ::=	t	$\mathtt{k}.m_c \leftarrow \mathtt{t}.m_c$	$\mathtt{k}.m_e \leftarrow \mathtt{t}.m_e$
	t $\mathtt{k}_1$	$\mathtt{k}.m_c \leftarrow \mathtt{t}.m_c\ \mathtt{k}_1.m_c$	$\mathtt{k}.m_e \leftarrow \mathtt{t}.m_e\ \mathtt{k}_1.m_e$
e ::=	{f}	$\mathtt{e}.m_c \leftarrow \underline{\leq}\dot{uv}(\mathtt{f}.q)\ \dot{ex}(\mathtt{f}.\mathrm{eq})\ \mathtt{f}.m_c\underline{\geq}$	$\mathtt{e}.m_e \leftarrow \underline{\leq}\dot{ex}(\mathtt{f}.\mathrm{eq})\ \mathtt{f}.m_e\underline{\geq}$
	{}	$\mathtt{e}.m_c \leftarrow \underline{<>}$	$\mathtt{e}.m_e \leftarrow \underline{<>}$
	false	$\mathtt{e}.m_c \leftarrow \underline{\mathtt{false}}$	$\mathtt{e}.m_e \leftarrow \underline{\mathtt{false}}$

Figure 4.12: Attribute rules for constructing the translation of a formula into N3 Core Logic according to Cwm (m_c) and according to EYE (m_e).

The range of these two functions are sets, for $\{\mathtt{x}, \mathtt{y}\}$ we get for example:

$$\dot{ex}(\{\mathtt{x}, \mathtt{y}\}) = \{\underline{\exists \mathtt{x}.\exists \mathtt{y}.}, \underline{\exists \mathtt{y}.\exists \mathtt{x}.}\}.$$

To be able to use the functions to construct from a set of variables a sequence of quantified variables, we thus need a selection function. Let $select : 2^{T*} \rightarrow T^*$ be such a function which, given a set, chooses one element of that set. We use the notation $\dot{ex}$ and $\dot{uv}$ to denote $select \circ ex$ and $select \circ uv$, respectively. For the empty set the selection function returns the empty string.

With the help of these functions we can define attributes which take the signs of the N3 formula, replace them by the respective sign of N3 Core Logic, and add, where necessary, explicit quantifiers. We use two different attributes:

m_c The value of m_c is the translation of the symbols of the alphabet occurring under a node according to Cwm.

m_e The value of m_e is the translation of the symbols of the alphabet occurring under a node according to EYE.

Both attributes are synthesized and defined for all symbols of the grammar. In Figure 4.12 we display the corresponding attribute rules. In most cases these rules look very similar: constants, existentials and universals are concatenated, the symbols { and } are replaced by < and >, and => by $\rightarrow$. A

different behaviour of the attributes can only be observed at the two places where quantifiers are added, the rules `e ::= {f}` and `s ::= f`.

For the first of these production rules, the rule for the attribute m_c adds quantifiers to the translated sub-formula. To get all universally quantified variables at that level, the value of the attribute q (universal variables for a formula according to Cwm) is used. For existentially quantified variables we use the value of eq (existential variables for a formula). As explained in Section 2.5.3 the translation puts first the universal and then the existential quantification. In contrast to that, the rule for the attribute m_e only adds the existential quantifier since according to its understanding, universal quantifiers are only set in front of the top formula.

For the second of these production rules, the starting rule, attribute m_c behaves as before: universal and existential quantifiers are set using the values of q and eq. The rules for attribute m_e also add universal quantifiers at that highest level: the quantifiers for the universal variables collected using u, the attribute gathering all universal variables occurring under a formula.

With these rules the quantifiers for a formula are only added when it is sure that the formula stands on its own and is not part of a conjunction. To understand the reason for that behaviour, recall the example in Formula 4.4: there, the universal quantifier for `y` caused by the second conjunct also counted for the first and needed thus to stand in front of both formulas. In case `f` is produced using the last rule `s ::= f` we know that there are no more conjuncts for the formula and the quantifiers can be added. For each formula, the translation to N3 Core Logic generated using the attributes m_c and m_e is the attribute value of $\mathtt{s}.m_c$, respectively $\mathtt{s}.m_e$.

We finish this section by an example: we again consider the syntax tree of Formula 4.3 displayed in Figure 4.9, this time to construct the translations using the attributes m_c and m_e. To improve readability, we omit the prefixes for the constants (“`:`”) and the introducing question-marks (“`?`”) for universal variables. Going from the bottom to the top of the tree, the first steps are the same for m_c and m_e and very easy to understand: the symbols are just captured, translated where needed and then concatenated. For $\mathtt{f}^4$ we get for example:

$$\mathtt{f}^4.m_{c,e} \leftarrow \underline{\mathtt{x\ q\ y}}$$

On the next higher level, both attributes do not add quantifiers:

$$\begin{aligned} \mathtt{e}^3.m_c \leftarrow \quad & \leq\!\dot{u}v(\mathtt{f}^4.q)\dot{e}x(\mathtt{f}^4.\mathsf{eq})\underline{\mathtt{x\ q\ y>}} \\ & = \leq\!\dot{u}v(\emptyset)\dot{e}x(\emptyset)\underline{\mathtt{x\ q\ y>}} = \underline{\mathtt{<x\ q\ y>}} \end{aligned}$$

and

$$\mathtt{e}^3.m_e \leftarrow \qquad \underline{\leq \dot{ex}(\mathtt{f}^4.\mathsf{eq})\mathtt{x\ q\ y}\!>} = \underline{\mathtt{<x\ q\ y>}}$$

This is also the case for $\mathtt{e}^2$ and $\mathtt{e}^4$. As the attributes work analogously for $\mathtt{e}^1$ with the only difference that at that level the attribute rule for m_c adds a universal quantifier as the value $\mathtt{f}^2.q$ is not empty, we also omit these values and take a closer look to the attribute values at $\mathtt{s}$. For $\mathtt{f}^1$ we have

$$\mathtt{f}^1.m_c \leftarrow \underline{\mathtt{<\forall y.<x\ q\ y> \rightarrow <x\ r\ c>> \rightarrow <x\ p\ a>}}$$

And get

$$\begin{aligned}\mathtt{s}.m_c &\leftarrow \dot{uv}(\mathtt{f}^1.q)\dot{ex}(\mathtt{f}^1.\mathsf{eq})\mathtt{f}^1.m_c\\ &= \dot{uv}(\{\mathtt{x}\})\dot{ex}(\emptyset)\mathtt{f}^1.m_c\\ &= \underline{\mathtt{\forall x.<\forall y.<x\ q\ y> \rightarrow <x\ r\ c>> \rightarrow <x\ p\ a>}}\\ &= \text{Formula 4.3'}\end{aligned}$$

And with

$$\mathtt{f}^1.m_e \leftarrow \underline{\mathtt{<<x\ q\ y> \rightarrow <x\ r\ c>> \rightarrow <x\ p\ a>}}$$

we get:

$$\begin{aligned}\mathtt{s}.m_e &\leftarrow \dot{uv}(\mathtt{f}^1.u)\dot{ex}(\mathtt{f}^1.\mathsf{eq})\mathtt{f}^1.m_e\\ &= \dot{uv}(\{\mathtt{x},\mathtt{y}\})\dot{ex}(\emptyset)\mathtt{f}^1.m_e\\ &= \underline{\mathtt{\forall x.\forall y.<<x\ q\ y> \rightarrow <x\ r\ c>> \rightarrow <x\ p\ a>}}\\ &= \text{Formula 4.3"}\end{aligned}$$

The attributes deliver the expected result.

4.5 N3, N3 Core Logic and other frameworks

By defining N3 Core Logic and formalising two mappings from N3 syntax to that formalism we provided two possible definitions of N3's semantics. We want to close this chapter by putting our work into context. We first take a look to logics which influenced the definition or are related to N3 Core logic and then discuss work related to the mapping mechanism we used.

4.5.1 N3 Semantics

When defining the semantics of N3 we took several other logics into account. The most important ones – RDF and FOL – have already been discussed earlier and we therefore do not got into their particular details here. Instead, we want to focus on two important aspects of N3 Core Logic: the presence of the grounding function and the fact that it is possible to refer to formulas.

Grounding function

The idea of grounding quantified variables, instead of directly mapping them to the domain of discourse using a validation function, is inspired by Herbrand semantics [102]. In Herbrand semantics the set of all ground terms forms the domain of discourse. This is different to our approach where we still have a separated domain of discourse. The relation between first order logic with its classical Tarskian semantics and Herbrand semantics is further discussed in a companion paper to the source mentioned above [103]. Every entailment under the classical Tarskian semantics of first order logic is also true under Herbrand semantics. But there are also differences: since the existential quantifier only refers to ground terms of the language and cannot be assigned to a nameless element of the domain of discourse, it is easy to use negation to construct a counter example for the compactness theorem in Herbrand semantics. How the results of Herbrand semantics can be applied to our core logic is subject to further research.

Cited formulas

The development of N3Logic has been influenced by the Knowledge Interchange Format (KIF) [104] and the related ISO standard Common Logic (CL) [105]. These are both influenced by McCarthy's Logic of context [106]. Both support, such as N3, quantified variables in predicate position and the citation of formulas. Their mechanisms to handle the former is similar to RDF: variables get mapped to resources of the domain of discourse. If resources are used as properties, a second mapping interprets these properties. Both formalisms support universal and existential quantification. A difference between them is, that in KIF the set of variables is disjoint from the set of constants, while CL, similar to N3 for explicit quantification (see Section 3.1.3), does not distinguish between variable names and constants. As a consequence, variables cannot occur freely in CL formulas. Free variables in KIF are universally bound on top level. Similar as it is done in core logic, cited formulas in KIF and in CL are interpreted as single elements of the universe of discourse. For KIF, this is done by mapping them directly to their string representation, which in KIF needs to be included in the universe of discourse. In CL, a citation is written as a pair of a name and a text (the cited logical formula(s)). This name can then be used in different contexts. Both logics also provide a mechanism to relate the quotation to the actual formula they quote. Being out of scope for the work presented here, which focusses on implicit quantification, it is also planned to add such a mechanism to our core logic. KIF, CL, but also their predeceasing logic of contexts are the most promising sources for this extension of our formalisation.

4.5.2 Mappings from Representations to Core Logics

The approach of translating a logical representation into a well defined core logic to explain its semantics, has been inspired by the formal description of programming languages where this practice is quite common. In programming languages, normally the lambda calculus and its extensions are used as a core logic. The general idea is, for example, explained by Pierce [107] and has been implemented for several programming languages: Sulzmann et al. [108] define System F_C, an extension of the polymorphic lambda calculus System F, to provide a way to express even rather complicated constructs contained in Haskell and other functional languages, as for example generalised algebraic data types (GADTs) and associated types, in terms of a well defined logic. Next to the definition and discussion of the basic properties of that logic, the authors also show how to translate the above mentioned examples from a source language into System F_C. Igarashi et al. [109] follow a similar approach for the programming language Java. To have a logical representation of the core features of Java, they define Featherweight Java. This logic can be used to describe and prove essential properties of the programming language. They furthermore discuss how the formalism can be extended by adding generic types and methods.

In the Semantic Web context, a similar approach to the one represented in this paper can be found for the definition of SPARQL's semantics: instead of defining the semantics directly on the language itself, expressions of SPARQL are first mapped to the SPARQL algebra for which an evaluation semantics is defined.[14] Another similar approach has been followed for RDF. De Bruijn et al. [110] embed RDF into F-Logic and first order logic. In contrast to our mapping of N3 to N3 Core Logic, the embedding defined in that work is not done to define the semantics of RDF – this is defined directly – but to research its relation to other logics. The authors show that RDF can be represented in the two frameworks.

4.6 Conclusion

In this chapter we introduced N3 Core Logic a core logic for N3. N3 Core Logic contains all important features of N3 – like for example a construct to cite formulas – but only supports explicit quantification. We provided a clear model theory for the logic and discussed its relation to RDF: N3 Core Logic is syntactically and semantically compatible with RDF.

[14] https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#sparqlDefinition

We furthermore defined an attribute grammar to map from N3 syntax to N3 Core Logic following the interpretations assumed by the reasoners Cwm and EYE. By that, we defined the semantics of N3 in two possible ways and made it possible to compare the interpretations of single formulas. Our formalism facilitates the discussion of different possibilities to understand N3 and thereby contributes to finding a common agreement.

Having identified the differences in the different implementations of N3 reasoners and formalised them, we are now interested in the practical consequences of these differences. Can we observe contradicting interpretations for files used in practical applications or is the problem we found of rather theoretical nature? This is the question we are going to answer in the following chapter.

This chapter was partly based on the publications:

D. Arndt, T. Schrijvers, J. De Roo, R. Verborgh, Implicit quantification made explicit: How to interpret blank nodes and universal variables in Notation3 Logic, Journal of Web Semantics 58 (2019) 100501. doi:10.1016/j.websem.2019.04.001.
URL https://authors.elsevier.com/c/1ZWg55bAYUaMkH

D. Arndt, R. Verborgh, J. De Roo, H. Sun, E. Mannens, R. Van de Walle, Semantics of Notation3 logic: A solution for implicit quantification, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 127–143.
URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_9

Chapter 5

The impact of having different N3 interpretations

In the previous chapter we specified how existing interpretations of formulas in N3Logic differ in their handling of implicit quantification. Here, we take a closer look at the practical impact of these differences: Is the problem we discussed only relevant in theoretical contexts – for example for defining the semantics of N3Logic – or do the interpretations of files written for practical applications also diverge? In which cases can we encounter differences? How can problems be avoided? In order to answer these questions we implemented the attribute grammar defined above and applied it to several N3 files written for industrial set-ups. Below, we explain our tests and discuss the observations we made: The interpretations of our files differ in some cases. These differences are caused by the use of built-in functions, by proof structures, but also by simply using nested formulas without such special contexts. Especially the last case is problematic: Users not making use of special constructs like reasoner-specific built-ins do most probably not write their files to be used with one reasoner but expect interoperability. The problem thus needs to be addressed. We therefore finish the chapter with a discussion how such a solution could look like.

5.1 Implementing the attribute grammar

For our evaluation we have implemented the attribute grammar as specified above. In order to stay close to our format, we used the Utrecht University Attribute Grammar (UUAG) and its compiler the Utrecht University Attribute Grammar Compiler (UUAGC) [111]. This framework enables the user to

specify attribute grammars which then get translated to Haskell code and can be used in all kinds of applications. All additional applications and functions were written in Haskell.

For every N3 formula, our program produces the syntax tree of the translated core logic formula as well as its string representation in our two interpretations. We furthermore implemented a function which compares these translations. Note that in N3, especially in Cwm's interpretation, every file needs to be treated as one formula. This is because the conjunction is done by putting triples after each other. It makes a difference whether

```
{:a :b {:c :d ?x}} => {:e :f :g}.
```

is followed by

```
:s :p {?x :p :o}.  or  :s :p {?y :p :o}.
```

In the first case Cwm's interpretation puts the quantifier for the variable `?x` on the top formula; in the second case it is inside the premise of the rule. We therefore cannot give the meaning of the first implication without taking its context into account. Our function thus always compares the meaning of an entire file and then displays the concrete differences between interpretations. All code can be accessed at https://github.com/IDLabResearch/N3CoreLogic.

5.2 Representative datasets

To test whether the differences described above can be observed in practical applications, we used two kinds of datasets: a test dataset of the reasoner EYE and several datasets used in previous research projects.

The *EYE dataset*[1] is a collection of test cases for the EYE reasoner. Some of the tests were created to challenge the reasoner (e.g. parsing of nested expressions, scoping of blank nodes and universals) and are therefore rather artificial, but the majority of test files was either sent by users of the reasoner to explain problems they had or are minimal examples of practical use cases from different parties. In that sense the content of the dataset reflects a big variety of applications created by different users. At the moment we tested, the dataset contained 359 N3 files in 50 folders of which 303 contained implicitly quantified universal variables. As the latest version of EYE does not support that feature, the tests cases do not include explicit quantification.

[1] Accessible at https://github.com/josd/eye/tree/master/reasoning/. Our tests are based on the version of October 15, 2017.

The *Project datasets* contain rules we specified in previous projects, in particular: the projects *ORCA (Ontology based Reasoning for nurse Call)* [8, 9], *Facts4Workers (Factories for Workers)* [112] and *DiSSeCt (Distributed Semantic software solutions for complex Service Composition)* [10]. The rules of the *ORCA dataset*[2] are written to perform an optimized version of OWL-RL reasoning and to follow a complex decision tree which, depending on the set-up and the current situation of a hospital, assign the best staff member to answer a patient call. We further explain that in Section 6.1. The aim of the rules from the *Facts4Workers* use-case[3] is to deal with the diverse infrastructure of modern factories in which different machines are able to perform a huge variety of tasks. These tasks are described via rules which can be combined to fulfil a desired goal. The idea behind that is further explained in Chapter 8. The last set of rules[4], used in the project *DiSSeCt*, is designed to check RDF datasets for user-specified constraints. These are the topic of Section 6.2.6. We chose these datasets because they were produced to be used in practical rather complex applications and not to merely test the reasoner. To ensure compatibility with EYE the datasets do not make use of explicit quantification.

For our tests, we selected only the files which contain universal variables. For the ORCA dataset this selection contained 130 files, for Facts4Workers 25 files, and for DiSSeCt 84 files.

5.3 Critical formulas

For the datasets introduced above, we tested whether the reasoners interpret every file in the exact same way, i.e. whether the two N3 Core Logic translations produced by our software were the same (accepting differences in the naming of variables and in the order of universal and of existential quantifiers on the same level of a formula), or whether we can identify differences. The results are displayed in Table 5.1. We see that every dataset contains files for which the interpretation differs between Cwm and EYE. Nevertheless, the portion of affected files varies per dataset and depends on the nature of the data. We have identified three kinds of constructs which can be related with disagreements in the interpretation:

[2] This project was in cooperation with an industry partner and the results might be used in a product. The rules are therefore not public.

[3] Available at https://github.com/IDLabResearch/Facts4Workers/tree/master/n3.

[4] Available at https://github.com/IDLabResearch/data-validation.

dataset	#files	#differences	percentage
EYE	303	81	27%
F4W	25	12	48%
ORCA	130	27	21%
DiSSeCt	84	50	60%
Total	542	179	31%

Table 5.1: Results of tests for differences in the interpretations of N3 files. By #differences we mean the number of files which are interpreted differently by the two reasoners.

Proofs formulas which represent a formal *proof*, i.e. an explanation of the derivation steps leading to a conclusion;

Built-ins formulas which contain built-in functions which have a special meaning for one or both reasoners;

Nesting formulas which, without using built-in functions, cite graph patterns, act on them or perform reasoning about rules.

Before taking a closer look at the distribution of these three cases in the different datasets, we explain what we mean by *proofs*, *built-ins* and *nesting* in more detail.

5.3.1 Proofs

In Section 2.5.4 we already explained that there is a proof vocabulary defined for N3. The reasoners Cwm and EYE make practical use of this vocabulary. From both reasoners the user can request a formal proof as an explanation of the derivations made. Such a proof contains all proof steps applied on the data provided to the reasoner leading to the output of the reasoner. We will further explain these steps and the underlying calculs in Chapter 7. For our tests in the current section, it is more important, how exactly these steps are represented. While Cwm proofs only contain explicit universal quantification and are therefore out of scope for our current evaluation, EYE proofs employ implicit universal quantification in nested constructs.

We already saw an example of an inference proof step in Listing 1, in Listing 7 we display another kind a proof step, an `r:Extraction`. This corresponds to conjunction elimination from common first-order calculi: if a bigger conjunction is known to be correct, so are its conjuncts. Here, this example step is based on another step, a `r:Parsing`, i.e. reading information from a file

```
1 PREFIX  : <http://example.org/ex#>
2 PREFIX r:  <http://www.w3.org/2000/10/swap/reason#>
3
4  <#lemma1> a r:Extraction;
5   r:gives { {:s :p ?x1} => {:s :pp ?x1}. };
6   r:because [ a r:Parsing; r:source <rules.n3> ].
```

Listing 7: Representation of a proof step.

which in the example has the symbolic name `rules.n3`. The proof steps yield the formula

$$\texttt{\{:s :p ?x1\} => \{:s :pp ?x1\}.} \tag{5.1}$$

indicated by the predicate `r:gives`. Interesting from a structural point of view is that the formula appears in a formula expression. For Cwm the variable `?x1` is therefore universally quantified in this expression. We get the interpretation:

$$\texttt{<L1> gives <}\forall\texttt{x. <s p x>} \rightarrow \texttt{<s pp x> >.}$$

Since the file from which the formula stems contains Formula 5.1 this interpretation is correct in this context. For EYE all variables are quantified on top level, we get:

$$\forall\texttt{x. <L1> gives < <s p x>} \rightarrow \texttt{<s pp x> >.}$$

Yet, if we consider the meaning of the predicate `r:gives` which indicates the consequences we can draw from the reasoning steps, then this interpretation is also right: from the fact that Formula 5.1 appears in our data we can for every `x` conclude that $\texttt{< <s p x>} \rightarrow \texttt{<s pp x> >.}$ Hence in this case the differences in the interpretations do not have practical consequences even if the proofs are used for further reasoning.

N3 reasoning mostly depends on rules containing universal variables. Like in the example, most proofs list the parsing and selection of such rules and are therefore often subject to the problem described. In our datasets, this is the case for all proofs.

5.3.2 Built-ins

Another big group of differences in the interpretations of a formula can be observed in connection with built-in functions. Both reasoners, Cwm and EYE, provide a set of predicates with predefined meanings[5] which can

[5] Available at https://www.w3.org/2000/10/swap/doc/CwmBuiltins for Cwm, and http://eulersharp.sourceforge.net/2003/03swap/eye-builtins.html for EYE.

be used to, for example, deal with lists (`rdf:first`), to compare terms (`log:equalTo`) or to do calculations (`math:product`). Some built-in predicates are reasoner-specific. Using these leads to different reasoning results. But even if a predicate is supported by different reasoners, we often get differences in connection with their usage. This is because many built-ins deal with graphs or graph patterns.

As an example consider the built-in predicate `log:includes`[6] which is supported by both reasoners and compares formula expressions. A triple `A log:includes B.` is correct iff the terms `A` and `B` are formula expressions and the triples occuring in `B` also occur in `A`. `{:a :b :c} log:includes {:a :b :c}.` is correct while `{:a :b :c} log:includes {:a :b :o}.` is not. The following rule contains a triple using `log:includes` in its antecedent:

```
{{:a :b :c} log:includes {:a :b ?x}} => {:d :e :f}.
```

The shape of this triple is similar to the examples given before with the difference that instead of `:o` or `:a`, the object of the triple in the right-hand side formula expression is the universal `?x`. Cwm interprets this formula as

```
<∀x.  <a b c> includes <a b x>>     →     <d e f>.
```

Since it is not true that for every `x` the expression `<a b x>` is included in `<a b c>` – think for example in the case above, $x = o$ – the consequent of the formula is not derived in Cwm. Opposed to that, EYE understands

```
∀x. < <a b c> includes <a b x> >     →     <d e f>.
```

Here, the antecedent of the implication is fulfilled for $x = c$. EYE derives the new triple `d e f.` The different interpretation of universals changes the reasoning result.

5.3.3 Nesting

Our examples contain a third kind of difference: rules which either operate on other rules or on graph structures. Many examples for this can be found in the ORCA dataset where rule-producing rules are applied (this will be further explained in Section 6.1) but also in the Facts4Workers dataset, where formula expressions refer to events. A (shortened) example of such a reference in a rule is shown in Listing 8.

[6] Prefix `log:<http://www.w3.org/2000/10/swap/log#>.`

```
1  PREFIX http:  <http://www.w3.org/2011/http#>
2  PREFIX math:  <http://www.w3.org/2000/10/swap/math#>
3  PREFIX     :  <http://example.org/ex#>
4
5  {
6   {?machine :hasProblem ?problem.} :eventId ?eid.
7   (?eid 1) math:sum ?nid
8  }
9  =>
10 {
11  _:request http:methodName "POST";
12            http:requestURI "https://f4w/actions";
13            http:body       ("stop" ?machine).
14
15  {?machine :stoppedBecause ?problem.} :eventId ?nid.
16 }.
```

Listing 8: Example rule using a nested graph (taken from the project Facts4Workers).

In this example we can perform actions on machines. Such actions can for example be to start or stop the machine, but also to simply do an observation of a problem. Each action performed gets an event id. The rule from the example expresses that, if a problem is observed, the very next action to be performed is to stop the machine (here via an http-call). Note, that the different actions or events in this example are expressed by formula expressions which contain universal variables. While in EYE the two occurrences of the variable `?problem` in Line 6 and Line 15 co-refer – here, the universal quantification of implicit universals is always on top level – this is not the case for Cwm whose interpretation is displayed in Listing 9. We clearly see that the two occurrences of `?problem` are understood as two different variables. The meaning of the rule differs between reasoners and the implementation of this use case does not work with Cwm.

5.3.4 Distribution of Cases

Having seen examples for common cases causing differences in the interpretation of N3 formulas, we return to the numbers of Table 5.1. These numbers depend on the nature of the datasets and the constructs they contain: If a dataset does not contain proofs at all our implementation will also not find any problems related to proof constructs there. The same holds for the two other kinds of potentially problematic constructs we describe above. We therefore show in Figure 5.1 how many files contain proofs, built-ins and

```
1  ∀ m. ∀ i. ∀ i2.
2  <
3   ∀ p.
4   <m hasProblem p> hasId i.
5   (i 1) sum i2
6  >
7  →
8  <
9   ∀ p1. ∃ r.
10  r methodBame 'POST'.
11  r requestURI 'https://f4w/actions'.
12  r body ('stop' m).
13  <m stoppedBecause p1> eventid i2.
14 >
```

Listing 9: Interpretation of Listing 8 according to Cwm. The occurrences of variable `?problem` are understood as two different variables.

nesting of any kind[7] (dark blue). Next to that information, we also display, how many of these constructs actually lead to conflicting interpretations (light blue). Note that the cases we consider are overlapping: proofs can contain built-ins, files can have deeply nested formula expressions at one place and use built-ins at other places. Due to the nature of proofs, every proof file is also subject to nesting.

We already mentioned earlier that proofs containing universal variables cause conflicting interpretations and that in our examples all proofs contain such variables. As a consequence, the dataset containing most proofs, the DiSSeCt data set, is also the one having the highest share of diverging interpretations. For the other kinds of constructs we discussed, built-in functions and nesting, we take a closer look at the numbers appearing in Figure 5.1 below, where we also explain how the values were calculated.

Disagreements per Formula Type

In order to understand how many of the built-ins present in our datasets are causing problems, we extended the attribute grammar by additional attributes. We give the definition of these attributes in Appendix B.1. Here, we only want to briefly discuss the idea. If a triple has a built-in function in predicate position, we test whether subject and object of that triple contains universal variables. If this is the case, we next need to know where exactly the interpretation according to Cwm quantifies these variables. If all these

[7] In that context we define nesting as containing at least one formula expression occurring in another one as in the example `:a :b {:c :d {:e :f :g.}.}.` where `{:e :f :g.}` is nested.

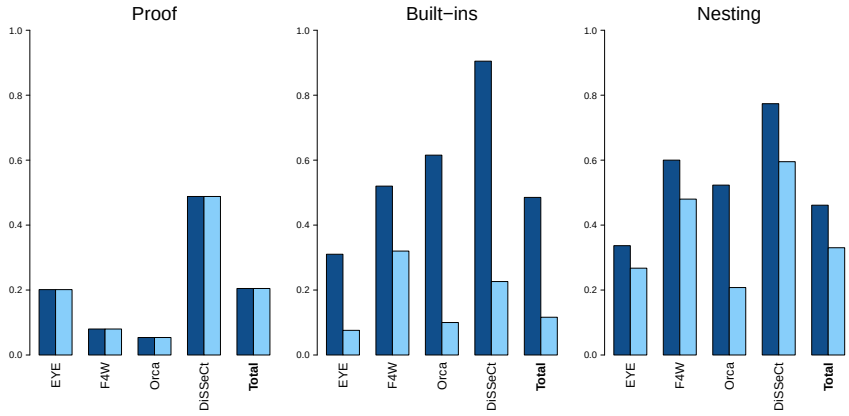


Figure 5.1: Distributions of proofs, built-ins and nesting in datasets. The share in dark blue always represents the files containing the respective feature while light blue is used to represent the cases where the feature leads to different interpretations.

universal variables are quantified on a higher level than the parent$_c$ level of the built-in then the built-in construct itself is not causing conflicting interpretations.

In Figure 5.1 where the numbers calculated by that method are displayed in the middle we already see that in many cases the presence of a built-in construct in a data set does not cause problems. To better understand how likely this presence of a built-in construct causes contradicting interpretations, we display the numbers from above in another way. Figure 5.2 shows in how many of the files containing built-in functions at least one of the built-in functions occurs in a construct which causes contradictory interpretations. We see that for the whole dataset and most subsets approximately a quarter of all files with built-ins contains at least one critical construct. Only in the Facts4Workers dataset this share is higher. This has to do with the fact that in that project one specific built-in is used very often: The built-in `e:whenGround`[8] tests whether its subject contains variables or is ground and calls the object if the latter is true. This built-in is implemented in EYE for a few very specific use cases. We expect that users employing such special predicates are aware of the fact that their rules only work with one specific reasoner. The cases counted here are therefore less critical for the problem than files written for use cases of the Semantic Web, which rely on interoperability, are interpreted contradictory by different reasoners.

[8] See http://eulersharp.sourceforge.net/2003/03swap/log-rules.html#whenGround.

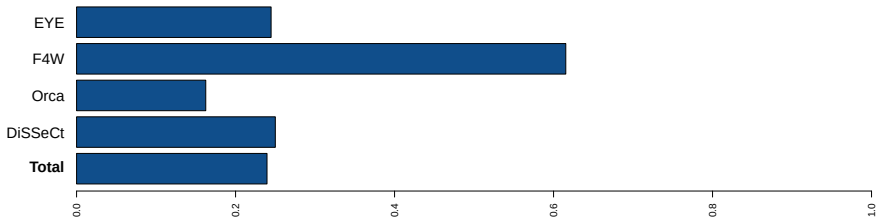


Figure 5.2: Share of files containing built-ins causing conflicting interpretations in files containing built-ins. Only in a quarter of all files containing built-ins these occur with deeply nested universals.

A similar figure as the one above can be produced for nested formulas. We display the share of formulas being subject of a nesting problem in all formulas containing nesting in Figure 5.3. In Figures 5.1 and 5.3 we understand a formula as nested if it contains a graph construct in a graph construct (i.e. a formula in nested brackets $\{\ldots\{\ldots\}\ldots\}$). As a nesting construction leading to the problems described in this section we understand any nested construct containing a universal variable for which the quantifier according to Cwm is on any other level than the top level. This rather broad definition qualifies all differences listed in Table 5.1 as subject to a nesting problem and these numbers are also used for the two figures. We observe that, when deep nesting is already present in a file, we suffer from conflicting interpretations in 72% of the cases. This is not very surprising since most nested graphs occur in rules which most likely also contain universals, but this figure shows once again, that when using nested graphs, users need to be careful with universal variables.

Categorisation of Errors

As a last point in this subsection we display how the different problem types are distributed over the files counted in Table 5.1. The reasons for conflicting interpretations of a formula can be overlapping. To be able to show a distribution we separate them in disjoint groups as follows:

Built-in Group Every formula containing a built-in construct which leads to contradictory interpretations is counted as such even if this construct occurs in a proof or additionally contains nested graph constructs without built-ins.

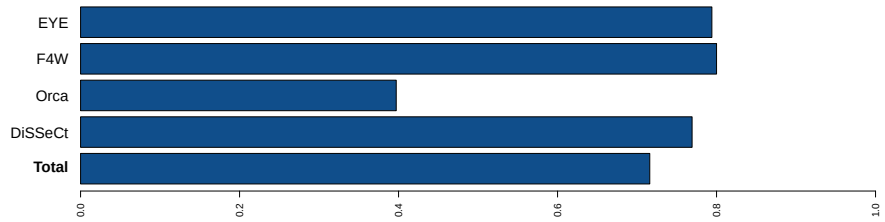


Figure 5.3: Share of files causing conflicting interpretations in files which have nested expressions. 72% of the files containing nesting are interpreted differently by both reasoners.

Proof Group Every proof formula for which the reason of the contradictory interpretations is only the proof structure itself. That means that the formula does not belong to the *Built-in Group* and that the results of the proof steps – i.e. the *result* part in the triples `<#lemma> r:gives {` *result* `}` – do not contain any universal variables which Cwm quantifies on a level which is nested inside these results.

Nesting Group Every formula not belonging to the two groups above is counted as a nested formula.

To test whether conflicting interpretations are caused by built-ins, we used the attributes introduced above. The method to determine whether a formula representing a proof belongs to the *Nesting Class* or the *Proof Class* is discussed in Appendix B.2.

Following this classification, the results of our tests are displayed in Figure 5.4. For the overall dataset (last line) half (51%) of the conflicts between reasoning results occur only because of the different interpretations of proofs. As discussed these can be considered as rather harmless. The next bigger group of differences occurs in connections with built-in functions (31%). Here, not all, but some of the conflicts are unavoidable, since some built-in functions are not supported by all reasoners. The last group of conflicts (13%), caused by the simple use of nested graphs or rules without direct involvement of built-in functions or proof predicates, is the most dangerous: While users employing built-in functions are often aware that the support of these could be limited to one reasoner and therefore also check carefully when they want to switch to a different one, this is not the case here. If nested rules and graphs are used without any special predicates, it is normally expected that the reasoning results from formulas containing these constructions do not

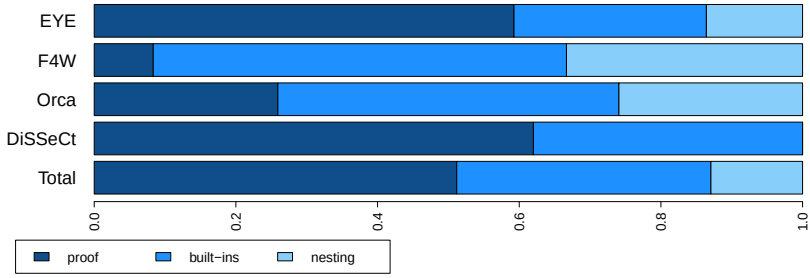


Figure 5.4: Distribution of different cases causing conflicting interpretations by the reasoners Cwm and EYE.

differ between reasoners. The user has no reason to do extra compatibility checks.

Whether these cases occur depends on the use case: In the DiSSeCt dataset, such cases do not occur. This has to do with the fact that the use case here is the test for constraints on RDF data, i.e. data without formula expressions or rules. The constraints themselves and the results are plain RDF and there is always only one reasoning run applied in order to find constraint violations. In contrast, the Facts4Workers dataset contains many constructs similar to the one displayed in Listing 8 and does therefore have a rather high occurrence of cases belonging to the last category (33%).

5.4 Possible solutions

In the previous section we examined the impact of having different interpretations for nested implicit universal quantification on practical cases and discovered that 31% of our test files contained at least one critical construct. This means that the problem is not only of theoretical nature and needs to be addressed. In this section, we discuss possible solutions from two perspectives: First, we take the perspective of a practitioner who – given the current situation – needs to make sure that his rules lead to the same result in both reasoners. Second, we take a broader perspective and clarify the different positions a standardisation could take.

5.4.1 Avoiding Conflicts in Practical Cases

Having seen which kinds of constructs lead to conflicts between the different interpretations of N3 we now discuss how to deal with those conflicts. One

option is to avoid them from the very beginning by not using nested formula expressions. The drawback of this solution is that this also means to not use the full power of N3 since constructs such as rule-producing rules [9] would not be available any more. If we want to support N3 as it is, we need a way to translate from one reasoner to the other.

EYE formulas interpreted by Cwm

In order to make N3 formulas written for the reasoner EYE be interpreted in the exact same way by Cwm, we can use a simple trick: Since we know that the interpretation of implicitly universally quantified variables also depends on their contexts, i.e. on their occurrence in the different conjuncts of a formula, we can add a dummy formula which lifts the scope of deeply nested variables to the top level without changing EYE's interpretation of the whole expression. To illustrate that idea we use an example we have seen earlier: remember that in Cwm's interpretation (Interpretation 4.3') of Formula 4.3 the quantifier for the universal variable `?y` was nested while for EYE it was on top level, i.e. in front of the overall formula (Interpretation 4.3"). If we now add the implication `{?y ?y ?y}=>{?y ?y ?y}.` to the formula this difference disappears. The formula

```
{?y ?y ?y.}=>{?y ?y ?y.}.                                    (4.3a)
{{?x :q ?y.} => {?x :r :c.}.}
                    =>{?x :p :a.}.
```

has in both reasoners the same interpretation, namely:

$$\forall x.\forall y. \qquad (4.3a')$$
$$<y\ y\ y> \rightarrow <y\ y\ y>.$$
$$<<\ x\ q\ y> \rightarrow <\ x\ r\ c>> \rightarrow <\ x\ p\ a>.$$

Since the antecedent and the consequent of the added rule are exactly the same, its addition does not change the meaning of the whole formula according to EYE. For Cwm, the meaning does change, the quantifier for `?y` is lifted to the top level and is no longer nested.

While here, the change of meaning has been performed on purpose – we wanted the formula to have the same interpretation in both reasoners – phenomena as the one above can also occur rather randomly: In our datasets there are several rules embedded in a bigger context which make use of nested universals but whose interpretation does not differ between Cwm and

EYE[9]. The reason is that they make use of rather arbitrary variable names such as `?x` and `?y` which are also used in other conjuncts of the same very long formulas. Having that in mind, users of Cwm who want to use nested implicit universal quantification need to be careful with the variable names they are using to avoid unwanted changes of scope.

Cwm formulas interpreted by EYE

Performing the other direction – making sure that EYE interprets a formula containing universal variables in nested graphs the same way Cwm does – is more difficult: The only way to do so is to use explicit universal quantification as one can easily see recalling Cwm's interpretation of Formula 4.3:

$$\forall x.\texttt{<}\forall y.\texttt{<x q y>} \quad \rightarrow \quad \texttt{<x r c>>} \quad \rightarrow \quad \texttt{<x p a>} \tag{4.3'}$$

Here the universal quantifier for `y` is nested, but EYE interprets all implicitly universally quantified variables as quantified on top level.

Due to the open issues mentioned in Section 3.1.3, the current version of EYE does not support nested explicit quantification which makes the desired task impossible. This was different in earlier versions[10]. A possible way to make sure that formulas written for Cwm are understood equally by EYE, could be to use such an older version of the reasoner. Then our implementation can be used to generate the representation of an N3 formula in core logic according to Cwm's interpretation, which we could then translate to explicit quantification in N3. But even when doing that, it cannot be guaranteed that this explicit quantification in N3 works exactly the same way explicit quantification in core logic does since a formal specification of the former is missing. Without a clearly defined explicit quantification in EYE, it is not possible for each nested formula to translate Cwm's interpretation to an N3 formula EYE interprets equally.

5.4.2 Definition of a Standard

As discussed above, in the current situation the user writing rules needs to either know beforehand with which reasoning engine he wants to use his rules, or he needs to apply the strategy discussed above of adding dummy rules which lift the quantifier of deeply nested implicitly universally quantified variables to the top level. Given that N3Logic was created for the

[9] A concrete example is the file https://github.com/josd/eye/blob/master/reasoning/n3p/sample.n3 in the eye dataset.

[10] In all versions before EYE-2014-12 nested explicit quantification is allowed.

Semantic Web where interoperability is a very crucial feature, this situation is not acceptable and the community needs to come to an agreement.[11] For such an agreement, we see three options, which we discuss below.

Semantics with Nested Universal Quantifiers

One possible solution is to follow Cwm. In that case we could use the specification provided above as the official semantics of N3.

One problem with that solution is that it is rather difficult to formalise. We needed to define an attribute grammar with four attributes to be able to handle Cwm's implicit universal quantification. Following the same approach, scoping on top level only required the definition of one attribute. Being a direct realisation of this grammar our implementation is equally complex and so far we did not encounter an easier way to implement the scoping as intended by Cwm. Even the Cwm reasoner itself has difficulties with implicit universal quantification in some cases (see Appendix A). To the best of our knowledge, there is no other logical framework supporting implicit universal quantification which interprets this feature the way Cwm does (we also discuss other frameworks supporting implicit quantification in Section 3.3). We therefore suspect that the users' intuition about implicit universal quantification could be opposed to Cwm's interpretation. We furthermore see the the difficulties when formalising the semantics and implementing a parser or reasoner following it as an indication that end users writing N3 rules could also have problems to understand and apply the formalisation of implicit universal quantification according to Cwm.

All these reasons make us to rather opt against the possibility to handle implicit universal quantification the way Cwm does.

Semantics with Quantification on Top Level

Another possible remedy for the problem at hand would be to strictly follow the interpretation which understands implicitly universally quantified variables as quantified on top level such as the reasoner EYE does.

One argument to do so is, that this is easier to formalise and also to implement. The attribute we used for the quantification of EYE was rather simple and just passed all implicitly universally quantified variables upwards in the syntax tree. A formalisation could also be made without employing attribute

[11] The first step towards such an agreement has already been taken by forming a W3C community group: https://www.w3.org/community/n3-dev/. Issues are discussed at: https://github.com/w3c/N3/issues/.

grammars by simply using a universal closure for all universal variables. As assuming the universal closure for variables freely occurring in formulas is common practice and done in many frameworks like for example Prolog [89] we expect that users writing N3 rules can easily understand this behaviour and write their rules accordingly.

We therefore favour the interpretation putting the quantifier of implicitly universally quantified variables on the top level.

Exclude Implicit Universal Quantification

The third solution for the problem explained in this thesis would be to either not allow implicit universal quantification at all or to at least only allow it in non-nested structures such that the scoping of every implicitly universally quantified variable is clearly defined in both interpretations. Instead, explicit quantification could be employed. The problem here is that, as explained in Section 3.1.3, the meaning of explicit quantification is not clearly defined in the W3C team submission either. Problems especially arise if explicit universal quantification is used in combination with explicit existential quantification – to be consistent with the implicit case universal quantification always dominates existential quantification if both occur on the same level – and if constants and quantified variables are not clearly distinguished. In order at least make this last opportunity an applicable option, these uncertainties need to be clarified. One possible way to do that would be to extend the attribute grammar discussed here.

5.5 Conclusion

In this chapter we investigated which impact the fact that we have different interpretations for implicit universal quantification on reasoning files used for practical applications.

In order to do so, we implemented the attribute grammar defined in the previous chapter. This implementations allows us to produce the concrete N3 Core Logic equivalent of an N3 formula following the different interpretations and used for testing: Whoever wants to provide N3 rules in the Web, can test whether the rules he or she created are subject to ambiguity.

We applied our implementation on different on the example use cases of EYE and on different sets of files which have been implemented in research projects to solve practical problems. We encountered that 31% of our files were subject to contradictions. We further investigated the contexts in which

we could observe these contradictions and identified one group of them as most harmful: if universal variables occur in constructs which do not make use of built-in functions and which are not part of a proof they are most likely produced by users who are not aware of the different possibilities to interpret the formulas. 13% of or formulas belonged to that group.

These investigations have shown that the the underspecified semantics of N3 is an actual problem which needs to be addressed. We therefore ended the chapter by discussing possible solutions for the practitioner but also for the community. The latter needs to come to an agreement.

With this and the previous two chapters we addressed research questions 1 and 2 which focus on the formal semantics of N3 and on N3's relation to RDF. In the following chapter, we take a more practical perspective and discuss the use cases N3Logic can solve, in particular: a Semantic nurse call system and data validation.

This chapter was partly based on the publications:

D. Arndt, T. Schrijvers, J. De Roo, R. Verborgh, Implicit quantification made explicit: How to interpret blank nodes and universal variables in Notation3 Logic, Journal of Web Semantics 58 (2019) 100501. doi:10.1016/j.websem.2019.04.001.
URL https://authors.elsevier.com/c/1ZWg55bAYUaMkH

D. Arndt, R. Verborgh, J. De Roo, H. Sun, E. Mannens, R. Van de Walle, Semantics of Notation3 logic: A solution for implicit quantification, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 127–143.
URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_9

Chapter 6

Querying and ontology reasoning with Notation3 Logic

In the previous chapters we discussed N3Logic, its relation to RDF and the possibilities to fix its formal semantics. A clearly defined semantics and compatibility with RDF were only two requirements on a *Unifying Logic* we identified in Section 2.4. A *Unifying Logic* for the Semantic Web should furthermore connect the logical building blocks, i.e. the blocks of *Querying*, *Ontologies/Taxonomies* and *Rules*, and it should support the layer of *Proofs*. The last point – proofs – will be discussed in the following chapter. Here, we focus on N3's relation to the logical blocks: Being a rule logic itself N3Logic covers the *Rules* block – we can perform rule-based reasoning with N3. For *Querying* and *Ontologies/Taxonomies* we need to investigate further: To what extend does N3Logic support these two blocks? Can it solve the same practical problems as the standards SPARQL and OWL-DL listed in the Semantic Web stack? If yes, is the performance comparable?

In order to answer these questions, we consider two practical use cases which both can be solved by applying a combination of ontology reasoning and querying:

1. a semantic nurse-call system, and
2. a system to perform RDF validation

Both use cases have been previously implemented by either using OWL-DL or RDFS reasoning and SPARQL querying and we aim to solve them

by applying N3 reasoning instead. Below, we describe the use cases in detail and discuss how N3 can be used to perform the same tasks as the existing implementations. Both sections finish with a comparison between the approaches. We then conclude our findings.

6.1 Use case 1: Semantic nurse call system

Our first use case is a semantic nurse call system in a hospital: The system is aware of certain details about personnel and patients. Such information can include: personal skills of a staff member, staff competences, patient information, special patient needs, and/or the personal relationship between staff members and patients. Furthermore, there is dynamic information available, for example, the current location of staff members and their status (busy or free). When a call is made, the nurse call system should be able to assign the best staff member to answer that call. The definition of this "best" person varies between hospitals and can be quite complex. The system additionally controls different devices. If for example staff members enter a room with a patient, a light should be switched on; if they log into the room's terminal, they should have access to the medical lockers in the room.

We next discuss the technical challenges which need to be resolved when implementing a system as described above. After that, we explain how this use case can be tackled by applying ontology reasoning and querying and present our implementation with N3Logic: We use OWL RL reasoning to be able to incorporate knowledge specified in an OWL ontology (ontology reasoning) and write extra rules to support decision trees (querying). We first perform the OWL RL reasoning by directly applying the rules specified on the Web site introducing this profile [41] and then add a preprocessing step to further improve performance. We compare both implementations with an implementation using OWL DL reasoning and SPARQL querying.

6.1.1 Technical challenges

An event-driven reasoning system for the use case described above needs to fulfil certain requirements:

scalability It should cope with data sets ranging from 1000 to 100,000 relevant triples (i.e. triples necessary to be included for the reasoning to be correct). Especially in bigger hospitals the number of staff members and patients and thereby also the amount of available information about those can be quite big. It is not always possible to divide this

knowledge into smaller independent chunks as this data is normally full of mutual dependencies.

functional complexity It should implement deterministic decision trees with varying complexities. The reasons to assign a nurse to a certain patient can be as manifold as the data. Previous work has shown that this complexity is not only theoretically possible but also desired by the parties interested in such a semantic system [113].

configuration It should support the ability to change these decision trees at configuration time. Different hospitals have different requirements and even in one single hospital those requirements can easily change due to, e.g. an increase of available information or a simple change in the hospital's organizational concepts or philosophy.

real-time It should return a response within 5 seconds to any given event. Especially in such a delicate sector as patient care, seconds can make a difference. Even though a semantic nurse call system will not typically be employed to assign urgent emergency calls through complex decision trees, a patient should not wait too long till his possibly pressing request is answered.

6.1.2 Benefits of a semantic approach

There are several options to implement a nurse call system as described above. Following a more classical approach, the system could be written in an object-oriented programming language such as Java or C++. An implementation like this can easily fulfil the real-time and scalability constraints. But such systems are traditionally hard-coded. They are implemented for a specific use case, and even though they might be able to support the required functional complexity, this implementation would be static. The possibility to configure complex decision trees as postulated by the complexity requirement is rather hard to fulfil using traditional programming. Even more difficult to satisfy is the semantic requirement: Most object oriented languages do not support enough logic to "understand" the available information. Knowledge must be stated explicitly, as even simple connections between statements such as "*nurse x has location y*" and "*y is location of nurse x*" cannot be found easily.

Especially the last argument motivates us to solve the described problem using Semantic Web technologies as they natively fulfil the semantics requirement. Knowledge can be represented in an OWL ontology which is understood by OWL DL reasoners. Complex decision trees can be handled

by subsequent SPARQL queries. It is easy to add new queries or to change the order of existing queries and to thereby accommodate for the configuration constraint. But our tests (see Section 6.1.6 below) have shown that such systems inherently are not fast and reliable enough to fulfil the scalability and real-time requirements. For bigger amounts of data or too complex decision trees, the reasoning times of traditional OWL DL reasoners grow exponentially, which is not scalable.

To keep the benefits of an OWL DL based implementation in a scalable and real-time way, we propose a rule-based solution. By using rules to reason over the ontology we can significantly decrease reasoning times. Complex decision trees can directly be implemented in rules. As rules are the most natural representations of such trees, it is easy for a user to understand and change the priorities of different assignments or to configure new rules. A further advantage of our approach is that all logic is represented in one single way. Instead of OWL DL reasoning plus SPARQL querying, we can implement the use case by only employing N3Logic. With the aforementioned system, we can meet all necessary requirements. Next, we describe the details of our solution.

6.1.3 Performing OWL RL reasoning with N3

If we use a traditional OWL DL reasoner like for example Pellet [42] or HermiT [43] to reason over an existing ontology this reasoner natively understands the concepts which are part of the OWL DL profile and applies different inference steps in a predefined order to derive new knowledge. As OWL DL is very expressive and contains a huge variety of different concepts, OWL DL reasoners are rather slow in comparison with rule-based reasoners. The OWL 2 profiles [41] aim to overcome this gap by defining less expressive but still powerful subsets of OWL DL. One of these profiles is OWL RL, which was designed to enable rule-based reasoners to cope with OWL ontologies.

For each OWL concept which forms part of the RL profile the specification lists one or more rules which can be used to draw conclusions from it. These rules written in a generic RDF rule language with the expressiveness of Datalog [114] – i.e. simple rules which do not allow functions or new existential variables in their conclusion – and can be translated to every rule language being at least equally expressive. These rules then either directly act on the syntactical representation of the ontology – this is what we do in our implementation – or on its translation to the input format of the rule language – this is for example done by OWLim [62] and Oracle's RDF Semantic Graph [115]. By using N3 which is compatible with RDF/Turtle we make sure that the reasoning performed does not depend on the way the

```
1 PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
3
4 {?C rdfs:subClassOf ?D. ?X a ?C} => {?X a ?D}.
```

Listing 10: OWL-RL rule for `rdfs:subClassOf` class axiom in N3.

triples are translated to an internal format. In that sense we are reasoner independent. This approach furthermore makes it easier for the data consumer to understand or choose which rules are applied.

Below, we explain OWL RL reasoning in N3 on a concrete example taken from our use case. The OWL RL rule we show, as well as the other used in our implementation, are taken from the EYE website [116]. Knowledge is represented using the ACCIO ontology[1] [117], an ontology which was designed to represent all aspects of patient care in a hospital. Listing 10 shows the class axiom rule[2] which is needed to deal with the RDFs concept `subclassOf`. For convenience we omit the prefixes in the formulas below. The empty prefix refers to the ACCIO ontology, `rdf` and `rdfs` have the same meaning as in Listing 10. Consider that we have the following triple stating that the class `:Call` is a subclass of the class `:Task`:

```
:Call rdfs:subClassOf :Task.
```
(6.1)

If the ontology contains an individual which is member of the class `:Call`

```
:call1 a :Call.
```
(6.2)

an OWL DL reasoner would make the conclusion that the individual also belongs to the class `Task`:

```
:call1 a :Task.
```
(6.3)

Our rule in Listing 10 does exactly the same: as Formula 6.1 and Formula 6.2 can be unified with the antecedent of the rule, a reasoner derives Formula 6.3.

6.1.4 Decision trees

The ACCIO ontology provides the user with a huge variety of concepts which can, e.g. be used to describe patients (social background, needs, disease),

[1] Available at: https://github.com/IBCNServices/Accio-Ontology/tree/gh-pages

[2] The rule is the N3 version of the cax-sco rule in Table 7 on the OWL 2 Profiles website [41].

```
1  PREFIX : <http://ontology/Accio.owl#>
2  PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4  {
5    ?c rdf:type :Call.
6    ?c :hasStatus :Active.
7    ?c :madeAtLocation ?loc.
8    ?p :hasRole [rdf:type :StaffMember].
9    ?p :hasStatus :Free.
10   ?p :closeTo ?loc.
11 }
12 =>
13 {
14   (?p ?c) :assigned 200.
15 }.
```

Listing 11: Rule assigning a preference value to a staff member with status "free" who is close to the call-location.

staff members (skills, relationships to patients), and situations (locations of persons, states of devices). If all this information is actually available, decision trees can use all of it and be therefore quite complex. Here, we provide two simple rules which could be part of such a tree and we explain how these rules can be modified depending on the needs of an individual hospital.

Listing 11[3] shows a rule which, given an active call, assigns a staff member with a certain preference to that call. As explained in Section 2.5.5, N3 reasoners can be used goal-driven i.e. the reasoner outputs all instances of the result of a filter rule which it can derive based on the input at hand, in our example all combinations of persons assigned to calls and their preference number. By using built-in functions we can now search for the highest or lowest preference number. In our example, lower numbers mean higher preferences. The antecedence of the given rule contains certain constraints: the active call is made on a certain location and there is a staff member, who is currently free and close to that location. In such a case, our rule assigns the number 200 to the combination of call and staff member.

Listing 12 displays another rule. This rule applies if the reason of the active call is known. If there is a staff member who has the required skills to answer the call, but this staff member is currently busy, our rule assigns the

[3]Here and in the remainder of Section 6.1 we use one single prefix for the Accio ontology. This is only to make our examples easier to represent. The Accio ontology consists of several subontologies each having its own name space. The actual name spaces of the concepts used in our examples can be determined by searching on the ontology's git repository: https://github.com/IBCNServices/Accio-Ontology/tree/gh-pages.

```
1  PREFIX : <http://ontology/Accio.owl#>
2  PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4  {
5    ?c rdf:type :Call.
6    ?c :hasStatus :Active.
7    ?c :hasReason [rdf:type :CareReason].
8    ?p rdf:type :Person.
9    ?p :hasStatus :Busy.
10   ?p :hasRole [rdf:type :StaffMember].
11   ?p :hasCompetence [rdf:type :AnswerCareCallCompetence].
12 }
13 =>
14 {
15   (?p ?c) :assigned 100.
16 }.
```

Listing 12: Rule assigning a preference value to a busy staff member who has the needed skills to answer the call.

number 100 to this combination of call and staff member. This means, in our current decision tree, that we prefer this assignment to the one described by Listing 11.

Now, it could be, that another hospital has different priorities. Imagine for example that in this new hospital, no busy staff should be called if there is still a free staff member available, regardless of the reason of the call. We could easily adapt our decision tree by simply changing the assignment number of one of the rules. If we replace the triple

```
(?p ?c) :assigned 100.
```

in line 15 of Listing 12 by the triple

```
(?p ?c) :assigned 300.
```

the reasoner would prefer the assignment expressed by Listings 11 to 12.

Similarly, we can add extra conditions to the rules. Currently, the rule in listing 12 does not take the location of the staff member into account. We can change that by only adding the triples

```
?c :madeAtLocation ?loc. ?p :closeTo ?loc.
```

to the antecedence of the rule. To give this new rule a higher priority than the existing one, we would again only have to change the assigned number in the consequence. Rules with the same number are treated equally.

6.1.5 Improving performance by pre-producing TBox-rules

By using the OWL RL rules as described above, we can already perform better than the solution employing OWL reasoning and SPARQL querying (see Section 6.1.6 below). However, to be able to fulfil the real-time constraint, we need further optimisation. To improve the performance of our solution we take advantage of a specific feature of N3: In N3 it is possible to write and apply rules having new rules in their consequences, we call these *rule-producing rules*.

To better understand how we can benefit from rule-producing rules we go back to the above example. By unifying the antecedent of the rule in Listing 10 with Formulas 6.1 and 6.2 we could derive Formula 6.3. But this unification is rather expensive: The antecedent contains three different variables occurring in two different triples which have to be instantiated with the data of the ontology. While information as stated in Formula 6.2 can change – patients will make new calls – statements as Formula 6.1 can be considered as fixed. The terminology does not change during the reasoning process, calls are tasks for our ontology. Triples as the above specifying the terminology of an DL ontology are known as the TBox [26, ch. 1]. Here the broader concepts (classes) and their relationships among each other are specified while the ABox contains triples over individuals. We consider the TBox as static knowledge which can be used for pre-processing. The idea of our solution is to do as much unification as possible before dealing with (possibly) dynamic data. We produce more specialized rules, in the case mentioned above, for example the rule

```
{?X a :Call.} => {?X a :Task.}.          (6.4)
```

which will derive for every new call, that it is also a task, just as the rule in Listing 10 does.

Below we describe our implementation. We perform two steps:

1. Produce a grounded copy of the TBox.
2. Use rules to translate the grounded TBox into specialized rules.

The need of the first step has to do with the fact that OWL ontologies represented in RDF often use *structural blank nodes* (see Section 3.2.2). As RDF follows a strict triple structure, compounded constructs like for example the union or the intersection of two classes cannot directly serve as subject of object in a triple. To represent such constructs, anonymous blank nodes are normally introduced. Used in rules, these blank node class names have

```
1  PREFIX : <http://ontology/Accio.owl#>
2  PREFIX owl:  <http://www.w3.org/2002/07/owl#>
3  PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4  PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
5
6  :Call  rdfs:subClassOf [
7                         rdf:type owl:Class ;
8                         owl:intersectionOf (
9                                     :PatientTask
10                                    :UnplannedTask
11                                )
12                        ].
```

Listing 13: ACCIO example: A call is both, a patient task and an unplanned task.

a limited scope (remember the examples from Section 3.1.1). It is therefore difficult to use them to reference the same class in different rules. We will give a more elaborate explanation in the next section. After that we will describe the translation step in more detail.

Grounding the Ontology

Before translating the TBox into rules we have to replace all blank nodes by URIs or literals. To understand the reason for this skolemization step, consider Listing 13. The example contains triples which further describe the class `:Call` from Formula 6.2. A call is a patient task and an unplanned task, or to be more specific: There exists an anonymous class of which the class `:Call` is a subclass and which is the intersection of the classes `:PatientTask` and `:UnplannedTask`. It is exactly this anonymous class which causes problems. As we discussed in Section 3.2.2 blank nodes are not simply variables, they are variables with an implicit existential quantifier and a specific scope. If we take the triple

```
:Call rdfs:subClassOf _:y.
```
(6.5)

from line 6 of Listing 13[4] and try to construct a rule from it in an analogous way as we did with Formula 6.1 and its resulting Rule 6.4 we would most probably come up with something like:

```
{ ?X a :Call. } => { ?X a _:y.}.
```
(6.6)

[4] Remember that `[ ]` stands for a "fresh" blank node, we chose to name it `_:y` to make it easier to refer to it.

```
1 PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3
4 {?C rdfs:subClassOf ?D.} => {{?X a ?C.}=>{?X a ?D.}.}.
```

Listing 14: Rule producing new rule for every occurrence of `rdfs:subClassOf`; based on the `rdfs:subClassOf` class axiom of Listing 10.

But here the blank node `_:y` is quantified in the consequence of the rule, in N3 Core Logic this translates to:

$$\forall \texttt{x}: \quad \texttt{<type(x,Call)>} \rightarrow \texttt{<}\exists \texttt{y}: \quad \texttt{type(x,y)>} \tag{6.6'}$$

This rule means, that every instance of the class `:Call` is also instance of *some* other class. This knowledge can already be derived from Formula 6.4 where we have an example of such a class. With the local scoping of the blank node in the consequence of the rule, it is impossible to make the connection between the class mentioned in the rule and the intersection class of patient tasks and unplanned tasks by using blank nodes. We need to choose a constant representing the intersection class mentioned in Listing 13 which does not not change its scoping when used in the consequence of the rule.

As OWL ontologies use a lot of such structural blank nodes, which here can be understood as anonymous classes, we perform a grounding step. Our implementation uses the EYE reasoner which provides the option to obtain a skolemized version of any input N3 file(s). The switch `--no-qvars` replaces every blank node by a unique skolem IRI following the naming convention as described in the RDF specification [4]. It additionally makes sure that equally named blank nodes only get assigned the same skolem IRI if they actually refer to the same thing. Producing a grounded version of the ontology enables us in further reasoning steps to use the new identifiers for (formally) anonymous classes in different rules.

Translation Step

As explained above, the next step after having produced a grounded version of the ontology's TBox is to produce the new specialized rules. Here we use rule-producing rules. To understand the idea, consider the following example:

$$\{\underbrace{\texttt{:Call rdfs:subClassOf :Task.}}_{\text{satisfied ontology triple(s)}}\}$$

```
=>{ {?X a :Call}=>{?X a :Task.}. }.
```
produced new rule(s)

Just as simple rules enable the reasoner to derive new triples from the fact that its antecedent is fulfilled, the rule above, applied on Formula 6.1, derives a new rule, namely Formula 6.4. Nevertheless, the rule as stated above is too specific to be used for our purpose: if we already knew that the ontology contained the triple in Formula 6.1 we could also write the rule in Formula 6.4 directly instead of writing a rule which will surely produce it. Our rule needs to be more general as we want to handle all `owl:subclassOf` triples in that same way and always produce a rule similar to the rule expressed in Formula 6.4. This more general rule can be found in Listing 14. Applied on Formula 6.1 the variable `?C` gets unified with the URI `:Call` and the variable `?D` gets unified with `:Task`, thus, Rule 6.4 can be derived. Similarly, an application of the rule in Listing 14 on triple

```
:UnplannedTask rdfs:subClassOf :Task.
```

results in a new rule

```
{?X a :UnplannedTask.} => {?X a :Task.}.
```

The same principle can be applied for other OWL concepts. Listing 15 shows a rule[5] which handles the concept `owl:intersectionOf`. Note that this rule uses a built-in predicate of Notation3, `list:in`. A triple using `list:in` as a predicate is true if and only if the object is a list and the subject is an entry of that list. If we apply this rule to the (now skolemized) intersection expressed in Listing 13

```
:InterClass1 owl:intersectionOf
                    (:PatientTask :UnplannedTask ).
```

two rules will be produced by that:

```
{?x a :InterClass1} => {?x a :PatientTask.}.
```

and

```
{?x a :InterClass1} => {?x a :UnplannedTask.}.
```

The above example illustrates how useful it is that Notation3 treats lists as first-class citizens and does not rely on reification here (see Sections 3.2.2

[5]The rule is motivated by the cls-int2 rule in Table 6 on [41].

```
1 PREFIX list:  <http://www.w3.org/2000/10/swap/list#>
2 PREFIX owl:  <http://www.w3.org/2002/07/owl#>
3
4 {?C owl:intersectionOf ?L. ?D list:in ?L} =>
5                                   {{?X a ?C.}=>{?X a ?D}}.
```

Listing 15: Rule-producing rule for `owl:intersectionOf`.

and 4.1). There are many built-in predicates which enable the user to write clear rules operating on lists. For working with OWL ontologies this is a real advantage as lists are normally used together with many OWL concepts like the above `owl:intersectionOf` or for example `owl:unionOf`.

If we apply the rule-producing rules as filter rules (see Section 2.5.5) in an N3 reasoning run having the ground version of the ontology's TBox as input, the reasoner produces a file only containing our specialized rules. These rules now replace the TBox and can be used for further reasoning.

6.1.6 Evaluation

The aforementioned methodology replaces generic and complex constructs in the TBox by specialized rules that provide the same functionality. To test how much performance we gain by using this pre-processing step and by using rules to implement the decision tree compared to the original implementation using OWL DL reasoning and SPARQL querying we tested a scenario of our use case with three implementations: the first using the DL reasoner Pellet [42] and SPARQL-querying, the second using the traditional rule set processing the triples of the original TBox while reasoning and the third using the precomputed rule set of specialized rules which replaces the TBox of the ontology. All experiments were run on the same hardware settings, namely Intel(R) Xeon(R) E5620@2.40GHz CPU with 12 GB RAM, Debian "Wheezy", using two different technology stacks. For the implementation with OWL DL+SPARQL: Pellet 3.0 and OWL-API 3.4.5; for the two implementations using N3: EYE-Autumn15 09261046Z and SWI-Prolog 6.6.6.

Ontology and Data

To represent the data as described above we make use of the ACCIO ontology. At the time of testing the ontology contained ca. 3,500 triples (414 named classes, 157 object properties, 38 data type properties). A full description is given by Ongenae et al. [117].

This ontology was filled with data describing wards in a hospital. This data was simulated, based on real-life situations, as deducted from user studies [113]. The data was scaled by increasing the amount of wards from 1 to 10 to fill the ABox with more data. The description of such a ward contains approximately 1,000 static triples. Additionally, there was dynamic data such as for example the location of nurses or the status of calls taken into account.

Test scenario

We compared the reasoning times of the three implementations by running a scenario, based on a real-life situation. This scenario consists of a sequence of events, which we list below. The expected outcome of the reasoning is indicated in brackets.

1. A patient launches a call (*assign nurse and update call status*)
2. The assigned nurse indicates that she is busy (*assign other nurse*)
3. The newly assigned nurse accepts the call task (*update call status*)
4. The nurse moves to the corridor (*update location*)
5. The nurse arrives at the patients' room (*update location, turn on lights and update nurse status*)
6. The nurse logs into the room's terminal (*update status call and nurse, open lockers*)
7. The nurse logs out again (*update status call and nurse, close lockers*)
8. The nurse leaves the room (*update location and nurse status and turn off lights*)

Results

The aforementioned scenario was run 35 times, consisting of 3 warm-up runs and 2 cool-down runs, for 1 ward and 10 wards, for both rule sets. By averaging the 30 remaining reasoning times per amount of wards and Implementation type, we provide the results shown as a table in Figure 6.1.

The reasoning times of our strategies using N3 are in most cases better than the reasoning times of DL+SPARQL. We furthermore observe, that the N3 implementations have far more predictable reasoning times, as they are more robust against more complex decision trees (e.g. the decision trees of the

wards	1 ward							
event	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
DL+SPARQL	45.8	21.4	0.1	2.4	2.4	3.0	1.6	2.2
RL+Rules	2.1	2.1	2.4	2.1	2.1	2.2	2.4	2.1
prep+RL+Rules	0.4	0.6	0.7	0.4	0.4	0.4	0.5	0.4
wards	**10 wards**							
event	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
DL+SPARQL	1125.7	755.6	0.1	67.0	66.7	25.6	174.9	65.8
RL+rules	30.7	30.7	34.9	30.6	30.6	30.7	35.0	30.5
prep+RL+rules	6.8	10.7	12.2	8.1	8.0	6.7	9.3	8.1

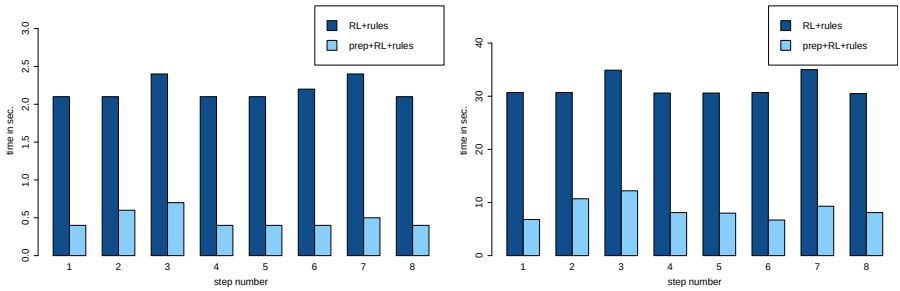
Figure 6.1: Reasoning times using (1) DL+SPARQL querying, (2) RL+rules in N3, and (3) a preprocessed version of the latter in seconds. OWL RL combined with rules performs better than OWL DL and SPARQL. Preprocessing further reduces reasoning times.

first two events are notably more complex than the other events' decision trees). The implementation using DL+SPARQL is much faster than both N3 implementations in the third event, because this event does not trigger any reasoning for DL+SPARQL, however, a full reasoning cycle is performed by the N3-implementations. With an average reasoning time of about 2 seconds, the N3 implementation without pre-processing achieves the real-time constraint within small-scale datasets.

To better understand the impact of the preprocessing step, we further compare the reasoning times of the two implementations using N3. These are depicted in Figure 6.2. The figure shows that preprocessing the rules improves reasoning times significantly, consistently requiring only a quarter of the reasoning time. This trend manifests itself regardless of the amount of dynamic data involved. Whereas the traditional rule set can no longer be used in a hospital with 10 wards the preprocessed rule set still provides reasonable reasoning times.

6.1.7 Discussion

Our analysis has shown that the implementation using OWL RL and rules performs much faster than the one using OWL DL and SPARQL querying. The results could be further improved by using rule-producing rules, a special feature of N3. In this particular use case we can positively answer our initial question whether N3 reasoning can solve the same practical problems as OWL DL reasoning and SPARQL querying with comparable performance.



(a) 1 ward, reasoning time per event. **(b)** 10 wards, reasoning time per event.

Figure 6.2: Comparison of reasoning times using preprocessed and traditional RL rules in N3. The preprocessing step improves reasoning times.

We can even say more: Here, the N3 solutions outperform OWL DL and SPARQL. But we also need to be careful: The OWL DL profile is of course more expressive than OWL RL and our good results were achieved by ignoring some derivations. In our use case these derivations were not relevant for the queries we wanted to answer and here we also see the actual advantage of using N3: We can customise the reasoning according to our needs. From the various informations which are stated by means of an OWL DL ontology we chose which ones we wanted to use for further reasoning. If we need to be more expressive than OWL RL we can extend our rules set and use – for example – rules with new blank nodes in the consequent to support OWL ER [118], an OWL profile which can be supported by existential rules. Powerful built-in functions provide us with even more options. By limiting the information we actually use for our conclusions, we can even reason over OWL Full ontologies and thereby accept an OWL format which is semantically compatible with RDF.

6.2 Use case 2: Data validation

Our second use case is about data validation. Even though the amount of publicly available Linked Open Data (LOD) sets is constantly growing[6], the diversity of the data employed in applications is very limited. Only a handful of RDF data is used frequently [119]. One of the reasons for this is that the datasets' quality and consistency varies significantly, ranging from expensively curated to relatively low quality data [120], and thus need to be validated carefully before use.

[6] See, e.g. statistics at: http://lod-cloud.net/.

One way to assess data quality is to check them against constraints: Users can verify that certain data are fit for their use case, if the data abide to their requirements. First approaches to do that were implementations with hard coded validation rules, such as Whoknows? [121]. Lately, attention has been drawn to formalizing RDF quality assessment, more specifically, formalizing RDF constraints languages, such as Shape Expressions (ShEx) [122] or Resource Shapes (ReSh) [123]. This detaches the specification of the constraints from its implementation.

Constraint languages allow users dealing with RDF data and vocabularies to express, communicate, and test their particular expectations. Such languages can either be (i) existing frameworks designed for different purposes, e.g. SPARQL [124, 125], or OWL [126], or they can be (ii) languages only designed for validation, e.g. ShEx [122], ReSh [123], Description Set Profiles (DSP) [127], or the W3C recommended Shapes Constraint Language (SHACL) [128]. These different languages can be compared by testing them on commonly supported constraints [125, 129], as conducted by Hartmann (né Bosch) et al. [130].

Depending on the users' needs, constraint languages have to be able to cope with very diverse kinds of constraints which imply certain *logical requirements*. Such requirements were investigated by Hartmann et al. [130] who identified the Closed World Assumption (CWA) (see also Section 2.3.1) and the Unique Name Assumption (UNA) – the assumption that each resource in the domain of discourse does have exactly one unique name in the logical language – as crucial for validation. Since both are not supported by many Web Logics, Hartmann et al. particularly emphasize the difference between reasoning and validation languages and favour SPARQL-based approaches for validation which – if needed – can be combined with RDF, RDFS or OWL entailment. In this section, we take a closer look into these findings from a rule-based perspective: We show that neither UNA nor CWA are necessary for validation if a rule-based framework containing predicates to compare URIs and literals, and supporting Scoped Negation as Failure (SNAF) is used. This enables us to – instead of combining separate, successive systems – do both RDF validation *and* reasoning in only one system which acts directly on a constraint language. We show the feasibility of this approach by providing an implementation in Notation3 Logic. We tackle those constraints identified by Hartmann et al. [130] which are also covered in RDFUnit [125], a validation system which is purely based on SPARQL querying. We then compare our implementation to the latter. Our solution functionally outperforms RDFUnit and our execution time is faster for small datasets containing 100.000 triples or less. We can thus perform a task which is often solved by using SPARQL querying with N3Logic and obtain comparable results.

6.2.1 An overview of RDF validation

Below we first present the state of the art around validation constraint languages. Then, we will give an overview of different languages and approaches used for RDF validation.

Data quality can be described in many dimensions, one of them being the intrinsic dimension, namely, the adherence to a data schema [120]. In the case of RDF data, this implies adhering to certain constraints. These have been carefully investigated by several authors (e.g. Hartmann et al. [124]). The formulation of (a subset of) these constraints can be done using existing languages (e.g. OWL) [25], the SPARQL Inferencing Notation (SPIN) [131], or SPARQL [15])), or via dedicated languages (e.g. Shape Expressions (ShEx) [122], Resource Shapes (ReSh) [123], Description Set Profiles (DSP) [127], or Shapes Constraint Language (SHACL) [128]. Their execution is either based on reasoning frameworks, or querying frameworks.

On the one hand, Motik et al. [132] and Sirin and Tau [133] propose alternative semantics for OWL which support the Closed World Assumption, and are therefore more suited for constraint validation than the original version. To know which semantics apply, constraints have to be marked as such. Using one standard to express both, validation and reasoning, is a strong point of this approach, however, this leads to ambiguity: If the exact same formula can have different meanings, one of the key properties of the Semantics Web – interoperability – is in danger. Another disadvantage of using (modified) OWL as a constraint language is its limited expressiveness. Common constraints such as mathematical operations or specific checks on language tags are not covered by OWL [124].

On the other hand, SPARQL based querying frameworks for validation execution emerged (e.g. Hartmann [124] or Kontokostas et al. [125]). Where Hartmann proposes SPIN as base language to support validation constraints, Kontokostas introduces a similar but distinct language to SPIN, more targeted to validation, so-called Data Quality Test Patterns (DQTP). DQTPs are generalized SPARQL queries containing an extra type of variables. In an extra step, these variables are instantiated based on the RDFS and OWL axioms used by the data schema and can then be employed for querying. As such, the authors assume a closed world semantics for OWL but in contrast to the approaches mentioned above, this special semantics cannot be marked in the ontology itself. They thus change the semantics of the common Web standard OWL. To also find implicit constraint validation an extra reasoning step could be added, but this step would then most probably assume the standard semantics of OWL, further increasing the possibly of experiencing conflicts between the two contradicting versions of the semantics. Hartmann

proposes a dedicated ontology to express integrity constraints, and as such, this method does not involve changing existing semantics. For both, only very limited reasoning can be included directly in the system.

6.2.2 RDF Validation Constraints

Based on the collaboration of the W3C RDF Data Shapes Working Group[7] and the DCMI RDF Application Profiles Task Group[8] with experts from industry, government, and academia, a set of validation requirements has been defined, based on which, 81 types of constraints were published, each of them corresponding to at least one of the validation requirements [129]. This set thus gives a realistic and comprehensive view of what validation systems should support.

Prior to this, the creators of RDFUnit [125] had provided their own set of constraint types they support. Given the usage of RDFUnit in real-world use cases [134], this set gives a good overview of what validation systems should minimally cover.

Table 6.1 shows the alignment of the 17 types of constraints as supported by RDFUnit with the relevant constraint types as identified by Hartmann et al. [130]. As can be seen, these types are not mapped one-to-one. One constraint from RDFUnit maps to at least one constraint as identified by Hartmann, except for PVT and TRIPLE, which are both not very complex constraints and could thus easily be added to the work of Hartmann et al. Here, we mainly focus on these 17 constraints which are all covered by our implementation. To make the topic of constraint validation more concrete, we discuss the examples (TYPEDEP), (INVFUNC) and (MATCH) in more detail below and refer the reader interested in the other constraint types to the above mentioned sources.

6.2.3 Features required for Validation

After having listed the kind of constraints relevant for RDF validation in the previous section, we will now focus on the suitability of rule-based logics for that task. Based on the work of Sirin and Tao [133], and Hartmann et al. [130] who identified the logical requirements constraint languages need to fulfil, we discuss why rule-based logic is a reasonable choice to validate RDF datasets.

[7] https://www.w3.org/2014/data-shapes/wiki/Main_Page

[8] http://wiki.dublincore.org/index.php/RDF_Application_Profiles

Table 6.1: Constraints Alignment. The first column lists the codes as used in RDFUnit; the second the constraints of Hartmann following his numbering [124, appendix]; and the third the description taken from RDFUnit.

RDFUnit	**Constraint Code**	**Description**
COMP	A11	Comparison between two literal values of a resource
MATCH	A20, A21	A resource's literal value (does not) matches a RegEx
LITRAN	A17, A18	The literal value of a resource (having a certain type) must (not) be within a specific range
TYPEDEP	A4	Type dependency: the type of a resource may imply the attribution of another type
TYPRODEP	A41	A resource of specific type should have a certain property
PVT	B1	If a resource has a certain value V assigned via a property P1 that in some way classifies this resource, one can assume the existence of another property P2
TRIPLE	B2	A resource can be considered erroneous if there are corresponding hints contained in the dataset
ONELANG	A28	A literal value has at most one literal for a language
RDFS-DOMAIN	A13	The attribution of a resource's property (with a certain value) is only valid if the resource is of a certain type
RDFS-RANGE	A14, A15	The attribution of a resource's property is only valid if the value is of a certain type
RDFS-RANGED	A23	The attribution of a resource's property is only valid if the literal value has a certain datatype
INVFUNC	A2	Some values assigned to a resource are considered to be unique for this particular resource and must not occur in connection with other resources
OWL-CARD	A1, A32–37	Cardinality restriction on a property
OWLDISJC	A70	Disjoint class constraint
OWLDISJP	A69	Disjoint property constraint
OWL-ASYMP	A57	Asymmetric property constraint
OWL-IRREFL	A64	Irreflexive property constraint

Reasoning

We start our discussion with reasoning. Hartmann [124, p. 181] points out that performing reasoning in combination with RDF validation brings several benefits: constraint violations may be solved, violations which otherwise would stay undetected can be found, and datasets do not need to contain redundant data to be accepted by a validation engine. To better understand these benefits, consider the following ontology example:[9]

```
:Reseacher rdfs:subClassOf :Person.        (6.7)
```

And the instance:

```
:Kurt a :Researcher; :name "Kurt01".       (6.8)
```

If we now have a type dependency constraint (TYPEDEP) saying that every instance of the class `:Researcher` should also be an instance of the class `:Person`, which we test on the data above, a constraint validation error would be raised since `:Kurt` is not declared as a `:Person`. If we perform the same constraint check after reasoning, the triple

```
:Kurt a :Person.                           (6.9)
```

would be derived and the constraint violation would be solved. Without the reasoning, Triple 6.9 would need to be inserted into the dataset to solve the constraint, leading to redundant data.

To understand how reasoning can help to detect implicit constraints, consider another restriction: suppose that we have a constraint stating that a person's name should not contain numbers[10]. Without reasoning, no constraint validation would be detected because even though the `:name` of `:Kurt` contains numbers, `:Kurt` would not be detected as an instance of `:Person`.

Hartmann's and many other validation approaches thus suggest to first perform a reasoning step and then do an extra validation step via SPARQL querying. The advantage of using rule-based reasoning instead is that validation can take place during the reasoning process *in one single step*. Relying on a rule which supports `rdfs:subClassOf` as presented in Section 6.1.3 the aforementioned problem could be detected. In general, OWL-RL [41] can be applied since it is supported by every rule language. If higher complexity is needed, rule languages with support for existential quantification can be used for OWL QL reasoning.

[9] As above, the prefix `rdfs:` stands for `http://www.w3.org/2000/01/rdf-schema#`.

[10] This could be expressed by an extended version of MATCH as for example the constraint "Negative Literal Pattern Matching" in [124].

Scoped Negation as Failure

Another aspect which is important for constraint validation is negation. Hartmann et al. claim that the Closed World Assumption is needed to perform validation tasks. Given that most Web logics assume the Open World Assumption, that would form a barrier for the goal of combining reasoning and validation mentioned in the previous section. Luckily, that is not the case. As constraint validation copes with the local knowledge base, Scoped Negation as Failure (SNAF) (see also Section 2.3.1 and for example [17, 55, 56]) is enough. This is for example supported by FLORA-2 [135] or N3Logic [5].[11]

In order to understand the idea behind Scoped Negation as Failure, consider the triples that form Formula 6.8 and suppose that these are the only triples in a knowledge base we want to validate. We now want to test the constraint from above that every individual which is declared as a researcher is also declared as a person (TYPEDEP). This means our system needs to give a warning if it finds an individual which is declared as a researcher, but not as a person:

$$\forall x : ((\texttt{ x a :Researcher}) \wedge \neg(\texttt{ x a :Person})) \rightarrow (\texttt{:constraint :is :violated.}) \quad (6.10)$$

In the form it is stated before, the constraint cannot be tested with the Open World Assumption. The knowledge base contains the triple

```
:Kurt a :Researcher.
```

but not Triple 6.9, but the rule is more general: given its open nature, we cannot guarantee that there is no document in the entire Web which declares Triple 6.9. This changes if we make an addition. Suppose that $\mathcal{K}$ is the the set of triples we can derive (either with or without reasoning) from our knowledge base consisting of Formula 6.8. Having $\mathcal{K}$ at our disposal, we can test:

$$\forall x : ((\texttt{ x a :Researcher}) \in \mathcal{K}) \wedge \neg((\texttt{ x a :Person}) \in \mathcal{K})) \rightarrow (\texttt{:constraint :is :violated.}) \quad (6.11)$$

[11] As we will discuss below, SNAF in N3 is mostly supported by built-in functions. As we explicitly excluded these from our discussion in Chapter 4, our semantics does not cover SNAF yet. To ensure monotonicity a proper formalisation would need to follow the contextually closed semantics with context sets as defined by Polleres et al. [55] using stable model semantics.

The second conjunct is not a simple negation, it is a negation with a certain scope, in this case $\mathcal{K}$. If we know that $\mathcal{K}$ is closed and has a closed context, i.e. it does not rely on any construct whose scope is not specified, we can guarantee monotonicity [55]. If we added new data to our knowledge like for example Triple 6.9, we would have a different scope $\mathcal{K}'$ for which other statements hold. The truth value of the formula above would not be touched since this formula explicitly mentions the closed scope $\mathcal{K}$. Scoped negation as failure is the kind of negation we actually need in RDF validation: we do not want to make and test statements in the Web in general, we just want to test the information contained in a local file or knowledge base.

Predicates for Name Comparison

Next to the Open World Assumption, Hartmann et al. [130] identify the fact that most Web logics do not base themselves on the Unique Names Assumption (UNA) as a barrier for them being used for constraint validation. This assumption is for example present in F-Logic [136] and basically states that every element in the domain of discourse can only have one single name (URI or Literal in our case). The reason, why this assumption is in general problematic for the Semantic Web lies in its distributed nature: different datasets can – and actually do – use different names for the same individual or concept. For instance, the URI `dbpedia:London` refers to the same place in England as for example `dbpedia-nl:London`.[12] In this case this fact is even stated in the corresponding ontologies using the predicate `owl:sameAs`.[13]

The impact of the Unique Name Assumption for RDF validation becomes clear if we take a closer look at OWL's inverse functional property and the related constraint (INVFUNC). Let us assume that `dbo:capital`[14] is an `owl:InverseFunctionalProperty` and our knowledge base contains:

```
:England dbo:capital :London.
:Britain dbo:capital :London.          (6.12)
```

Since `:England` and `:Britain` are both stated as having `:London` as their capital and `dbo:capital` is an inverse functional property, an OWL reasoner would derive

```
:England owl:sameAs :Britain.          (6.13)
```

[12] `dbpedia` stands for `http://dbpedia.org/resource/`, `dbpedia-nl` for `http://nl.dbpedia.org/resource/`.

[13] As above `owl` is the prefix for `http://www.w3.org/2002/07/owl#`.

[14] With `dbo` standing for `http://dbpedia.org/ontology/`.

Such a derivation cannot be made if the Unique Name Assumption is valid, since the former explicitly excludes this possibility.

The constraint (INVFUNC) is related to the OWL concept above, but it is slightly different: while OWL's inverse functional property refers to the elements of the domain of discourse denoted by the name, the validation constraint (INVFUNC) refers to the representation itself. Formula 6.12 thus violates the constraint. Even if our logic does not follow the Unique Name Assumption, this violation can be detected if the logic offers predicates to compare names. In N3Logic, `log:equalTo` and `log:notEqualTo`[15] are such predicates: in contrast to `owl:sameAs` and `owl:differentFrom`, they do not compare the resources they denote, but their representation. The idea to support these kinds of predicates is very common. So does, for example, the Rule Interchange Format (RIF) cover several functions which can handle URIs and strings, as we will discuss in the next subsection.

RIF Built-ins

In the previous subsection we indicated that a special predicate of a logic, in this case `log:notEqualTo`, can be used to do URI comparisons and thereby support a concept which would otherwise be difficult to express. Such built-in functions are widely spread in rule-based logics and play an important role in RDF validation which very often deals with string comparisons, calculations or operations on URI level. While it normally depends on the designers of a logic which features are supported, there are also common standards.

The most important standard for rules in the Semantic Web is RIF which was developed to exchange rules in the Web. Being the result of a W3C working group consisting of developers and users of different rule-based languages, RIF can also be understood as a reference for the most common features rule based logics might have. This makes the list of predicates [137] supported by the different RIF dialects particularly interesting for our analysis. And it is indeed the case that by only using RIF predicates many of the constraints listed in Section 6.2.2 can already be checked: negative pattern matching (MATCH) can be implemented by using the predicate `pred:matches`,[16] the handling of language tags as required for the constraint ONELANG can be done using `func:lang-from-PlainLiteral`,[17] and for the comparison of literal

[15] https://www.w3.org/2000/10/swap/doc/CwmBuiltins.

[16] `PREFIX pred: <http://www.w3.org/2007/rif-builtin-predicate#>`.

[17] `PREFIX func: <http://www.w3.org/2007/rif-builtin-function#>`

values (COMP) there are several predicates to compare strings, numbers or dates.

To illustrate how powerful RIF is when it comes to string validation, we take a closer look at the predicate `log:notEqualTo` from the previous section. In the example above it is used to compare two URI representations and succeeds if these these two are different. To refer to a URI value, RIF provides the predicate `pred:iri-string` which converts a URI to a string and and vice versa. In N3 notation[18] that could be expressed by:

```
(:England "http://exmpl.com/England")
                              pred:iri-string true.   (6.14)
```

To compare the newly generated strings, the function `func:compare` can be used. This function takes two string values as input, and returns -1 if the first string is smaller than the second one regarding a string order, 0 if the two strings are the same, and 1 if the second is smaller than the first. The example above gives:

```
("http://exmpl.com/Britain"
            "http://exmpl.com/England")
                                  func:compare -1.   (6.15)
```

To enable a rule to detect whether the two URI names are equal, one additional function is needed: the reasoner has to detect whether the result of the comparison is not equal to zero. That can be checked using the predicate `pred:numeric-not-equal` which is the RIF version of $\neq$ for numeric values. In the present case the output of the comparison would be `true` since $0 \neq 1$, a rule checking for the name equality of `:England` and `:Britain` using the three predicates would therefore be triggered.

Even though we needed three RIF predicates to express one N3 predicate, the previous example showed how powerful built-ins in general – but also the very common RIF predicates in particular – are. Whether a rule based Web logic is suited for RDF validation highly depends on its built-ins. If it supports all RIF predicates, this can be seen as a strong indication that it is expressive enough.

6.2.4 Validation with N3Logic

In the previous section we analysed the requirements on a rule-based Web logic to be able to combine validation and reasoning: it should support scoped

[18] The EYE reasoner supports RIF predicates. These need to be used in N3 syntax.

```
PREFIX rdfcv:  <http://www.dr-thomashartmann.de/phd-thesis/
rdf-validation/vocabularies/rdf-constraints-vocabulary#>
PREFIX : <http://example.org/ex#>
PREFIX dbo:  <http://dbpedia.org/ontology/>

:example_constraint a rdfcv:SimpleConstraint ;
  :constraintType :InverseFunctionalProperties ;
  rdfcv:constrainingElement :inverse-functional-properties ;
  rdfcv:leftProperties ( dbo:capital ) ;
  rdfcv:contextClass  dbo:Country .
```

Listing 16: Example inverse functional property constraint: No city can be the capital of two countries.

negation as failure, it should provide predicates to compare different URIs and strings, and its built-in functions should be powerful enough to, inter alia, access language tags and do string comparison as they are supported by RIF. N3Logic as it is implemented in the EYE reasoner [88] fulfils all these conditions. With that logic, we were able to implement rules for all the constraints listed in Section 6.2.2, and thus provide similar functionality as RDFUnit using rule-based Web logics. Below we discuss the details of this implementation. The rules for our implementation can be accessed at https://github.com/IDLabResearch/data-validation.

Expressing Constraints

Before we can detect violations of constraints using N3 logic, these constraints first need to be stated. This could either be done by directly expressing them in rules – and thereby creating a new constraint language next to the ones presented in Section 6.2.1 – or on top of existing RDF-based conventions. We opt for the latter and base our present implementation on the work of Hartmann [124, p.167 ff]: in his PhD thesis, Hartmann presents a lightweight vocabulary to describe any constraint, the RDF Constraints Vocabulary (RDF-CV)[19]. The reason why we chose that vocabulary over the upcoming standard SHACL is its expressiveness. We aim to tackle the 81 constraints identified by Hartmann which are not all expressible in SHACL or any other of the constraint languages mentioned in Section 6.2.1 [124, p.52, appendix].[20] As will be shown in the following section, it is not difficult to

[19] https://github.com/boschthomas/RDF-Constraints-Vocabulary

[20] After Hartmann published his PhD thesis, SHACL was further improved. An apdated version of the comparison of the expressivity of the different constraint languaeges can be found in our recent paper [11].

adopt the rules to different constraint languages as long as they are based on RDF and as such valid N3 expressions.

RDF-CV supports the concept of so called *simple constraints* which are all the constraints expressible by the means of the vocabulary, in particular the ones mentioned in Section 6.2.2. Each simple constraint has a constraining element. Where applicable, the names of these elements are inspired by their related DL names, but the constraining element can also be for example the name of a SPARQL function. In some cases, the same constraint type can be marked by different constraining elements as for example the constraint COMP whose constraining element is the relation used to compare values (e.g. the usual numerical orders: $<, >, \leq$, and $\geq$) or there can be different constraint types sharing the same constraining element. To be sure that cases like this do not cause any ambiguity we additionally assign a *constraint type* to every constraint. The names of these types follow the names used by Hartmann [124, appendix]. The TYPEDEP constraint from Section 6.2.3 is for example of constraint type `:Subsumption`.

In addition to constraining element and constraint type, there are several predicates to assign the constraints to individuals and classes: *context class*, *classes*, *leftProperties*, *rightProperties*, and *constraining values*. The *context class* of a constraint fixes the set of individuals for which a constraint must hold. For the subsumption constraint mentioned above, that would be the class `:Researcher`, the constraint talks about *every* individual labelled as *researcher*. There could be other classes involved. In our subsumption example that is the superclass the individuals should belong to, `:Person`. Every researcher should also be labelled as *person*. Since these kinds of properties can be multiple, they are given in a list. How and if the predicate *classes* is used depends on the constraint. The predicates *leftProperties* and *rightProperties* are used to do similar statements about properties. The constraint INVFUNC as displayed in Listing 16 makes for example use of it to relate the constraint specified to the predicate it is valid for. The objects of the predicates *leftProperties* and *rightProperties* are lists. The predicate *constraining value* is used for the predicates where a literal value is needed to further specify a constraint. An example for such a constraint is MATCH as described in Section 6.2.3. To express, that a name should not contain numbers, the predicate *constraining value* connects the constraint to the string pattern, `"[1-9]"` in the present case.

Constraint Rules

Having seen in the last section one possible way to describe constraints on RDF datasets, this section explains how these descriptions can be used. We

```
1  PREFIX rdfcv:  <http://www.dr-thomashartmann.de/phd-thesis/
2  rdf-validation/vocabularies/rdf-constraints-vocabulary#>
3  PREFIX : <http://example.org/ex#>
4  PREFIX list:  <http://www.w3.org/2000/10/swap/list#>
5  PREFIX log:  <http://www.w3.org/2000/10/swap/log#>
6
7  {
8  ?constraint a rdfcv:SimpleConstraint;
9    :constraintType :InverseFunctionalProperties;
10   rdfcv:constrainingElement :inverse-functional-properties;
11   rdfcv:leftProperties ?list;
12   rdfcv:contextClass  ?Class.
13
14 ?object a ?Class.
15 ?property list:in ?list.
16 ?x1 ?property ?object.
17 ?x2 ?property ?object.
18 ?x1 log:notEqualTo ?x2
19 }
20 =>
21 {
22 [] a :constraintViolation;
23     :violatedConstraint ?constraint.
24 }.
```

Listing 17: Rule for inverse functional property (INVFUNC). The predicate `log:notEqualTo` compares the resources based on their URI and thereby supports the Unique Name Assumption.

employ rules which take the expressed constraints and the RDF dataset to be tested into account and generate triples indicating constraint validations, if present. We illustrate that by an example: In Listing 17 we provide a rule handling the constraint INVFUNC. Lines 7–11 contain the details of the constraint. The rule applies for simple constraints of the type *inverse functional properties* for which a context class `?Class` and a list `?list` of left properties is specified. This part of the rule's antecedence unifies with the constraint given in Listing 16. Lines 13–18 describe which situation in the tested data causes a constraint violation: for an `?object` which is an instance of `?Class`, there are two subjects, `?x1` and `?x2`, defined which are both connected to `?object` via `?property`. This `?property` is an element of `?list`, and the names, i.e. the URI- or string-representations, of `?x1` and `?x` differ. The latter is expressed using the predicate `log:notEqualTo`[21] (Line 18). Together with Listing 16 that violation is thus detected if two different resource names for resources of the class `dbo:Country` are connected via

[21] As explained in Section 6.2.3 there are alternative ways to express the predicate `log:notEqualTo` in N3, the antecedence of the entire rule could also be expressed only using RIF predicates.

```
1 PREFIX rdfcv:  <http://www.dr-thomashartmann.de/phd-thesis/
2 rdf-validation/vocabularies/rdf-constraints-vocabulary#>
3 PREFIX : <http://example.org/ex#>
4
5 _:example_constraint_2 a rdfcv:SimpleConstraint ;
6   :constraintType :Subsumption ;
7   rdfcv:constrainingElement :sub-class ;
8   rdfcv:contextClass  :Researcher;
9   rdfcv:classes :Person .
```

Listing 18: Example constraint (TYPEDEP). This constraint tests whether the derivations implied by Formula 6.7 have been produced.

the predicate `dbo:capital` to the same object. Assuming that `:Britain` and `:England` are both instances of the class `dbo:Country`, the triples in Formula 6.12 lead to the violation:

```
_:x a :violaton;
      :violatedConstraint :example_constraint.   (6.16)
```

The example relies on descriptions following the vocabulary Hartmann suggests, but our approach can easily be adapted for other constraint vocabularies whose syntax is based on RDF. We only need rules to translate from one concept to another. Similarly, we can generate constraints we want to test later on using rules, this is what we explain below.

Generating constraint descriptions

One use case for constraint testing – apart from simple quality assurance – is to test whether a reasoner or any other application which automatically adds triples to a triple store works properly. To better understand this special case and how this can be approached using N3, we come back to our reasoning example above. Via subclass reasoning an OWL DL reasoner should derive Formula 6.9 from the TBox-triple 6.7 and the ABox-formula 6.8.

Above, we discussed, how this reasoning can also be performed by using N3. Here, instead of applying this reasoning, we want to test, whether the data set which is produced by another system contains Formula 6.9 given that the Formulas 6.7 and 6.8 are also present. To do so, we can declare a TYPEDEP constraint as displayed in Listing 18. But in this case – testing the output of a reasoner – we already know that we want to perform a similar test for all `rdfs:subClassOf`-declarations in our ontology's TBox. We can thus use a rule to automatically produce the constraint descriptions. We display such a rule in Listing 19. Applied on Formula 6.7 we get an instance of The

```
PREFIX rdfcv:  <http://www.dr-thomashartmann.de/phd-thesis/
rdf-validation/vocabularies/rdf-constraints-vocabulary#>
PREFIX : <http://example.org/ex#>
PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>

{
?x rdfs:subClassOf ?y
}
=>
{
_:constraint a rdfcv:SimpleConstraint ;
  :constraintType :Subsumption ;
  rdfcv:constrainingElement :sub-class ;
  rdfcv:contextClass  ?x;
  rdfcv:classes ?y .
}.
```

Listing 19: Rule producing a (TYPEDEP) constraint from an `rdfs:subClassOf` declaration. Such rules can be used to test reasoners.

formula displayed in Listing 18. We can write similar rules for other ontology concepts and we can translate other constraint vocabularies to RDF-CV.

This translation can be performed in one single reasoning run together with the validation. By using translation or generation rules together with validation rules we which can be selected according to the user's need we have a very strong and at the same time very flexible system.

6.2.5 Evaluation

In the previous section we described how we can use N3 to perform constraint testing and – if necessary – combine these tests with constraint generation or ontology reasoning. We call our implementation "Validatrr": a validator using rule-based reasoning. Validatrr makes use of the EYE reasoner. To be able to compare our implementation with existing approaches, a Node.js JavaScript framework was created to discover and retrieve vocabularies and ontologies, manage command line arguments and run tests. This framework is available at https://github.com/IDLabResearch/validatrr.

For our evaluation, we compare Validatrr with an implementation based on SPARQL querying: RDFUnit [125]. We want to know how Validatrr performs in terms of accuracy – that is, does it correctly detect the same constraint violations – and execution times. Below, we discuss both aspects separately.

Table 6.2: Comparison of RDFUnit and Validatrr applying no extra inference $\emptyset$, and using the rule set υ. Validatrr(υ) finds more violations. Test cases where Validatrr outperforms RDFUnit are starred. Rows where the results differ are marked grey.

Test Case	# found violations		
	RDFUnit	Validatrr	
	υ	$\emptyset$	υ
INVFUNC_correct	0	0	0
INVFUNC_wrong	2	0	2
OWLCARDT_correct	0	0	0
OWLCARDT_wrong_exact	6	6	6
OWLCARDT_wrong_max	2	2	2
OWLCARDT_wrong_min	2	2	2
OWLDISJC_correct	0	0	0
OWLDISJC_wrong	6	2	6
OWLQCARDT_correct	0	0	0
OWLQCARDT_wrong_exact	6	6	6
OWLQCARDT_wrong_max	2	2	2
OWLQCARDT_wrong_min	2	2	2
RDFLANGSTRING_correct	0	0	0
RDFLANGSTRING_wrong	2	2	2
*RDFSRANGE-MISS_wrong**	1	3	3
RDFSRANGED_correct	0	0	0
*RDFSRANGED_wrong**	2	3	3
*RDFSRANGE_correct**	0	5	4
*RDFSRANGE_wrong**	1	3	3
RDFSRANG_LIT_correct	0	0	0
RDFSRANG_LIT_wrong	3	3	3

Accuracy

To test for both implementations whether they correctly detect all constraint violations they are supposed to find when provided with a set of constraint descriptions and files to check for those we used the unit tests written for RDFUnit which we directly took from the RDFUnit repository[22]. As mentioned above, constraints in RDFUnit are expressed using SPARQL patterns, so-called DQPTs. We translated these DQTPs to RDF-CV according to the mapping given in Table 6.1. Apart from this, there was no difference in the input sets of both validation systems and we used all test patterns available[23]. RDFUnit furthermore uses a custom set of inference rules: each resource is an `rdfs:Resource` and the construct `rdfs:subclassOf` is understood according to its meaning defined in RDFS semantics [35]. We call this set of rules υ.

In Table 6.2 we display the results of our tests using RDFUnit, Validatrr

[22] https://github.com/AKSW/RDFUnit/tree/master/rdfunit-core/src/test/resources/org/aksw/rdfunit/validate/data

[23] For a detailed description of these test patterns we refer to the original publication of RDFUnit [125].

without extra reasoning ($\emptyset$) and Validatrr using the rules υ. We list the name of the test cases together with the number of constraint violations each implementation found. We marked each test case for which Validatrr finds more violations than RDFUnit with a star. These are the cases *rdfsrange-miss_wrong*, *rdfsranged_wrong*, *rdfsrange_correct* and *rdfsrange_wrong*. All these cases contain the restriction type *multiple range* which is not yet supported by RDFUnit: when the predicate `:multRanges` is used, each resource linked as an object to that predicate should be classified into multiple classes. For all other cases the number of encountered constraints is the same for both implementations when applying the same set of inference rules. Our approach thus outperforms RDFUnit.

Performance

After comparing both implementations in terms of accuracy, we now want to have a closer look at their execution time. To test this, we again follow RDFUnit's original evaluation method. RDFUnit derives constraints from the existing schemas FOAF, GeoSPARQL, OWL, DC terms, SKOS, and Prov-O. The idea behind that is similar as we discussed it above when we showed how a rule can be used to generate a constraint from the concept `rdfs:subClassOf`. RDFUnit uses a step to first generate DQPTs based on the concepts found in an ontology while we use extra rules.

For our concrete test we used graphs following these schemas. At most ten different RDF graphs – per schema, per RDF graph size – were downloaded, by querying LODLaundromat's SPARQL endpoint [138]. In total we downloaded 300 graphs with sizes ranging from ten to one million triples. For each of these datasets constraints were created according to the schema they were following and then tested. The tests were executed on a machine consisting of 24 cores (Intel Xeon CPU E5-2620 v3 @ 2.40GHz) and 128GB RAM. All evaluations were performed using untampered docker images for both approaches to maintain reproducibility, the different tests were orchestrated using custom scripts. All timings include the docker images' initialization time. The data is available at https://github.com/IDLabResearch/validation-benchmark/tree/master/data/validation-journal.

In Figure 6.3 we show the results of our tests: We grouped the different datasets of all schemas by their size and compare the median execution time when performing validation taking into account the rule set υ. We clearly see that for small datasets containing less than 100,000 triples our solution is faster than RDFUnit.

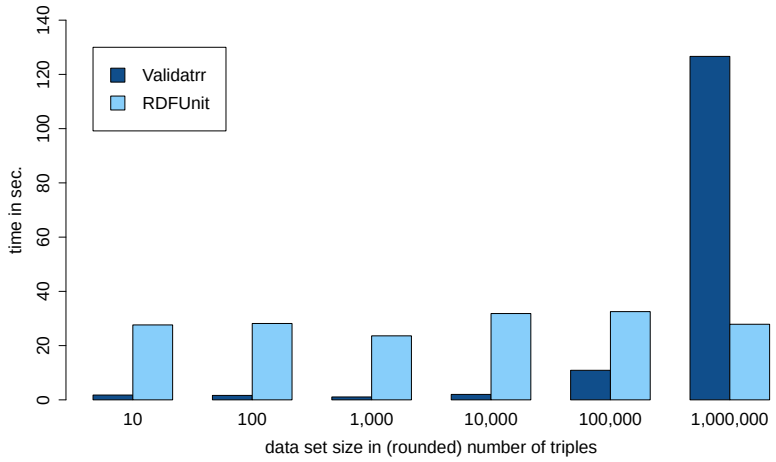


Figure 6.3: Validatrr's execution speed (dark blue) is up to an order of magnitude faster than RDFUnits's (light blue) when the number of triples per RDF graph is below 100,000 triples.

6.2.6 Discussion

Above we compared our rule-based RDF validation system Validatrr with another state-of-the art system providing that service. The difference between Validatrr and RDFUnit lies in the technologies used: RDFUnit first instantiates SPARQL patterns (DQPTs) using a programming script and then applies these queries while Validatrr uses N3Logic for both of these steps. This quality of a *Unifying Logic*– supporting constraint generation and testing at the same time – made that our approach performed faster on small datasets. Similarly, more advanced reasoning as for example discussed in Section 6.1.3 could be added. Validatrr was furthermore more accurate on the test cases we performed and could correctly spot more constraint violations.

If we come back to our initial question "Can N3 solve the same practical tasks as SPARQL querying without suffering a loss of performance?" we can again give a positive answer for our use case: RDF validation can be performed by using N3 reasoning and for smaller datasets N3 outperforms the solution using SPARQL. But we need again to be careful with that answer: Our positive results could be achieved because N3 as implemented in the EYE reasoner offers three important features which are crucial for data validation: It supports SNAF, it provides predicates to compare names (and thereby makes an UNA unnecessary), and it has powerful built-ins to for example,

do string comparisons or access language tags. Not all of these features are covered in the original specification about N3Logic [5, 71] and it depends on the discussion within the community[24] whether or not these become an official part of that logic. But if this is the case, N3 is powerful enough to support all constraint types which are covered by RDFUnit.

6.3 Conclusion

In order to answer the question whether N3Logic can solve the same practical problems as SPARQL and OWL DL reasoning with a comparable performance we considered two use cases: A semantic nurse call system and a system for RDF validation.

The nurse call system has been previously implemented using OWL DL reasoning to draw conclusions from the ontology involved and SPARQL querying to select a nurse based on the reasoning result. We implemented a system which combined ontology reasoning and querying using only one framework: N3Logic. To draw conclusions from the knowledge expressed in the ontology, we applied OWL RL reasoning. The decision tree for assigning nurses to a call depending on the current context has been implemented by directly using N3 rules. We compared the execution times of both implementations and saw that the solution using N3 performed faster which was caused by two reasons: Firstly, we only took the OWL concepts which were relevant for our use case into account and not all inference steps included in the profile OWL DL. This optimisation could be made because N3Logic is very flexible. Using rules we can choose which OWL concepts we want to support. Since N3 also supports existential rules and is in most reasoners enriched with various built-ins, we can go beyond OWL RL if needed (see also Section 8.6.2 below). The second reason for the good performance was that two different tasks – querying and ontology reasoning – could be performed by one single framework and not by two different ones as in the original implementation. Here, N3 completes the task of a *Unifying Logic* and connected the building blocks *Querying* and *Ontologies/Taxonomies*.

For the second use case we considered – RDF data validation – SPARQL is normally seen as the method of choice [124, p. 143]. By implementing rules for the constraint types supported by RDFUnit – a very popular engine to test constraints on RDF datasets – we showed that this typical use case for querying can also be supported by N3Logic. The comparison of our implementation with RDFUnit showed that both implementations are comparable in terms of accuracy and execution times. For small datasets our implementation was

[24] See also: https://github.com/w3c/N3/issues/11.

faster, for datasets containing more than 100,000 triples RDFUnit performed better. The real advantage of using N3 reasoning instead of SPARQL querying was in this case again the fact that our system can be easily extended to perform more complex reasoning.

Both use cases we considered were originally approached using OWL DL reasoning and/or SPARQL querying and we could retrieve comparable – and in some aspects even better – results by applying N3Logic instead. This indicates that N3Logic– if equipped with the right set of built-in predicates – can connect the layers *Querying* and *Ontologies/Taxonomies* for the Semantic Web stack just as we expect it from the *Unifying Logic*.

This chapter was partly based on the publications:

D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle, E. Mannens, Ontology reasoning using rules in an eHealth context, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 465–472.
URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_31

D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle, E. Mannens, Improving OWL RL reasoning in N3 by using specialized rules, in: V. Tamma, M. Dragoni, R. Gonçalves, A. Ławrynowicz (Eds.), Ontology Engineering: 12th International Experiences and Directions Workshop on OWL, Vol. 9557 of Lecture Notes in Computer Science, Springer, 2016, pp. 93–104. doi:10.1007/978-3-319-33245-1_10.
URL http://dx.doi.org/10.1007/978-3-319-33245-1_10

D. Arndt, B. De Meester, A. Dimou, R. Verborgh, E. Mannens, Using rule-based reasoning for RDF validation, in: S. Costantini, E. Franconi, W. Van Woensel, R. Kontchakov, F. Sadri, D. Roman (Eds.), Proceedings of the International Joint Conference on Rules and Reasoning, Vol. 10364 of Lecture Notes in Computer Science, Springer, 2017, pp. 22–36. doi:10.1007/978-3-319-61252-2_3

B. De Meester, P. Heyvaert, D. Arndt, A. Dimou, R. Verborgh, RDF graph validation using rule-based reasoning, submitted to: Semantic Web Journal

Chapter 7

Proofs in N3

By now, we have discussed three of the four characteristics we identified as crucial for a *Unifying Logic* for the Semantic Web (Section 2.4): A clearly defined semantics, syntactic and semantic compatibility with RDF, and the power to connect the logical building blocks *Querying*, *Ontologies/Taxonomies* and *Rules*. The fourth aspect in this list, the support of the layer *Proof*, is the topic of the following two chapters. It should be possible to express and exchange proofs by making use of the *Unifying Logic*. These proofs should not only support the layer of *Trust* but should also be suitable to be used in practical applications.

In the Semantic Web there are different ways how reasoners provide explanations for their conclusions: OWL DL reasoners mostly either provide a *justification* [139, 140] – a set of facts from which the new knowledge follows – or a *proof* [141, 142] – a list of the aforementioned facts together with the concrete reasoning steps applied on them leading to the new data. While verifications leave the work of constructing a more concrete description of how inference steps have been applied to the user who needs to know the underlying calculus to make sense of it, DL-proofs have another disadvantage: they are mostly represented in formats which are not native to the Semantic Web and can therefore not easily be exchanged, reused or automatically interpreted in this context.

For N3 this is different: N3 reasoners produce proofs using the SWAP [72] proof vocabulary[1]. This vocabulary enables them to explain all their derivations in N3. By that the proofs themselves become again part of the Semantic Web, they can be exchanged between different parties. They can be checked or used further by other reasoners.

[1] http://www.w3.org/2000/10/swap/reason#

In this chapter we focus on the formal aspects of proofs: We identify a set of formulas for which the semantics of N3 is not ambiguous, *simple formulas*. For these *simple formulas* we define the direct semantics. This definition also covers the semantics according to EYE. We then formally define the calculus the SWAP proof vocabulary expresses and prove its correctness. In the following chapter we then investigate the practical applications of N3 proofs.

7.1 Simple formulas

To be able to discuss about the possible applications of proofs in N3, we first need to clarify the calculus behind it and – most important – prove its correctness based on the semantics of N3. We already know, that the semantics of N3 is rather problematic: In Section 4 we formalised two possible alternatives to understand the meaning of nested implicitly universally quantified variables and it is very likely, that there are even more ways to understand implicit quantification in N3. Since we want to keep our findings valid for all implementations – proofs should be exchangeable among the different reasoners – we first define a set of formulas whose implicitly universally quantified variables are not ambiguous. If we make sure that the formulas we consider do not contain deeply nested universals, we can guarantee that Cwm and EYE quantify these variables at the same position: on the top formula. We call formulas fulfilling this condition *simple formulas*. Below, we explain the idea of such *simple formulas* in more detail.

Based on our considerations in the previous chapters, we know at this point that implicit universal quantification in N3 always leads to contradicting interpretations if the quantified variables only occur deeply nested in a formula. While example Formula 2.7 from Section 2.5.3

```
{:Kurt :knows ?x.} => {?x :knows :Kurt.}.
```

had a clear meaning in both interpretations we considered, namely

$$\forall x : \texttt{<Kurt knows x>} \rightarrow \texttt{<x knows Kurt>}$$

Formula 4.3 led to discrepancies:

```
{{?x :q ?y.} => {?x :r :c.}.} =>{?x :p :a.}.
```

The problem with that last formula was that Cwm and EYE differ in their understanding of the concept *parent* for universal variables which only occur

nested – i.e. surrounded by at least two pairs of curly brackets `{ }` – in a formula. Here, that is the case for the universal variable `?y`. For Cwm, the next higher formula surrounded by curly brackets is the parent, it carries the quantifier – that is the formula `{?x :q ?y.} => {?x :r :c.}.` in the example – while for EYE the top formula is always the parent. *Simple formulas* are now the formulas where universal variables do not occur nested and which are therefore interpreted equally by both reasoners.

In order to formally define *simple formulas*, we first introduce the concept of *nested components*. All definitions in this chapter rely on the syntax definitions we gave in Section 4.2, in particular the N3 alphabet (Definition 16) and the grammar given in Figure 4.2. We will furthermore make use of two different types of brackets: brackets occurring in N3's syntax `{, }, (, )` and auxiliary brackets we use in mathematical expressions. To emphasize the difference between these two kinds of brackets, we will underline the N3 brackets in all definitions where both kinds of brackets occur.

Definition 19 (Components of a formula). *Given an N3 alphabet $\mathcal{A}$. Let $\mathcal{F}$ be the set of N3 formulas and $\mathcal{T}$ the set of terms over $\mathcal{A}$. Let $f \in \mathcal{F}$ be a formula and $c : \mathcal{T} \to 2^{\mathcal{T}}$ a function such that:*

$$c(t) = \begin{cases} c(t_1) \cup \ldots \cup c(t_n) & \text{if } t = \underline{(}t_1 \ldots t_n\underline{)} \text{ is a list,} \\ \{t\} & \text{otherwise.} \end{cases}$$

We define the set comp$(f) \subset \mathcal{T}$ of direct components *of f as follows:*

- *If f is an atomic formula of the form $t_1\ t_2\ t_3$., comp$(f) = c(t_1) \cup c(t_2) \cup c(t_3)$.*
- *If f is an implication of the form e_1=> e_2., then comp$(f) = \{e_1, e_2\}$.*
- *If f is a conjunction of the form $f_1 f_2$, then comp(f) = comp$(f_1) \cup$ comp(f_2).*

Likewise, for $n \in \mathbb{N}_{>0}$, we define the components of level n *as:*

$$\text{comp}^n(f) := \{t \in \mathcal{T} | \exists f_1, \ldots, f_{n-1} \in \mathcal{F} : t \in \text{comp}(f_1) \wedge \underline{\{}f_1\underline{\}} \in \text{comp}(f_2) \wedge \ldots \wedge \underline{\{}f_{n-1}\underline{\}} \in \text{comp}(f)\}$$

For all $n > 1$ we call the components $t \in$ comp$^n(f)$ nested components *of f.*

Now, we can distinguish between direct components and nested components. Formula 4.3 from above has `{{?x :q ?y.} => {?x :r :c.}.}` and `{?x :p :a.}` as direct components, nested components of

level two are `{?x :q ?y.}`, `{?x :r :c.}`, `?x`, `:p`, and `:a`, and nested components of level three are `?x`, `:q`, `?y`, `:r` and `:c`. Our definition of *simple formulas* now excludes all formulas having universal variables at level three or deeper:

Definition 20 (Simple formulas). *Let $\mathcal{A}$ be an N3 alphabet and let U be the set of universal variables. We call an N3 formula f* simple *iff* $\mathrm{comp}^n(f) \cap U = \emptyset$ *for all $n \in \mathbb{N}$, $n > 2$.*

For $f =$ Formula 4.3 we have:

$$\mathit{comp}^3(f) = \{\texttt{?x}, \texttt{:q}, \texttt{?y}, \texttt{:r}, \texttt{:c}\}$$

and thus:

$$\mathit{comp}^3(f) \cap U = \{\texttt{?x}, \texttt{:q}, \texttt{?y}, \texttt{:r}, \texttt{:c}\} \cap U = \{\texttt{?x}, \texttt{?y}\} \neq \emptyset$$

Formula 4.3 is not a *simple formula*. Since Formula 2.7 does not contain any components of a higher level than 2, this formula is *simple*. For *simple formulas* we now have:

Lemma 21 (Universal quantification of simple formulas). *The N3 Core Logic interpretations according to Cwm and EYE always quantify all universal variables occurring in* simple formulas *on top level.*

Proof. For the interpretation according to EYE the claim is trivial since EYE quantifies all universal variables exclusively on top level.

To see that Cwm also quantifies all universal variables occurring in a *simple formula* on top level we take a closer look at the attributes from Section 4.4.2 defined in Table 4.10. Remember that for each formula node `f` the value $\texttt{f}.q$ of attribute q is exactly the set of universal variables for which `f` carries a universal quantifier in the N3 Core Logic translation. We use the definition of this attribute in our proof.

Let us assume that g_1 is a simple formula containing a variable `?x` as nested component which is not quantified on top level. This means, that for some $k \geq 1$ there exists a component $\{g_2\} \in \mathit{comp}^k(g_1)$ for which $\texttt{?x} \in g_2.q$. Based on the fact that g_2 is surrounded by brackets, we know that the value $g_2.q$ is computed using the attribute rule $\texttt{f}.q \leftarrow \texttt{f}.v_2 \setminus \texttt{e}.s$ on the production rule `e ::={f}`. We therefore know that $\texttt{?x} \in g_2.v_2$. If we now look at the attribute rules for v_2, we furthermore see that this value can only be not empty if the attribute rule $\texttt{e}.v_2 \leftarrow \texttt{f}.v_1$ has been applied on a lower level in the syntax tree. As this rule is an attribute rule for the production rule `e::={f}` we know that there exists a direct component $\{g_3\} \in \mathit{comp}(g_2) \subset$

comp$^{k+1}(g_1)$ for which ?x $\in g_3.v_1$. According to the definition of v_1 which only passes values upwards to the next expression $\ominus$ in the syntax tree this means that ?x is direct component of g_3. We therefore know that ?x $\in$ *comp*$(g_3) \subset$ *comp*$^2(g_2) \subset$ *comp*$^{k+2}(g_1)$. This means for $k+2>2$ we have *comp*$^{k+2}(g_1) \cap U \neq \emptyset$ which contradicts the assumption that g_1 is a *simple formula*.

□

7.2 The direct semantics of N3

Below, we define a direct semantics for N3. We do this to make the later proofs concerning the proof calculus easier to follow. For our direct semantics we assume that universal variables are always quantified on top level. Using the result from the previous section we can nevertheless guarantee that this semantics is compatible with both reasoners – EYE and Cwm – if we limit the input to *simple formulas*. By this limitation proofs remain interchangeable.[2]

We start our formalisation by introducing ground formulas for N3. These are similar to the same concept in N3 Core Logic (Definition 7) and will become relevant in the following sections. We make use of the concept of *components* as specified in Definition 19.

Definition 22 (Ground and universal-free formulas and terms). *Let $\mathcal{A}$ be an N3 alphabet and let E be the set of existential and U the set of universal variables.*

- *We call an N3 formula f over $\mathcal{A}$* ground *iff* $\mathrm{comp}^n(f) \cap E = \emptyset$ *and* $\mathrm{comp}^n(f) \cap U = \emptyset$ *for all* $n \in \mathbb{N}$. *We denote the set of ground formulas by* $\mathcal{F}_g$.
- *We call a term t* ground *iff t is either a constant or a formula expression $\{f\}$ where $f \in \mathcal{F}_g$. We denote the set of all ground terms by* $\mathcal{T}_g$.
- *We call an N3 formula f over $\mathcal{A}$* universal-free *iff* $\mathrm{comp}^n(f) \cap U = \emptyset$ *for all* $n \in \mathbb{N}$. *We denote the set of universal-free formulas by* $\mathcal{F}_e$.
- *We call a term t* universal-free *iff t is either a constant, an existential or a formula expression $\{f\}$ where $f \in \mathcal{F}_e$. We denote the set of all ground terms by* $\mathcal{T}_e$.

[2] As we already know from the discussion in Section 5.3.1 the proof vocabulary adds an extra level to the formulas, proofs are therefore most likely not *simple formulas*. But as also discussed in that section, Cwm and EYE interpret the implicitly universally quantified which only appear nested in a proof because of the proof vocabulary almost equally.

A *ground* formula is thus a formula which does neither contain universal nor existential variables. Similarly, a universal-free formula is a formula which does not contain implicitly universally quantified variables.

As a next step, we define *substitutions* which are similar to *grounding functions* as we introduced to define the semantics of N3 Core Logic (Definition 9). Substitutions can be used to replace universals or existentials by terms. We distinguish two ways how such substitutions can be applied: we either apply them to all nested or direct components of a formula or – as a second option – only to its direct components. We do that to reflect how universals and existentials are treated if variables with the same name occur on different levels. In Formula 3.7

```
{?x :q :b.} =>{ :a :b {?x :s :d.}.}.
```

the name `?x` stands twice for the same variable, the formula means:

$$\forall \texttt{x.} \quad \texttt{<x q b>} \rightarrow \texttt{< a b <x s d>>.}$$

In contrast to that in Formula 3.1 from earlier:

```
_:x :says {_:x :knows :Albert.}.
```

the name `_:x` stands for different variables.

$$\exists \texttt{x1.} \quad \texttt{x1 says <}\exists \texttt{x2.} \quad \texttt{x2 knows Albert>}$$

This motivates the following definition:

Definition 23 (Substitution). *Let $\mathcal{A}$ be an N3 alphabet and $f \in \mathcal{F}$ an N3 formula over $\mathcal{A}$.*

- *A* substitution *is a finite set of pairs of terms $\{v_1/t_1, \ldots, v_n/t_n\}$ where each t_i is an term and each v_i is either an existential or a universal such that $v_i \neq t_i$ and $v_i \neq v_j$, if $i \neq j$.*
- *For a formula f and a substitution $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$, we obtain the* component application *of σ to f, $f\sigma^c$, by simultaneously replacing each v_i which occurs as a* direct component *in f by the corresponding expression t_i.*
- *For a formula f and a substitution $\sigma = \{v_1/t_1, \ldots, v_n/t_n\}$, we obtain the* total application *of σ to f, $f\sigma^t$, by simultaneously replacing each v_i which occurs as a* direct or nested component *in f by the corresponding expression t_i.*

As the definition states, component application of a substitution only changes the direct components of a formula. For a substitution $\mu = \{\texttt{\_:x}/\texttt{:Kurt}\}$ we obtain:

$$(\texttt{\_:x :says \{\_:x :knows :Albert.\}.})\mu^c = \\ (\texttt{ :Kurt :says \{\_:x :knows :Albert.\}.})$$

A total application of $\sigma = \{\texttt{?x}/\texttt{:Kurt}\}$ in contrast, replaces each *occurrence* of a variable in a formula:

$$(\texttt{?x :says \{?x :knows :Albert.\}.})\sigma^t = \\ (\texttt{:Kurt :says \{:Kurt :knows :Albert.\}.})$$

We can now define interpretation and semantics:

Definition 24 (Interpretation). *An interpretation $\mathfrak{I}$ of an N3 alphabet $\mathcal{A}$ consists of:*

1. *A set $\mathcal{D}$ called the domain of $\mathfrak{I}$.*
2. *A function $\mathfrak{a} : \mathcal{T} \setminus (U \cup E) \to \mathcal{D}$ called the object function.*
3. *A function $\mathfrak{p} : \mathcal{D} \to 2^{\mathcal{D}\times\mathcal{D}}$ called the predicate function.*

Note that the above definition is very close to the definition of a structure in N3 Core Logic (Definition 8) and thereby also to structures in RDF (Section 3.2.1 and [35]). The main difference is that our domain does not distinguish between properties (IP) and resources (IR). The definitions are nevertheless compatible, as we assume $\mathfrak{p}(p) = \emptyset \in 2^{\mathcal{D}\times\mathcal{D}}$ for all resources p which are not properties (i.e. $p \in \text{IR}\setminus\text{IP}$ in the RDF-sense). As for N3 Core formulas, we define the structure of simple N3 formulas only for ground terms and use a substitution (grounding function) instead of a classical valuation function which maps the variables directly into the domain of discourse. The reason for that is the same as discussed above: cited formulas should not be "*referentially transparent*" [5, p.7], i.e. an interpretation needs to treat the formula expressions `{:Superman :can :fly}` and `{:ClarkKent :can :fly}` differently even if `:Superman` and `:ClarkKent` refer to the same element of the domain of discourse. As a consequence, the representation level also needs to be taken into account when we assign the meaning to variables. For our definition of N3's direct semantics for simple formulas, we now take the two different ways of applying such a substitution into account:

Definition 25 (Direct semantics of N3). *Let $\mathfrak{I} = (\mathcal{D}, \mathfrak{a}, \mathfrak{p})$ be an interpretation of an N3 alphabet $\mathcal{A}$, let U be the set of universal variable of $\mathcal{A}$ and E the set of existentials, and let f be a* simple *formula over $\mathcal{A}$. Then the following holds:*

1. *If f contains universal variables then $\mathfrak{I} \models f$ iff $\mathfrak{I} \models f\sigma^t$ for every substitution $\sigma : U \to \mathcal{T}_e$.*
2. *If f is universal-free and $W = comp(f) \cap E \neq \emptyset$ then $\mathfrak{I} \models f$ iff $\mathfrak{I} \models f\mu^c$ for some substitution $\mu : W \to \mathcal{T}_g$.*
3. *If f is universal free and $comp(f) \cap E = \emptyset$:*
   (a) *If f is an atomic formula $t_1\, t_2\, t_3$, then $\mathfrak{I} \models t_1\, t_2\, t_3$. iff $(\mathfrak{a}(t_1), \mathfrak{a}(t_3)) \in \mathfrak{p}(\mathfrak{a}(t_2))$.*
   (b) *If f is a conjunction $f_1 f_2$, then $\mathfrak{I} \models f_1 f_2$ iff $\mathfrak{I} \models f_1$ and $\mathfrak{I} \models f_2$.*
   (c) *If f is an implication then*
      - $\mathfrak{I} \models \{f_1\} \Rightarrow \{f_2\}$ *iff* $\mathfrak{I} \models f_2$ *if* $\mathfrak{I} \models f_1$.
      - $\mathfrak{I} \models \{f_1\} \Rightarrow \texttt{false}$. *iff* $\mathfrak{I} \not\models f_1$.
      - $\mathfrak{I} \models \{\} \Rightarrow \{f_2\}$. *iff* $\mathfrak{I} \models f_2$.
      - $\mathfrak{I} \models \texttt{false} \Rightarrow e$ *and* $\mathfrak{I} \models e \Rightarrow \{\}$., *for every expression e.*

In point 1 in the definition we first ground all universal variables. This is the same as assuming all universals to always be quantified on top level. Our semantics is thus following EYE's interpretation but is also correct for all *simple formulas* if we follow the semantics as implemented in Cwm. By grounding the universals first and then treating existentials afterwards (point 2) we furthermore make sure that in case of conflicts the universal quantifier is outside of the existential as it was stated in Quote I from Section 3.1.

With the above we can now define models:

Definition 26 (Model). *Let Φ be a set of N3 formulas. We call an interpretation $\mathfrak{I} = (\mathcal{D}, \mathfrak{a}, \mathfrak{p})$ a model of Φ iff $\mathfrak{I} \models f$ for every formula $f \in \Phi$.*

As in first order logic, we can define the notion of logical implication:

Definition 27 (Logical implication). *Let Φ be a set of N3 formulas and ϕ a formula over the same N3 alphabet A. We say that Φ (logical) implies ϕ ($\Phi \models \phi$) iff every model $\mathfrak{I} \models \Phi$ is also a model of ϕ.*

7.3 A proof calculus for N3

Having the direct semantics of (simple) N3 formulas defined we next want to consider proofs. When new knowledge is derived, a proof is a list of the different concrete steps applied which led to this new knowledge. This proof can be used by others to understand and either verify or decline the derivations. A proof calculus is now a set of all possible conceptual steps allowed to be used in a proof.[3] The proof calculus we discuss below is based on the steps which can be described by the N3 proof vocabulary which we further describe in the next section.

The N3 proof calculus contains four different kinds of proof steps. We write them as deduction rules, using "$\vdash$".

Definition 28 (Proof steps). *Let $\mathcal{F}$ be the set of simple formulas over an N3 alphabet $\mathcal{A}$, U the set of universals and E the set of existentials in $\mathcal{A}$, $\Gamma \subset \mathcal{F}$ a set of formulas and $f, f_1, f_2, g \in \mathcal{F}$. A proof step is one of the following inference rules:*

1. Axiom: *If $f \in \Gamma$ then $\Gamma \vdash f$.*
2. Conjunction elimination: *If $\Gamma \vdash f_1 f_2$ then $\Gamma \vdash f_1$ and $\Gamma \vdash f_2$.*
3. Conjunction introduction: *Let $\Gamma \vdash f_1$ and $\Gamma \vdash f_2$ and let*

   $$\sigma : U \to U \setminus (\bigcup_{n \in \mathbb{N}_{>0}} \mathit{comp}^n(f_1)) \text{ and } \mu : E \to E \setminus \mathit{comp}(f_1)$$

   be substitutions. Let $f_2' = f_2 \sigma^t \mu^c$ then

   $$\Gamma \vdash f_1 f_2'$$

4. Generalized modus ponens: *If $\Gamma \vdash \{ f_1 \} \Rightarrow \{ f_2 \}$. and $\Gamma \vdash g$ and there exists a substitution $\sigma : \mathit{comp}(f_1) \cap U \to \mathcal{T}_e$ such that*

   $$(\{ f_1 \} \Rightarrow \{ f_2 \} \, .) \sigma^t = (\{ f_1' \} \Rightarrow \{ f_2' \} \, .) \text{ and } f_1' = g$$

   then $\Gamma \vdash f_2'$.

When we define proof steps in order to explain derivations done in a logic, the most important requirement these steps need to fulfil is that they need to be correct: Only if we can be sure that the application of a proof step on valid formulas leads again to valid formulas we can trust the proofs which have been produced applying this calculus. We therefore state the following theorem:

[3] More about proof calculi can be found in most introductions to FOL, for example [73–75].

Theorem 29 (Correctness of proof calculus). *Let Φ be a set of N3 formulas and ϕ a formula over the same N3 alphabet $\mathcal{A}$. Let U be the set of universals and E the set of existentials in $\mathcal{A}$. Then the following holds:*

$$\textit{If } \Phi \vdash \phi \textit{ then } \Phi \models \phi.$$

Proof. We prove that every proof step is correct.

1. *Axiom:* For the axiom step the claim is trivial, as it corresponds to Definition 27.
2. *Conjunction elimination:* Let $\Phi \models f_1 f_2$ and let $\mathfrak{I} = (\mathcal{D}, \mathfrak{a}, \mathfrak{p})$ be a model for Φ and $f_1 f_2$.
   - If $f_1 f_2$ is universal-free and $\text{comp}(f_1 f_2) \cap E = \emptyset$, the claim follows immediately from Definition 25.3b.
   - If $f_1 f_2$ universal-free and $\text{comp}(f_1 f_2) \neq \emptyset$ let $\mu : \text{comp}(f_1 f_2) \to \mathcal{T}_g$ be a substitution such that $\mathfrak{I} \models (f_1 f_2)\mu^c$. Then $\mathfrak{I} \models (f_1\mu^c)(f_2\mu^c)$, thus $\mathfrak{I} \models (f_1\mu^c)$ and $\mathfrak{I} \models (f_2\mu^c)$.
   - If $f_1 f_2$ are not universal-free, then $\mathfrak{I} \models (f_1 f_2)\sigma^t$ for all substitutions $\sigma : U \to \mathcal{T}_e$.

     The claim follows by the same argument as above.
3. *Conjunction introduction:* Let $\Phi \models f_1$, $\Phi \models f_2$ and let $\mathfrak{I} = (\mathcal{D}, \mathfrak{a}, \mathfrak{p})$ be a model for Φ, f_1 and f_2. As the renaming substitutions σ and μ do not change the meaning of a formula, for $f_2' = f_2\sigma^t\mu^c$ the following holds: $\mathfrak{I} \models f_2'$. It immediately follows that $\mathfrak{I} \models f_1 f_2'$.
4. *Generalized modus ponens:* the claim follows directly from Definitions 25.1 and 25.3c.

□

The proof calculus is thus correct. The natural next step here would be to also prove the completeness of the proof calculus. Unfortunately, the calculus is not complete. To see this, consider the following rule containing a blank node in its antecedent:

```
{_:x :knows :Kurt.}=>{:Kurt :is :known}.
```

Before they apply this rule, the reasoners Cwm and EYE internally translate it to:

```
{?x :knows :Kurt.}=>{:Kurt :is :known}.
```

According to all definitions of N3's semantics we have given so far, this translation is correct. The step (or combination of steps depending on the calculus) performed here is not part of the proof calculus described by the SWAP proof vocabulary. Therefore, it is impossible for reasoners applying this step to communicate this application to different parties. To also be complete or to at least reflect the actual reasoning steps performed by the reasoners, the SWAP vocabulary would need to be extended by proof steps which act on existential variables.

Before we introduce the proof vocabulary, we make an observation about the generalised modus ponens which will become important in the following chapter:

Lemma 30. *Let $\mathcal{A}$ be an N3 alphabet, U the set of universal variables, $f \in \mathcal{F}_g$ a ground formula and $\{f_1\} => \{f_2\} \in \mathcal{F}$ an implication formula where all universal variables which occur in f_2 also occur in f_1. If the generalized modus ponens is applicable to f and $\{f_1\} => \{f_2\}$ then the resulting formula does not contain universal variables.*

Proof. Let $\sigma : U \to \mathcal{T}_e$ be a substitution such that $(\{f_1\} => \{f_2\}\,.)\sigma^t = \{f\} => \{f_2'\}\,.$ As f is a ground formula, $\text{range}(\sigma|_{U \cap (\bigcup_{i \geq 1} \text{comp}^i(f_1))}) \subset E_g$. With the condition that every universal variable of f_2 is also in f_1, that is

$$((\bigcup_{i \leq 1} \text{comp}^i(f_2)) \cap U) \subset ((\bigcup_{i \leq 1} \text{comp}^i(f_1)) \cap U)$$

the claim follows. □

7.4 The proof vocabulary

After having introduced the concrete proof steps which can be described by it in the last section, we now introduce the SWAP proof vocabulary.[4] This proof vocabulary makes it possible to express proofs directly in N3 and thereby make them accessible for Semantic Web technology. Semantic Web reasoners can process proofs, exchange them among each others or – for example – check them for correctness. By expressing proofs in N3, we thus make them part of the Semantic Web.

Below, we define the N3 names of the steps listed in Definition 28. From a technical point of view, we can only be aware of *axioms* if we previously parse them. In the proof vocabulary, the axiom step is therefore named after this action.

[4] The vocabulary's RDF definition can be found at: http://www.w3.org/2000/10/swap/reason#.

Definition 31 (Proof vocabulary). *Let $\mathcal{A}$ be an N3 alphabet and $\mathfrak{I} = (\mathcal{D}, \mathfrak{a}, \mathfrak{p})$ be an interpretation of its formulas. Let C be the set of constants in $\mathcal{A}$. Let $\mathtt{x}, \mathtt{y}, \mathtt{y}_1, \ldots, \mathtt{y}_n \in C$ be N3 representations of proof steps and $\mathtt{z}_1, \mathtt{z}_2, \mathtt{z}_3 \in C$.*

1. *Proof step types:*
   - $\mathfrak{I} \models$ `x a r:Proof.` *iff* `x` *is the proof step which leads to the proven result.*
   - $\mathfrak{I} \models$ `x a r:Parsing.` *iff* `x` *is a parsed axiom.*
   - $\mathfrak{I} \models$ `x a r:Conjunction.` *iff* `x` *is a conjunction introduction.*
   - $\mathfrak{I} \models$ `x a r:Inference.` *iff* `x` *is a generalized modus ponens.*
   - $\mathfrak{I} \models$ `x a r:Extraction.` *iff* `x` *is a conjunction elimination.*
2. *Proof predicates:*
   - $\mathfrak{I} \models$ `x r:gives {`f`}.` *iff $f \in F$ is the formula obtained by applying* `x`*.*
   - $\mathfrak{I} \models$ `x r:source u.` *iff* `x` *is a parsed axiom and* `u` $\in U$ *is the URI of the parsed axiom's source.*
   - $\mathfrak{I} \models$ `x r:component y.` *iff* `x` *is a conjunction introduction and* `y` *is a proof step which gives one of its components.*
   - $\mathfrak{I} \models$ `x r:rule y.` *iff* `x` *is a generalized modus ponens and* `y` *is the proof step which leads to the applied implication.*
   - $\mathfrak{I} \models$ `x r:evidence (`$\mathtt{y}_1, \ldots, \mathtt{y}_n$`).` *iff* `x` *is a generalized modus ponens and* $\mathtt{y}_1, \ldots, \mathtt{y}_n$ *are the proof steps which lead to the formulas whose conjunction was used for the unification with the antecedent of the implication.*
   - $\mathfrak{I} \models$ `x r:because y.` *iff* `x` *is a conjunction elimination and* `y` *is the proof step which yields the to-be-eliminated conjunction.*
3. *Substitutions:*
   - $\mathfrak{I} \models$ `x r:binding` $\mathtt{z}_1$. *iff* `x` *includes a substitution* $\mathtt{z}_1$.
   - $\mathfrak{I} \models \mathtt{z}_1$ `r:variable` $\mathtt{z}_2$. *iff* $\mathtt{z}_1$ *is a substitution whose domain is* $\{\mathtt{z}_2\}$.
   - $\mathfrak{I} \models \mathtt{z}_1$ `r:boundTo` $\mathtt{z}_3$. *iff* $\mathtt{z}_1$ *is a substitution whose range is* $\{\mathtt{z}_3\}$.

An N3 proof is now a conjunction of all the proof steps leading to the proven formula. To illustrate that, we discuss an example proof. We come back to the formulas we used in Chapter 2 of this thesis. From Formula 2.3

```
:Kurt :knows :Albert.
```

```
1  PREFIX : <http://example.org/ex#>
2  PREFIX r:  <http://www.w3.org/2000/10/swap/reason#>
3
4  [] a r:Proof, r:Conjunction;
5    r:component <#lemma1>;
6    r:gives {
7      :Albert :knows :Kurt.
8    }.
9
10 <#lemma1> a r:Inference;
11   r:gives {
12     :Albert :knows :Kurt.
13   };
14   r:evidence (
15     <#lemma2>
16   );
17   r:rule <#lemma3>.
18
19 <#lemma2> a r:Extraction;
20   r:gives {
21     :Kurt :knows :Albert.
22   };
23   r:because [ a r:Parsing; r:source <Formula_2.1>].
24
25 <#lemma3> a r:Extraction;
26   r:gives {
27     {:Kurt :knows ?x_0_1} => {?x_0_1 :knows :Kurt}.
28   };
29   r:because [ a r:Parsing; r:source <Formula_2.7>].
```

Listing 20: Example proof: Formula 2.9 can be derived by applying Rule 2.7 on Formla 2.3.

and Rule 2.7

```
{:Kurt :knows ?x.} => {?x :knows :Kurt.}.
```

we could derive Formula 2.9.

```
:Albert :knows :Kurt.
```

The proof for that derivation is displayed in Listing 20. To produce this proof we used Formula 2.7 as a goal. The proof is produced by EYE. We go through the proof from the bottom to the top:

`<#lemma3>` in line 25 is an `r:Extraction` which results in the formula `{:Kurt :knows ?x_0_1} => {?x_0_1 :knows :Kurt}.` as indicated by the predicate `r:gives`. This formula was picked (`r:because`) as one conjunct from another formula produced by another proof step which here is denoted by an anonymous blank node. This proof

step is an r:Parsing having as source (r:source) the source file <Formula_2.7> which contains Formula 2.7.

The description of <#lemma2> in line 19 is very similar to the above. We again have an r:Extraction, this times made on a parsed formula contained in the file <Formula_2.1>. This step results in the formula :Kurt :knows :Albert.

<#lemma1> in line 10 now makes use of the two formulas: The lemma is an inference step (r:Inference) which applies the rule (r:rule) produced by <#lemma3> on the formula produced by <#lemma1> (r:evidence). This results in the formula :Albert :knows :Kurt.

This last formula is also the result of out proof as indicated in line 4. The last proof step we apply (r:Proof) is a r:Conjunction with only one component, the result of <#lemma1>. We get :Albert :knows :Kurt.

The proof above has been automatically produced and in our example some of the proof steps have not really been necessary. We do for example not need the conjunction of only one component indicated in line 4. The proof did on the other hand not make use of the different predicates describing a substitution. It renamed the variable ?x from our input Formula 2.7 to ?x_0_1 (line 27) and then unified this variable ?x_0_1 with :Albert when executing <#lemma1> (line 10) without letting us know.

Such and other simplifications are often made by N3 reasoners producing proofs. However, to the best of our knowledge, all reasoners' proofs contain all applications of r:Inference leading to a goal g, which allows us to measure a proof's length by counting applications of the generalized modus ponens.

7.5 Conclusion

In this chapter we introduced and explained the N3 proof vocabulary: This vocabulary makes it possible to express the derivations performed by a reasoner in a Semantic Web format. By using N3 to express proofs, we make sure that proofs produced by one reasoner can be used as input by another reasoner. Such a reasoner can then verify the proofs or use them for further reasoning.

To allow this exchangeability of proofs, these proofs do not only need to share the same syntactic format, they also need to be interpreted equally by all reasoners. From our consideration in the earlier chapter, we know that this can be problematic: especially the interpretation of implicitly quantified variables differs between reasoner. To make the exchange of proofs possible, despite

the discrepancies in the interpretations of N3 formulas, we introduced the concept of *simple formulas*: By excluding deeply nested universals, we can guarantee for universals to always quantified on top level. For these formulas we then defined a direct semantics which assumes top level quantification for universals. With this direct semantics we proved that the calculus behind the proof vocabulary is correct. It is therefore save to exchange proofs only containing *simple formulas* among different reasoners.

We know that N3 reasoners can produce proofs, exchange and verify them and use them for further applications. In the next chapter, we want to take a look at possibilities for such *further applications*. We consider a concrete example: If we describe possible API operation in N3 we can use proofs to automatically combine these APIs and thereby use proofs as a kind of plan. We discuss this idea in detail in the following chapter and also explain how this idea of using proofs for planning can be applied in other scenarios.

This chapter was partly based on the publications:

R. Verborgh, D. Arndt, S. Van Hoecke, J. De Roo, G. Mels, T. Steiner, J. Gabarró Vallés, The pragmatic proof: Hypermedia API composition and execution, Theory and Practice of Logic Programming 17 (1) (2017) 1–48. doi:10.1017/S1471068416000016.
URL http://arxiv.org/pdf/1512.07780v1.pdf

D. Arndt, R. Verborgh, J. De Roo, H. Sun, E. Mannens, R. Van de Walle, Semantics of Notation3 logic: A solution for implicit quantification, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 127–143.
URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_9

Chapter 8

Applications of proofs: Adaptive planning

Having discussed the theory behind proofs in N3Logic in the previous chapter, we now turn to practical applications of proofs: Since the proofs produced by N3 reasoners are again written in this format, they can easily be used in other Semantic Web applications. The application we want to discuss here is a system to automatically combine and execute RESTful API operations on the Web. If we describe such operations by using N3 rules, a proof applying these rules can be understood as a plan. This plan can then be executed and – if needed – updated. As our proofs use Semantic Web technology, background information, like the preferences of the person using the system or simple information about the resources involved in the plan, can always be taken into account. By only using *simple rules* in our descriptions, these plans are furthermore reasoner-independent and can be exchanged, adapted and reused throughout the Web. This last property makes our procedure to create plans particularly well suited for RESTful Web APIs as these are designed to cope with the distributed and fast changing nature of the Web and to be used and combined for different applications and by different parties in an open environment.

The remainder of this chapter is structured as follows: In the next section, we give a short introduction to Web APIss and ways to semantically describe them. We then introduce hypermedia API descriptions in Section 8.2. Section 8.3 explains the use of proofs to validate hypermedia API compositions, the generation of which is detailed in Section 8.4. We then discuss the limits and shortcomings of our approach (Section 8.5) and another possible application of the idea to use proofs for planning, a system to detect sensors

relevant for fault detection in complex environments (Section 8.6). Finally, we end the chapter in Section 8.7 with conclusions.

8.1 Hypermedia APIs and their semantic descriptions

In this chapter we use N3Logic to describe Application Programming Interfaces (APIs) which are designed following the Representational State Transfer (REST) architectural style [143]. Before we come to the details of our approach we first introduce the terms about Web APIs which are relevant for our considerations:

A Web server uses the HTTP protocol [144] to offer data and/or actions, which are often exclusively located on this server. In other words, the server performs a data lookup or computation that another device cannot or does not.

A Web client consumes resources offered by a Web server.

A Web API is a machine-accessible interface to data and/or actions offered by a server through HTTP.

A Web API operation is a single HTTP request from a Web client to a Web API. Interactions between a client and a server consist of one or more operations.

A hypermedia control is a hyperlink or form that allows clients to navigate from one resource to another.

A hypermedia API is a Web API that follows all REST architectural constraints [143]. In particular, it is designed such that its (human or machine) clients should perform the interaction through hypermedia controls.

Our approach is designed for hypermedia APIs. While for more lightweight Web APIs several semantic description formats exist [145] the description of hypermedia APIs is a relatively new field. Hydra [146] is a vocabulary to support API descriptions, but does not directly support automated composition. RESTdesc [147] is a description format for hypermedia APIs that describes them in terms of resources and links. RESTdesc is expressed in N3 and will be used as a description format in this chapter. The Resource Linking Language

(ReLL, [148]) features media types, resource types, and link types as first-class citizens for descriptions. The authors of ReLL also propose a method for ReLL API composition [149] using Petri nets to describe the machine-client navigation. However, in contrast to RESTdesc, it does not support automatic, functionality-based composition. Roman and Kifer introduced ServLog [150], a framework for service modelling, contracting, and reasoning based on Concurrent Transaction Logic (CTL) [151]. In their earlier work they furthermore described how CTL can be used to automatically combine Web services [152]. The description format for services used in both papers rather focusses on their role in workflows (e.g. which services rely on each other) than on their functionality (e.g. which result do they produce). In that sense this approach can be understood as complementary to RESTdesc.

8.2 Describing Hypermedia APIs with RESTdesc

After having discussed the background and most important terms when talking about hypermedia APIs, we now explain how such APIs can be described by using N3 rules. We start by explaining the example use case we are going to base the explanations in this chapter on.

8.2.1 Example Use Case

As a guiding example, we introduce an exemplary hypermedia API in the domain of image processing. It offers functionality such as the following:

- uploading images
- resizing an image
- changing the colors of an image
- combining multiple images
- …

Since hypermedia APIs follow the REST architectural constraints, this functionality is realized in a resource-oriented, hypermedia-driven way. For instance, the API does *not* offer methods such as `resizeImage`; rather, it exposes `image` and `thumbnail` resources, which are connected to each other through hyperlinks and/or forms. In order to enable machines to interpret responses of this API, these resources are available as representations in the machine-readable format RDF.

The problem statement is to make a generic Web client that can autonomously reach a complex goal such as "obtain a scaled, black-and-white version of `image.jpg`", without hard-coding any knowledge about this API. The client should be able to execute complex instructions on (combinations of) other hypermedia APIs as well.

Since this API is a hypermedia API, we cannot create a detailed plan in advance, because the exact steps are not known at design time. However, at the same time, an automated client cannot only follow hypermedia controls, because there would be no way to know whether the controls it chooses lead to the given goal. In other words, while hypermedia gives machines access to resources via links, it does not explain them the functionality offered through those links.

RESTdesc descriptions [147, 153] allow to express the functionality of hypermedia APIs by explaining the role of a hypermedia control in an API. That way, if a machine client encounters a hypermedia control, it can interpret the results of following it. Furthermore, RESTdesc descriptions allow the composition of a high-level plan that can guide machine clients through an interaction with a hypermedia API.

In the remainder of Section 8.2, we formalise RESTdesc. Sections 8.3 and 8.4 detail hypermedia API composition and execution, using the above image processing hypermedia API as an example.

8.2.2 RESTdesc Descriptions

RESTdesc descriptions are designed to explain how a hypermedia API can be used to perform a specific action. Such a process of using an API consists of different steps: Given all needed information, the client sends an HTTP request to the hypermedia API on a server. The API interprets the request and, if possible, reacts by fulfilling the indicated task or retrieving the information requested. The server sends a response and thereby creates a new situation. As mentioned in Section 8.1, this process is called an API operation. Hypermedia APIs are commonly able to perform different kinds of operations.

To describe such an operation in N3 a formalization of HTTP is needed. We use the RDF vocabulary as defined in the corresponding W3C Working Draft [154]. In order to facilitate the understanding of the following sections, we give a short overview of the HTTP predicates used in this paper.

Definition 32 (HTTP predicates). *Let $\mathcal{A}$ be an N3 alphabet, M be a set of HTTP method names, $\{$"GET", "POST", "PUT",*

"DELETE", "HEAD", "PATCH" $\} \subset M \subset C$, $\mathtt{u} \in C$ *an HTTP message,* $\mathtt{v} \in C$ *and* $\mathfrak{I} = (\mathcal{D}, \mathfrak{a}, \mathfrak{p})$ *an interpretation of* $\mathcal{A}$*. Then:*

- $\mathfrak{I} \models$ `u http:headers v.` *iff* `v` *is an HTTP header of* `u`*.*
- $\mathfrak{I} \models$ `u http:body v.` *iff* `v` *is the HTTP body of* `u`*.*
- $\mathfrak{I} \models$ `u http:methodName v.` *iff* `u` *is a request and* $\mathtt{v} \in M$ *its method name.*
- $\mathfrak{I} \models$ `u http:requestURI v.` *iff* `u` *is a request and* `v` *is its URL.*
- $\mathfrak{I} \models$ `u http:resp v.` *iff* `u` *is a request and* `v` *is its responding HTTP message.*

This vocabulary can be used to describe HTTP-requests. Such a request must always have a method name and a request URI. These two properties also have to be specified in an HTTP request description:

Definition 33 (HTTP request description). *Let* $\mathcal{A}$ *be an N3 alphabet which contains a set* H *of HTTP predicates including those defined in Definition 32. Let* U *be the set of universals and* E *be the set of existentials in* $\mathcal{A}$*.*

- *An* HTTP request description *is a conjunction* $f = f_1 f_2 \dots f_n \in \mathcal{F}$ *of atomic formulas with the following properties:*
  - *All atomic formulas* f_i *share the same existential variable* $\mathtt{\_:x} \in E$ *as a subject.*
  - *The predicate of each formula* f_i *is an HTTP-predicate, i.e.* f_i *has the form* $\mathtt{\_:x\ :h_i:o_i.}$ *with* $\mathtt{:h_i} \in H$*.*
  - *The conjunction* f *contains one atomic formula* f_i *with the predicate* `http:methodName` *and one formula* f_j *with the predicate* `http:requestURI`*.*
  - *The object of every atomic formula* $f_i = \mathtt{\_:x\ :h_i:o_i.}$ *with* $\mathtt{h_i} \neq$ `http:resp` *is either a universal variable, or a constant (i.e., a URI or a literal),* $\mathtt{:o_i} \in U \cup C$*.*
- *We call an HTTP request description* $f = f_1 \dots f_n$ *sufficiently specified, if* $\mathtt{:o_i} \in \mathcal{T}_g$ *for every triple* $f_i = \mathtt{\_:x\ :h_i:o_i}$ *with* $\mathtt{:p_i} \neq$ `http:resp`*.*

The definition reflects the syntactical requirements to an HTTP request description, it should contain the URL and the method name of the described request and it can contain additional information which can be described

using the HTTP-predicates. If the object of these formulas are instantiated, i.e. sufficiently specified, they can be sent to a server and, if they contain all necessary information, executed by an API which will return the HTTP response. RESTdesc descriptions enable us to specify the intended functionality of a hypermedia API's operations:

Definition 34 (RESTdesc description). *Let $\mathcal{A}$ be an N3 alphabet containing the predicates defined in Definition 32 and $\mathcal{F}$ the set of formulas over $\mathcal{A}$. A RESTdesc description $f \in \mathcal{F}$ of a hypermedia API operation is a* simple *N3 formula of the form:*

```
{ precondition } => { http-request  postcondition }.
```

where `precondition`*,* `http-request` *and* `postcondition` *are N3 formulas over $\mathcal{A}$ with the following properties:*

1. `precondition` *describes the resources needed to execute the operation and does not contain any existential variable.*

2. `http-request` *is an HTTP request description which describes a request which can be used to obtain the desired result of executing the operation. All universal variables which occur in* `http-request` *do also occur in* `precondition`*.*

3. `postcondition` *describes one or more results obtained by the execution of the operation. All universal variables contained in* `postcondition` *also occur in* `precondition`*.*

Note that a RESTdesc description is an existential rule as defined in the Datalog$^{\pm}$ framework [155]: our restriction on universal variables, that all universals in the head of the rule should also occur in the body, is very similar to Datalog [114] and our rules allow (and expect) new existentials in the consequence.

We make the restrictions on variables occurring in RESTdesc descriptions because we want to use the proofs applying these descriptions as execution plans. This idea will be further explained later in Section 8.3.3. To be able to do so, we need to be sure that whenever we apply a RESTdesc description to a ground term it results in a sufficiently specified HTTP request and that the postcondition will not contain any universal variables. As HTTP requests in RESTdesc descriptions only contain one leading existential to represent the HTTP message, and RESTdesc descriptions fulfil the conditions of Lemma 30 we arrive at the following corollary:

```
PREFIX dbpedia:  <http://dbpedia.org/resource/>
PREFIX dbpedia-owl:  <http://dbpedia.org/ontology/>
PREFIX ex:  <http://example.org/image#>
PREFIX http:  <http://www.w3.org/2011/http#>

{ ?image ex:smallThumbnail ?thumbnail. }
=>
{
  _:request http:methodName "GET";
            http:requestURI ?thumbnail;
            http:resp [ http:body ?thumbnail ].

  ?image dbpedia-owl:thumbnail ?thumbnail.
  ?thumbnail a dbpedia:Image;
             dbpedia-owl:height 80.0.
}.
```

Listing 21: RESTdesc description of the action "obtaining a thumbnail" *(desc_thumbnail.n3)*

Corollary 35. *Every application of a RESTdesc description to a ground formula results in a sufficiently specified HTTP request and a postcondition which does not contain any universal variables.*

To illustrate how such a RESTdesc description looks like, we consider an example: Listing 21 shows a description that explains the `smallThumbnail` relation in a hypermedia API. The *precondition* demands the existence of a `smallThumbnail` hyperlink between an `?image` resource and a `?thumbnail` resource. The HTTP request is a `GET` request to the URL of `?thumbnail`. The response to this request will be a representation of `?thumbnail`. These characteristics of this representation are detailed in the remainder of the *postcondition*. This states that the original `?image` will be in a `thumbnail` relationship (the meaning of which is defined by the DBpedia ontology [156]) with `?thumbnail`. Furthermore, `?thumbnail` will be an `Image` and have a `height` of `80.0`.

There are two different ways to interpret this description: First the declarative, static way as defined in Section 7.2, which could be phrased as "the existence of the `smallThumbnail` relationship implies the existence of a `GET` request which leads to an 80px-high thumbnail of this image."

The second interpretation is the operational, dynamic way. In this case, a software agent has a description of the world, against which the description is *instantiated*, i.e., the rule is applied.

Thus, given a concrete set of triples, such as:

```
</photos/37> ex:smallThumbnail </photos/37/thumb>.
```

```
1  PREFIX dbpedia:  <http://dbpedia.org/resource/>
2  PREFIX dbpedia-owl:  <http://dbpedia.org/ontology/>
3  PREFIX ex:  <http://example.org/image#>
4  PREFIX http:  <http://www.w3.org/2011/http#>
5
6  { ?image a dbpedia:Image. }
7  =>
8  {
9    _:request http:methodName "POST";
10             http:requestURI "/images/";
11             http:body ?image;
12             http:resp [ http:body ?image ].
13   ?image ex:comments _:comments;
14          ex:smallThumbnail _:thumb.
15 }
```

Listing 22: RESTdesc description of the action "uploading an image" *(desc_images.n3)*

the description in Listing 21 would be instantiated to:

```
_:request http:methodName "GET";
          http:requestURI </photos/37/thumb>;
          http:resp [ http:body </photos/37/thumb> ].

</photos/37> dbpedia-owl:thumbnail </photos/37/thumb>.
</photos/37/thumb> a dbpedia:Image;
                   dbpedia-owl:height 80.0.
```

Thereby, the description has been instantiated into a concrete HTTP request that can be executed by the agent. Note how this instantiation directly results in RDF triples, which can be interpreted by any RDF-compatible client. The request has been sufficiently specified as defined in Definition 33. In addition, the instantiated postcondition explains the properties realized by this concrete request. Here, an HTTP `GET` request to `/photos/37/thumb` will result in a thumbnail of the image `/photos/37` that will have a height of 80 pixels. This dynamic interpretation is helpful to agents that want to understand the impact of performing a certain action on resources they have at their disposition.

RESTdesc descriptions are not limited to `GET` requests. They can also describe state-changing operations, for instance, those realized through the `POST` method. Listing 22 shows a description for an image upload action. The postconditions contain existential variables that are not referenced (`_:comments` and `_:thumb`), which might appear strange at first sight. However, these triples are important to an agent as they convey an *expectation* of what happens when an image is uploaded. Concretely, any uploaded image will receive a `comments` link and a `smallThumbnail` link. Even

though the exact values will only be known at runtime when the actual `POST` request is executed, at design-time, we are able to determine that there will be several links. The meaning of those links is in turn expressed by other descriptions, such as the one in Listing 21 discussed earlier.

8.3 Proofs as plans

Having seen in the previous section how we can describe hypermedia API operations by means of existential rules, we now want to explain how proofs containing these RESTdesc descriptions can serve as plans towards the fulfilment of a goal. We start by further defining the problem we want to solve.

8.3.1 Composition problems

Given a set of possible API operations, we want to achieve a goal from an initial state. Furthermore, we might have some additional knowledge that can be incorporated. The above can be expressed in N3 as follows:

Definition 36 (API composition problem). *Let $\mathfrak{F}$ be the set of simple N3 formulas over an alphabet $\mathcal{A}$ which contains the predicates defined in Definition 32. An API composition problem consists of the following formulas:*

- *A set $H \subset \mathfrak{F}_g$ of ground formulas capturing all resource and application states the client is currently aware of, the initial state.*
- *A formula $g \in \mathfrak{F}$ with $comp^n(g) = \emptyset$ for all $n \geq 2$, which does not contain existential variables, the goal state which indicates on a symbolic level what the client wants to achieve.*
- *A set R of RESTdesc descriptions or conjunctions of RESTdesc descriptions, describing all hypermedia APIs available to the client, the description formulas.*
- *A (possibly empty) set of N3 formulas B, the background knowledge, where each $b \in B$ is either a ground formula or an implication e_1=>e_2. which does not contain existential variables and where each universal variable e_2 contains does also occur in e_1.*

Note that we put syntactical restrictions on our definitions: as already mentioned in Section 8.2.2, RESTdesc descriptions are existential rules. The constraints put on the background knowledge make the rules contained in it expressible in Datalog [114]. The initial state contains only ground formulas

and as the goal does not contain nested constructions or existential variables it can also be expressed in a Datalog rule. This makes the whole problem at our disposal expressible in Datalog$^{\pm}$ [155]. The reason for this restrictions will become clear in Section 8.4.
If we talk about a proof as explained above, we mean evidence for the fact that from $H \cup R \cup B$ follows g', where g' is a valid instance of g. As a normal hypermedia API composition problem tries to actually achieve real states and obtain instantiated objects, our final target is to make g' ground.

8.3.2 Pre-proofs versus Post-proofs

In this section, we extend this classical notion or proofs to also include *dynamic* information, *i.e.,* data generated by Web APIs. As a consequence of this dynamic nature, we introduce two different kinds of proofs for an API composition problem (H, g, R, B) as defined in Definition 36:

a pre-execution proof *("pre-proof")*, in which the assumption is made that execution of all API operations will behave as expected, i.e., a proof in a classical sense which provides evidence for

$$H \cup R \cup B \models g'$$

a post-execution proof *("post-proof")*, in which an additional evidence for the goal is provided by the API operations' actual execution results, which are purely static data. This means the resulting proof itself confirms that

$$H \cup R \cup B \cup \{\text{execution results}\} \models g''$$

with g' and g'' being instances of g. Note that technically speaking, for the same API operation every pre-proof is also a post-proof but, if its execution actually yields a non-empty result, not every post-proof is a pre-proof. We are especially interested in the post-proofs of an API operation which are not its pre-proofs, and call those proofs **proper post-execution proofs**. In other words, proper post proofs are those proofs that actually make use of the information gained by the API call. Intuitively, we expect those proofs to be shorter than the initial pre-proof, as they have more relevant knowledge at their disposal.

The distinction between pre- and (proper) post-proof exists because, although error handling is possible, one can never *guarantee* that a composition that has proven to work in theory will *always* and reliably achieve the desired

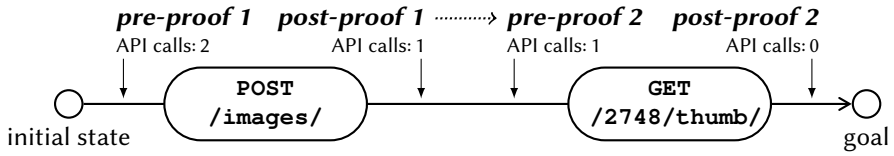


Figure 8.1: A proper post-proof of one operation becomes the pre-proof of the next.

result in practice, since the individual steps can fail. Some errors (such as disk failures or power outages) cannot be predicted and may cause a composition not to reach a goal that would normally be possible. Furthermore, a composition can only be as adequate as its individual descriptions, which could contain mistakes. Therefore, the pre-proof necessarily has to make the additional assumption that all APIs will function according to their description. The pre-proof's objective thus becomes: *"assuming correct behavior of all APIs, the composition must lead to the fulfilment of the goal."*

While a pre-proof can be validated before a composition's execution, the creation and validation of a proper post-proof can only happen when the execution's results are available. At that stage, however, the environment's nature is no longer dynamic, since the APIs' results are effectively available as data. A proper post-proof is therefore equivalent to a data-based proof, wherein the executed API operations contribute to the provenance information. This provenance can be used to link the proper post-proof to the pre-proof, indicating whether the non-failure assumption has corresponded to reality. In the ideal case, this assumption indeed holds and the proper post-proof is essentially a revision of the pre-proof in which the actual values returned by the hypermedia APIs are filled in. The objective of the post-proof is thus *"given the execution results of some API operations, the composition must lead to the fulfilment of the goal."* Thereby, a proper post-proof after one operation becomes the pre-proof of the next operation, as indicated in Figure 8.1.

Regular proofs do not contain dynamic information that needs to be obtained at runtime. The extension to pre-proofs that contain dynamic information, necessary to verify the correctness of a composition *before* it is executed, requires a mechanism to express when API operations are performed. REST-desc descriptions can be considered rules that *simulate* the execution of a hypermedia API, using existentially quantified variables as placeholders for the API's results, which are still unknown at the time the pre-proof is to be verified.

```
1 PREFIX dbpedia:  <http://dbpedia.org/resource/>
2
3 <lena.jpg> a dbpedia:Image.
```

Listing 23: The initial knowledge of the agent *(agent_knowledge.n3)*

```
1 PREFIX dbpedia-owl:  <http://dbpedia.org/ontology/>
2
3 { <lena.jpg> dbpedia-owl:thumbnail ?thumbnail. }
4 =>
5 { <lena.jpg> dbpedia-owl:thumbnail ?thumbnail. }.
```

Listing 24: A filter rule expressing the agent's goal *(agent_goal.n3)*.

8.3.3 Hypermedia API Operations inside a Proof

We now want to use proofs as plans for the execution of API calls. RESTdesc descriptions are designed in a way that allows us to know from the proofs they are used in, which API operations will bring us closer to our desired goal. To further explain this idea, we use an example.

Listing 25 displays a hypermedia API composition proof. In addition to the two RESTdesc descriptions from Listings 21 and 22 (R in an API composition problem), it involves the goal file Listing 24 (`{g} => {g}.`) and the input file Listing 23 (the initial knowledge H). The proof, generated by EYE [88], explains how, given an image, a thumbnail of this image can be obtained. The proof makes use of the proof vocabulary introduced in Section 7.4. We now explain the different elements in detail.

Overview The main element of the proof is `r:Proof` [*line 8*], which is a conjunction of different components, indicated by `r:component`. In this example, there is only one, but there can be multiple. The conclusion of the proof (object of the `r:gives` relation) is the triple `<lena.jpg> dbpedia-owl:thumbnail _:sk3` (with `_:sk3` an existential variable). This captures the fact that `lena.jpg` has a certain thumbnail, which is the consequent of the filter rule in Listing 24.

Extractions and inferences EYE represents `r:Extractions` and `r:Inferences` as lemmata, which can be reused in different proof steps. The extractions, Lemmata 4 to 7 [*lines 39–45*], are fairly trivial as none of the underlying formulas is a real conjunction and will therefore not be discussed in detail. In this proof, they just give the formulas specified in Listing 21 to 24 with renamed variables. The inferences describe the actual reasoning carried out and thus merit a closer inspection. We will follow the path backwards

```
PREFIX dbpedia:  <http://dbpedia.org/resource/>
PREFIX dbpedia-owl:  <http://dbpedia.org/ontology/>
PREFIX ex:  <http://example.org/image#>
PREFIX http:  <http://www.w3.org/2011/http#>
PREFIX n3:  <http://www.w3.org/2004/06/rei#>
PREFIX r:  <http://www.w3.org/2000/10/swap/reason#>

<#proof> a r:Proof, r:Conjunction;
  r:component <#lemma1>;
  r:gives { <lena.jpg> dbpedia-owl:thumbnail _:sk3. }.

<#lemma1> a r:Inference;
  r:gives { <lena.jpg> dbpedia-owl:thumbnail _:sk3. };
  r:evidence (<#lemma2>);
  r:rule <#lemma7>.

<#lemma2> a r:Inference;
  r:gives { _:sk4 http:methodName "GET".
            _:sk4 http:requestURI _:sk3.
            _:sk4 http:resp _:sk5.
            _:sk5 http:body _:sk3.
            <lena.jpg> dbpedia-owl:thumbnail _:sk3.
            _:sk3 a dbpedia:Image.
            _:sk3 dbpedia-owl:height 80.0. };
  r:evidence (<#lemma3>);
  r:rule <#lemma4>.

<#lemma3> a r:Inference;
  r:gives { _:sk0 http:methodName "POST".
            _:sk0 http:requestURI "/images/".
            _:sk0 http:body <lena.jpg>.
            _:sk0 http:resp _:sk1.
            _:sk1 http:body <lena.jpg>.
            <lena.jpg> ex:comments _:sk2.
            <lena.jpg> ex:smallThumbnail _:sk3. };
  r:evidence (<#lemma6>);
  r:rule <#lemma5>.

<#lemma4> a r:Extraction;
  r:because [ a r:Parsing; r:source <desc_thumbnail> ].
<#lemma5> a r:Extraction;
  r:because [ a r:Parsing; r:source <desc_images> ].
<#lemma6> a r:Extraction;
  r:because [ a r:Parsing; r:source <agent_knowledge> ].
<#lemma7> a r:Extraction;
  r:because [ a r:Parsing; r:source <agent_goal> ].
```

Listing 25: Example hypermedia API composition proof

from the proof's conclusion, tracing back inferences until we arrive at the atomic facts that are the starting point of the proof.

Filter rule The justification for the conclusion consists in this case of a single component, namely Lemma 1 [*line 12*]. This lemma is an application of the modus ponens using the implication of Lemma 7, which is the file `agent_goal.n3` shown in Listing 24. This implication has been applied on a part of the result of Lemma 2 as indicated by `r:evidence>` This rule application leads to the desired conclusion `<lena.jpg> dbpedia-owl:thumbnail _:sk3`, which contributes to the final result.

Thumbnail operation Lemma 2 [*line 17*] is another `r:Inference`, this time applying the rule from Lemma 4, which is the REST-desc description in Listing 21 that explains what happens when an image is uploaded. This rule application relies on the fact that `<lena.jpg> ex:smallThumbnail _:sk3` is proven (Lemma 3).

Upload operation Lemma 3 [*line 28*] explains the derivation of the triple that is a necessary condition for Lemma 2: `<lena.jpg> ex:smallThumbnail _:sk3`. The rule used for the inference is that from the upload action in Listing 22 (Lemma 5), instantiated with the initial knowledge of the agent that it has access to an image (Listing 23 / Lemma 6). Substituting `?image` by `lena.jpg`, gives the triples of the instantiated consequent, as shown by the `r:gives` predicate. Note in particular the newly created existential `_:sk3`; this entails the triple `<lena.jpg> ex:smallThumbnail _:sk3` which is needed for Lemma 2. Lemma 3 itself is justified by Lemma 6 (listing 23), which is a simple extraction that stands on itself. Hence, we have reached the starting proof's starting point.

From start to end As a summary, we will briefly follow the proof in a forward way. Extracted from the background knowledge in Listing 23, the triple `<lena.jpg> a dbpedia:Image` triggers the image upload rule from Listing 21, which yields `<lena.jpg> dbpedia-owl:thumbnail _:sk3` for some existential variable `_:sk3`. This triple in turn triggers the thumbnail rule from Listing 21, which yields `<lena.jpg> ex:smallThumbnail _:sk3`. Finally, this is the input for the filter rule from Listing 24, which yields the final result of the proof: the image has some thumbnail. So given the background knowledge and the two hypermedia API descriptions, this proof verifies that the goal follows from them.

The proof in Listing 25 is special in the sense that some of its implication rules, namely Listings 21 and 22, are actually hypermedia API descriptions. That means they do not fulfil an actual ontological implication. Instead, they convey dynamic information. Therefore, those steps in the proof can be interpreted as HTTP requests that should be performed in order to achieve the desired result. This proof is indeed a pre-proof: it is valid under the assumption that the described HTTP requests will behave as expected, which can never be guaranteed on an environment such as the Internet. The instantiation of a hypermedia API description turns it into the description of a concrete API operation. For instance, Lemma 3 [*line 28*] contains the following operation:

```
_:sk0 http:methodName "POST".
_:sk0 http:requestURI "/images/".
_:sk0 http:body <lena.jpg>.
_:sk0 http:resp _:sk1.
_:sk1 http:body <lena.jpg>.
<lena.jpg> ex:comments _:sk2.
<lena.jpg> ex:smallThumbnail _:sk3.
```

This instructs an agent to perform an HTTP `POST` request (`_:sk0`) to `/images/` with `lena.jpg` as the request body. This request will return a response (`_:sk1`) with a representation of `lena.jpg`, which will contain `ex:comments` and `ex:smallThumbnail` links. Since this is a hypermedia API, the link targets are not yet known at this stage. Therefore, they are represented by generated existential variables `_:sk2` and `_:sk3`. At runtime, the HTTP request will return the actual link targets, but at the pre-proof stage, it suffices to know that some target for those links will exist.

Indeed, the outcome of this rule – the fact that *some* `smallThumbnail` links will exist – serves as input for the next API operation in Lemma 2 [*line 17*]. The consequent of this rule is instantiated as follows:

```
_:sk4 http:methodName "GET".
_:sk4 http:requestURI _:sk3.
_:sk4 http:resp _:sk5.
_:sk5 http:body _:sk3.
<lena.jpg> dbpedia-owl:thumbnail _:sk3.
_:sk3 a dbpedia:Image.
_:sk3 dbpedia-owl:height 80.0
```

This describes a `GET` request (`_:sk4`) to the URL `_:sk3`, which will return a representation of a thumbnail that is 80 pixels high. This request is interesting because it is *incomplete*: `_:sk3` is not a concrete URL that can be filled in. However, this identifier is the same variable as the one in Lemma 3, so this description essentially states that whatever will be the target of the

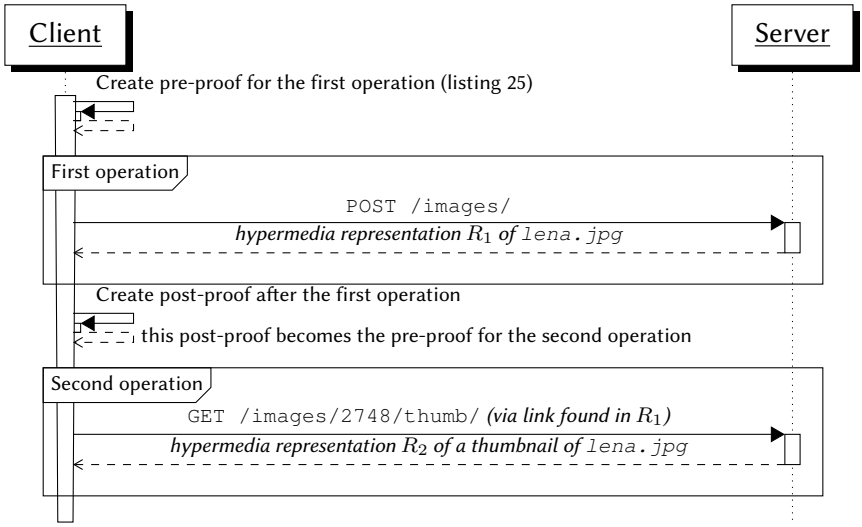


Figure 8.2: The *a posteriori* UML sequence diagram of an example execution shows how the second operation happened through a link in the first operation.

`smallThumbnail` link in the previous `POST` request should be the URL of the present `GET` request. The existential variables thus serve as placeholders for values that will be the result of actual API operations.

While the proof above is a pre-proof, a proper post-proof can be obtained by actually executing the `POST` HTTP request, which has all values necessary for execution (as opposed to the `GET` request where the URL is still undetermined). This execution will result in a concrete value for the `comments` and `smallThumbnail` link placeholders `_:sk2` and `_:sk3`. They lead to a proper post-proof that uses these concrete values, and hence that proof does not need the assumption that the `POST` request will execute successfully (because evidence shows it did). Figure 8.2 shows the UML sequence diagram of an example interaction between the client and the server.

As stated in its definition, a pre-proof implicitly assumes that each API will indeed deliver the functionality as stated in its RESTdesc description. The proof thus only holds under that assumption. For example, if a power outage occurs during the calculation of the aspect ratio, the placeholder will not be instantiated with an actual value during the execution, which can pose a threat to subsequent hypermedia API operations that depend on this value. However, the failure of a single API operation does not necessarily imply the intended result cannot be achieved. Rather, it means the assumption of the pre-proof was invalid and an alternative pre-proof – a new hypermedia API

composition – should be created, starting from the current application state. Such a pragmatic approach to proofs containing hypermedia API operations is unavoidable: no matter how low the probability of a certain operation to fail, failures can never be eliminated. Therefore, pragmatism ensures that planning in advance is possible. Each proof should be stored along with its assumptions in order to understand the context it which it can be used.

8.4 Hypermedia-driven Composition Generation and Execution

In contrast to fully plan-based methods, the steps in the composition obtained through reasoner-based composition of hypermedia APIs are not executed blindly. Instead, the interaction is driven by the hypermedia responses from the server; the composition in the proof only serves as *guidance* for the client, and as a guarantee (to the extent possible) that the desired goal can be reached. The composition that starts from the current state helps an agent decide what its next step towards that goal should be. Once this step has been taken, the rest of the pre-proof is discarded because it is based on outdated information. After the request, the state is augmented with the information inside the server's response. This new state becomes the input for a new pre-proof that takes into account the actual situation, instead of the expected (and incomplete) values from the hypermedia API description. In this section, we will detail this iterative composition generation and execution.

8.4.1 Goal-oriented Composition Generation

Creating a composition that satisfies a goal comes down to generating a proof that supports the goal. Inside this proof, the necessary hypermedia API operations will be incorporated as instantiated rules. Proof-based composition generation, unlike other composition techniques, requires no composition-specific tools or algorithms. A generic reasoner that supports the rule language in which the hypermedia APIs are described is capable of generating a proof containing the composition. For example, since RESTdesc descriptions are expressed in the N3 language, compositions of hypermedia APIs described with RESTdesc can be performed by any N3 reasoner with proof support. The fact that proof-based composition can be performed by existing reasoners is an advantage in itself, because no new software has to be implemented and tested. Furthermore, this offers the following benefits.

Incorporation of external knowledge Existing RDF knowledge can directly be incorporated into the composition process. Whereas com-

position algorithms that are specifically tailored to certain description models usually operate on closed worlds, generic Semantic Web reasoners are built to incorporate knowledge from various sources. For example, existing OWL and RDF ontologies can be used to compose hypermedia APIs described with different vocabularies.

Evolution of reasoners Many implementations of reasoners exist and they continue to be updated to allow enhanced performance and possibilities. The proof-based composition method directly benefits from these innovations. This also counters the problem that many single-purpose composition algorithms are seldom updated after their creation because they are so specific.

Independent validation When dealing with proof and trust on the Web, it is especially important that the validation can happen by an independent party. Since different reasoners and validators exist, the composition proof can be validated independently. This contrasts with other composition approaches, whose algorithms have to be trusted.

In order to make a reasoner generate a pre-proof of a composition, it must be invoked with the initial state, the available hypermedia APIs, and the desired goal. Here, we will examine the case for N3 reasoners and hypermedia APIs described with RESTdesc N3 rules, but the principle of proof-based composition is generalizable to all families of inference rules.

The hypermedia API descriptions include all those APIs available to the client. In practice, the number of supplied available hypermedia APIs would be substantially higher than the number of APIs in the resulting composition. The background knowledge can, for example, consist of ontologies and business rules. The reasoner will try to infer the goal state, asserting the other inputs as part of the ground truth. The initial state and background knowledge should correspond to reality, regardless of the results of the actual execution, provided the descriptions are accurate. In contrast, the API description rules only hold under the assumption of successful execution, due to the nature of the pre-proof.

If the reasoner can infer the goal state given the ground truth, we can conclude that a composition *exists*. To obtain the details of the composition, the reasoner must return the proof of the inference, i.e., the data and rules applied to achieve the goal. Inside this proof, there will be placeholders for return values by the server that are unknown at design-time. The proof will be structured as in Listing 25, where the initial state was Listing 23, the goal state Listing 24, and the descriptions Listings 21 and 22. No background knowledge was needed, but it could have been useful for instance if the

image of the initial state was described in different ontology, in which case the conversion to the DBpedia ontology would be necessary.

8.4.2 Hypermedia-driven Execution

In order to achieve a certain goal in a hypermedia-driven way, the following process steps can be followed.

Definition 37 (Pragmatic proof algorithm). *Given an API composition problem with an initial state H, goal g, description formulas R and background knowledge B, we define the pragmatic proof algorithm as follows:*

1. *Start an N3 reasoner to generate a pre-proof for (R, g, H, B).*
   (a) *If the reasoner is not able to generate a proof, halt with* failure.
   (b) *Else scan the pre-proof for applications of rules of R, set the number of these applications to n_{pre}.*
2. *Check n_{pre}:*
   (a) *If $n_{pre} = 0$, halt with* success.
   (b) *Else continue with Step 3.*
3. *Out of the pre-proof, select a sufficiently specified HTTP request description which is part of the application of a rule $r \in R$.*
4. *Execute the described HTTP request and parse the (possibly empty) server response to a set of ground formulas G.*
5. *Invoke the reasoner with the new API composition problem $(R, g, H \cup G, B)$ to produce a post-proof.*
6. *Determine n_{post}:*
   (a) *If the reasoner was not able to generate a proof, set $n_{post} := n_{pre}$.*
   (b) *Else scan the proof for the number of inference steps which are using rules from R and set this number of steps to n_{post}.*
7. *Compare n_{post} with n_{pre}:*
   (a) *If $n_{post} \geq n_{pre}$ go back to Step 1 with the new API composition problem $(R \setminus \{r\}, g, H, B)$.*
   (b) *If $n_{post} < n_{pre}$, the post-proof can be used as the next pre-proof. Set $n_{pre} := n_{post}$ and continue with Step 2.*

Before having a more theoretical look at the results of this algorithm, let us run through a possible execution of the composition example introduced previously.

- *(Step 1)* Given the background knowledge, initial state, and goal, the reasoner generates the pre-proof from Listing 25, which contains $n_{\text{pre}} = 2$ API operations.
- *(Step 2)* $n_{\text{pre}} \neq 0$, so continue with Step 3.
- *(Step 3)* The HTTP request to upload the image is the only one that is sufficiently instantiated, so it is selected.
- *(Step 4)* Execute the HTTP request by posting the image to `/images/`, and retrieve a hypermedia response. Inside this hypermedia response, there is a `comments` link to `/comments/about/images/37` and a `smallThumbnail` link to `/images/37/thumb/`. They are added to G.
- *(Step 5)* A post-proof is produced from the new state, revealing that the goal can now be completed with one API operation. Indeed, only an HTTP `GET` request to `/image/37/thumb` is needed.
- *(Step 6)* The above means that $n_{\text{post}} = 1$.
- *(Step 7)* $n_{\text{post}} = 1 < n_{\text{pre}} = 2$, so set $n_{\text{pre}} := 1$ and continue with Step 2.
- *(Step 2)* $n_{\text{pre}} = 1 \neq 0$, so continue with Step 3.
- *(Step 3)* Select the only remaining HTTP request in the pre-proof.
- *(Step 4)* Execute the `GET` request to `/image/37/thumb` and thereby obtain a representation of the thumbnail of the image.
- *(Step 5)* Generate the post-proof; it consists entirely of data as the necessary information to reach the goal has been obtained.
- *(Step 6)* No rules of R are applied, so $n_{\text{post}} = 0$.
- *(Step 7)* $n_{\text{post}} = 0 < n_{\text{pre}} = 1$, so set $n_{\text{pre}} := 0$ and continue with Step 2.
- *(Step 2)* $n_{\text{pre}} = 0$, so halt with *success*.

This example shows how the proof guides the process, but hypermedia drives the interaction. For instance, the URL needed for the `GET` request was not hard-coded: it was obtained as a hypermedia control from the server. This means that, even if the server changes its internal URL structure or the layout of the representation, the interaction can still take place. The client needs the RESTdesc descriptions to find out whether the complex goal is possible and what first steps it should take. Otherwise, it would have no way of knowing that the upload of an image results in a link to the thumbnail. However, once this expectation is there, the client navigates through hypermedia. We can compare this to driving with a map: the map gives the overall picture, but the actual wayfinding happens based on the actual roads and scenery when somebody undertakes the journey.

There are several reasons, why the situations in 1(a) or 6(a) can occur: the reasoner could have a technical problem, it could detect inconsistencies (a fulfilled antecedent of a rule with false in the consequent), it could simply be that there is no instance g' of g which is a logical consequence of $H \cup R \cup B$, but even if such a g' exists the reasoning problem is undecidable. This is because RESTdesc descriptions are rules with new existential variables in the consequence, which makes the problem in general undecidable, as discussed by Baget et al. [157].

However, we can show that the following holds:

Theorem 38. *Given an execution of the algorithm in theorem 37 that requires n executions of the reasoner (in Steps 1 and 5). If all n reasoning runs terminate, the algorithm terminates as well. Furthermore, if the algorithm halts with* success*, its output is a ground instance of the goal state.*

Proof. We first show termination: if the algorithm does not terminate, it especially never reaches the Steps 1(a) and 2(a) and Steps 1 and 2 always result in option (b). All formulas in $H \cup B$ are either ground formulas or simple rule formulas which do not contain existentials and which fulfil Lemma 30. Therefore no formula or conjunction of formulas which can be obtained by applying the proof steps from Definition 28 on $H \cup B$ contains universals or existentials. If for a pre-proof $n_{\text{pre}} > 0$ holds, it must by Corollary 35 contain at least one sufficiently specified HTTP request description. So, Step 3 and the following Steps 4–6 can always be executed. Step 7(a) reduces the set of RESTdesc descriptions, it can only be performed $|R|$ times. Starting from a fixed pre-proof pre_0, Step 7(b) can only be applied n_{pre_0} times. Thus, the algorithm terminates.

As every operation which changes the API composition problem for the pre-proof to be checked in Step 2, preserves the syntactic properties of the sets of

formulas involved, it is enough to show that the result of every pre-proof of an API composition problem which does not contain applications of RESTdesc rules is ground. This follows by the same arguments as above. As $H \cup B \cup \{\{g\} \Rightarrow \{g\}.\}$ only contains ground formulas and rules which fulfil the conditions of Lemma 30, it is not possible to derive formulas or conjunctions of formulas which contain existentials or universals thus the result of the proof is ground. □

8.5 Limits of the approach

Having introduced the pragmatic proof algorithm and discussed its advantages for hypermedia APIs – the algorithm is highly adaptive and can cope well with rapidly changing environments – we now want to discuss shortcomings and possible problems of the approach. Our goal is to clearly identify for which kinds of use cases we can perform planning by using proofs and for which cases this method is not well suited.

8.5.1 Existential rules

Our approach relies on existential rules. While initial tests have shown that the EYE reasoner performs well when applying automatically generated RESTdesc descriptions [12], it is in general rather easy to construct examples which – when applying classical inferencing algorithms – lead to infinite loops. To illustrate that, we discuss a classical example problem for existential rules. Consider the following rule:

```
{?x a :Person}=>{?x :mother _:y.  _:y a :Person.}.
```

This rule means:

"Every person has a mother and this mother is again a person."

The moment we have a triple which declares someone as a person, the rule will trigger and add a triple representing the mother of this person. As this mother again has a mother which then again has a mother and so on, we get into an infinite loop and the reasoning process will not stop.

While the loop we created here can be avoided – it is rather unusual to write a rule which triggers on its own results – more complicated loops are more difficult to detect. A lot of research focusses on how to restrict existential rules in order to guarantee termination (e.g. [157–159]). These

restrictions mostly take combinations of rules into account and cannot be imposed separately on individual rules. Therefore such approaches cannot be used in the Web with its distributed nature: As we do not know which APIs can and will be combined using our algorithm, we cannot ask the provider of one service to write its description in a way that it does not lead to infinite loops when being combined with another service.

Instead of restricting the rules, we work with different reasoning tactics: A reasoning tactic determines when a reasoner stops (for example after a certain number of rule-applications or directly after one instance of the goal is found), it controls the order in which rules are applied (following Prolog's left-most selection rule [90] or any other similar rule) and it can exclude rule applications based on their consequence (for example no applying a rule for whose consequence the dataset already contains a ground instance). Such strategies are implemented in EYE [79] and have proven to be useful in practice. The aim of the pragmatic proof algorithm is not to find all plans or the best plan towards a goal, its aim is to find one possible plan and then execute it. In this set-up it makes sense to try – if needed – different reasoning strategies on a API composition problem till one solution is found.

8.5.2 Choices

In the previous section we already explained, that the pragmatic proof algorithm is not designed to find different paths towards a goal but to find *one* valid path. In settings where different alternatives need to be provided, the approach cannot be applied: The purpose of a proof in the classical sense is to provide evidence for a derived fact. Once such an evidence is found, it does not really make sense to search for an alternative evidence. Therefore, N3 reasoners are not optimised to find alternative proofs. But even if they were, the fact that we combine reasoning on background knowledge and application of RESTdesc descriptions would quickly lead to performance problems: From the reasoner's point of view there is no difference between RESTdesc descriptions and other rules. Alternative proofs would also need to take all alternative ways to derive background knowledge into account. If we for example have the following knowledge about `:Kurt`

```
:Kurt a :Researcher; :Man; :Mathematician.
```

and we know that every `:Researcher`, every `:Man` and also every `:Mathematician` is a `:Person`, we need to always generate three different proofs whenever the knowledge that `:Kurt` is a `:Person` is required. These alternatives then need to be combined with all alternatives we have for all other derivations involved in a proof. The complex problem of calculating

different combinations of RESTdesc descriptions in order to fulfil a goal gets even more complex and – depending on the number of rules involved – not solvable by a normal computer.

8.5.3 Change

Another shortcoming of the approach of using classical proofs as plans towards a goal is that in this set-up it is not possible to express *change*. It is always possible to add new data and triples but it is not possible to remove triples. We can therefore not (or at least not in an easy way) describe that an API operation changes, for example, the state of the light from off to on. We cannot express that the triple

```
:myLight :status :off.
```

gets *replaced* by

```
:myLight :status :on.
```

The reason for that is that N3Logic– such as most Web logics – is based on first order logic where we have the principle of monotonicity: If a formula ϕ follows from another formula ψ this derivation still needs to be true when we add further knowledge to the premise. From $\gamma \wedge \psi$ we still need to be able to conclude ϕ regardless of the concrete shape of γ. In order to properly model change which includes the possibility to invalidate data we would need a non-monotonic logic.

For our API compositions this means that the approach works well if we have settings where many `GET`-requests are included – these result in new knowledge – while it can be problematic to – for example – include a `DELETE`-request.

8.6 Proofs for Source Selection

Having had a closer look to the advantages and shortcomings of the idea of using proofs as plans we now discuss another application of the idea: We can use existential rules to describe possible sensor measurements and their meaning for the context they are used in. In complex scenarios where reasoning with all results of such sensors would be too slow, proofs enable us to identify the sensors which are relevant to retrieve the information we need. We explain that idea on an example taken from the healthcare domain.

With the development of new technologies and the possibility to rapidly produce and store more and many different kinds of data, healthcare systems

have new opportunities. Next to the classical electronic health records (EHR) of patients, they can use background data clarifying diseases, their diagnosis and possible cure, organisational data (e.g. the availability and competences of staff members in a hospital), but also data produced by sensors for example to measure values on patients (e.g. their blood pressure or pulse), or to control the patient's ambience in a hospital (e.g. the illumination or temperature in his room). To combine these in a meaningful way, Semantic Web reasoning systems are particularly well suited: their declarative approach allows the user to explain concepts represented by the data and to specify how for example the known EHR and newly measured values of a patient relate to each other. If the system knows how to interpret the values of a specified kind of sensor, it is easy to add and remove new instances of this type. But as most computer systems, Semantic Web implementations also have to cope with the challenges of Big data[1]: if they have to deal with with a huge *volume* and different forms of data (*variety*), which can come in a high *velocity*, for example in data streams, and in different levels of quality (*veracity*), the performance of the systems can be rather poor.

Our approach aims to tackle this problem: in a setting where many sensors are present and a request constantly needs to be answered, we find the sensors relevant for this specific goal. These sensors and their results can then be monitored more closely while others, not relevant for the problem, can be ignored. By that, the amount of data taken into account can be reduced. To do so, we developed a special format to describe possible sensor queries consisting of three parts: the context in which a sensor query becomes relevant, the query itself, and the consequence in terms of the ontology used. Together with the related knowledge the descriptions enable a reasoner to produce a formal proof which we use to find the sensor queries contributing to the goal.

The remainder of this section is structured as follows: In Section 8.6.1 we illustrate a scenario in which our application can be used. This serves as a running example throughout the whole section. In Section 8.6.2 we explain the details of our implementation including our new description format and the use of formal proofs to determine relevant sensor queries. In Section 8.6.3 we discuss the relation of our approach to other research.

8.6.1 Use case

Our use case is set in a hospital. This hospital is equipped with a large number of different sensors to monitor, among others, the temperature and

[1] http://www.ibmbigdatahub.com/infographic/four-vs-big-data

light intensity in a room, values related to the patient's health, e.g. pulse, blood pressure and body temperature, and the location of staff members and patients. We want to enable a computer to use this data in order to answer requests or to detect critical situations. These situations do not only depend on the sensor values themselves, but also on their surrounding: patients, their diseases, settings in the hospital like rooms and locations of the sensors and many more aspects could influence the decision, whether a sensor value is normal or alarming. Such surroundings can change: patients enter and leave the hospital, sensors are added or removed. Our system should thus take the context into account, and easily adapt to possible changes. We therefore use Semantic Web technology and describe the data itself and the context of the sensor setting in a machine understandable way. If the system *understands* the concept of a light sensor, it knows how to deal with a new instance of such a device. Only the new sensor and its surrounding need to be described, the concrete use of the sensor, e.g. threshold values, can be deducted from the knowledge. The ontology we use includes the hospital and its rooms, the patients and their diseases, and the descriptions and locations of the different sensors. We aim to find in the current setting all possible kinds of problems which could be detected by a sensor.

In our ontology, we call these problems *faults*. Such faults could be caused by many reasons like an alarming sensor value (e.g. the luminance, sound or temperature in a room is a above or below a critical threshold), or faulty sensors (e.g. a temperature, luminance or sound sensor does not work properly). As stated, faults depend on context like patient profiles and locations or the general sensor set-up in the hospital: While a pulse of 110/min might be alarming for a grown-up, it is totally normal for an infant. Similarly, whether the light in a room is too bright depends on who is using the room: a patient suffering from a concussion is more sensitive to light than a patient treated for Parkinson. In an empty room the light intensity does not influence the well-being of the patients and is therefore irrelevant in our case.

This relevance of context makes the detection of faults a complex task. While in a small or simple setting where either context is irrelevant or the number of sensors is small, it is no problem to perform reasoning on the sensor data whenever a value changes, this is different for the given situation. Each reasoning task takes time (depending on its complexity the task can take seconds on a common computer (remember for example the performance results we had in Section 6.1) and the performance of computers present in hospitals is often limited.

We therefore want to find out early in the current setting (e.g. rooms of patients, patients' diagnosis):

```
PREFIX : <http://example.org#>
PREFIX Ar:  <...RoleCompetenceAccio.owl#>
PREFIX Du:  <...ontologies/DUL.owl#>
PREFIX SN: <...SSNiot.owl#>
PREFIX WA: <...WSNextensionAccio.owl#>

:bob Du:hasRole [ a Ar:PatientRole];
   Du:hasLocation :room21;
   WA:lightTreshold 200.

:lightsensor17
    Du:hasLocation :room21;
    a SN:LightSensor.
```

Listing 26: Instance of a patient with a light sensivity of 200 who is located in a room with a light sensor.

1. Which sensors are relevant to our request, i.e. which sensors could detect a fault?

2. Which values are critical for these sensors?

This allows us to monitor only these sensors and queries, and just perform complex reasoning when critical values are measured.

8.6.2 Implementation

Our implementation uses N3Logic to identify the sensors and sensor queries which can be used to detect alarming situations. The basic idea of our approach is, given a description of the hospital's setting, to use existential rules to describe possible sensor queries and their meaning in terms of the ontology. We call that format SENSdesc. Together with a general goal, i.e. a request to be answered, a reasoner can employ all the knowledge to produce a proof. If this proof contains the instantiated version of a sensor query, this occurrence indicates that in the given context this query is relevant and needs to be monitored. We explain the details of this idea below.

Description of context

To be able to use the knowledge about context in a hospital to determine which sensors need to be observed carefully and for which values, this context knowledge needs to be described in a machine understandable way. Like

```
PREFIX WA: <...WSNextensionAccio.owl#>
PREFIX SN: <...SSNiot.owl#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>

WA:LuminanceAboveThresholdFault
 a owl:Class ;
 owl:equivalentClass [
  a owl:Restriction;
  owl:onProperty SN:hasSymptom;
  owl:someValuesFrom
   WA:LuminanceAboveThresholdSymptom
 ].
```

Listing 27: Axiom using `owl:someValuesFrom`.

in Section 6.1, our implementation again uses the ACCIO ontology[2] [117]. ACCIO makes use of other well established ontologies e.g. the Semantic Sensor Ontology [160], and was designed for continuous care settings. It covers all concepts relevant to our use case. A simple example of the use of the ontology is given in Listing 26.[3] We see specifications about someone named `:bob` who is a patient and currently present at a location called `:room21`. In this `:room21` there is furthermore a light sensor (`:lightsensor17`). We also see that `:bob` has a light threshold value of 200 assigned. Such an assignment can either be derived by reasoning (e.g. if Bob has a concussion and all patients with a concussion have this light threshold value) or it can be set individually to a patient (e.g. by a doctor).

Ontology reasoning

To incorporate the knowledge specified in the ontology, we again use the OWL-RL rules introduced in Section 6.1. But for this use case, we need to extend them: For the OWL predicate `owl:someValuesFrom` OWL RL only supports the subclass version. While we also need the other direction.

The property restriction `owl:someValuesFrom` for a predicate p on a class C means, that each instance of C has at least one value connected to it via p. The object of `owl:someValuesFrom` determines the class to which that connected value belongs. To illustrate that we display the description of the example class `WA:LuminanceAboveThresholdFault` in Listing 27. This class covers a special kind of faults and is defined

[2] Available at: https://github.com/IBCNServices/Accio-Ontology/tree/gh-pages

[3] In this and all the following listings the dots (...) in the prefix declaration stand for http://IBCNServices.github.io/Accio-Ontology/.

```
PREFIX owl:<http://www.w3.org/2002/07/owl#>

{?D owl:equivalentClass ?C.
 ?C owl:someValuesFrom ?Y.
 ?C owl:onProperty ?P.
 ?U ?P ?V.
 ?V a ?Y.
}=>{?U a ?D}.
```

Listing 28: OWL RL rule for `owl:someValuesFrom`.

(using `owl:equivalentClass`) as the class of all instances which have a symptom (`SN:hasSymptom`) being an instance of the class `WA:LuminanceAboveThresholdSymptom`. Given

```
:observation1 SN:hasSymptom [
 a WA:LuminanceAboveThresholdSymptom].        (8.1)
```

the description enables the reasoner to apply the RL rule in Listing 28 [4] and conclude that

```
:observation1 a
              WA:LuminanceAboveThresholdFault.  (8.2)
```

But the mechanism also needs to work the other way around. Given an instance of a class with a `owl:someValuesFrom` restriction on a property *p* like in Formula 8.2 where `:observation1` is a `WA:LuminanceAboveThresholdFault`, it can be derived that there *exists* an object of the class indicated in the class axiom which is connected to that instance via *p*. In our example, that is an instance of the class `WA:LumincanceAboveThresholdSymptom` connected to our `:observation1` via the property `SN:hasSymptom`. The existential rule displayed in Listing 29 produces the expected result, it derives Formula 8.1 from Formula 8.2 and Listing 27. By implementing this and other rules, we extended the set of OWL concepts supported by our rule based implementation.

[4] In the actual implementation we have two rules handling this case: one to derive an `owl:subClass`-relation form a `owl:equivalentClass`-statement and one rule like the one displayed but acting on `owl:subClass` instead of `owl:equivalentClass`. We shortened it here to one rule keep the examples short.

```
1 PREFIX owl:<http://www.w3.org/2002/07/owl#>
2
3 {?x a ?C
4  ?C owl:equivalentClass
5    [ owl:onProperty ?property;
6      owl:someValuesFrom ?from ].
7 }=>{?x ?property _:e. _:e a ?from.}.
```

Listing 29: Existential rule for `owl:someValuesFrom`.

Sensor queries

Our concept, SENSdesc, describes possible sensor queries via existential rules of the form

```
{ pre-condition } =>{ query-description.
                      post-condition.   }
```

where the three parts are as follows:

Pre-condition: This part specifies the kind of sensor the description is valid for, the situation in which the query can be done and additional knowledge relevant for the query.

Query description: This part specifies how exactly a query has to look like. Additional parameters relevant for the query can also be added here.

Post-condition: Here we describe the consequences of the query-result.

For pre- and post-condition it is crucial to use vocabulary specified in the surrounding ontology. Only with further background knowledge about the terms and contexts used, reasoning can take place. We illustrate the concept of a SENSdesc rule in Listing 30.

The *pre-condition* (lines 9–14) describes the situation in which our description is relevant: A patient has a light threshold defined and there is a light sensor at the same location as the patient. In our example this is the case for our patient *Bob* from Listing 26.

The *query-description*, lines 16–19, states which query has to be performed to the light sensor. The example query here tests if the measured value is above the threshold. Note that sensor and threshold value are expressed by universal variables taking values from pre-condition. Both depend on the context data.

The *post-condition*, lines 21–24, describes the consequence of a successful sensor query: if the query triggers, a

```
PREFIX : <http://example.org#>
PREFIX Ar:  <...RoleCompetenceAccio.owl#>
PREFIX Du:  <...ontologies/DUL.owl#>
PREFIX SN: <...SSNiot.owl#>
PREFIX sosa:  <http://www.w3.org/ns/sosa/>
PREFIX WA: <...WSNextensionAccio.owl#>

{
#pre-condition
?p Du:hasRole [ a Ar:PatientRole];
   Du:hasLocation ?l;
   WA:lightTreshold ?t.
?s Du:hasLocation ?l;
   a SN:LightSensor.
}=>{
#query-description
?s :sensorQuery {
  ?s :value _:v.
  _:v :greaterThan ?t }.

#post-condition
?s sosa:hasObservation _:o.
_:o SN:hasSymptom [ a
 WA:LuminanceAboveThresholdSymptom].
}.
```

Listing 30: Example description. For a patient for whom a lightThreshold is defined, light sensors at his location can test whether the light is below this threshold.

`WA:LuminanceAboveThresholdSymptom` is observed. In the ontology, the observation of this symptom can—depending on the further context—have several consequences. Here, via Listing 27, we can get a `WA:LuminanceAboveThresholdFault`.

Goal

As mentioned in Section 8.6.1, we want to know in our example, which faults could be detected by a sensor using the current data consisting of ontology, description of the context and SENSdesc rules to describe possible sensor queries. To do so, we search for all instances of the class `SN:Fault` the reasoner can detect. The goal for this looks as follows:

```
{?x a SN:Fault}=>{?x a SN:Fault}.
```
(8.3)

Remember that even though antecedence and consequence are the same, the rule is not redundant because it is marked as a goal. The reasoner looks for cases where this rule can be applied and provides a proof for them.

Making use of proofs

Having the goal, the different SENSdesc descriptions including the one from above, the OWL ontology and the N3 rules to perform OWL reasoning at our disposal, we can start an N3 reasoner and produce a proof. This proof enables us to find the sensors and the concrete sensor queries relevant to our problem. We illustrate that idea on our example. Additionally to the axioms mentioned above, our ontology also contains the triple:

```
WA:LuminanceAboveThresholdFault
                        rdfs:subClassOf SN:Fault.   (8.4)
```

This indicates that every instance of the class `WA:LuminanceAboveThresholdFault` is also an instance of the class `SN:Fault`. Using the data displayed in this section and the goal in Formula 8.3 the reasoning process produces the proof in Listing 31. From the different proof steps (lemmas) composing the proof, there is one particularly interesting: Lemma 4 (lines 27–38) describes the application of our SENSdesc rule (`r:Inference`) leading to:

```
:lightsensor17 :sensorQuery {
  :lightsensor17 :value _:sk_0.
  _:sk_0 :greaterThan 200 }.
:lightsensor17 sosa:hasObservation _:sk_1.
_:sk_1 SN:hasSymptom _:sk_2.
_:sk_2 a WA:LuminanceAboveThresholdSymptom.
```

These triples contain a concrete sensor query to the sensor `:lightsensor17`: if the value of this sensor is greater than 200 we need to add the observation of a `WA:LuminanceAboveThresholdSymptom` to the knowledge. We thus know from the the proof that in our current setting this particular sensor is important and needs to be monitored with the indicated value.

This technique of looking for inference steps applying SENSdesc rules can be generalised: If we additionally had another sensor in another room where another patient with a defined light threshold value was located, this sensor with the corresponding threshold value would also appear in the proof. At the same time, light sensors located in rooms with no patients or with patients without a defined luminance threshold value will not be listed. In that way we find the sensors and queries relevant to our particular problem.

```
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX SN: <http://IBCNServices.github.io/Accio-Ontology
/SSNiot.owl#>
PREFIX sosa:  <http://www.w3.org/ns/sosa/>
PREFIX WA: <http://IBCNServices.github.io/Accio-Ontology
/WSNextensionAccio.owl#>
PREFIX : <http://example.org#>
PREFIX r:  <http://www.w3.org/2000/10/swap/reason#>

[] a r:Proof, r:Conjunction;
  r:component <#lemma1>;
  r:gives { _:sk_1 a SN:Fault. }.

<#lemma1> a r:Inference;
  r:gives { _:sk_1 a SN:Fault. };
  r:evidence (<#lemma2>); r:rule <#lemma5>.

<#lemma2> a r:Inference;
  r:gives { _:sk_1 a SN:Fault.};
  r:evidence (<#lemma3> <#lemma7>); r:rule <#lemma8>.

<#lemma3> a r:Inference;
  r:gives { _:sk_1 a WA:LuminanceAboveThresholdFault.};
  r:evidence (<#lemma4> <#lemma6>); r:rule <#lemma9>.

<#lemma4> a r:Inference;
  r:gives {
    :lightsensor17 :sensorQuery {
      :lightsensor17 :value _:sk_0.
      _:sk_0 :greaterThan 200
    }.
    :lightsensor17 sosa:hasObservation _:sk_1.
    _:sk_1 SN:hasSymptom _:sk_2.
    _:sk_2 a WA:LuminanceAboveThresholdSymptom.
  };
  r:evidence (<#lemma10>); r:rule <#lemma11>.

<#lemma5>  a r:Extraction;
           r:because [ a r:Parsing; r:source <Formula6>].
<#lemma6>  a r:Extraction;
           r:because [ a r:Parsing; r:source <Listing2>].
<#lemma7>  a r:Extraction;
           r:because [ a r:Parsing; r:source <Formula7>].
<#lemma8>  a r:Extraction;
           r:because [ a r:Parsing; r:source <owl_sub.n3>].
<#lemma9>  a r:Extraction;
           r:because [ a r:Parsing; r:source <Listing3>].
<#lemma10> a r:Extraction;
           r:because [ a r:Parsing; r:source <Listing5>].
<#lemma11> a r:Extraction;
           r:because [ a r:Parsing; r:source <Listing6>].
```

Listing 31: (Simplified) Example proof for the goal displayed in Formula 8.3 taking into account the data from Listings 2–6 and an additional rule to handle `owl:subclassOf`. Lemma 4 includes an instatiated version of the SENSdesc description.

8.6.3 Streams in the Semantic Web

We discuss the connection between our work and Semantic Web stream processing and reasoning. A complete overview of this topic can be found in Margara et al. [161]. RDF stream processing systems are classically divided into two categories: Complex Event Processing systems (CEP) and Data Stream Management Systems (DSMS). While the former normally have timestamped triples and the detection of patterns within these as subject, the latter focus on the data produced in fixed time periods, so called windows. An overview of systems can be found at [162]. To query on streams, several languages have been developed, e.g. C-SPARQL, CEQLS and $\text{SPARQL}_{\text{stream}}$. Recent research aims to unify them into the language RSP-QL [163]. Our approach is independent of these languages since reasoning only happens on top of query results whose consequences are defined in the SENSdesc descriptions. In that aspect it also differs from ontology based data access (OBDA) on streams such as e.g. [164]. OBDA takes sensor queries as input which are rewritten based on ontology knowledge to easier sensor queries to perform on the stream. In our approach, we do not have a sensor query on top of ontology and streams, we have a simple goal, which can not directly contain informations about streams and or windows. This kind of information can only be expressed in the query description part of our SENSdesc rules.

8.6.4 SENSdesc and RESTdesc

The approach presented follows the idea of RESTdesc – proofs are used to select services – but it only suffers from one of the main problems which can occur when applying RESTdesc (Section 8.5): All sensor data can be understood as new information which can be added to the knowledge at hand, we do not need to take care of *change*. In the given scenario we do not need alternative proofs (*choice*), if we have a set of sensors which we can monitor to detect a fault, we do not need an alternative. Nevertheless, we can use the approach to search for alternative ways to detect a fault if one sensor is failing, we only need to re-run the reasoner excluding the description of that particular sensor. Only the problems caused by the fact that we use *existential rules* here can form an obstacle in the presented set-up, but for this case we suggest, as above, to apply different reasoning strategies till sensors detecting a fault can be found.

8.7 Conclusion

In this chapter we investigated how proofs produced by N3 reasoners can be used in practice. The applications we discussed used proofs as plans: We describe possible API operations or possible sensor queries by means of existential rules where the existentials represent the knowledge gained by performing the operation of executing the query. In the pre- and post-conditions, this knowledge is set into context. We express under which circumstances we can execute operations and queries and what their outcome means for our set-up. By that, we connect the rules to the context knowledge such that plans produced with them adapt to the present situation. In combination with a goal, we use descriptions, background knowledge and rules to generate proofs. As every description which is relevant for the goal appears in such a proof, we can understand the proof as a plan and use the information about specific query parameters or call methods as instructions to execute these plans. By constantly updating such plans we can always take all necessary information into account and adapt in case the situation changes.

The use cases we described here were only two possible applications of N3 proofs. The fact that N3 reasoners produce proofs which are themselves written in N3 provides many opportunities. Proofs can be used to generate explanations together with the nurse assignments we discussed in Section 6.1 (the author of this thesis was involved in the development of the so-called *Why-Me?-button* which exactly does that and was submitted for patent application [165]) or they can help to find the causes of constraint violations as discussed in Section 6.2 (this is further explained in [11]). There are many set-ups in in which N3 proofs can be beneficial.

Coming back to our initial discussion in Chapter 2 we can conclude that if we realise the *Proof layer* of the Semantic Web stack by using N3, we have a clear and strong connection to the highest layer of the stack, the layer of *Applications*. As this is the most important layer which ultimately needs to be supported by all levels if we want to realise the vision of the Semantic Web, using N3Logic can bring us one step closer to fully realising the Semantic Web.

This chapter was partly based on the publications:

R. Verborgh, D. Arndt, S. Van Hoecke, J. De Roo, G. Mels, T. Steiner, J. Gabarró Vallés, The pragmatic proof: Hypermedia API composition and execution, Theory and Practice of Logic Programming 17 (1) (2017) 1–48. doi:10.1017/S1471068416000016.
URL http://arxiv.org/pdf/1512.07780v1.pdf

D. Arndt, P. Bonte, A. Dejonghe, R. Verborgh, F. De Turck, F. Ongenae, SENSdesc: connect sensor queries and context, in: R. Zwiggelaar, H. Gamboa, S. Fred, Ana Bermúdez i Badia (Eds.), Proceedings of the 11th International Joint Conference on Biomedical Engineering Systems and Technologies, SCITEPRESS, 2018, pp. 671–679. doi:10.5220/0006733106710679.
URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006733106710679

Chapter 9

Conclusions

In this thesis we investigated N3Logic and its potential to become the *Unifying Logic* for the Semantic Web. Many properties of N3 make it a very strong candidate for this role. It is based on RDF, the most established standard of the Web, and it extends this framework by rules and citations. This connection makes it possible to reason about the RDF representation of OWL ontologies by using rules which support the different concepts forming part of that standard. N3 furthermore supports querying. It is possible to define filter rules such that the reasoners only provide all valid consequences of these rules. N3 supports the layer of proofs. N3 reasoners provide proofs for their derivations. These proofs are again written in N3 which makes it easy to share and exchange them and to use them in further applications.

We also discussed the most important open problem when it comes to N3: As its semantics is not formally defined the implementations using this logic differ in their understanding of implicit quantification. We tackled this problem by introducing a format close to N3 which only supports explicit quantification, N3 Core Logic. For this Logic we provided the formal semantics and defined concrete mappings from N3 syntax to it which followed the different interpretations. By doing that we made the differences between interpretations explicit. Our N3 Core Logic makes it possible to discuss different ways to define N3's semantics which then – hopefully – leads to an agreement.

Below, we connect these achievements to the research questions we posed in Section 2.6 and answer these in detail.

9.1 Review of the research questions

We focussed on four general requirements on the *Unifying Logic* for the Semantic Web: (1) clear semantics, (2) compatibility with RDF, (3) connection of the logical layers and (4) support of proofs. To understand in how far N3 meets these requirements and is thus a candidate to fulfil that role, we posed research questions for each of these aspects. We now discuss how these research questions can be answered based on the research performed in this doctoral project.

For the first requirement, clear semantics, we posed two questions. The first of these was Research Question 1:

"How can the differences between the individual interpretations of N3 assumed by different reasoners following the W3C team submission [71] and the journal paper [5] be formally expressed?"

Our analysis (Chapter 3) of the problem of diverging interpretations showed that the reasoners Cwm and EYE differ in their understanding of implicit universal quantification and that this difference is caused by the underspecification of the term *parent formula* in the W3C team submission. Universals are quantified on their *parent formula*. For EYE the overall formula is the *parent* and all universal variables are quantified on that level. For Cwm universal quantification depends on the syntactic structure of a formula: Cwm's parent formula is the second next formula in curly brackets { } surrounding the universal variable. But, if this parent is already in scope of another universal quantification for the same variable this quantification also counts for its descendants. The interpretations thus disagree on where exactly they assume a nested universal to be quantified. To clarify the differences between interpretations it therefore makes sense to make implicit quantification explicit. To do so we defined N3 Core Logic – a logic which is very similar to N3 and only differs from the latter by the fact that it only supports explicit quantification – and defined two mappings from N3 syntax to that logic, one following Cwm's semantics and one following EYE (Chapter 4). Especially to be able to cope with Cwm's interpretation of implicit universal quantification we used attribute grammars to define these mappings as these act directly on a formula's syntax tree. Our attribute grammar assigns one N3 Core Logic formula following the interpretation of EYE and one following Cwm to any given N3 formula. These two translations can easily be compared as the position of every quantifier for every implicitly universally quantified variable is stated explicitly. We thus accept Hypothesis 1: *"Existing interpretations differ in their understanding of implicit quantification and this difference can be expressed by mapping formulas containing such constructs to a core logic which only supports explicit quantification."* By defining N3 Core Logic we provided

a tool which enables different parties implementing or using N3Logic to communicate their understanding of formulas containing implicit universal quantification. This makes it possible to identify and discuss differences, and to understand their consequences.

The consequences we are particularly interested in are related to the practical use of N3. For the two interpretations we considered in this thesis we posed Research Question 2 which is the second research question in the context of a clear semantic definition:

"How big is the impact of having different interpretations for N3 formulas in practice?"

In order to answer this question we implemented the attribute grammar we defined earlier (Chapter 5). For any given N3 formula our implementation returns the interpretations in N3 Core Logic according to Cwm and to EYE. To understand the impact of the difference we applied our implementation to two datasets: one dataset consisting of different formulas which were used in industry-related research projects – these files were not created to test the reasoners but to solve practical problems – the other one consisting of example formulas provided for the EYE reasoner. The latter formulas contain examples for the different applications the users of the reasoner have reported and therefore cover a huge variety of different ways to apply the logic. For these datasets we compared the two N3 Core Logic translations of each formula and discovered that these do not concur for 31% of all our files tested. We thus accept Hypothesis 2: *"For at least one quarter of all N3 formulas contained in our dataset of practical examples the interpretations by the reasoners EYE and Cwm differ."* From a practical point of view this result means that the way we define the semantics of N3 directly influences the formulas we considered and thus the working of the applications they are used for. This means that fixing the semantics of N3 is more than just an academical exercise. In order to guarantee interoperability as required by the Semantic Web we need an agreement and to come to this agreement we should involve all parties applying N3Logic on a practical level. We also discussed how such an agreement could look like: we could either follow Cwm or EYE, or exclude implicit quantification in all constructs where it leads to ambiguity and use explicit quantification instead. Unfortunately, the meaning of explicit quantification in N3 is not properly defined either which made us discard this solution. When choosing between EYE's and Cwm's interpretation of implicit universal quantification, we give a clear preference in quantifying all universals on top level, mainly because that is easier to implement, easier to understand for the user and also much more in-line with other formalisms containing implicit universal quantification as for example Prolog. A drawback of this solution is that – at least if we exclude

explicit quantification – it makes the logic less expressive. Whether or not the expressivity of Cwm's interpretation of N3 is needed in practice is subject to future research. Independent of that result, it is also necessary to agree on the meaning of N3's explicit quantification, a subject we omitted in this thesis.

In order understand in how far N3 logic addresses the second requirement we had on the *Unifying Logic*, the compatibility with RDF, we raised Research Question 3:

"How can we define the semantics of N3Logic in a way which is compatible to RDF?"

To answer this research question we defined the semantics of N3 Core Logic using a model theoretic approach and formalised two mappings from N3 syntax to that logic (Chapter 4). By doing so, we provided two possible definitions for the semantics of N3. The definitions we used when defining the model theory for N3 Core Logic were compatible with those from RDF with one exception: Instead of mapping implicitly quantified variables directly to elements of the domain of discourse we added a grounding step which assigned N3 constants to these variables before mapping them to the actual resources they represent. We thus accept Hypothesis 3: *"The model theory for N3Logic can be specified in such a way that it is compatible with the model theory of RDF with the only exception that RDF blank nodes to refer to named instances in the domain of discourse."* We made this exception because cited N3 formulas should not be referentially transparent: Even if cited formulas refer to the same statement they should be handled differently if they contain different constants. As a consequence, we need to be able to assign constants to variables occurring in such cited formulas. For that reason, we had to deviate from RDF. We expect the practical impact of that deviation to be rather small given that such a grounding step is also implemented in all N3 reasoners we are aware of. To understand the impact from a theoretical point of view, further research is needed and this research could start by studying the properties of Herbrand Logic [102].

As N3 syntactically extends RDF, the result from above that once we agreed on a semantics of N3 this semantics can be defined in accordance with RDF – with the exception we mentioned – shows that N3 is truly compatible with RDF. This compatibility was something we expected from the *Unifying Logic*. As N3 is compatible with RDF we can use that logic to reason about anything which can be expressed in RDF/Turtle syntax. In particular, we can reason about OWL ontologies. We used this property when approaching Research Question 4 which focuses of the third requirement we had on the *Unifying Logic*, its ability to connect the building blocks *Querying*, *Ontologies/Tax-*

onomies and *Rules* of the semantic Web stack:

"In which aspects can Notation3 Logic cover and connect the building blocks Querying, Ontologies/Taxonomies *and* Rules *of the Semantic Web stack?"*

In order to realise the vision of the Semantic Web it is crucial that its concepts and technologies can be used in practical applications. We therefore tackled this research question from a practical point of view. Being a rule-based formalism, N3 covers the building block *Rules*. To understand the relationship of N3 to querying and ontology reasoning we considered two practical use cases which previously have been tackled by using OWL DL reasoning and SPARQL querying: a semantic nurse call system and a system to perform RDF-validation (Chapter 6). We approached both use cases by applying rule-based reasoning instead. In our implementation we made use of N3 as it is implemented in EYE. Our resulting systems were able to provide the same functionality as the original ones. The performance of our nurse call system was faster in all cases. The RDF-validation system was faster for datasets below 100,000 triples. We thus accept Hypothesis 4: *"With N3Logic we can tackle the same use cases as with OWL DL reasoning and/or SPARQL querying without suffering a loss of performance in terms of execution times or correctness of the result."* However, this acceptance comes with a condition: The EYE reasoner supports several built-ins which were crucial to implement the use cases we discussed. As N3Logic is not standardised yet, it is also not fixed which of the built-in functions we used form part of the logic. Such as the semantics of N3, this decision should be made by the Semantic Web community. If N3 supports some form of scoped negation as failure and predicates with a similar expressivity as those defined for RIF, then N3 supports the applications of querying and ontology reasoning we investigated and many others.

The last requirement on the *Unifying Logic* for the Semantics Web we covered with our research questions was the connection to the layer of *Proofs*. For N3 there is a proof vocabulary defined, the SWAP vocabulary, which makes it possible to express proofs in N3. This vocabulary is used by different reasoners to explain their derivations. This explanation is only meaningful if we know that the steps it describes are themselves correct. This correctness has not been proven so far. We therefore asked in Research Question 5:

"How can we verify that the proof steps defined by the SWAP vocabulary are correct?"

The term *correctness* itself depends on the semantics we assume for a logic. As this semantics is not fixed yet for N3 and even differs between reasoners we focussed on a set of N3 formulas which are not subject to ambiguity, *simple formulas* (Chapter 7). These are formulas for which both reasoners EYE and

Cwm quantify all universal variables on top level. For these formulas we defined the direct semantics. Using that definition we then formally defined the proof calculus which is expressed by the SWAP vocabulary and proved its correctness. We thus accept Hypothesis 5: *"The proof steps included in the SWAP vocabulary can be formally defined on top of the model theory for N3. That formalisation allows us to prove that the calculus is correct."*

Knowing that the calculus itself is correct we can trust proofs which list the correct applications of the different proof steps in order to verify a goal. As the SWAP proof vocabulary makes it possible to express these applications directly in N3, a Semantic Web format, such proofs can be easily exchanged and verified between reasoners, but also used in further applications. To emphasize this last aspect that proofs written in N3 can not only help to establish the layer of *Trust*, but also the very important layer of *Applications* we provided examples for the latter. As a use case for N3 proofs we studied automatic composition and execution of hypermedia APIs (Chapter 8). We presented a way to describe the operations such APIs can perform by means of existential rules which then could be combined in proof for a desired goal. As all rules contributing to that goal also appear in such a proof, we can understand that proof as a plan, execute and – if needed – update it. We defined an algorithm to exactly do that and we showed that if the proof generation itself terminates, then our algorithm does as well. We discussed the limits of our approach: by using N3 which is based on FOL we cannot express change, we furthermore cannot provide different options to the user. We concluded that use cases which do not require these two properties can be tackled by our method and presented one instance of such a use case: the localisation of critical sensors in complex set-ups where context is relevant. By discussing these two use cases we showed that there are many applications which can benefit from N3 proofs.

The research we conducted in this doctoral project showed that N3Logic is a promising candidate to become the *Unifying Logic* of the Semantic Web: We were able to propose two definitions for N3's semantics which were mostly compatible with RDF. It depends on the agreement of the community which of these definitions – if any – is chosen. We furthermore showed that N3 can be used to combine *Querying*, reasoning about *Ontologies/Taxonomies*, and rule-based inferencing. However, to properly support querying we needed built-in functions. As N3 is not standardised (yet) there is also no agreement on the built-in functions which are part of the logic. This issue again needs to be decided by the community. N3 furthermore does not only support the layer of proof, by providing the to possibility to cite formulas it even makes such proofs part of the Semantic Web. Proofs can be used for further reasoning, they can be exchanged and they can be used in all kinds of

applications. If the Semantic Web community comes to an agreement about the meaning of implicit universal quantification and the built-ins which form part of the logic, this logic has the potential to become the *Unifying Logic* for the Semantic Web.

9.2 Open challenges and future directions

Having answered the research questions in the previous section, we now take an outlook to the future and discuss the challenges which lie ahead.

In this thesis we only considered the question of N3's potential to become the *Unifying Logic* of the Semantic Web from a practical point of view and had promising results which of course need to be further refined. Next to the practice, we also need to work on the theory: We need to investigate on the formal relationships to other logics and on the properties of N3 itself. This includes for example the relationship of N3 and SPARQL. Here it would be interesting to directly translate SPARQL queries to N3 rules and the other way around. To do so, the relationship between N3 built-ins and the functions defined for the SPARQL filter needs to be taken into account. The former are not yet covered in the formalisation we provided in this book. Also the relationship of N3 and RIF could be further studied. Here it is interesting to note that RIF in contrast to N3 already includes RDF datatypes in its core semantics [47]. Datatype reasoning in N3 could be supported by built-in functions. Another possible direction of future research is the relation of N3 and OWL DL. To better understand how these two play together a joint model theory for N3 and OWL DL could be defined as it has also been done for RIF [49] and it could be investigated how far beyond OWL RL we can go with our approach of using N3 rules to support OWL DL reasoning. As a last theoretical aspect we want to mention decidability. To better understand N3 and its properties and is useful to find subsets of N3 which remain decidable. Next to the research on Datalog^{+-} [157–159], the works related to answer set programming in the Semantic Web context could provide meaningful insides [166–168].

This investigation of N3's theory, can only be performed if an agreement on its meanings is reached and its semantics is formalised. In order to come to such an agreement one important challenge we need to tackle is the standardisation of N3. The research presented in this thesis has shown that N3 is worth that effort. Even though rule-based reasoning is very powerful, it has so far been widely neglected in the Semantic Web in favour of other frameworks like OWL DL which scarify user-friendliness in order to guarantee decidability. Here, we do not say that one framework is better

than the other – there are use cases in which it makes sense to use an OWL DL reasoner instead of an rule-based reasoner – we just say that both frameworks should co-exist and that in cases where reasoning needs to be combined with querying, N3 could be the logic of choice.

We made a first step towards the standardisation of N3 and started a W3C community group[1]. We hope to soon be able to come to agreements in the questions discussed[2] to then provide a formalisation of the logic. The most important points which need to be addressed are the meaning of implicit quantification, the agreement on the built-in functions which form part of the logic – here it is in particular important whether or not we want to include scoped negation as failure – and the meaning of cited formulas. This last topic has not been discussed in detail here in this thesis even though the semantic definition of N3 Core Logic gives a hint how we want to understand cited formulas.

Apart from the standardisation of N3Logic, we also see the need to extend the SWAP proof vocabulary: Reasoners should at least have the option to express all proof steps they perform when deriving new knowledge. Even with a more complex vocabulary the implementers of reasoners can decide to omit proof steps they consider trivial. But the calculus such descriptions represent should be complete for a big part of the logic, if possible, even for the entire logic.

To provide a last suggestion for further research we come back to the limitations of RESTdesc: we saw that there are cases where it makes sense for a logic to have a notion of change. Even though the concept of time and invalid information is already part of the Semantic Web, we do not have any Semantic Web Logic which can deal with that concept and use it – for example – for planning. Of course the solution of that problem requires us to leave our comfort zone of monotonic logics and try something new. But establishing this new direction as an addition to the different logical frameworks we already have we bring the Semantic Web one step further towards its real materialisation.

Looking at the different challenges we tackled in this thesis as well as the ones which lie ahead, we see this thesis as a starting point towards the thorough investigation of N3, its logical properties and its potential to become a *Unifying Logic*. We tackled the practical aspects and made a first but very important step towards the definition of N3's model theoretic semantics. This work forms the base for further investigation to finally answer, whether

[1] https://www.w3.org/community/n3-dev/

[2] The discussions can be found on the group's git repository: https://github.com/w3c/N3/issues/

N3 can be the *Unifying Logic* for the Semantic Web. If this question can be positively answered, it still depends on the take up of the logic by the community, whether N3 will actually establish itself in this role. We hope that with this thesis we provided the arguments to any potential user of this logic to actually apply it and thereby let it contribute towards the full realisation of the Semantic Web.

Appendices

Appendix A

Problems in Cwm

In this section we explain where the Cwm's interpretations of N3 formulas as specified in this paper differ from the actual implementation. We start with a simple example. Consider the formula:

```
{?x :p :o}=>{?x :q :o}.
            :a :b {:c :d {:e :f {?x :g :h}}}.   (A.1)
```

According to the formalisation provided in this paper, the occurrences of the variable `?x` in the implication are universally quantified on its parent$_c$ formula which is the overall formula. This quantification then also counts for the third occurrence of `?x` which is more deeply nested. In core logic this formula can be represented as:

```
∀x.  <x p o> → <x q o>.
                    <a b <c d <e f <x g h>>>>.   (A.1')
```

In this case this is also the result Cwm gives. In Listing 32 we display the reasoning result of Cwm[1] when provided with Formula A.1. The output is produced by using the option `--think` which makes the reasoner derive the deductive closure of its input dataset. During this process Cwm also translates all implicit universal quantification into its explicit counterpart. But if we now change the order of the conjunction to

```
:a :b {:c :d {:e :f {?x :g :h}}}.
                        {?x :p :o}=>{?x :q :o}.   (A.2)
```

[1] Version: v 1.197 2007/12/13 15:38:39 syosi

```
1 @prefix :  <http://example.org/ex#>
2 @prefix c:  <#>
3
4 @forAll c:x .
5 :a :b {
6   :c :d {:e :f {c:x :g :h.}.}.
7 }.
8 {c:x :p :o.} => {c:x :q :o.}.
```

Listing 32: Output of Cwm when provided with Formula A.1.

```
1 @prefix :  <http://example.org/ex#>
2 @prefix c:  <#>
3
4 @forAll c:x .
5 :a :b {
6   :c :d {
7     @forAll c:x.
8     :e :f {c:x :g :h.}.
9   }.
10 }.
11 {c:x :p :o.} => {c:x :q :o.}.
```

Listing 33: Output of Cwm when provided with Formula A.2.

we get a different result. According to the formalisation this formula has the same meaning as the previous one namely Interpretation A.1'. In Listing 33 we display the reasoning output of Cwm for this formula. We clearly see that in this interpretation an extra universal quantifier for the deeply nested occurrence of `?x` is added (line 7). For Cwm the interpretation of the conjunction depends on the order of its conjuncts. Cwm does its translation from implicit to explicit quantification while parsing. Whenever a new universal variable is encountered which is not quantified yet a universal quantifier is added on its parent_c level. This mechanism does not take the later encounter of universal variables on higher levels into account. Such a mechanism would have a negative impact on the performance of Cwm. Similar examples can be easily constructed.

Appendix B

Attributes and Methods for Evaluation

For the evaluation in Section 5.3.4 we defined several attributes. In this section, we display the definition of these attributes and explain how they have been used for the categorisation of different cases.

B.1 Critical Built-in Constructs

To identify the built-in constructs in our datasets which are causing conflicting interpretations we extend the attribute grammar by the two synthesized attributes b and bi. We display the rules for these attributes in Figure B.1.

Attribute b is used to pass the set of universal variables occurring in the subject or object position of a built-in predicate upwards in the syntax tree to the parent$_c$ formula of that built-in. Here, it is most interesting how the attribute behaves for the production rule of a simple triple, i.e. the rule `f := t1t2t3`. In that case we make use of the attributes m_c and u which have been defined in Section 4.4.2 and apply function $f_1 : \mathcal{T} \times 2^U \times 2^U \rightarrow 2^U$ which is defined as follows:

$$f_1(t, s_1, s_2) := \begin{cases} s_2 & \texttt{if } t \text{ is no built-in symbol} \\ s_1 & \text{else} \end{cases}$$

The first argument of the function is the actual value of the predicate which can either be a built-in or not. In the case it is not a built-in, the function simply passes the information collected by the attribute so far upwards. If the predicate is a built-in, the values of the attribute u for subject and object

production rule		rules for b	rules for bi
s ::=	f	$\mathtt{s}.b \leftarrow \emptyset$	$\mathtt{s}.bi \leftarrow \mathtt{f}.bi$
f ::=	$\mathtt{t}_1\mathtt{t}_2\mathtt{t}_3$.	$\mathtt{f}.b \leftarrow f_1(\mathtt{t}_2.m_c, (\mathtt{t}_1.u \cup \mathtt{t}_3.u), (\mathtt{t}_1.b \cup \mathtt{t}_2.b \cup \mathtt{t}_3.b))$	$\mathtt{f}.bi \leftarrow \max(\mathtt{t}_1.bi, \mathtt{t}_2.bi, \mathtt{t}_3.bi)$
	$\mathtt{e}_1$=>$\mathtt{e}_2$.	$\mathtt{f}.b \leftarrow \mathtt{e}_1.b \cup \mathtt{e}_2.b$	$\mathtt{f}.bi \leftarrow \max(\mathtt{e}_1.bi, \mathtt{e}_2.bi)$
	$\mathtt{f}_1\mathtt{f}_2$	$\mathtt{f}.b \leftarrow \mathtt{f}_1.b \cup \mathtt{f}_2.b$	$\mathtt{f}.bi \leftarrow \max(\mathtt{f}_1.bi, \mathtt{f}_2.bi)$
t ::=	uv	$\mathtt{t}.b \leftarrow \emptyset$	$\mathtt{t}.bi \leftarrow 0$
	ex	$\mathtt{t}.b \leftarrow \emptyset$	$\mathtt{t}.bi \leftarrow 0$
	c	$\mathtt{t}.b \leftarrow \emptyset$	$\mathtt{t}.bi \leftarrow 0$
	e	$\mathtt{t}.b \leftarrow \mathtt{e}.b$	$\mathtt{t}.bi \leftarrow \mathtt{e}.bi$
	(k)	$\mathtt{t}.b \leftarrow \mathtt{k}.b$	$\mathtt{t}.bi \leftarrow \mathtt{k}.bi$
	()	$\mathtt{t}.b \leftarrow \emptyset$	$\mathtt{t}.bi \leftarrow 0$
k ::=	t	$\mathtt{k}.b \leftarrow \mathtt{t}.b$	$\mathtt{k}.bi \leftarrow \mathtt{t}.bi$
	t $\mathtt{k}_1$	$\mathtt{k}.b \leftarrow \mathtt{t}.b \cup \mathtt{k}_1.b$	$\mathtt{k}.bi \leftarrow \max(\mathtt{t}.bi, \mathtt{k}_1.bi)$
e ::=	{f}	$\mathtt{e}.b \leftarrow \emptyset$	$\mathtt{e}.bi \leftarrow f_2(\mathtt{f}.bi, (\mathtt{f}.b \setminus \mathtt{e}.s))$
	{}	$\mathtt{e}.b \leftarrow \emptyset$	$\mathtt{e}.bi \leftarrow 0$
	false	$\mathtt{e}.b \leftarrow \emptyset$	$\mathtt{e}.bi \leftarrow 0$

Figure B.1: Attribute rules for the synthesized attributes b (middle) and bi (right), and their corresponding production rules (left) from the N3 grammar (Figure 4.2). The attributes test whether a formula contains built-ins whose subject or object has universals which are not quantified on the parent or any higher level of the built-in.

get collected, i.e. all universal variables which occur in these two positions. For the other production rules the attribute value is either the empty set – in case we have a rule resulting in a terminal node of the tree – or it gets passed upwards through the tree. The only exception is the production rule e ::= {f} for which the attribute value of e is the empty set since f is a parent_c formula. For this formula the attribute value $\mathtt{e}.s$ with s defined as in Section 4.4.2 gives information which variables are already quantified on a higher level. This is used in the definition of attribute bi.

Attribute bi carries the information whether or not a built-in predicate has been found whose subject or object contains one or more universal variable which Cwm's interpretation quantifies either on the same level the built-in function occurs or any lower level. If this is the case, the value of the attribute is 1, else it is 0. If we look at Figure B.1 we see that the rules for this attribute mostly simply pass information upwards with one exception: For the production rule e ::= {f} we apply the function $f_2 : \mathbb{N} \times 2^{U \times U} \rightarrow \mathbb{N}$ which is defined as follows:

$$f_2(n, s) := \begin{cases} 0 & \text{if } n = 0 \text{ and } s = \emptyset \\ 1 & \text{else} \end{cases}$$

To better understand this definition, we take a closer look to the arguments the function is used with in the attribute definition: The first argument is simply used to pass the information that a problematic built-in construct has been found upwards in the syntax tree. So, if its value is 1, the application of function f_2 should also result in 1 (second case). As the second argument, we have the difference of $\mathtt{f}.b$ – the variables we collected using attribute b – and $\mathtt{e}.s$ – the variables which are quantified on the parent$_c$ level of the formula. If this difference is not empty, this means that we found a variable which occurs in the subject or object position of a built-in (collected by b) which Cwm quantifies either on the same level as the built-in or on a lower level. These are the constructs which are problematic in our consideration. In these cases function f_2 will result in 1, else its value is 0.

B.2 Nested Universals

The reasons for conflicting interpretations of a formula can be overlapping. Especially for proofs we want to know whether conflicts are only caused by the proof vocabulary which normally uses graphs – these cases are rather harmless – or if either a built-in construct in combination with a formula expression containing a universal variable which is quantified within this expression or some other occurrence of a universal variable quantified on level nested inside the result of a proof step is causing the disagreement of interpretations.

For built-ins we use the attributes defined in B.1. To find deeply nested universals we introduce the concept of depth: we measure the depth of a universal variable as one plus the number of formula expressions its quantifier is enclosed in when translating the N3 formula to its core logic translation according to Cwm. If a variable does not occur in a formula at all, it has depth 0. As an example consider Formula 4.3 from Section 4.4.2:

```
{{?x :q ?y.} => {?x :r :c.}.} =>{?x :p :a.}.   (4.3)
```

Here, variable `?z` has depth 0 – it does not occur in the formula – variables `?y` and `?x` have depths 2 and 1, respectively.

In order to determine the depth of the deepest nested variable in a formula we define two attributes: the inherited attribute c and the synthesized attribute d. The first attribute c simply goes downwards in the syntax tree and counts every formula expression it encounters on the way down. To understand attribute d we first take a closer look to the formula involved, $f_3 : 2^U \times \mathbb{N} \times$

production rule		rules for d	rules for c
s ::=	f	$\mathtt{s}.d \leftarrow f_3(\mathtt{f}.q, 1, \mathtt{f}.d)$	$\mathtt{f}.c \leftarrow 1$
f ::=	$\mathtt{t}_1\mathtt{t}_2\mathtt{t}_3$.	$\mathtt{f}.d \leftarrow \max_i \mathtt{t}_i.d$	$\mathtt{t}_i.c \leftarrow \mathtt{f}.c$
	$\mathtt{e}_1$=>$\mathtt{e}_2$.	$\mathtt{f}.d \leftarrow \max_i \mathtt{e}_i.d$	$\mathtt{e}_i.c \leftarrow \mathtt{f}.c$
	$\mathtt{f}_1\mathtt{f}_2$	$\mathtt{f}.d \leftarrow \max_i \mathtt{f}_i.d$	$\mathtt{f}_i.c \leftarrow \mathtt{f}.c$
t ::=	uv	$\mathtt{t}.d \leftarrow 0$	
	ex	$\mathtt{t}.d \leftarrow 0$	
	c	$\mathtt{t}.d \leftarrow 0$	
	e	$\mathtt{t}.d \leftarrow \mathtt{e}.d$	$\mathtt{e}.c \leftarrow \mathtt{t}.c$
	(k)	$\mathtt{t}.d \leftarrow \mathtt{k}.d$	$\mathtt{k}.c \leftarrow \mathtt{t}.c$
	()	$\mathtt{t}.d \leftarrow 0$	
k ::=	t	$\mathtt{k}.d \leftarrow \mathtt{t}.d$	$\mathtt{t}.c \leftarrow \mathtt{k}.c$
	t $\mathtt{k}_1$	$\mathtt{k}.d \leftarrow \max(\mathtt{t}.d, \mathtt{k}_1.d)$	$\mathtt{t}.c, \mathtt{k}.c \leftarrow \mathtt{k}.c$
e ::=	{f}	$\mathtt{e}.d \leftarrow f_3(\mathtt{f}.q, \mathtt{f}.c, \mathtt{f}.d)$	$\mathtt{f}.c \leftarrow \mathtt{e}.c + 1$
	{}	$\mathtt{e}.d \leftarrow 0$	
	false	$\mathtt{e}.d \leftarrow 0$	

Figure B.2: Attribute rules for the synthesized attribute d (middle) and the inherited c (right), and their corresponding production rules (left) from the N3 grammar (Figure 4.2). The attributes keep track of the deepest nested universal in a formula.

$\mathbb{N} \to \mathbb{N}$ is defined as follows:

$$f_3(s, n, m) := \begin{cases} m & \text{if } s = \emptyset \\ \max(n, m) & \text{else} \end{cases}$$

Attribute d now takes this formula to check for every formula expression and for the top formula whether according to Cwm's translation there is at least one quantifier for universal variables at that level (remember that the set of universals quantified at each level is captured by attribute q). If so, it takes the depth of the current formula which is captured by attribute c and compares it with the values for d already found in the depending nodes. From these two values it takes the maximum. If there are no universal quantifiers at a node, the attribute only passes the maximum value for d of the depending nodes upwards. The value $\mathtt{s}.d$ is now the depth of the deepest nested universal in the formula.

Because of the nature of the proof we know that if the maximum depth of the universal quantifiers is two or lower, then the only reason for conflicting

interpretations is the use of the predicate `r:gives` which has a graph as object. If the maximum depth of universal variables is bigger than two, we know that the conflict found in the proof is more serious and caused by a deeper nesting in one of the formulas occurring in the object of `r:gives`.

Bibliography

[1] T. R. Gruber, A translation approach to portable ontology specifications, Knowledge acquisition 5 (2) (1993) 199–220.

[2] T. Berners-Lee, J. Hendler, O. Lassila, The Semantic Web, Scientific American 284 (5) (2001) 34–43.
URL http://www.jstor.org/stable/26059207

[3] T. Berners-Lee, R. Cailliau, J.-F. Groff, The world-wide web, Computer Networks and ISDN Systems 25 (4–5) (1992) 454–459.
URL http://www.sciencedirect.com/science/article/pii/016975529290039S

[4] R. Cyganiak, D. Wood, M. Lanthaler, RDF 1.1: Concepts and Abstract Syntax, W3C Recommendation, http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (Feb. 2014).

[5] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf, J. Hendler, N3Logic: A logical framework for the World Wide Web, Theory and Practice of Logic Programming 8 (3) (2008) 249–269. doi:http://dx.doi.org/10.1017/S1471068407003213.

[6] D. Arndt, T. Schrijvers, J. De Roo, R. Verborgh, Implicit quantification made explicit: How to interpret blank nodes and universal variables in Notation3 Logic, Journal of Web Semantics 58 (2019) 100501. doi:10.1016/j.websem.2019.04.001.
URL https://authors.elsevier.com/c/1ZWg55bAYUaMkH

[7] D. Arndt, R. Verborgh, J. De Roo, H. Sun, E. Mannens, R. Van de Walle, Semantics of Notation3 logic: A solution for implicit quantification, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 127–143.

URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_9

[8] D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle, E. Mannens, Ontology reasoning using rules in an eHealth context, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Vol. 9202 of Lecture Notes in Computer Science, Springer, 2015, pp. 465–472.
URL http://link.springer.com/chapter/10.1007/978-3-319-21542-6_31

[9] D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle, E. Mannens, Improving OWL RL reasoning in N3 by using specialized rules, in: V. Tamma, M. Dragoni, R. Gonçalves, A. Ławrynowicz (Eds.), Ontology Engineering: 12th International Experiences and Directions Workshop on OWL, Vol. 9557 of Lecture Notes in Computer Science, Springer, 2016, pp. 93–104. doi:10.1007/978-3-319-33245-1_10.
URL http://dx.doi.org/10.1007/978-3-319-33245-1_10

[10] D. Arndt, B. De Meester, A. Dimou, R. Verborgh, E. Mannens, Using rule-based reasoning for RDF validation, in: S. Costantini, E. Franconi, W. Van Woensel, R. Kontchakov, F. Sadri, D. Roman (Eds.), Proceedings of the International Joint Conference on Rules and Reasoning, Vol. 10364 of Lecture Notes in Computer Science, Springer, 2017, pp. 22–36. doi:10.1007/978-3-319-61252-2_3.

[11] B. De Meester, P. Heyvaert, D. Arndt, A. Dimou, R. Verborgh, RDF graph validation using rule-based reasoning, submitted to: Semantic Web Journal.

[12] R. Verborgh, D. Arndt, S. Van Hoecke, J. De Roo, G. Mels, T. Steiner, J. Gabarró Vallés, The pragmatic proof: Hypermedia API composition and execution, Theory and Practice of Logic Programming 17 (1) (2017) 1–48. doi:10.1017/S1471068416000016.
URL http://arxiv.org/pdf/1512.07780v1.pdf

[13] D. Arndt, P. Bonte, A. Dejonghe, R. Verborgh, F. De Turck, F. Ongenae, SENSdesc: connect sensor queries and context, in: R. Zwiggelaar, H. Gamboa, S. Fred, Ana Bermúdez i Badia (Eds.), Proceedings of the 11th International Joint Conference on Biomedical Engineering

Systems and Technologies, SCITEPRESS, 2018, pp. 671–679. doi:10.5220/0006733106710679.
URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006733106710679

[14] A. Gerber, A. van der Merwe, A. Barnard, A functional semantic web architecture, in: S. Bechhofer, M. Hauswirth, J. Hoffmann, M. Koubarakis (Eds.), The Semantic Web: Research and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 273–287.

[15] S. Harris, A. Seaborne, SPARQL 1.1 Query Language, w3c Recommendation, http://www.w3.org/TR/sparql11-query/ (Mar. 2013).

[16] I. Horrocks, B. Parsia, P. Patel-Schneider, J. Hendler, Semantic web architecture: Stack or two towers?, in: F. Fages, S. Soliman (Eds.), Principles and Practice of Semantic Web Reasoning, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 37–41.

[17] M. Kifer, J. de Bruijn, H. Boley, D. Fensel, A realistic architecture for the semantic web, in: A. Adi, S. Stoutenburg, S. Tabet (Eds.), Rules and Rule Markup Languages for the Semantic Web: First International Conference, RuleML 2005, Galway, Ireland, November 10-12, 2005. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 17–29. doi:10.1007/11580072_3.
URL http://dx.doi.org/10.1007/11580072_3

[18] A. Hogan, Linked data and the semantic web standards, in: Linked Data and the Semantic Web Standards, Chapman and Hall/CRC Press, 2013.

[19] T. Berners-Lee, R. Fielding, L. Masinter, Uniform Resource Identifier (URI): Generic Syntax, https://www.ietf.org/rfc/rfc3986.txt (Jan. 2005).

[20] M. Duerst, M. Suignard, Internationalized Resource Identifiers (IRIs), http://www.ietf.org/rfc/rfc3987.txt (Jan. 2005).

[21] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, J. Cowan, Extensible Markup Language (XML) 1.1 (Second Edition), w3c Recommendation, https://www.w3.org/TR/xml11/ (Aug. 2006).

[22] Javascript object notation (json), https://json.org/.

[23] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, G. Carothers, Turtle - Terse RDF Triple Language, W3C Recommendation, http://www.w3.org/TR/turtle/ (Feb. 2014).

[24] D. Brickley, R. V. Guha, RDF Schema 1.1, W3C Recommendation, http://www.w3.org/TR/rdf-schema/ (Feb. 2014).

[25] W3C OWL Working Group, OWL 2 Web Ontology Language, W3C Recommendation, https://www.w3.org/TR/owl2-overview/ (Dec. 2012).

[26] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, New York, NY, USA, 2003.

[27] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission, https://www.w3.org/Submission/SWRL/ (May 2004).

[28] M. Kifer, Rule interchange format: The framework, in: D. Calvanese, G. Lausen (Eds.), RR 2008: Web Reasoning and Rule Systems, Vol. 5341 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 1–11.

[29] R. L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, Communications of the ACM 21 (2) (1978) 120–126.

[30] E. Rescorla, HTTP Over TLS, https://www.ietf.org/rfc/rfc2818.txt (May 2000).

[31] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, N. Lindström, A json-based serialization for linked data, W3C Recommendation, https://www.w3.org/TR/json-ld/ (Jan. 2014).

[32] A. Abele, P. Buitelaar, R. Cyganiak, A. Jentzsch, V. Andryushechkin, State of the LOD cloud, https://lod-cloud.net/ (2018).

[33] D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee, E. Simperl (Eds.), Proceedings of the 17th International Semantic Web Conference, Vol. 11137 of Lecture Notes in Computer Science, Springer, 2018.

[34] A. Gangemi, R. Navigli, M.-E. Vidal, P. Hitzler, R. Troncy, L. Hollink, A. Tordai, M. Alam (Eds.), Proceedings of the 15th ESWC, Vol. 10843 of Lecture Notes in Computer Science, Springer, 2018.

[35] P. J. Hayes, P. F. Patel-Schneider, RDF 1.1 Semantics, W3C Recommendation, http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/ (Feb. 2014).

[36] H. Knublauch, J. Hendler, K. Idehen, SPIN - Overview and Motivation, W3C Member Submission, https://www.w3.org/Submission/spin-overview/ (Feb. 2011).

[37] M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein, OWL Web Ontology Language, W3C Recommendation, https://www.w3.org/TR/owl-ref/ (Feb. 2004).

[38] P. F. Patel-Schneider, B. Motik, B. C. Grau, I. Horrocks, B. Parsia, A. Ruttenberg, M. Schneider, Owl 2 web ontology language mapping to rdf graphs (second edition), W3C Recommendation, https://www.w3.org/TR/owl-mapping-to-rdf/ (Dec. 2012).

[39] I. Horrocks, B. Parsia, U. Sattler, B. Motik, P. F. Patel-Schneider, B. Cuenca Grau, Owl 2 web ontology language direct semantics (second edition), W3C Recommendation, https://www.w3.org/TR/owl2-direct-semantics/ (Dec. 2012).

[40] B. Motik, P. F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, M. Smith, Owl 2 web ontology language structural specification and functional-style syntax (second edition), W3C Recommendation, https://www.w3.org/TR/owl-syntax/ (Dec. 2012).

[41] D. Calvanese, J. Carroll, G. Di Giacomo, J. Hendler, I. Herman, B. Parsia, P. F. Patel-Schneider, A. Ruttenberg, U. Sattler, M. Schneider, OWL 2 Web Ontology Language Profiles (second edition), W3C Recommendation, www.w3.org/TR/owl2-profiles/ (Dec. 2012).

[42] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, Pellet: A practical owl-dl reasoner, Journal of Web Semantics 5 (2) (2007) 51–53. doi:10.1016/j.websem.2007.03.004.
URL http://dx.doi.org/10.1016/j.websem.2007.03.004

[43] B. Motik, R. Shearer, I. Horrocks, Hypertableau Reasoning for Description Logics, Journal of Artificial Intelligence Research 36 (2009) 165–228.

[44] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, J. Banerjee, Rdfox: A highly-scalable rdf store, in: M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, S. Staab (Eds.), The Semantic Web - ISWC 2015, Springer International Publishing, Cham, 2015, pp. 3–20.

[45] J. Carroll, I. Herman, P. F. Patel-Schneider, M. Schneider, Owl 2 web ontology language rdf-based semantics (second edition), w3c Recommendation, https://www.w3.org/TR/owl2-rdf-based-semantics/ (Dec. 2012).

[46] C. de Sainte Marie, G. Hallmark, A. Paschke, Rif production rule dialect (second edition), w3c Recommendation, https://www.w3.org/TR/rif-prd/ (Feb. 2013).

[47] H. Boley, G. Hallmark, M. Kifer, A. Paschke, A. Polleres, D. Reynolds, Rif core dialect (second edition), w3c Recommendation, https://www.w3.org/TR/rif-core/ (Feb. 2013).

[48] S. Hawke, A. Polleres, Rif in rdf (second edition), w3c Recommendation, https://www.w3.org/TR/rif-in-rdf/ (Feb. 2013).

[49] J. de Bruijn, C. Welty, Rif rdf and owl compatibility (second edition), w3c Recommendation, https://www.w3.org/TR/rif-rdf-owl/ (Feb. 2013).

[50] B. N. Grosof, I. Horrocks, R. Volz, S. Decker, Description logic programs: Combining logic programs with description logic, in: Proceedings of the 12th International Conference on World Wide Web, WWW '03, ACM, New York, NY, USA, 2003, pp. 48–57. doi:10.1145/775152.775160.
URL http://doi.acm.org/10.1145/775152.775160

[51] M. Knorr, P. Hitzler, F. Maier, Reconciling owl and non-monotonic rules for the semantic web, in: L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, P. J. F. Lucas (Eds.), ECAI 2012 - 20th European Conference on Artificial Intelligence., Vol. 242, IOS Press, IOS Press, Montpellier, France, 2012, pp. 474–479. doi:10.3233/978-1-61499-098-7-474.
URL http://dx.doi.org/10.3233/978-1-61499-098-7-474

[52] M. Krötzsch, F. Maier, A. Krisnadhi, P. Hitzler, A better uncle for owl: Nominal schemas for integrating rules and ontologies, in: Proceedings of the 20th International Conference on World Wide Web, WWW '11, ACM, New York, NY, USA, 2011, pp. 645–654. doi:10.1145/1963405.1963496.
URL http://doi.acm.org/10.1145/1963405.1963496

[53] B. Motik, R. Rosati, Reconciling description logics and rules, J. ACM 57 (5) (2008) 30:1–30:62. doi:10.1145/1754399.1754403.
URL http://doi.acm.org/10.1145/1754399.1754403

[54] A. Krisnadhi, F. Maier, P. Hitzler, OWL and Rules, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 382–415. doi:10.1007/978-3-642-23032-5_7.
URL https://doi.org/10.1007/978-3-642-23032-5_7

[55] A. Polleres, C. Feier, A. Harth, Rules with Contextually Scoped Negation, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 332–347. doi:10.1007/11762256_26.
URL http://dx.doi.org/10.1007/11762256_26

[56] C. V. Damásio, A. Analyti, G. Antoniou, G. Wagner, Supporting Open and Closed World Reasoning on the Web, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 149–163. doi:10.1007/11853107_11.
URL http://dx.doi.org/10.1007/11853107_11

[57] R. Rosati, Dl+log: Tight integration of description logics and disjunctive datalog, in: Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning, KR'06, AAAI Press, 2006, pp. 68–78.
URL http://dl.acm.org/citation.cfm?id=3029947.3029960

[58] I. Horrocks, P. F. Patel-Schneider, A proposal for an owl rules language, in: Proceedings of the 13th International Conference on World Wide Web, WWW '04, ACM, New York, NY, USA, 2004, pp. 723–731. doi:10.1145/988672.988771.
URL http://doi.acm.org/10.1145/988672.988771

[59] R. Angles, C. Gutierrez, The expressive power of sparql, in: A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin, K. Thirunarayan (Eds.), The Semantic Web - ISWC 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 114–129.

[60] A. Polleres, J. P. Wallner, On the relation between sparql1.1 and answer set programming, Journal of Applied Non-Classical Logics 23 (1-2) (2013) 159–212. arXiv:https://doi.org/10.1080/11663081.2013.798992, doi:10.1080/11663081.2013.798992.
URL https://doi.org/10.1080/11663081.2013.798992

[61] Kaon2, http://kaon2.semanticweb.org/.

[62] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, R. Velkov, Owlim: A family of scalable semantic repositories., Semantic Web 2 (1) (2011) 33–42.

[63] B. Parsia, E. Sirin, B. C. Grau, E. Ruckhaus, D. Hewlett, Cautiously approaching swrl, Technical report, University of Maryland (2005) 1–30.

[64] U. Hustadt, B. Motik, U. Sattler, Reducing shiq-description logic to disjunctive datalog programs, KR 4 (2004) 152–162.

[65] G. Gottlob, A. Pieris, Beyond sparql under owl 2 ql entailment regime: Rules to the rescue, in: Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.

[66] B. Bishop, S. Bojanov, Implementing owl 2 rl and owl 2 ql rule-sets for owlim., in: OWLED, Vol. 796, 2011.

[67] B. Motik, U. Sattler, R. Studer, Query answering for owl-dl with rules, Web Semant. 3 (1) (2005) 41–60. doi:10.1016/j.websem.2005.05.001.
URL http://dx.doi.org/10.1016/j.websem.2005.05.001

[68] J. De Bruijn, T. Eiter, A. Polleres, H. Tompits, On representational issues about combinations of classical theories with nonmonotonic rules, in: International Conference on Knowledge Science, Engineering and Management, Springer, 2006, pp. 1–22.

[69] J. De Bruijn, E. Franconi, S. Tessaris, Logical reconstruction of normative rdf, in: OWL: Experiences and Directions Workshop (OWLED-2005), Galway, Ireland, 2005.

[70] F. Gandon, G. Schreiber, RDF 1.1 XML Syntax, w3c Recommendation, https://www.w3.org/TR/rdf-syntax-grammar/ (Feb. 2014).

[71] T. Berners-Lee, D. Connolly, Notation3 (N3): A readable RDF syntax, W3C Team Submission, http://www.w3.org/TeamSubmission/n3/ (Mar. 2011).

[72] T. Berners-Lee, Semantic Web Application Platform, http://www.w3.org/2000/10/swap/ (2000).

[73] H. B. Enderton, A Mathematical Introduction to Logic (Second Edition), 2nd Edition, Academic Press, Boston, 2001. doi:https://doi.org/10.1016/B978-0-08-049646-7.50005-9.
URL http://www.sciencedirect.com/science/article/pii/B9780080496467500059

[74] E. Mendelson, Introduction to Mathematical Logic, 5th Edition, Chapman & Hall/CRC, 2009.

[75] H.-D. Ebbinghaus, J. Flum, W. Thomas, Einführung in die mathematische Logik (5. Aufl.), Spektrum Akademischer Verlag, 2007.

[76] T. Berners-Lee, *cwm*, http://www.w3.org/2000/10/swap/doc/cwm.html (2000–2009).

[77] FuXi 1.4: A Python-based, bi-directional logical reasoning system for the semantic web, http://code.google.com/p/fuxi/.

[78] G. Klyne, swish: A semantic web toolkit., http://hackage.haskell.org/package/swish (2003–2018).

[79] J. De Roo, Euler yet another proof engine, http://eulersharp.sourceforge.net/ (1999–2019).

[80] H. Sun, K. Depraetere, J. D. Roo, G. Mels, B. D. Vloed, M. Twagirumukiza, D. Colaert, Semantic processing of ehr data for clinical research, Journal of Biomedical Informatics 58 (2015) 247 – 259. doi:https://doi.org/10.1016/j.jbi.2015.10.009.
URL http://www.sciencedirect.com/science/article/pii/S1532046415002312

[81] N. Douali, E. I. Papageorgiou, J. D. Roo, M. Jaulent, Case based fuzzy cognitive maps (cbfcm): New method for medical reasoning: Comparison study between cbfcm/fcm, in: 2011 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2011), 2011, pp. 844–850. doi:10.1109/FUZZY.2011.6007710.

[82] P. Pauwels, D. V. Deursen, R. Verstraeten, J. D. Roo, R. D. Meyer, R. V. de Walle, J. V. Campenhout, A semantic rule checking

environment for building performance checking, Automation in Construction 20 (5) (2011) 506 – 518. doi:https://doi.org/10.1016/j.autcon.2010.11.017.
URL http://www.sciencedirect.com/science/article/pii/S0926580510001962

[83] S. Mayer, R. Verborgh, M. Kovatsch, F. Mattern, Smart configuration of smart environments, IEEE Transactions on Automation Science and Engineering 13 (3) (2016) 1247–1255. doi:10.1109/TASE.2016.2533321.

[84] R. Kontchakov, M. Rodríguez-Muro, M. Zakharyaschev, Ontology-Based Data Access with Databases: A Short Course, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 194–229. doi:10.1007/978-3-642-39784-4_5.
URL https://doi.org/10.1007/978-3-642-39784-4_5

[85] T. Berners-Lee, Notation 3 logic, http://www.w3.org/DesignIssues/N3Logic (2005).

[86] D. Brickley, R. V. Guha, RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation, http://www.w3.org/2000/10/swap/reason# (Feb. 2004).

[87] A. Zimmermann, RDF 1.1: On Semantics of RDF Datasets, W3C Working Group Note, http://www.w3.org/TR/2014/NOTE-rdf11-datasets-20140225/ (Feb. 2014).

[88] R. Verborgh, J. De Roo, Drawing conclusions from Linked Data on the Web, IEEE Software 32 (5) (2015) 23–27.
URL http://online.qmags.com/ISW0515?cid=3244717&eid=19361&pg=25

[89] W. F. Clocksin, C. S. Mellish, Programming in PROLOG, Springer, 1994.

[90] U. Nilsson, J. Małuszyński, Logic, programming and Prolog, Wiley Chichester, 1990.

[91] G. L. Steele, Jr., Common LISP: The Language (2Nd Ed.), Digital Press, Newton, MA, USA, 1990.

[92] C. Bizer, R. Cygniak, RDF 1.1 TriG, W3C Recommendation, http://www.w3.org/TR/trig/ (Feb. 2014).

[93] D. Arndt, W. Van Woensel, Towards supporting multiple semantics of named graphs using n3 rules, in: Proceedings of the 13th RuleML+RR

2019 Doctoral Consortium and Rule Challenge, Vol. 2438 of CEUR Workshop Proceedings, 2019.
URL http://ceur-ws.org/Vol-2438/paper6.pdf

[94] O. Hartig, B. Thompson, Foundations of an alternative approach to reification in RDF, CoRR abs/1406.3399. arXiv:1406.3399.
URL http://arxiv.org/abs/1406.3399

[95] O. Hartig, Rdf* and sparql*: An alternative approach to annotate statements in RDF, in: Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 23rd - to - 25th, 2017., 2017.
URL http://ceur-ws.org/Vol-1963/paper593.pdf

[96] V. Nguyen, O. Bodenreider, A. Sheth, Don't like rdf reification?: making statements about statements using singleton property, in: Proceedings of the 23rd international conference on World wide web, ACM, 2014, pp. 759–770.

[97] A. Hogan, M. Arenas, A. Mallea, A. Polleres, Everything you always wanted to know about blank nodes, Web Semantics: Science, Services and Agents on the World Wide Web 27 (2014) 42–69.

[98] L. Henkin, Completeness in the theory of types, Journal of Symbolic Logic 15 (2) (1950) 81–91. doi:10.2307/2266967.

[99] D. E. Knuth, Semantics of context-free languages, Mathematical systems theory 2 (2) (1968) 127–145. doi:10.1007/BF01692511.
URL http://dx.doi.org/10.1007/BF01692511

[100] D. E. Knuth, Semantics of context-free languages: Correction, Mathematical systems theory 5 (2) (1971) 95–96. doi:10.1007/BF01702865.
URL http://dx.doi.org/10.1007/BF01702865

[101] J. Paakki, Attribute grammar paradigms—a high-level methodology in language implementation, ACM Comput. Surv. 27 (2) (1995) 196–255. doi:10.1145/210376.197409.
URL http://doi.acm.org/10.1145/210376.197409

[102] M. Genesereth, E. Kao, The herbrand manifesto, in: N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, D. Roman (Eds.), Rule Technologies: Foundations, Tools, and Applications, Springer International Publishing, Cham, 2015, pp. 3–12.

[103] M. Genesereth, E. Kao, Herbrand Semantics, http://logic.stanford.edu/herbrand/herbrand.html (2015).

[104] M. R. Genesereth, R. E. Fikes, et al., Knowledge interchange format-version 3.0: reference manual (1992) 1–68.

[105] ISO/IEC 24707:2007 Information technology – Common Logic (CL), standards.iso.org/ittf/PubliclyAvailableStandards/c039175_ISO_IEC_24707_2007%28E%29.zip (2007).

[106] J. McCarthy, Notes on formalizing context, in: Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1, IJCAI'93, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, pp. 555–560.
URL http://dl.acm.org/citation.cfm?id=1624025.1624103

[107] B. C. Pierce, Types and Programming Languages, 1st Edition, The MIT Press, 2002.

[108] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, K. Donnelly, System f with type equality coercions, in: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, TLDI '07, ACM, New York, NY, USA, 2007, pp. 53–66. doi:10.1145/1190315.1190324.
URL http://doi.acm.org/10.1145/1190315.1190324

[109] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight java: A minimal core calculus for java and gj, ACM Trans. Program. Lang. Syst. 23 (3) (2001) 396–450. doi:10.1145/503502.503505.
URL http://doi.acm.org/10.1145/503502.503505

[110] J. de Bruijn, S. Heymans, Logical foundations of (e)rdf(s): Complexity and reasoning, in: K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi, G. Schreiber, P. Cudré-Mauroux (Eds.), The Semantic Web: 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 86–99. doi:10.1007/978-3-540-76298-0_7.
URL https://doi.org/10.1007/978-3-540-76298-0_7

[111] S. D. Swierstra, P. R. A. Alcocer, J. Saraiva, Designing and implementing combinator languages, in: International School on Advanced Functional Programming, Springer, 1998, pp. 150–206.

[112] D. Arndt, J. Van Herwegen, R. Verborgh, E. Mannens, R. Van de Walle, Using rules to generate and execute workflows in smart factories, in: Proceedings of the RuleML 2016 Challenge, Doctoral Consortium and Industry Track hosted by the 10th International Web Rule Symposium, Vol. 1620 of CEUR Workshop Proceedings, 2016.
URL http://ceur-ws.org/Vol-1620/paper12.pdf

[113] F. Ongenae, L. Bleumers, N. Sulmon, M. Verstraete, M. Van Gils, A. Jacobs, S. De Zutter, P. Verhoeve, A. Ackaert, F. De Turck, Participatory design of a continuous care ontology.

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

[115] Z. Wu, G. Eadon, S. Das, E. I. Chong, V. Kolovski, M. Annamalai, J. Srinivasan, Implementing an inference engine for RDFS/OWL constructs and user-defined rules in Oracle, in: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, IEEE, 2008, pp. 1239–1248.

[116] J. De Roo, EYE and OWL 2, http://eulersharp.sourceforge.net/2003/03swap/eye-owl2.html (1999–2019).

[117] F. Ongenae, P. Duysburgh, N. Sulmon, M. Verstraete, L. Bleumers, S. De Zutter, S. Verstichel, A. Ackaert, A. Jacobs, F. De Turck, An ontology co-design method for the co-creation of a continuous care ontology, Applied Ontology 9 (1) (2014) 27–64.
URL http://dx.doi.org/10.3233/AO-140131

[118] J.-F. Baget, A. Gutierrez, M. Leclère, M.-L. Mugnier, S. Rocher, C. Sipieter, Datalog+, RuleML and OWL 2: Formats and Translations for Existential Rules, in: RuleML: Web Rule Symposium, Berlin, Germany, 2015.
URL https://hal.archives-ouvertes.fr/hal-01172069

[119] L. Rietveld, W. Beek, S. Schlobach, LOD lab: Experiments at LOD scale, in: International Semantic Web Conference, Springer, 2015, pp. 339–355.

[120] A. Zaveri, A. Rula, A. Maurino, R. Pietrobon, J. Lehmann, S. Auer, Quality assessment for linked data: A survey, Semantic Web 7 (1) (2015) 63–93. doi:10.3233/SW-150175.
URL http://www.semantic-web-journal.net/system/files/swj556.pdf

[121] M. Ketterl, L. Knipping, N. Ludwig, R. Mertens, J. Waitelonis, N. Ludwig, M. Knuth, H. Sack, Whoknows? evaluating linked data heuristics with a quiz that cleans up dbpedia, Interactive Technology and Smart Education 8 (4) (2011) 236–248.

[122] H. Solbrig, E. Prud'hommeaux, Shape expressions 1.0 definition. w3c member submission, World Wide Web Consortium, June.

[123] A. Ryman, Resource shape 2.0. w3c member submission, World Wide Web Consortium, Feb.

[124] T. Hartmann, Validation framework for rdf-based constraint languages, Ph.D. thesis, Dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2016 (2016).

[125] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, A. Zaveri, Test-driven evaluation of linked data quality, in: Proceedings of the 23rd international conference on World Wide Web, ACM, 2014, pp. 747–758.

[126] J. Tao, Integrity constraints for the semantic web: an owl 2 dl extension, Ph.D. thesis, Rensselaer Polytechnic Institute (2012).

[127] M. Nilsson, Description set profiles: A constraint language for dublin core application profiles, DCMI Working Draft.

[128] H. Knublauch, D. Kontokostas, Shapes constraint language (shacl), Tech. rep., W3C, accessed March 3rd, 2017 (2017).
URL https://www.w3.org/TR/shacl/

[129] T. Bosch, A. Nolle, E. Acar, K. Eckert, Rdf validation requirements-evaluation and logical underpinning, arXiv preprint arXiv:1501.03933.

[130] T. Bosch, E. Acar, A. Nolle, K. Eckert, The role of reasoning for rdf validation, in: Proceedings of the 11th International Conference on Semantic Systems, SEMANTICS '15, ACM, New York, NY, USA, 2015, pp. 33–40. doi:10.1145/2814864.2814867.
URL http://doi.acm.org/10.1145/2814864.2814867

[131] H. Knublauch, J. A. Hendler, K. Idehen, SPIN – overview and motivation, Tech. rep., W3C, accessed April 18th, 2016 (Feb. 2011).
URL https://www.w3.org/Submission/2011/SUBM-spin-overview-20110222/

[132] B. Motik, I. Horrocks, U. Sattler, Adding integrity constraints to owl., in: OWLED, Vol. 258, 2007.

[133] E. Sirin, J. Tao, Towards integrity constraints in owl, in: Proceedings of the 6th International Conference on OWL: Experiences and Directions - Volume 529, OWLED'09, CEUR-WS.org, Aachen, Germany, Germany, 2009, pp. 79–88.
URL http://dl.acm.org/citation.cfm?id=2890046.2890055

[134] D. Kontokostas, C. Mader, C. Dirschl, K. Eck, M. Leuthold, J. Lehmann, S. Hellmann, Semantically enhanced quality assurance in the jurion business use case, in: 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 2016, 2016, pp. 661–676. doi:10.1007/978-3-319-34129-3_40.
URL http://svn.aksw.org/papers/2016/ESWC_Jurion/public.pdf

[135] M. Kifer, Nonmonotonic Reasoning in FLORA-2, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 1–12. doi:10.1007/11546207_1.
URL http://dx.doi.org/10.1007/11546207_1

[136] M. Kifer, G. Lausen, J. Wu, Logical foundations of object-oriented and frame-based languages, J. ACM 42 (4) (1995) 741–843. doi:10.1145/210332.210335.
URL http://doi.acm.org/10.1145/210332.210335

[137] A. Polleres, H. Boley, M. Kifer, Rif datatypes and built-ins 1.0 (second edition), w3c Recommendation, https://www.w3.org/TR/rif-dtb/ (Feb. 2013).

[138] W. Beek, L. Rietveld, H. R. Bazoobandi, J. Wielemaker, S. Schlobach, LOD Laundromat: A Uniform Way of Publishing Other People's Dirty Data, in: Proceedings of the 13th International Semantic Web Conference, Springer, Springer International Publishing, 2014, pp. 213–228.

[139] A. Kalyanpur, B. Parsia, M. Horridge, E. Sirin, Finding all justifications of owl dl entailments, in: K. Aberer, K.-S. Choi, N. Noy, D. Allemang, K.-I. Lee, L. Nixon, J. Golbeck, P. Mika, D. Maynard, R. Mizoguchi,

G. Schreiber, P. Cudré-Mauroux (Eds.), The Semantic Web, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 267–280.

[140] B. Suntisrivaraporn, G. Qi, Q. Ji, P. Haase, A modularization-based approach to finding all justifications for owl dl entailments, in: J. Domingue, C. Anutariya (Eds.), The Semantic Web, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 1–15.

[141] A. Borgida, D. Calvanese, M. Rodriguez-Muro, Explanation in the dl-lite family of description logics, in: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems: OTM 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 1440–1457.

[142] M. Horridge, B. Parsia, U. Sattler, Justification oriented proofs in owl, in: P. F. Patel-Schneider, Y. Pan, P. Hitzler, P. Mika, L. Zhang, J. Z. Pan, I. Horrocks, B. Glimm (Eds.), The Semantic Web – ISWC 2010, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 354–369.

[143] R. T. Fielding, R. N. Taylor, Principled design of the modern Web architecture, Transactions on Internet Technology 2 (2) (2002) 115–150. doi:10.1145/514183.514185.

[144] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, http://www.ietf.org/rfc/rfc2616.txt (Jun. 1999).

[145] R. Verborgh, A. Harth, M. Maleshkova, S. Stadtmüller, T. Steiner, M. Taheriyan, R. Van de Walle, Survey of semantic description of REST APIs, in: C. Pautasso, E. Wilde, R. Alarcón (Eds.), REST: Advanced Research Topics and Practical Applications, Springer, 2014, pp. 69–89.

[146] M. Lanthaler, C. Gütl, Hydra: A vocabulary for hypermedia-driven Web APIs, in: Proceedings of the 6th Workshop on Linked Data on the Web, 2013.
URL http://ceur-ws.org/Vol-996/papers/ldow2013-paper-03.pdf

[147] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. Gabarró Vallés, R. Van de Walle, Functional descriptions as the bridge between hypermedia APIs and the Semantic Web, in: Proceedings of the Third International Workshop on RESTful Design, ACM, 2012, pp. 33–40. doi:10.1145/2307819.2307828.

[148] R. Alarcón, E. Wilde, RESTler: crawling RESTful services, in: Proceedings of the 19th international conference on World Wide Web, ACM, 2010, pp. 1051–1052. doi:10.1145/1772690.1772799.

[149] R. Alarcón, E. Wilde, J. Bellido, Hypermedia-driven RESTful service composition, in: Service-Oriented Computing, Vol. 6568 of Lecture Notes in Computer Science, Springer, 2011, pp. 111–120.

[150] D. Roman, M. Kifer, Servlog: A unifying logical framework for service modeling and contracting, Semantic Web 9 (2) (2018) 257–290.

[151] A. J. Bonner, M. Kifer, Concurrency and communication in transaction logic., in: JICSLP, 1996, pp. 142–156.

[152] D. Roman, M. Kifer, Semantic web service choreography: Contracting and enactment, in: A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin, K. Thirunarayan (Eds.), The Semantic Web - ISWC 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 550–566.

[153] R. Verborgh, T. Steiner, D. Van Deursen, J. De Roo, R. Van de Walle, J. Gabarró Vallés, Capturing the functionality of Web services with functional descriptions, Multimedia Tools and Applicationsdoi:10.1007/s11042-012-1004-5.

[154] J. Koch, C. A. Valesco, P. Ackermann, Http vocabulary in rdf 1.0, w3c Working Draft, http://www.w3.org/TR/HTTP-in-RDF10/ (May 2011).

[155] A. Calì, G. Gottlob, T. Lukasiewicz, A. Pieris, Datalog+/-: A family of languages for ontology querying, in: Datalog Reloaded, Springer, 2011, pp. 351–368.

[156] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. Ives, DBpedia: A nucleus for a Web of open data, in: The Semantic Web, Vol. 4825 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 722–735, http://dx.doi.org/10.1007/978-3-540-76298-0_52. doi:10.1007/978-3-540-76298-0_52.

[157] J.-F. Baget, M. Leclère, M.-L. Mugnier, E. Salvat, On rules with existential variables: Walking the decidability line, Artificial Intelligence 175 (9–10) (2011) 1620–1654. doi:http://dx.doi.org/10.1016/j.artint.2011.03.002.
URL http://www.sciencedirect.com/science/article/pii/S0004370211000397

[158] M. Krötzsch, S. Rudolph, Extending decidable existential rules by joining acyclicity and guardedness, in: Twenty-Second International Joint Conference on Artificial Intelligence, 2011.

[159] G. Gottlob, M. Manna, A. Pieris, Combining decidability paradigms for existential rules, Theory and Practice of Logic Programming 13 (4-5) (2013) 877–892.

[160] Compton M. et al., The SSN ontology of the W3C semantic sensor network incubator group, Web Semantics: Science, Services and Agents on the World Wide Web 17. doi:https://doi.org/10.1016/j.websem.2012.05.003.
URL https://www.sciencedirect.com/science/article/pii/S1570826812000571

[161] D. Dell'Aglio, E. D. Valle, F. van Harmelen, A. Bernstein, Stream reasoning: a survey and outlook: A summary of ten years of research and a vision for the next decade, Data Science Journal 1. doi:http://www.doi.org/10.3233/DS-170006.

[162] A. Margara, J. Urbani, F. van Harmelen, H. Bal, Streaming the Web: reasoning over dynamic data, Journal of Web Semanticsdoi:https://doi.org/10.1016/j.websem.2014.02.001.
URL http://www.sciencedirect.com/science/article/pii/S1570826814000067

[163] D. Dell'Aglio, J.-P. Calbimonte, E. Della Valle, O. Corcho, Towards a Unified Language for RDF Stream Query Processing, 2015. doi:10.1007/978-3-319-25639-9_48.
URL https://doi.org/10.1007/978-3-319-25639-9_48

[164] M. Bienvenu, B. T. Cate, C. Lutz, F. Wolter, Ontology-based data access: A study through disjunctive Datalog, CSP, and MMSNP, ACM Trans. Database Syst.doi:10.1145/2661643.
URL http://doi.acm.org/10.1145/2661643

[165] W. Dereuddre, J. Bhatti, P. Crombez, R. Verborgh, D. Arndt, B. De Meester, Task management system and method for assigning tasks in a task management system (2016).

[166] R. Schindlauer, Answer-set programming for the semantic web, Ph.D. thesis, Dissertation, Technische Universität Wien (2006).

[167] P. A. Bonatti, Reasoning with infinite stable models, Artificial Intelligence 156 (1) (2004) 75–111.

[168] P. A. Bonatti, Erratum to: Reasoning with infinite stable models [artificial intelligence 156 (1)(2004) 75–111], Artificial Intelligence 172 (15) (2008) 1833–1835.