Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics
Department of Computer Science and Information Theory

H-1117 Budapest
Magyar tudósok körútja 2. B-132.sz
Tel.: (36-1) 463-2585 Fax: (36-1) 463-3157

Prolog Based Reasoning

by

Zsolt Zombori

PhD Thesis

Adviser: Dr. Péter Szeredi

Department of Computer Science and Information Theory
Faculty of Electrical Engineering and Informatics
Budapest University of Technology and Economics
Budapest, Hungary

Nyilatkozat

Alulírott Zombori Zsolt kijelentem, hogy ezt a doktori értekezést magam készítettem, és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint vagy azonos tartalomban, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2013. március 1.

Zombori Zsolt

A disszertáció bírálatai és a védésről készült jegyzőkönyv megtekinthető a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Dékáni Hivatalában.

Abstract

This dissertation is about automated reasoning. We present reasoning algorithms and ways in which such algorithms can be useful for knowledge intensive applications. We will discuss two main topics. The first topic is Description Logic reasoning. Description Logics (DLs) is a family of logic languages, a knowledge representation formalism that is widely used for building domain ontologies. We developed various reasoning algorithms that allow for querying DL ontologies, as well as checking the consistency of an ontology. The second topic is type inference for functional languages. Here, the task is to analyse an input program and discover as many errors as possible in compile time, to make program development easier. These topics seem very different at first sight, but they are quite similar at their cores: in both cases we start out from some initial knowledge (a set of DL axioms in the first case, an input program and a set of type restrictions in the second), and we aim to discover some logical properties of the input through automated reasoning.

The results related to these topics have been implemented in two software systems. We built a DL data reasoner called DLog and a type inference tool for the Q functional language called `qtchk`. After presenting the theoretical foundations and algorithms, we report on the developed systems as well.

Acknowledgments

I owe special thanks to my adviser, Péter Szeredi. All the work reported in this dissertation has been carried out under his supervision and in collaboration with him. He has always been very rigorous and paid attention even to the smallest details. Many times, when I already felt a paper or a proof was good enough, he insisted on making further amendments, to make what seemed to be clear and obvious even clearer and more obvious. This extra work always proved to pay off, resulting in a better final outcome.

I greatly enjoyed the close cooperation with János Csorba in developing the `qtchk` type analysis tool. We spent many hours discussing challenges and trying to find good solutions together. Our teamwork made a real difference in the development work.

The `qtchk` type inference tool has been developed in a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest. We are especially grateful to András G. Békés, Balázs G. Horváth and Ferenc Bodon for their encouragement and help.

The work reported in this thesis has been developed in the framework of the project "Talent care and cultivation in the scientific workshops of BME" project. This project is supported by the grant TÁMOP - 4.2.2.B-10/1–2010-0009.

Contents

1 Introduction **1**
- 1.1 Problem Formulation . . . 1
- 1.2 Thesis Overview . . . 2
- 1.3 List of Publications . . . 4

I Prolog Based Description Logic Reasoning **7**

2 Introducing Resolution and Description Logics **9**
- 2.1 Resolution and the Prolog Language . . . 9
  - 2.1.1 Resolution Theorem Proving . . . 9
  - 2.1.2 Programming in Prolog . . . 11
  - 2.1.3 Prolog Technology Theorem Proving . . . 11
- 2.2 Description Logics . . . 12
  - 2.2.1 The $\mathcal{AL}$ language and its extensions . . . 13
  - 2.2.2 The $\mathcal{SHIQ}$ language . . . 14
  - 2.2.3 The $\mathcal{RIQ}$ language . . . 16
  - 2.2.4 The reasoning task . . . 16
- 2.3 Resolution Based Reasoning for Description Logics . . . 17
  - 2.3.1 Separating TBox and ABox Reasoning . . . 19

3 Resolution based Methods for Description Logic Reasoning **21**
- 3.1 Related work . . . 21
- 3.2 Translating an $\mathcal{ALCHIQ}$ TBox to function-free clauses . . . 22
  - 3.2.1 Where Do Functions Come From? . . . 22
  - 3.2.2 The Modified Calculus . . . 23
  - 3.2.3 Implementing Two-Phase Reasoning . . . 25
  - 3.2.4 Benefits of Eliminating Functions . . . 26
  - 3.2.5 Summary . . . 27
- 3.3 Reduction of $\mathcal{RIQ}$ DL reasoning to $\mathcal{ALCHIQ}$ DL reasoning . . . 27
  - 3.3.1 Building automata to represent RIAs . . . 27
  - 3.3.2 A Motivating Example . . . 28
  - 3.3.3 Translating automata to concept inclusion axioms . . . 29
  - 3.3.4 Summary . . . 31
- 3.4 A Resolution Based Description Logic Calculus . . . 31
  - 3.4.1 DL Calculus . . . 32
  - 3.4.2 Termination . . . 35
  - 3.4.3 Soundness . . . 36
  - 3.4.4 The Completeness of the DL Calculus . . . 36
  - 3.4.5 Towards a DL Calculus for ABox Reasoning . . . 40
  - 3.4.6 Summary . . . 41

4 Loop Elimination, a Sound Optimisation Technique for PTTP Related Theorem Proving 43
4.1 Termination of Logic Programs . . . 43
4.1.1 Sources of infinite execution . . . 43
4.1.2 Termination in DLog . . . 44
4.1.3 Eliminating Loops . . . 45
4.2 Loop Elimination . . . 45
4.2.1 Proof Trees . . . 46
4.2.2 The Soundness of Loop Elimination . . . 48
4.3 Summary . . . 53

5 The DLog Description Logic Reasoner 55
5.1 Architecture of the DLog System . . . 55
5.2 Terminology Reasoning – the First Phase . . . 57
5.2.1 Data Representation . . . 59
5.2.2 Saturation . . . 59
5.2.3 Optimising saturation via indexing . . . 60
5.3 Future Work . . . 62
5.4 Summary . . . 62

II Static Type Inference 63

6 Introducing the Q Language and Constraint Logic Programming 65
6.1 The Q Programming Language . . . 65
6.2 Constraint Satisfaction Problems . . . 67
6.3 Constraint Handling Rules (CHR) . . . 67

7 Type Inference for the Q Functional Language 71
7.1 Work Related to Type Inference . . . 71
7.2 Necessary Restrictions of the Q Language for Type Reasoning . . . 73
7.3 Extending Q with a Type Language . . . 73
7.3.1 Type Declarations . . . 74
7.4 Type Checking for the Q Language . . . 74
7.4.1 Type Analysis Proper . . . 75
7.4.2 Constraints . . . 76
7.4.3 Issues about Type Declarations . . . 77
7.5 Type Inference for the Q Language . . . 78
7.5.1 Type Inference as a Constraint Satisfaction Problem . . . 78
7.6 Summary . . . 81

8 The `qtchk` Static Type Inference Tool for the Q Functional Language 83
8.1 Architecture . . . 83
8.2 Representing variables and constraint reasoning . . . 84
8.3 Error Handling . . . 85
8.4 Labeling . . . 85
8.5 Difficulties . . . 86
8.5.1 Handling Meta-Constraints . . . 86
8.5.2 Copying Constraints over Variables . . . 87
8.5.3 Handling Equivalence Classes of Variables . . . 88
8.5.4 Labeling . . . 89
8.6 Evaluation . . . 90
8.7 Summary . . . 91

Summary and List of Contributions 91

Bibliography **94**

Appendix **98**

A ALCHQ tableau rules **99**

B Syntax of the Q Type Language **101**

Chapter 1

Introduction

Reasoning is the magic word that binds the chapters of this thesis together. Reasoning is the ability to use available knowledge to infer something true that has not been stated explicitly. Today, there are numerous information based systems that aim to represent knowledge in a machine processable way. For such systems, automated reasoning support is very important: it allows for discovering hidden knowledge, as well as hidden errors in the knowledge base. Besides, automated reasoning can be used to answer complex user queries.

In this dissertation I will present work related to two main topics. The first topic is Description Logic reasoning. Description Logics (DLs) is a family of logic languages, a knowledge representation formalism that appeared in the early 1990's and gained wide popularity during the past two decades. These languages were designed with the intention to provide a convenient tool for building domain ontologies, while having clear and well defined semantics. DL ontologies form the basis of the Semantic Web initiative. They also play an important role in creating a unified vocabulary for medical applications. There are several futher domains, such as software verification and configuration of complex systems, where Description Logics were successfully deployed. For all such knowledge intensive applications, efficient reasoning support plays a crucial role and the success of DLs is strongly tied to the available reasoning algorithms. We developed various reasoning algorithms that allow for querying DL ontologies, as well as checking the consistency of an ontology.

Our second topic is type inference for functional languages. Here, the task is to analyse an input program and discover as many errors as possible in compile time, to make program development easier. Although our methods can be applied for functional language in general, we formulate them in the context of the Q language. Q is a vector processing language that appeared in 2003. It serves as a query language for kdb+ database. Q allows for extremely fast processing of large arrays of numeric data and has gained popularity in the financial sector over the past decade. By now, several large investment banks (Morgan Stanley, Goldman Sachs, Deutsche Bank, Zurich Financial Group, etc.) store and manipulate their data using Q. Q has a particularly terse syntax that allows for implementing complex calculations quickly, however, it is very challenging to find programming errors. Automatic error detection can hence be greatly beneficial for the Q programming community.

These topics seem very different at first sight, but they are quite similar at their cores: in both cases we start out from some initial knowledge (a set of DL axioms in the first case, an input program and a set of type restrictions in the second), and we aim to discover some logical properties of the input through automated reasoning.

In the following, I list the main research objectives. This is followed by an overview of the dissertation, where I also highlight my own results. Afterwards, I list my publications. A precise formulation of my theses will be provided at the very end of the dissertation.

1.1 Problem Formulation

This section summarises the main research objectives that resulted in the dissertation.

Large scale Description Logic reasoning Description Logics is an important and widely used formalism for knowledge representation. While existing reasoning support for Description Logics is very sensitive to the size of the available data, there are lots of application domains – such as reasoning over the web – that has to cope with really huge amounts of data. The goal of this work is to explore novel reasoning techniques that are applicable in such situations. In particular, it is crucial that the reasoner be not affected by the size of irrelevant data and to find a way to transform user queries into direct database queries.

1. The primary goal of my work is to find a tranformation scheme of Description Logic axioms into function-free clauses of first-order logic.

2. This transformation should primarily target the $\mathcal{SHIQ}$ Description Logic language.

3. After a successful $\mathcal{SHIQ}$ transformation, the results should be extended to incorporate more refined language elements, such as complex role inclusion axioms.

4. The results should be implemented in the DLog data reasoning system.

Optimised PTTP execution The Prolog Technology Theorem Prover (PTTP) is a complete first-order theorem prover built on top of Prolog. This technique plays an important role in the DLog reasoner, hence any optimisations to this technique have the potential to greatly increase the performance of DLog.

1. Some of the PTTP implementations use an optimisation called loop elimination. However, the soundness of this optimisation has not yet been proved. My goal was to find a rigorous proof of the soundness of loop elimination.

Static Type Analysis for the Q functional language Q is a dynamically typed functional programming language with a very terse and irregular syntax. While the language is widespread in financial applications, there is no built-in support for debugging and compile-time detection of errors, which makes program maintenance very difficult. The goal of this work is to provide Q with a tool that discovers static type errors in compile-time.

1. The first task is to examine the possibility of static type analysis of Q programs. This task also involves identifying the type discipline that should be enforced on Q programmers.

2. Devise an algorithm to verify the correctness of user provided type information.

3. Devise an algorithm that discovers as many type errors as possible, without any input from programmers.

4. Implement all the algorithms in a tool that can be deployed in industrial environment at Morgan Stanley Business and Technology Centre, Budapest.

1.2 Thesis Overview

Part 1, which consists of Chapters 2-5, presents our work done in the field of Description Logic reasoning. Part 2 deals with our results related to type analysis and consists of Chapters 6-8.

Chapter 2

This chapter contains all necessary background information that will be important for understanding the first part of the thesis. We first introduce resolution theorem proving and logic programming. Afterwards, we summarise the Description Logic formalism.

Chapter 3

In this chapter we present two reasoning calculi that can be used for deciding the consistency of a DL knowledge base. The first calculus, that we will refer to as the *modified calculus* is based on first-order resolution and supports the $\mathcal{ALCHIQ}$ DL language. We show how this calculus can be used for a two-phase data reasoning, which scales well and allows for reasoning over really large data sets. Using well known techniques that reduce a $\mathcal{SHIQ}$ knowledge base to an equisatisfiable $\mathcal{ALCHIQ}$ knowledge base, we easily extend our results to the $\mathcal{SHIQ}$ language, which is the most widely used DL variant.

> **Result 1.A:** I designed the modified calculus. I proved that it is sound, complete and always terminates.

Afterwards, we present a transformation that reduces the task of consistency checking of a $\mathcal{RIQ}$ knowledge base into that of an $\mathcal{ALCHIQ}$ knowledge base. The benefit of this reduction is that the latter task can be solved using our modified calculus. Our results yield a well scaling reasoning algorithm for the $\mathcal{RIQ}$ language.

> **Result 1.B:** I designed the $\mathcal{RIQ}$ to $\mathcal{ALCHIQ}$ transformation. I showed that the transformation preserves the satisfiability of the knowledge base.

In the end of this chapter, we introduce a second calculus called the *DL calculus*, which is defined directly on DL expressions, without recourse to first-order logic. The DL calculus decides the consistency of a $\mathcal{SHQ}$ terminology.

> **Result 1.C:** I designed the DL calculus. I proved that it is sound, complete and always terminates.

Chapter 4

In this chapter we present an important optimisation technique for the Prolog Technology Theorem Prover (PTTP), called *loop elimination*. This technique allows for avoiding certain kinds of infinite looping in the reasoning process and is the single most important optimisation for PTTP. We give a thorough proof of the soundness of loop elimination.

> **Result 2:** I proved that loop elimination is sound, i.e., that it can be employed without missing a valid solution.

Chapter 5

In this chapter we present the DLog data reasoner system that can be used to query $\mathcal{RIQ}$ DL knowledge bases with really large data sets.

> **Result 1.D:** I implemented the TBox saturation module of the DLog system, which performs the first phase of reasoning.

Chapter 6

In this chapter we give some background about type inference.

Chapter 7

In this chapter we present a reasoning algorithm that we developed to analyse programs written in the Q programming language for type correctness. We first present a type checker that builds on user provided type information. Afterwards, we introduce a type inference algorithm that can detect type errors even without any user provided information.

> **Result 3.A:** I designed a type checking algorithm for the Q language.

> **Result 3.B:** I designed a type inference algorithm for the Q language.

Chapter 8

We present the `qtchk` type inference tool that analyses Q programs and discovers type errors.

> **Result 3.C:** I implemented both type checking and type inference in the type analysis module of the `qtchk` system.

1.3 List of Publications

Journal Articles

1. [58] Zsolt Zombori: A Resolution Based Description Logic Calculus. In ACTA CYBERNETICA-SZEGED 19: pp. 571-588. (2010)
2. [68] Zsolt Zombori, Péter Szeredi: Loop Elimination, a Sound Optimisation Technique for PTTP related Theorem Proving. In ACTA CYBERNETICA-SZEGED 20: pp. 441-458. Paper 3. (2012)
3. [55] Zsolt Zombori: Expressive Description Logic Reasoning Using First-Order Resolution. Submitted to JOURNAL OF LOGIC AND COMPUTATION.

Reviewed International Conference Proceedings

1. [56] Zsolt Zombori: Efficient Two-Phase Data Reasoning for Description Logics. In Artificial Intelligence in Theory and Practice II: World Computer Congress 2008. Milano, Italy, 2008.09.07-2008.09.10. Springer, pp. 393-402.(ISBN: 978-0-387-09694-0)
2. [65] Zsolt Zombori Zsolt, Gergely Lukácsy: A Resolution Based Description Logic Calculus. In Proceedings of the 22nd International Workshop on Description Logics (DL2009). Oxford, UK, 2009.07.27-2009.07.30. pp. 27-30.
3. [60] Zombori Zsolt: Two Phase Description Logic Reasoning for Efficient Information Retrieval. In 27th International Conference on Logic Programming (ICLP'11): Technical Communications. Lexington, USA, 2011.07.06-2011.07.10. Wadern: Schloss Dagstuhl Leibniz-Zentrum für Informatik, pp. 296-300.(ISBN: 978-3-939897-31-6)
4. [59] Zsolt Zombori: Two Phase Description Logic Reasoning for Efficient Information Retrieval. In Extended Semantic Web Conference (ESWC) 2011: AI Mashup Challenge 2011. Heraklion, Greece, 2011.05.29-2011.06.02. pp. 498-502.
5. [69] Zsolt Zombori, Péter Szeredi, Gergely Lukácsy: Loop elimination, a sound optimisation technique for PTTP related theorem proving. In Hungarian Japanese Symposium on Discrete Mathematics and Its Applications. Kyoto, Japan, 2011.05.31-2011.06.03. Kyoto: pp. 503-512.
6. [61] Zsolt Zombori, János Csorba, Péter Szeredi: Static Type Checking for the Q Functional Language in Prolog. In 27th International Conference on Logic Programming (ICLP'11): Technical Communications. Lexington, USA, 2011.07.06-2011.07.10. Wadern: Schloss Dagstuhl Leibniz-Zentrum für Informatik, pp. 62-72.(ISBN: 978-3-939897-31-6)
7. [14] János Csorba, Zsolt Zombori, Péter Szeredi: Using Constraint Handling Rules to Provide Static Type Analysis for the Q Functional Language. In Proceedings of the 11th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2011). Lexington, USA, 2011.07.10. Paper 5.
8. [64] Zsolt Zombori, János Csorba, Péter Szeredi: Static Type Inference for the Q language using Constraint Logic Programming. In Technical Communications of the 28th International Conference on Logic Programming (ICLP'12): Leibniz International Proceedings in Informatics (LIPIcs). Budapest, Hungary, 2012.09.04-2012.09.08. Dagstuhl: pp. 119-129. Paper 8. (ISBN: 978-3-939897-43-9)

9. [15] János Csorba, Zsolt Zombori, Péter Szeredi: Pros and Cons of Using CHR for Type Inference. In Proceedings of the 9th workshop on Constraint Handling Rules (CHR 2012). Budapest, Hungary, 2012.09.04 Paper 4.

Domestic Conference Proceedings

1. [66] Zsolt Zombori, Gergely Lukácsy, Péter Szeredi: Hatékony következtetés ontológiákon. In 17th Networkshop Conference 2008. Dunaújváros, Hungary, 2008.03.17-2008.03.19.

2. [63] Zsolt Zombori, János Csorba, Péter Szeredi: Static Type Inference as a Constraint Satisfaction Problem. In Proceedings of the TAMOP PhD Workshop: TAMOP-4.2.2/B-10/1-2010-0009. Budapest, Hungary, 2012.03.09.

Course Handouts

1. [67] Zsolt Zombori, Péter Szeredi: Szemantikus és deklaratív technológiák oktatási segédlet. Course handout. 2012. Budapest, pp. 1-32.

Part I

Prolog Based Description Logic Reasoning

Chapter 2

Introducing Resolution and Description Logics

This chapter contains background information that will be important for understanding the subsequent chapters. First, in Section 2.1 we present first-order resolution, its connection to the Prolog programming language and the Prolog Technology Theorem Prover (PTTP). Afterwards, in Section 2.2 we introduce the Description Logic language family. In Section 2.3 we briefly present previous work done in applying resolution for description logic reasoning.

2.1 Resolution and the Prolog Language

Resolution is one of the first and most widely used methods for proving first order theorems. In this section we briefly introduce resolution and some of its variants. Afterwards, we present a logic programming language called Prolog that is based on resolution. In fact, the execution of a Prolog program corresponds to a resolution proof search for a sublanguage of first-order logic that consists of Horn clauses only. Finally, we present the Prolog Technology Theorem Prover, a full first order theorem proving technology that is built on top of Prolog. The definitions in this section will be important for understanding Chapter 3, where we present some resolution calculi specialised for Description Logic reasoning and also for understanding Chapter 4, which discusses an improvement on PTTP.

2.1.1 Resolution Theorem Proving

Resolution [46] is a powerful method for proving first-order theorems. Directly, it is used to check the satisfiability of a set of first-order clauses, i.e., whether there is a model satisfying all the clauses. However, all common reasoning tasks – such as entailment analysis – can be easily reduced to satisfiability checking.

Clauses are first-order formulae satisfying the following properties: all variables are universally quantified, all quantifiers are at the beginning of the formula and the quantifier-free part is a disjunction of *literals*, i.e., possibly negated atomic predicates. It is well known that any set of first-order formulae can be translated into a set of clauses (for example, see [18]) that preserves the satisfiability of the initial formula set; in other words they are *equisatisfiable*. Since all variables in clauses are universally quantified, it is customary to omit the quantifiers. We will do so in the following.

Resolution defines two inference rules, called *Binary Resolution* and *Positive Factoring*, presented in Figure 2.1. In the figure, the clauses above the bar are the premises of the inference and the clause under the bar is the conclusion. σ is the *most general unifier* of B and C, i.e., a variable substitution to terms that satisfies two properties: (1) after the substitution B and C are identical, i.e., $B\sigma = C\sigma$, and (2) σ is a most general substitution that satisfies (1). In Figure 2.2, we illustrate the application of the the two inference rules. On the left side the Binary Resolution rule is used and on the right side the Positive Factoring rule fires. The most general unifier is the same in both examples: variable y is mapped to x and every other variable is mapped to itself.

$$\frac{A \vee B \qquad \neg C \vee D}{A\sigma \vee D\sigma} \qquad\qquad \frac{A \vee B \vee C}{A\sigma \vee C\sigma}$$

Figure 2.1: Binary Resolution and Positive Factoring

$$\frac{A(x) \vee B(x) \qquad \neg B(y) \vee D(y)}{A(x) \vee D(x)} \qquad\qquad \frac{A(x) \vee B(x) \vee B(y) \vee D(y)}{A(x) \vee B(x) \vee D(x)}$$

Figure 2.2: Examples illustrating the Binary Resolution and Positive Factoring inference rules

Theorem 1. *Binary Resolution and Positive Factoring yield a calculus that is* sound *and* complete*. This means that a set of clauses is inconsistent if and only if there is a finite series of clauses $C_1, C_2, \ldots, C_n = \square$, where $\square$ denotes the empty clause, such that each clause is either a member of the initial clause set or is obtained as a conclusion of Binary Resolution or Positive Factoring with premises selected from preceding clauses.*

A proof of Theorem 1 can be found, for example, in [46].

Linear resolution As Theorem 1 indicates, resolution captures logical entailment very well. However, finding a deduction of the empty clause to show inconsistency can be rather tedious as we are given no guidance as to what clauses should be resolved in what order. To address this, various selection strategies have been devised, among them *linear resolution*.

Linear resolution is motivated by the idea that if we add a clause to a set of clauses that is considered consistent, then we only have to check the interactions that the new clause can have with the rest. Hence, in the first step, we resolve the new clause with some other, and in all subsequent steps, one of the premises will be the conclusion of the preceding step. Unfortunately, while in linear resolution the number of possible deductions is greatly decreased, we lose completeness. However, linear resolution remains complete for a restricted type of clauses that contain at most one positive literal, called *Horn clauses*. Besides, as it is shown in [34], linear resolution can be extended with a technique called *ancestor resolution* (see below in Subsection 2.1.3) which yields a complete calculus for the whole of first-order logic.

Ordered Resolution Ordered resolution [5] refines this technique by imposing an order in which the literals of a clause have to be resolved. This reduces the search space while preserving completeness. It is parametrised with an *admissible ordering ($\succ$)* on literals and a *selection function*.

Basic Superposition Basic superposition [4] is an extension of ordered resolution which has explicit inference rules for handling equality. The rules are summarised in Figure 2.3, where $E|_p$ is a subexpression of E in position p, $E[t]_p$ is the expression obtained by replacing $E|_p$ in E with t, C and D denote clauses, A and B denote literals without equality and E is an arbitrary literal. The necessary conditions for the applicability of each rule are given in the following list:

Hyperresolution: *(i)* σ is the most general unifier such that $A_i\sigma = B_i\sigma$, *(ii)* each $A_i\sigma$ is maximal in $C_i\sigma$, and there is no selected literal in $(C_i \vee A_i)\sigma$, *(iii)* either every $\neg B_i$ is selected, or $n = 1$ and nothing is selected and $\neg B_1\sigma$ is maximal in $D\sigma$.

Positive factoring: *(i)* $\sigma = \mathrm{MGU}(A, B)$, *(ii)* $A\sigma$ is maximal in $C\sigma$ and nothing is selected in $A\sigma \vee B\sigma \vee C\sigma$.

Equality factoring: *(i)* $\sigma = \mathrm{MGU}(s, s')$, *(ii)* $t\sigma \not\succ s\sigma$, *(iii)* $t'\sigma \not\succ s'\sigma$, *(iv)* $(s = t)\sigma$ is maximal in $(C \vee s' = t')\sigma$ and nothing is selected in $(C \vee s = t \vee s' = t')\sigma$.

Reflexivity resolution: *(i)* $\sigma = \mathrm{MGU}(s, t)$, *(ii)* in $(C \vee s \neq t)\sigma$ either $(s \neq t)\sigma$ is selected or nothing is selected and $(s \neq t)\sigma$ is maximal in $C\sigma$.

Superposition: *(i)* $\sigma = \mathrm{MGU}(s, E|_p)$, *(ii)* $t\sigma \not\succ s\sigma$, *(iii)* if $E = {}'w = v'$ and $E|_p$ is in w then $v\sigma \not\succ w\sigma$ and $(s\sigma = t\sigma) \not\succ (w\sigma = v\sigma)$, *(iv)* $(s = t)\sigma$ is maximal in $C\sigma$ and nothing is selected in $(C \vee s = t)\sigma$,

Hyperresolution	$\frac{(C_1 \vee A_1) \ldots (C_n \vee A_n) \quad (D \vee \neg B_1 \vee \cdots \vee \neg B_n)}{(C_1 \vee \cdots \vee C_n \vee D)\sigma}$
Positive factoring	$\frac{A \vee B \vee C}{A\sigma \vee C\sigma}$
Equality factoring	$\frac{C \vee s = t \vee s' = t'}{(C \vee t \neq t' \vee s' = t')\sigma}$
Reflexivity resolution	$\frac{C \vee s \neq t}{C\sigma}$
Superposition	$\frac{(C \vee s = t) \quad (D \vee E)}{(C \vee D \vee E[t]_p)\sigma}$

Figure 2.3: Inference rules of Basic Superposition

(v) in $(D \vee E)\sigma$ either $E\sigma$ is selected or nothing is selected and $E\sigma$ is maximal, *(vi)* $E|_p$ is not a variable position.

An important feature of basic superposition is that it remains complete even if we do not allow superposition into variables or terms substituted for variables. For this reason we keep track of such positions, by surrounding them with '[ ]' and refer to them as *variable positions* or *marked positions*. So, for instance, applying substitution $\sigma = \{x/g(y)\}$ to clause $C = R(x,y) \vee P(x)$ results in $C\sigma = R([g(y)],y) \vee P([g(y)])$.

2.1.2 Programming in Prolog

Prolog [45] is a declarative programming language equipped with a built-in logical inference mechanism that corresponds to linear resolution. This mechanism is complete for Horn clauses, which correspond directly to Prolog rules. A rule has three parts: a head containing the only positive literal, the symbol ':-' and a body which is the list of negative literals without negation, separated by commas. So, for instance, the Horn clause $P(X) \vee \neg Q_1(X) \vee \neg R(X,Y) \vee \neg Q_2(Y)$ corresponds to the Prolog rule

$$\mathtt{P(X) :- Q_1(X),\ R(X,Y),\ Q_2(Y).}$$

The semantics of this rule is as follows: if all atoms in the body are true, then so is the atom in the head. A Prolog program is a set of rules that can be used to prove a query atom, called *goal*. The program will try to unify the goal with some rule head, and in case of a successful unification, it will recursively try to prove each statement in the body. If the goal matches more than one rule head, then the program remembers this by creating a so called *choice point* and proceeds with the first matching rule. If we manage to unify the goal with a bodiless rule head, then the goal is proved. If the inference fails, because there is no matching rule head, then we roll back to the last choice point and proceed with the next matching rule. This algorithm corresponds to linear resolution that starts from the negation of the query and that is always resolved in its first literal. This mechanism is very efficient in that it starts out from the goal and examines only those rules that have a potential to answer it.

2.1.3 Prolog Technology Theorem Proving

The Prolog Technology Theorem Prover approach (PTTP) was developed by Mark E. Stickel in the late 1980's [51]. PTTP is a sound and complete first-order theorem prover, built on top of Prolog. An arbitrary set of general clauses is transformed into a set of Horn-clauses that correspond to Prolog rules. Prolog execution on these rules yields first-order logic reasoning.

In PTTP, to each first-order clause we assign a set of Horn-clauses, the so-called *contrapositives*. The first-order clause $L_1 \vee L_2 \vee \cdots \vee L_n$ has n contrapositives of the form $L_k \leftarrow \neg L_1, \ldots, \neg L_{k-1}, \neg L_{k+1}, \ldots, \neg L_n$, for each $1 \leq k \leq n$. Having removed double negations, the remaining negations are eliminated by introducing new predicate names for negated literals. For each predicate name P a new predicate name *not_P* is introduced, and all occurrences of $\neg P(X)$ are replaced with *not_P*(X), both in the head and in the body.

The link between the separate predicates P and *not_P* is provided using *ancestor resolution*, see below. For example, the clause $A(X) \vee \neg B(X) \vee \neg R(X,Y)$ is translated into three Prolog rules, each with different rule head:

```
A(X)       :-  B(X), R(X,Y).
not_B(X)   :-  not_A(X), R(X,Y).
not_R(X,Y) :-  not_A(X), B(X).
```

Thanks to using contrapositives, each literal of a first-order clause appears in the head of a Horn clause. This ensures that each literal can participate in a resolution step, in spite of the restricted selection rule of Prolog.

Next, let us see how PTTP implements positive factoring. Suppose we want to prove the goal A and during execution we obtain the subgoal $\neg A$. What this means is that by this time we have inferred a rule, according to which if a series of goals starting with $\neg A$ is true, then A follows:

$$\texttt{A} \leftarrow \texttt{not\_A}, \texttt{P}_1, \texttt{P2}, \ldots \texttt{P}_k.$$

The logically equivalent first-order clause is

$$A \vee A \vee \neg P_1 \vee \neg P2 \vee \cdots \vee \neg P_k$$

from which we see immediately that the two occurrences of A can be unified, so there is no need to prove the subgoal `not_A`. This step is called *ancestor resolution* [34], which corresponds to the positive factoring inference rule. Ancestor resolution is implemented in Prolog by building an *ancestor list* which contains *open* predicate calls (i.e. goals that we started but have not yet finished proving).

Ancestor resolution is the inference step that checks if the ancestor list contains a goal which can be matched with the negation of the current goal. If this is the case, then the current goal succeeds and the unification with the ancestor element is performed. Note that in order to retain completeness, as an alternative to ancestor resolution, one has to try to prove the current goal using normal resolution, too. This is important if the ancestor element contains variables and a different proof can yield a different variable substitution.

There are two further features in the PTTP approach. First, to avoid infinite loops, iterative deepening is used instead of the standard depth-first Prolog search strategy. Second, in contrast with most Prolog systems, PTTP uses occurs check during unification, i.e., for example terms X and $f(X)$ are not allowed to be unified because this would result in a term of infinite depth.

To sum up, PTTP uses five techniques to build a first-order theorem prover on the top of Prolog: contrapositives, renaming of negated literals, ancestor resolution, iterative deepening, and occurs check.

2.2 Description Logics

Description Logics (DLs) [26] is family of logic languages designed to be a convenient means of knowledge representation. These languages can be embedded into first-order logic, but – contrary to the latter – they are mostly decidable which gives them a great practical applicability. Description Logics provide the logical background for the Web Ontology Language (OWL [27] and OWL2 [21]).

A DL knowledge base KB consists of two parts: the TBox (terminology box) and the ABox (assertion box). The TBox contains universal knowledge that holds in a specific domain. The ABox stores knowledge about individuals. We refer to the TBox part of the knowledge base as $KB_{\mathcal{T}}$ and to the ABox as $KB_{\mathcal{A}}$.

The main building blocks of a DL knowledge base are *concepts*, that represent sets of individuals and *roles* that represent binary relations, i.e., sets of pairs of individuals. Complex concepts and roles can be built from simpler ones using concept and role constructors: the set of available constructors determines the expressivity of the language and naturally defines a language family. In the following we introduce the most important DL languages. These definitions will be important for understanding Chapter 3, where we present various calculi to perform Description Logic reasoning.

2.2.1 The $\mathcal{AL}$ language and its extensions

The $\mathcal{AL}$ language allows for describing simple relationships between concepts and roles. In particular, we can state that two concepts are identical or that one concept is a subset of another. These statements constitute the TBox. Besides, the ABox holds assertions stating that some named individual belongs to the extension of a concept or that the relationship represented by a role holds between two named individuals.

The syntax of the $\mathcal{AL}$ language is given with respect to a set $\mathcal{N}_I$ of individual names, a set $\mathcal{N}_C$ of atomic concept names and a set $\mathcal{N}_R$ of atomic role names. From these, we define the set of $\mathcal{AL}$-concepts to be the smallest set such that 1) every concept name is a concept, 2) $\top$ and $\bot$ are concepts, and 3) if C, D are concepts, A is an atomic concept and R is a role, then $\neg A$, $C \sqcap D$, $\forall R.C$ and $\exists R.T$ are also concepts.

Let a, b be individual names, C, D concept names and R a role name. An $\mathcal{AL}$ TBox is a list of axioms of the form $C \equiv D$ (*concept equivalence axiom*) and $C \sqsubseteq D$ (*concept inclusion axiom*). An $\mathcal{AL}$ ABox contains axioms of the form $C(a)$ and $R(a,b)$.

The semantics of $\mathcal{AL}$ is defined as follows:

Definition 1 (interpretation)**.** *An interpretation $I = (\Delta^I, \cdot^I)$ consists of a set Δ^I called the* domain *of I and a valuation $\cdot^I$ which maps every concept to a subset of Δ^I, every role to a subset of $\Delta^I \times \Delta^I$ and every individual name to a member of Δ^I such that, for all concepts C, D roles R, S and nonnegative integers n, the following equations hold, where $\sharp S$ denotes the cardinality of a set S:*

$$
\begin{aligned}
\top^I &= \Delta^I \\
\bot^I &= \emptyset \\
(\neg A)^I &= \Delta^I \setminus C^I \\
(C \sqcap D)^I &= C^I \cap D^I \\
(\forall R.C)^I &= \left\{ x \mid \forall y : \langle x,y \rangle \in R^I \rightarrow y \in C^I \right\} \\
(\exists R.\top)^I &= \left\{ x \mid \exists y : \langle x,y \rangle \in R^I \right\}
\end{aligned}
$$

An interpretation I satisfies

- *terminology $\mathcal{T}$ if and only if $C^I \subseteq D^I$ for each $C \sqsubseteq D \in \mathcal{T}$ and $C^I = D^I$ for each $C \equiv D \in \mathcal{T}$. In this case we say that I is a model of $\mathcal{T}$.*
- *ABox $\mathcal{A}$ if and only if $a^I \in C^I$ for each $C(a) \in \mathcal{A}$ and $\langle a^I, b^I \rangle \in R^I$ for each $R(a,b) \in \mathcal{A}$. In this case we say that I is a model of $\mathcal{A}$.*

A knowledge base KB is said to be *satisfiable* in case there exists an interpretation I which is a model of $KB_\mathcal{T}$ and $KB_\mathcal{A}$.

A concept equivalence axiom $C \equiv D$ is logically equivalent to two concept inclusion axioms: $C \sqsubseteq D$ and $D \sqsubseteq C$. Consequently, without loss of generality, we will sometimes treat the TBox as containing concept inclusion axioms only.

Let R be a role name and a, b individual names. If $\langle a^I, b^I \rangle \in R^I$ for some interpretation I, then we say that b^I is an *R-successor* of a^I in that interpretation. When it leads to no misunderstanding we will simply say that b is an R-successor of a.

Several language extensions exist for the $\mathcal{AL}$ language, which add new concept constructors. Each extension has a letter associated with it and the name of an extended language is obtained by adding the corresponding letters to the $\mathcal{AL}$ prefix. So, for example, $\mathcal{AL}$ extended with $\mathcal{C}$ and $\mathcal{Q}$ yields the $\mathcal{ALCQ}$ language. In the following, we introduce the most important language extensions.

$\mathcal{C}$ The $\mathcal{AL}$ language only allows negation in front of atomic concept. We can lift this restriction, by allowing negation to be applied to any concept. Thus we obtain the $\mathcal{ALC}$ language.

$\mathcal{U}$ This extension introduces the union constructor ($\sqcup$), i.e., a concept can be the union of two concepts. The syntax and semantics of this language extension is:

$$(C \sqcup D)^I = \qquad\qquad C^I \cup D^I$$

$\mathcal{E}$ The simple existential restriction ($\exists R.\top$) in $\mathcal{AL}$ allows for describing individuals who appear in the domain of some relation R. With full existential restriction ($\exists R.C$), we can also prescribe that the R-successors be in some concept C. Syntactically, what changes is that we allow arbitrary concepts in place of the $\top$ concept:

$$(\exists R.C)^I = \qquad\qquad \{x \mid \exists y : \langle x,y\rangle \in R^I \wedge y \in C^I\}$$

$\mathcal{N}$ We add unqualified number restrictions of the form ($\leq nR$) and ($\geq nR$) that define the set of individuals having at least or at most n R-successors:

$$(\leq nR)^I = \qquad\qquad \{x \mid \sharp\{y \mid \langle x,y\rangle \in R^I\} \leq n\}$$
$$(\geq nR)^I = \qquad\qquad \{x \mid \sharp\{y \mid \langle x,y\rangle \in R^I\} \geq n\}$$

$\mathcal{Q}$ Qualified number restrictions extend unqualified number restrictions. We can provide a concept that the R-successors have to satisfy:

$$(\leq nR.C)^I = \qquad\qquad \{x \mid \sharp\{y \mid \langle x,y\rangle \in R^I \wedge y \in C^I\} \leq n\}$$
$$(\geq nR.C)^I = \qquad\qquad \{x \mid \sharp\{y \mid \langle x,y\rangle \in R^I \wedge y \in C^I\} \geq n\}$$

One can easily show that the expressive power of full negation ($\mathcal{C}$) is equivalent to that of the union constructor ($\mathcal{U}$) and full existential restriction ($\mathcal{E}$) using De Morgan like equivalences, hence we have $\mathcal{ALC} = \mathcal{ALUE} = \mathcal{ALCE} = \mathcal{ALCU} = \mathcal{ALCUE}$. $\mathcal{AL}$, $\mathcal{ALE}$ and $\mathcal{ALU}$ are real sublanguages of $\mathcal{ALC}$ and all these languages are extended by the $\mathcal{N}$ and $\mathcal{Q}$ extensions.

2.2.2 The $\mathcal{SHIQ}$ language

The $\mathcal{AL}$ language family presented in the previous section is rather simple, which makes reasoning rather straightforward. However, there are many domains that can only be described using more sophisticated language constructs. In this section we introduce the $\mathcal{SHIQ}$ language that is probably the best known DL variant, thanks to a good compromise between complexity and expressivity. We define the language as a series of language extensions, introducing the $\mathcal{S}$, $\mathcal{SH}$, $\mathcal{SHI}$ and $\mathcal{SHIQ}$ languages.

$\mathcal{S}$ This language extends the $\mathcal{ALC}$ language with transitivity axioms of the form $Trans(R)$ for some role R. Such an axiom is satisfied by all interpretations where R is a transitive role.

$\mathcal{SH}$ The $\mathcal{H}$ extension introduces role hierarchies. The knowledge base can contain axioms of the form $S \sqsubseteq R$ where S and R are roles, which expresses that the relation represented by S is a subset of the relation represented by R, i.e., $(S \sqsubseteq R)^I \Leftrightarrow S^I \subseteq R^I$. Given a knowledge base KB, we will call the set of all role inclusion axioms the *RBox* ($KB_{\mathcal{R}}$). Note that defined this way, the RBox is part of the TBox.

$\mathcal{SHI}$ So far, we only saw constructors for concepts, that is, we could describe complex concepts, but not roles. Now, we introduce the inverse role constructor (R^-). This is an important extension since ontology developers often need to refer to the inverse of some role:

$$(R^-)^I = \qquad\qquad \{(x,y) \mid (y,x) \in R^I\}$$

$\mathcal{SHIQ}$ The $\mathcal{SHIQ}$ language is obtained by extending $\mathcal{SHI}$ with qualified number restrictions ($\mathcal{Q}$). This language has been the most important DL language of the last decade – it also forms the logical basis of OWL1, the first Web Ontology Language – and only recently has attention shifted towards even more expressive variants. The subsequent chapters will deal greatly with $\mathcal{SHIQ}$ reasoning, hence we now repeat and summarize the definitions that make up this language:

Definition 2 ($\mathcal{SHIQ}$)**.** *Let $\mathcal{N}_C$ be a set of concept names and $\mathcal{N}_\mathcal{R}$ a set of role names. The set of roles is $\mathcal{N}_\mathcal{R} \sqcup \{R^- \mid R \in \mathcal{N}_\mathcal{R}\}$. We define the function Inv on roles such that $Inv(R) = R^-$ if $R \in N_R$ and $Inv(R^-) = R$.*

A role inclusion axiom *(RIA) is an expression of the form $R \sqsubseteq S$, where R,S are roles. A* transitivity axiom *is of the form $Trans(R)$ where R is a role. A $\mathcal{SHIQ}$-*role hierarchy*, also called a $\mathcal{SHIQ}$ RBox, is a set of role inclusion axioms together with a set of transitivity axioms. For an RBox $\mathcal{R}$ we define $\sqsubseteq^*$ to be a transitive-reflexive closure of $\sqsubseteq$ over $\mathcal{R} \cup \{Inv(R) \sqsubseteq Inv(S) | R \sqsubseteq S \in \mathcal{R}\}$. Role S is called a* sub-role *of R if $S \sqsubseteq^* R$. A role is* simple *if it has no sub-role S such that $Trans(S) \in \mathcal{R}$.*

The set of $\mathcal{SHIQ}$-concepts is the smallest set such that 1) every concept name is a concept, 2) $\top$ and $\bot$ are concepts and 3) if C,D are concepts, R is a role, S is a simple role and n is a nonnegative integer, then $C \sqcup D$, $C \sqcap D$, $\neg C$, $\forall R.C$, $\exists R.C$, $\leq nS.C$ and $\geq nS.C$ are also concepts.

*A general concept inclusion axiom (GCI) is an expression of the form $C \sqsubseteq D$ for two $\mathcal{SHIQ}$-concepts C,D. A $\mathcal{SHIQ}$-*terminology*, also called a $\mathcal{SHIQ}$ TBox is a set of GCIs, extended with a $\mathcal{SHIQ}$ RBox.*

Let $\mathcal{N}_I = \{a,b,c,\ldots\}$ be a set of individual names. An assertion *is of the form $C(a)$, $R(a,b)$, $a = b$ or $a \neq b$ for $a,b \in \mathcal{N}_I$, a role R and $\mathcal{SHIQ}$-concept C. An* ABox *is a set of assertions.*

A $\mathcal{SHIQ}$ knowledge base KB can be broken into two parts: an ABox ($KB_\mathcal{A}$) and a terminology ($KB_\mathcal{T}$). The part of the terminology that relates to roles is called the role hierarchy ($KB_\mathcal{R}$).

The semantics of $\mathcal{SHIQ}$ is defined as follows:

Definition 3 (interpretation)**.** *An interpretation $I = (\Delta^I, \cdot^I)$ consists of a set Δ^I called the* domain *of I and a valuation $\cdot^I$ which maps every concept to a subset of Δ^I, every role to a subset of $\Delta^I \times \Delta^I$ and every individual name to a member of Δ^I such that, for all concepts C,D roles R,S and nonnegative integers n, the following equations hold, where $\sharp S$ denotes the cardinality of a set S:*

$$
\begin{aligned}
(R^-)^I &= \{(x,y) \mid (y,x) \in R^I\} \\
\top^I &= \Delta^I \\
\bot^I &= \emptyset \\
(\neg C)^I &= \Delta^I \setminus C^I \\
(C \sqcap D)^I &= C^I \cap D^I \\
(C \sqcup D)^I &= C^I \cup D^I \\
(\forall R.C)^I &= \{x \mid \forall y : \langle x,y \rangle \in R^I \rightarrow y \in C^I\} \\
(\exists R.C)^I &= \{x \mid \exists y : \langle x,y \rangle \in R^I \wedge y \in C^I\} \\
(\leq nR.C)^I &= \{x \mid \sharp\{y \mid \langle x,y \rangle \in R^I \wedge y \in C^I\} \leq n\} \\
(\geq nR.C)^I &= \{x \mid \sharp\{y \mid \langle x,y \rangle \in R^I \wedge y \in C^I\} \geq n\}
\end{aligned}
$$

An interpretation I satisfies

- *role hierarchy $\mathcal{R}$ if and only if $S^I \subseteq R^I$ for each $S \sqsubseteq R \in \mathcal{R}$ and R^I is transitive for each $Trans(R) \in \mathcal{R}$. In this case we say that I is a model of $\mathcal{R}$.*
- *terminology $\mathcal{T}$ if and only if it satisfies the contained role hierarchy and $C^I \subseteq D^I$ for each GCI $C \sqsubseteq D \in \mathcal{T}$. In this case we say that I is a model of $\mathcal{T}$.*
- *ABox $\mathcal{A}$ if and only if $a^I \in C^I$ for each $C(a) \in \mathcal{A}$, $\langle a^I, b^I \rangle \in R^I$ for each $R(a,b) \in \mathcal{A}$, $a^I = b^I$ for each $a = b \in \mathcal{A}$ and $a^I \neq b^I$ for each $a \neq b \in \mathcal{A}$. In this case we say that I is a model of $\mathcal{A}$.*

A $\mathcal{SHIQ}$ knowledge base *KB* is said to be *satisfiable* in case there exists an interpretation I which is a model of $KB_\mathcal{T}$ and $KB_\mathcal{A}$.

2.2.3 The $\mathcal{RIQ}$ language

In $\mathcal{SHIQ}$ we only allowed RIAs of the form $S \sqsubseteq R$ for (possibly inverse) roles R and S. Besides, we could declare that some roles are transitive. With transitivity, we can make statements like: *The friend of a friend is a friend* or that *If A is located in B and B is located in C then A is located in C.* However, often it would also be convenient to be able to say things like *The wife of a friend is a friend as well* or that *If A is located in B and B is a subdivision of C, then A is located in C.* It might also be useful in an ontology of family relations to say that *The mother of a spouse is a mother-in-law.* Motivated by these examples, we introduce *generalised RIA*s:

Definition 4 (generalised role inclusion axiom)**.** *A generalised role inclusion axiom (RIA) is of the form $w \sqsubseteq R$ where R is an atomic role name and $w = S_1 \circ S_2 \circ \ldots S_n$, i.e., w is obtained by composing n roles. A generalised role hierarchy is a set of generalised RIAs.*

In the following, when it leads to no ambiguity, we will indicate composition of roles by simply writing them after each other, i.e., instead of $S_1 \circ S_2$ we will write S_1S_2. Restricting R to atomic roles is no real restriction and is only meant to make the syntax simpler. Note that the axiom $w \sqsubseteq R$ is equivalent to $Inv(w) \sqsubseteq Inv(R)$, hence we can always choose the one in which the right hand side is an atomic role name. In the presence of generalised RIAs, there is no need for transitivity axioms, since the RIA $RR \sqsubseteq R$ captures the transitivity of R.

Introducing generalised RIAs to $\mathcal{SHIQ}$ leads to undecidability in general ([28]). However, we will focus on an important decidable subcase, when the role hierarchy is *regular*:

Definition 5 (regular role hierarchy)**.** *Let $\prec$ be a strict partial order on roles. A generalised RIA of the form $w \sqsubseteq R$ is $\prec$-regular if*

- $w = RR$ *or*
- $w = R^-$ *or*
- $w = S_1S_2 \ldots S_n$ *and* $S_i \prec R$ *for* $i \in \{1 \ldots n\}$ *or*
- $w = RS_1S_2 \ldots S_n$ *and* $S_i \prec R$ *for* $i \in \{1 \ldots n\}$ *or*
- $w = S_1S_2 \ldots S_nR$ *and* $S_i \prec R$ *for* $i \in \{1 \ldots n\}$

A generalised role hierarchy is regular if there exists a strict partial order $\prec$ such that each RIA is $\prec$-regular. The semantics is defined analogously to $\mathcal{SHIQ}$, i.e., a model I satisfies a RIA $w \sqsubseteq R$ if $w^I \subseteq R^I$. The Description Logic $\mathcal{RIQ}$[1] is obtained from $\mathcal{SHIQ}$ by replacing role hierarchies and transitivity axioms with regular role hierarchies.

With the change of the role hierarchy, the definition of simple roles changes as well:

- *Every role name that does not occur on the right hand side of a RIA is simple.*
- *A role name R is simple if, for each RIA $w \sqsubseteq R, w = S$ for some simple role S.*
- *An inverse role S^- is simple if S is simple.*

2.2.4 The reasoning task

We list the most important DL reasoning tasks:

1. Is a set of DL axioms satisfiable? In other words, can one find a model that satisfies all the axioms?
2. Is an axiom logically entailed by some set of axioms?
3. Is one concept subsumed by another, i.e., is it true that any individual that belongs to the first necessarily belongs to the second?

[1] Note that $\mathcal{RIQ}$ is sometimes defined to only allow RIAs where the left hand side contains at most two roles. Our more general definition follows [28].

4. Are two concepts equivalent, i.e., do they have the same extensions in all interpretations?

5. Are two concepts disjoint?

6. Is a concept satisfiable with respect to a TBox?

For sufficiently expressive languages, one can easily show that these reasoning tasks can be reduced to each other. In particular, all reasoning problems can be rephrased as satisfiability checking.

Theorem 2. *For any DL language that contains full negation (from $\mathcal{ALC}$ upwards), the following reasoning tasks can be reduced to satisfiability checking:*

1. *Is axiom Q logically entailed by the set of axioms KB?*

2. *Is concept C subsumed by concept D ($C \sqsubseteq D$)?*

3. *Are concepts C and D equivalent?*

4. *Are concepts C and D disjoint?*

5. *Is concept C satisfiable with respect to TBox $\mathcal{T}$?*

Proof. (1) Q is logically entailed by KB exactly when $KB \cup \{\neg Q\}$ is not satisfiable. (2) C is subsumed by D exactly when $(C \sqcap \neg D)$ is not satisfiable. (3) C and D are equivalent exactly when neither $(C \sqcap \neg D)$ nor $(D \sqcap \neg C)$ is satisfiable. (4) C and D are disjoint exactly when $(C \sqcap D)$ is not satisfiable. (5) Take a role R that appears neither in $\mathcal{T}$ nor in C. Let us consider a new TBox $\mathcal{T}' = \mathcal{T} \cup \{\top \sqsubseteq \exists R.C\}$. Given that R is a new role name, it is easy to see that the newly added axiom will only introduce inconsistency to the TBox if C is unsatisfiable. C is satisfiable in the presence of TBox $\mathcal{T}$ if and only if $\mathcal{T}'$ is consistent. □

Thanks to Theorem 2, we can afford to address only satisfiability checking when we build algorithms for DL reasoning. We will do so in the rest of the dissertation.

2.3 Resolution Based Reasoning for Description Logics

In [41] a resolution based theorem proving algorithm for the $\mathcal{SHIQ}$ DL language is presented. Our results presented in Chapter 3 provide various extensions to this algorithm.

In the first step, transitivity axioms are eliminated, at the expense of adding some new GCIs. The language obtained from $\mathcal{SHIQ}$ by eliminating transitivity is called $\mathcal{ALCHIQ}$. The obtained $\mathcal{ALCHIQ}$ knowledge base is not logically equivalent to the original one, however [41] proves that the two knowledge bases are equisatisfiable.

In the following definition, *NNF(C)* denotes the negation normal form of *C*, i.e., negation is pushed inwards to atomic concepts.

Definition 6 (concept closure)**.** *For a $\mathcal{SHIQ}$ knowledge base KB,* clos(KB) *denotes the smallest set of concepts that satisfies the following conditions:*

- *if $C \sqsubseteq D \in KB$, then $NNF(\neg C \sqcup D) \in clos(KB)$;*
- *if $C \equiv D \in KB$, then $NNF(\neg C \sqcup D) \in clos(KB)$ and $NNF(\neg D \sqcup C) \in clos(KB)$;*
- *if $C(a) \in KB$ then $NNF(C) \in clos(KB)$;*
- *if $C \in clos(KB)$ and D is a subconcept of C then $D \in clos(KB)$;*
- *if $(\leq nR.C) \in clos(KB)$ then $NNF(\neg C) \in clos(KB)$;*
- *if $\forall R.C \in clos(KB)$, $S \sqsubseteq^* R$, and $Trans(S) \in KB_{\mathcal{R}}$, then $\forall S.C \in clos(KB)$.*

We call $clos(KB)$ the concept closure of KB.

Definition 7 ($\Omega(KB)$). *For any $\mathcal{SHIQ}$ DL knowledge base KB, $\Omega(KB)$ is an $\mathcal{ALCHIQ}$ knowledge base constructed as follows:*

- $\Omega(KB)_{\mathcal{T}}$ *is obtained from* $KB_{\mathcal{T}}$ *by removing all axioms* $Trans(R)$ *and adding for each concept* $\forall R.C \in clos(KB)$ *and role S such that* $S \sqsubseteq^* R$ *and* $Trans(S) \in KB_{\mathcal{R}}$ *the axiom* $\forall R.C \sqsubseteq \forall S.(\forall S.C)$*;*
- $\Omega(KB)_{\mathcal{A}} = KB_{\mathcal{A}}$

Proposition 1. *KB is satisfiable if and only if $\Omega(KB)$ is satisfiable.*

Proof. See [41]. □

After eliminating transitivity axioms, the knowledge base, together with the negation of the query is transformed into a set of first-order clauses with a characteristic structure. These are referred to as $\mathcal{ALCHIQ}$ clauses and are summarised in Figure 2.4, where:

- **P**(t) is a possibly empty disjunction $(\neg)P_1(t) \vee \cdots \vee (\neg)P_n(t)$ of unary literals;
- **P(f**(x)): is a possibly empty disjunction $\mathbf{P_1}(f_1(x)) \vee \cdots \vee \mathbf{P_n}(f_n(x))$;
- t is a term that is surely not marked;
- [t] is a term that is surely marked;
- <t> is a term that may or may not be marked;
- $\# \in \{=, \neq\}$;

Figure 2.4: $\mathcal{ALCHIQ}$ clauses

$$\neg R(x,y) \vee S(y,x) \tag{c1}$$
$$\neg R(x,y) \vee S(x,y) \tag{c2}$$
$$\mathbf{P}(x) \vee R(x, < f(x) >) \tag{c3}$$
$$\mathbf{P}(x) \vee R([f(x)], x) \tag{c4}$$
$$\mathbf{P_1}(x) \vee \mathbf{P_2}(< \mathbf{f}(x) >) \vee \bigvee (< f_i(x) > \# < f_j(x) >) \tag{c5}$$
$$\mathbf{P_1}(x) \vee \mathbf{P_2}([g(x)]) \vee \mathbf{P_3}(< \mathbf{f}([g(x)]) >) \bigvee (< t_i > \# < t_j >) \tag{c6}$$
where t_i and t_j are of the form $f([g(x)])$ or of the form x
$$\mathbf{P_1}(x) \vee \bigvee_{i=1}^{n} (\neg R(x,y_i) \vee \bigvee_{i=1}^{n} \mathbf{P_2}(y_i) \vee \bigvee_{i,j=1}^{n \times n} (y_i = y_j) \tag{c7}$$
$$\mathbf{R}(< a >, < b >) \vee \mathbf{P}(< t >) \vee \bigvee (< t_i > \# < t_j >) \tag{c8}$$
where t, t_i and t_j are either a constant or a term $f_i([a])$

The reasoning task is reduced to deciding whether the obtained first-order clauses are satisfiable. This is answered using basic superposition (see Subsection 2.1.1) extended with a method called *decomposition*. [41] shows that the set of $\mathcal{ALCHIQ}$ clauses is bounded and that any inference with premises taken from a subset N of $\mathcal{ALCHIQ}$ results in either *(i)* an $\mathcal{ALCHIQ}$ clause or *(ii)* a clause redundant in N[2] or *(iii)* a clause that can be decomposed to, i.e., substituted with two $\mathcal{ALCHIQ}$ clauses without affecting satisfiability. These results guarantee that the saturation of an $\mathcal{ALCHIQ}$ set terminates.

[2] A redundant clause is a special case of other clauses in N and can be removed.

2.3.1 Separating TBox and ABox Reasoning

The drawback of the resolution algorithm outlined above is that it can be painfully slow. In general, resolution with saturation is a bottom-up strategy and computes all logical consequences of the clause set, many of which are irrelevant to deciding our question. It would be nice to be able to use some more efficient, query oriented, top-down mechanism. Unfortunately, such mechanisms are available only for more restrictive languages, such as Horn Clauses. One can get around this problem by breaking the reasoning into two tasks: first perform a saturation based preprocessing to deduce whatever could not be deduced otherwise and then use a fast top-down reasoner.

Note that complex reasoning is required because of the rules (TBox) of the knowledge base and that in a typical real life situation there is a relatively small TBox and a large ABox. Furthermore, the rules in the TBox are likely to remain the same over time while the ABox data can change continuously. Hence we would like to move forward all inferences involving the TBox only, perform them separately and then let the fast reasoner (whatever that will be) do the data related steps when a query arrives.

In the framework of basic superposition, when more than one inference steps are applicable, we are free to choose an order of execution, providing a means to achieve the desired separation. Elements from the ABox appear only in clauses of type (c8). [41] gives two important results about the role of ABox axioms in the saturation process:

Proposition 2. *An inference from $\mathcal{ALCHIQ}$ clauses results in a conclusion of type* (c8) *if and only if there is a premise of type* (c8)*.*

Proposition 3. *A clause of type* (c8) *cannot participate in an inference with a clause of type* (c4) *or* (c6)*.*

In light of Proposition 2, we can move forward ABox independent reasoning by first performing all inference steps involving only clauses of type (c1) – (c7). [41] calls this phase the saturation of the TBox. Afterwards, Proposition 3 allows us to eliminate clauses of type (c4) and (c6). Besides making the clause set smaller, this elimination is crucial because in the remaining clauses there can be no function symbol embedded into another (this only occurred in clauses of type (c6)). The importance of this result comes out in the second phase of the reasoning, because the available top down mechanisms are rather sensitive to the presence of function symbols.

By the end of the first phase, DL reasoning has been reduced to deciding the satisfiability of first-order clauses of type (c1) - (c3), (c5), (c7) and (c8), where every further inference involves at least one premise of type (c8). For the second phase, i.e., data reasoning, [41] uses a Datalog engine which requires function-free clauses. Therefore (unary) functional relations are transformed to new binary predicates and new constant names are added: for each constant a and each function f the new constant a_f is introduced to represent $f(a)$. Note that this transformation requires processing the whole ABox.

Chapter 3

Resolution based Methods for Description Logic Reasoning

In this chapter we present our results related to resolution based Description Logic reasoning. After an overview of related work, we present a resolution calculus in Section 3.2 that is a modified version of basic superposition. This calculus is specialised for $\mathcal{ALCHIQ}$ reasoning. In Section 3.3 we extend these results to the $\mathcal{RIQ}$ language by giving a transformation that maps any $\mathcal{RIQ}$ DL knowledge base to an equisatisfiable $\mathcal{ALCHIQ}$ knowledge base. Section 3.4 presents our work that aims to improve efficiency by moving the resolution based reasoning from the level of first-order clauses to DL axioms, i.e. define a calculus directly on DL expressions.

3.1 Related work

Description Logic languages are used more and more frequently for knowledge representation, which creates an increasing demand for efficient automated DL reasoning. The Tableau Method [2] has long provided the theoretical background for DL reasoning and most existing DL reasoners implement some of its numerous variants. The typical DL reasoning tasks can be reduced to consistency checking and this is exactly what the Tableau Method provides. While the Tableau itself has proven to be very efficient, the reduction to consistency check is rather costly for some reasoning tasks. In particular, the ABox reasoning task *instance retrieval* requires running the Tableau Method for every single individual that appears in the knowledge base. Several techniques have been developed to make tableau-based reasoning more efficient on large data sets, (see e.g. [22]), that are used by the state-of-the-art DL reasoners, such as RacerPro [23] or Pellet [50].

Other approaches use first-order resolution for reasoning. A resolution-based inference algorithm is described in [30] which is not as sensitive to the increase of the ABox size as the tableau-based methods. The system KAON2 [41] is an implementation of this approach, providing reasoning services over the description logic language $\mathcal{SHIQ}$. The algorithm used in KAON2 in itself is not any more efficient for instance retrieval than the Tableau, but several steps that involve only the TBox can be performed before accessing the ABox, after which some axioms can be eliminated because they play no further role in the reasoning. This yields a qualitatively simpler set of axioms which then can be used for an efficient, query driven data reasoning. For the second phase of reasoning KAON2 uses a disjunctive datalog engine and not the original calculus. Thanks to the preprocessing, query answering is very focused, i.e., it accesses as little part of the ABox as possible. However, in order for this to work, KAON2 still needs to go through the whole ABox once at the end of the first phase.

Reading the whole ABox is not a feasible option in case the ABox is bigger than the available memory or the content of the ABox changes so frequently that on-the-fly ABox access is an utmost necessity. Typical such scenarios include reasoning on web-scale or using description logic ontologies directly on top of existing information sources, such as in a DL based information integration system.

During the past years, we have developed a DL ABox reasoner called DLog [39], that can be freely downloaded from `http://dlog-reasoner.sourceforge.net`, which is built on principles similar to those of KAON2. We only highlight two main differences. First, instead of a Datalog engine, we use the reasoning mechanism of the Prolog language [11] to perform the second phase (see [37]). Second, we use a modified resolution calculus (see [57]) that allows us to perform more inference steps in the first phase, thanks to which more axioms can be eliminated, yielding an even simpler set of axioms to work with in the second phase. The important difference is that while the approach of [41] can only guarantee that there are no nested functional symbols, our calculus ensures that no function symbols remain at all. This makes the subsequent reasoning easier and we can perform focused, query driven reasoning without any transformation that would require going through the ABox even once.

3.2 Translating an $\mathcal{ALCHIQ}$ TBox to function-free clauses

Following the framework presented in Section 2.3, we break the reasoning task into two parts: the first phase works only with the terminology part (TBox) of the knowledge base and the second phase constitutes the data (ABox) reasoning. Note that complex reasoning is required due to the complex background knowledge stored in the TBox, while in a typical real life situation there is a relatively small TBox and a large ABox. Furthermore, the rules in the TBox are likely to remain the same over time while the ABox data can change continuously. Hence, if we manage to move forward all inferences involving the TBox only and perform them separately, then the slow reasoning algorithm required by the complexity of the TBox does not take unacceptably much time due to the potentially large size of the ABox. Furthermore, these inferences need only to be performed once, in a preprocessing phase. Afterwards, the second phase of reasoning can be performed by a fast and focused data reasoner. Each time queries arrive, only the second phase is repeated, to reflect the current state of the ABox.

The input of the reasoner is a DL knowledge base and we want to decide whether the knowledge base is satisfiable. As we have seen in Theorem 2, this is sufficient for solving all other DL reasoning tasks. In the first step we translate the knowledge base into a set of $\mathcal{ALCHIQ}$ clauses, as presented in Section 2.3. We know that any $\mathcal{SHIQ}$ knowledge base can be translated into $\mathcal{ALCHIQ}$ clauses, hence the calculus to be presented supports the $\mathcal{SHIQ}$ language.

Instead of the standard basic superposition calculus of Section 2.3, we introduce a new, slightly modified calculus, that allows us to perform more inferences before accessing the ABox. This is not just a mere regrouping of tasks, we will see that the algorithm produces a crucially simpler input for the second phase, allowing for more efficient data reasoning algorithms. The improvement is achieved by eliminating function symbols from the clauses derived from the TBox.

3.2.1 Where Do Functions Come From?

The initial $\mathcal{SHIQ}$ DL knowledge base contains no functions.[1] However, after translating TBox axioms to first-order logic, we have to eliminate existential quantifiers using skolemisation which introduces new function symbols. For example, consider the following axiom which states that rich people have a rich parent:

$$\mathit{Rich} \sqsubseteq (\exists \mathit{hasParent}.\mathit{Rich})$$

This can be expressed using the following existentially quantified first-order formula:

$$\forall x(\neg \mathit{Rich}(x) \vee \exists y(\mathit{hasParent}(x,y) \wedge \mathit{Rich}(y)))$$

Resolution requires that first-order formulae be translated into clause form, which involves eliminating existential quantification at the expense of introducing skolem functions. We obtain:

$$\neg \mathit{Rich}(x) \vee \mathit{hasParent}(x, f(x))$$
$$\neg \mathit{Rich}(x) \vee \mathit{Rich}(f(x))$$

[1] Altough constants are sometimes treated as nullary function symbols, we will not do so. Hence, whenever we refer to function symbols, constant symbols are not considered.

The ABox remains function-free, hence everything that is to know about the functions is contained in the TBox. This means we should be able to perform all function-related reasoning before accessing the ABox.

3.2.2 The Modified Calculus

We modify basic superposition presented in Section 2.3 by altering the necessary conditions to apply each rule. The new conditions are given below, with the newly added conditions underlined:

> **HyperresolutionTBox**: *(i)* σ is the most general unifier such that $A_i\sigma = B_i\sigma$, *(ii)* each $A_i\sigma$ is maximal in $C_i\sigma$, and either there is no selected literal in $(C_i \vee A_i)\sigma$ or A_i contains a function symbol, *(iii)* either every $\neg B_i$ is selected, or $n = 1$ and $\neg B_1\sigma$ is maximal in $D\sigma$ *(iv)* none of the premises contain constants.
>
> **HyperresolutionABox**: *(i)* σ is the most general unifier such that $A_i\sigma = B_i\sigma$, *(ii)* each $A_i\sigma$ is maximal in $C_i\sigma$, and there is no selected literal in $(C_i \vee A_i)\sigma$, *(iii)* either every $\neg B_i$ is selected, or $n = 1$ and nothing is selected and $\neg B_1\sigma$ is maximal in $D\sigma$, *(iv)* each A_i is ground, *(v)* $D\sigma$ is function-free.
>
> **Positive factoring**: *(i)* $\sigma = \mathrm{MGU}(A, B)$, *(ii)* $A\sigma$ is maximal in $C\sigma$ and either nothing is selected in $A\sigma \vee B\sigma \vee C\sigma$ or A contains a function symbol.
>
> **Equality factoring**: *(i)* $\sigma = \mathrm{MGU}(s, s')$, *(ii)* $t\sigma \not\succ s\sigma$, *(iii)* $t'\sigma \not\succ s'\sigma$, *(iv)* $(s = t)\sigma$ is maximal in $(C \vee s' = t')\sigma$ and either nothing is selected in $C\sigma$ or $s = t \vee s' = t'$ contains a function symbol.
>
> **Reflexivity resolution**: *(i)* $\sigma = \mathrm{MGU}(s, t)$, *(ii)* in $(C \vee s \neq t)\sigma$ either $(s \neq t)\sigma$ is selected or $s \neq t$ contains a function symbol or nothing is selected and $(s \neq t)\sigma$ is maximal in $C\sigma$.
>
> **Superposition**: *(i)* $\sigma = \mathrm{MGU}(s, E|_p)$, *(ii)* $t\sigma \not\succ s\sigma$, *(iii)* if $E = {}'w = v'$ and $E|_p = w|_{p'}$ then $v\sigma \not\succ w\sigma$ and $(s\sigma = t\sigma) \not\succ (w\sigma = v\sigma)$, *(iv)* $(s = t)\sigma$ is maximal in $C\sigma$ and either nothing is selected in $(C \vee s = t)\sigma$ or $s = t$ contains a function symbol, *(v)* in $(D \vee E)\sigma$ either $E\sigma$ is selected or nothing is selected and $E\sigma$ is maximal, *(vi)* $E|_p$ is not a variable position.

Note that hyperresolution is broken into two rules (HyperresolutionTBox and HyperresolutionABox) which differ only in the necessary conditions. In the following by *original calculus* we refer to the basic superposition presented in Section 2.3 and by *modified calculus* we mean the rules of basic superposition with the restrictions listed above.

We illustrate the difference between the two calculi using a small example. Suppose we know that people have at most one child and we also know that everybody has a clever child. We have the following axioms:

$$\top \sqsubseteq (\leq 1 \mathit{hasChild}.\top)$$
$$\top \sqsubseteq (\exists \mathit{hasChild}.\mathit{Clever})$$

From these we obtain three first-order clauses:

$$(1)\mathit{hasChild}(x, f(x))$$
$$(2)\mathit{Clever}(f(x))$$
$$(3)\neg \mathit{hasChild}(x,y) \vee \neg \mathit{hasChild}(x,z) \vee y = z$$

For any child b, i.e., for any $\mathit{hasChild}(a,b)$ axiom in the ABox, we can deduce $\mathit{Clever}(b)$ using the original basic superposition calculus through the following steps (the relevant inference rule is indicated after the conclusion):

$(4)\mathit{hasChild}(a,b)$	
$(5)\, f(a) = b$	Hyperresolution(3,1,4)
$(6)\mathit{Clever}(b)$	Superposition(5,2)

What basic superposition cannot deduce is the general rule that every child is clever. This, however, is easy with the modified calculus:

$$(7)\neg hasChild(x,y) \vee y = f(x) \qquad \text{HyperresolutionTBox(3,1)}$$
$$(8)\neg hasChild(x,y) \vee Clever(y) \qquad \text{Superposition(7,2)}$$

From the newly deduced general rule, we can easily obtain the cleverness of b using HyperresolutionABox with premises (8) and (4).

The benefit of the modified calculus is that once we deduce (8), we can dispose of (1), (2) and (7), the clauses containing function symbols. When we start adding the ABox clauses to the reasoning, the TBox has reduced to the following axioms:

$$(3)\neg hasChild(x,y) \vee \neg hasChild(x,z) \vee y = z$$
$$(8)\neg hasChild(x,y) \vee Clever(y)$$

In the following we prove that the new calculus can be used to solve the reasoning task.

Proposition 4. *The modified calculus remains sound and complete.*

Proof. The inference rules of basic superposition are all valid even if we do not impose any restrictions on their applicability. Since in the new calculus only the conditions are altered, it remains sound.

The modifications that weaken the firing requirements of a rule only extend the deducible set of clauses, so they do not affect completeness.

In case of hyperresolution, let us first consider only the new condition *(iv)* and disregard condition *(v)* on HyperresolutionABox. A hyperresolution step in its original form has a main premise of type (c7), some (possibly zero) side premises of type (c3) – (c4) and some (possibly zero) side premises of type (c8). This one step can be broken into two by first resolving the main premise with all side premises of type (c3) and (c4) (by one HyperresolutionTBox inference step) and then resolving the rest of selected literals with side premises of type (c8) (applying a HyperresolutionABox step). A hyperresolution step in the original calculus can be replaced by two steps in the modified one, so completeness is preserved.

All that remains to be proved is that condition *(v)* on HyperresolutionABox does not invalidate completeness. For this, let us consider a refutation in the original calculus that uses a hyperresolution step. If all side premises are of type (c3) and (c4) then it can be substituted with a HyperresolutionTBox step. Similarly, if all side premises are of type (c8), then we can change it to HyperresolutionABox, as clauses of type (c7) are function-free, satisfying condition *(v)*. The only other option is that there are both some premises of type (c3) and of type (c8)[2]. The result of such step is a clause of the following type:

$$\mathbf{P_1}(x) \vee \bigvee \mathbf{P_2}(a_i) \vee \bigvee \mathbf{P_2}([f_i(x)]) \vee$$
$$\vee \bigvee (a_i = a_j) \vee \bigvee ([f_i(x)] = [f_j(x)]) \vee \bigvee ([f_i(x)] = a_j)$$

At some point each function symbol is eliminated from the clause (by the time we reach the empty clause everything gets eliminated). In the modified calculus we will be able to build an equivalent refutation by altering the order of the inference steps: we first apply HyperresolutionTBox which introduces all the function symbols, but none of the constants, then we bring forward the inference steps that eliminate function symbols and finally we apply HyperresolutionABox. The intermediary steps between HyperresolutionTBox and HyperresolutionABox are made possible by the weakening of the corresponding necessary conditions. Notice, that by the time HyperresolutionABox is applied, functions are eliminated so condition *(v)* is satisfied.

We conclude that for any proof tree in the original calculus we can construct a proof tree in the modified calculus, so the latter is complete. □

Proposition 5. *Saturation of a set of $\mathcal{ALCHIQ}$ clauses using the modified calculus terminates.*

[2] It is shown in [41] that clauses of type (c8) and (c4) participating in an inference result in a redundant clause so we need not consider this case.

Proof. We build on the results in [41], that a $\mathcal{SHIQ}$ knowledge base can be transformed into first-order clauses of type (c1) – (c8) and that clauses of type (c8) are of the form $C(a), R(a,b), \neg S(a,b), a = b$ or $a \neq b$, i.e., initially they do not contain any function symbols. We will also use the fact that in the original calculus any inference with premises taken from a subset N of $\mathcal{ALCHIQ}$ results in either *(i)* an $\mathcal{ALCHIQ}$ clause or *(ii)* a clause redundant in N or *(iii)* a clause that can be substituted with two $\mathcal{ALCHIQ}$ clauses via decomposition.

All modifications (apart from breaking hyperresolution into two) affect clauses having both function symbols and selected literals, in that we can resolve with the literal containing the function symbol before eliminating all selected literals. Such a clause can only arise as a descendant of a HyperresolutionTBox step. After applying HyperresolutionTBox, we can obtain the following clauses:

$$\begin{aligned} &\mathbf{P_1}(x) \vee \bigvee(\neg R(x,y_i)) \vee \bigvee \mathbf{P_2}(y_i) \vee \bigvee \mathbf{P_2}([f_i(x)]) \vee \\ &\vee \bigvee(y_i = y_j) \vee \bigvee([f_i(x)] = [f_j(x)]) \vee \bigvee([f_i(x)] = y_j) \end{aligned} \tag{c9}$$

In the following, it will be comfortable for us to consider a clause set that is somewhat broader than (c9), in which function symbols can appear in inequalities as well. This set is:

$$\begin{aligned} &\mathbf{P_1}(x) \vee \bigvee(\neg R(x,y_i)) \vee \bigvee \mathbf{P_2}(y_i) \vee \bigvee \mathbf{P_2}([f_i(x)]) \vee \\ &\vee \bigvee(y_i = y_j) \vee \bigvee(< f_i(x) > \# < f_j(x) >) \vee \bigvee(< f_i(x) > \# y_j) \end{aligned} \tag{c10}$$

where $\# \in \{=, \neq\}$. Of course, every clause of type (c9) is of type (c10) as well.

Let us see what kind of inferences can involve clauses of type (c10). First, it can be a superposition with a clause of type (c3) or (c5). In the case of (c3) the conclusion is decomposed (in terms of [41]) into clauses of type (c3) and (c10), while in the case of (c5) we obtain a clause of type (c10). Second, we can resolve clauses of type (c10) with clauses of type (c10) or (c5). The conclusion is of type (c10). Finally, we can apply HyperresolutionABox with some side premises of the form $R(a,b_i)$, but notice that only if the literals with function symbols are missing. The result is of type (c8). This means that during saturation, we will only produce clauses of type (c1) – (c8) and (c10).

It is easy to see that there can only be a limited number of clauses of type (c10) over a finite signature. Hence the modified calculus will only generate clauses from a finite set, so the saturation will terminate. □

3.2.3 Implementing Two-Phase Reasoning

We will use the modified calculus to solve the reasoning task in two phases. Our separation differs from that of [41] in that function symbols are eliminated during the first phase, without any recourse to the ABox. Our method is summarised in Algorithm 1, where steps (1) - (3) constitute the first phase of the reasoning and step (4) is the second phase, i.e., the data reasoning. Note that one does not necessarily have to use the modified calculus for the second phase: any calculus that more effectively exploits the fact that no function symbols remain is applicable.

Algorithm 1 $\mathcal{SHIQ}$ reasoning

1. Transform the $\mathcal{SHIQ}$ knowledge base to a set of clauses of types (c1) – (c8), where clauses of type (c8) are function-free.
2. Saturate the TBox clauses (types (c1) – (c7)) with the modified calculus. The obtained clauses are of type (c1) – (c7) and (c10).
3. Eliminate all clauses containing function symbols.
4. Add the ABox clauses (type (c8)) and saturate the set.

To show that our method is adequate, we first formulate the following proposition:

Proposition 6. *A function-free ground clause can only be resolved with function-free clauses. Furthermore, the resolvent is ground and function-free.*

Proof. It follows simply from the fact that a constant a cannot be unified with a term $f(x)$ and from condition *(v)* on HyperresolutionABox. □

We are now ready to state our main claim:

Theorem 3. *Algorithm 1 is a correct, complete and finite $\mathcal{SHIQ}$ DL theorem prover.*

Proof. We know from Proposition 5 that saturation with the modified calculus terminates. After saturating the TBox, every further inference will have at least one premise of type (c8), because the conclusions inferred after this point are of type (c8) (Proposition 6). From this follows, (using Proposition 6) that clauses with function symbols will not participate in any further steps, hence they can be removed. In light of this and taking into account that the modified calculus is correct and complete (Proposition 4), so is Algorithm 1. □

By the end of the first phase of reasoning, we obtain clauses of the following types:

$$\neg R(x,y) \vee S(y,x) \tag{c11}$$
$$\neg R(x,y) \vee S(x,y) \tag{c12}$$
$$\mathbf{P}(x) \tag{c13}$$
$$\mathbf{P_1}(x) \vee \bigvee_i (\neg R(x,y_i)) \vee \bigvee_i \mathbf{P_2}(y_i) \vee \bigvee_{i,j} (y_i = y_j) \tag{c14}$$
$$(\neg) R(a,b) \tag{c15}$$
$$C(a) \tag{c16}$$
$$a = b \tag{c17}$$
$$a \neq b \tag{c18}$$

We have completely eliminated function symbols and are now ready to start the data reasoning.

3.2.4 Benefits of Eliminating Functions

The following list gives some advantages of eliminating function symbols before accessing the ABox.

1. It is more **efficient**. Whatever ABox independent reasoning we perform after having accessed the data will have to be repeated for every possible substitution of variables.

2. It is **safer**. A top-down reasoner that has to be prepared for arguments containing function symbols is very prone to fall into infinite loops. Special attention needs to be paid to ensure the reasoner does not generate goals with ever increasing number of function symbols.

3. We get **equality handling** for free. In the resulting TBox only clauses of type (c14) contain equality that can be eliminated by a mere check whether two constants from the ABox refer to the same object which is usually well known by the creators of the database. Note that equality treatment in general makes the reasoning task much more complex. This is why we had to use basic superposition.

4. ABox reasoning without functions is **qualitatively easier**. Some algorithms, such as those for Datalog reasoning are not available in the presence of function symbols. We have seen in Section 2.3.1 that [41] solves this problem by syntactically eliminating functions, but this has two drawbacks: first, equality reasoning is required (an introduced constant might be equal to an ABox constant) and second, this transformation requires scanning through the whole ABox, which might not be feasible when we have a lot of data.

3.2.5 Summary

In this section we have presented a saturation algorithm for $\mathcal{ALCHIQ}$ clauses that can be used to transform a $\mathcal{SHIQ}$ TBox to a set of function-free clauses. The transformation is independent of the ABox, and hence of the size of the ABox. It can be seen as a preprocessing for ABox reasoning and hence any resolution based ABox reasoning algorithm can make use of it. The main benefit is that without functions the ABox reasoning can be more focused, i.e., less sensitive to the size of the ABox.

3.3 Reduction of $\mathcal{RIQ}$ DL reasoning to $\mathcal{ALCHIQ}$ DL reasoning

$\mathcal{RIQ}$ is a Description Logic language that is obtained by extending $\mathcal{SHIQ}$ with complex role inclusion axioms. This extension significantly increases the expressive power of the language and is particularly important in medical ontologies. It is well known that the complexity of reasoning also increases, namely by an exponential factor. We designed an algorithm that maps any $\mathcal{RIQ}$ knowledge base into an equisatisfiable $\mathcal{ALCHIQ}$ knowledge base, which is $\mathcal{SHIQ}$ without transitivity axioms. The transformation time is exponential in the size of the initial knowledge base, hence it is asymptotically optimal. The transformation provides a means to reduce $\mathcal{RIQ}$ reasoning to $\mathcal{ALCHIQ}$ reasoning.

Most of the definitions that will be introduced in the following are based on [28], which gives a tableau procedure for deciding $\mathcal{RIQ}$. For each role R, the authors define a non-deterministic finite automaton (NFA) that captures the role paths that are subsumed by R. These automata are used during the construction of a tableau, to "keep track" of role paths. In the following we will show that the automata can be used to transform the initial $\mathcal{RIQ}$ knowledge base to an equisatisfiable $\mathcal{ALCHIQ}$ knowledge base. The main benefit is that the treatment of the role hierarchy becomes independent of the tableau algorithm. Hence, any algorithm that decides satisfiability for an $\mathcal{ALCHIQ}$ knowledge base can be used for satisfiability checking of a $\mathcal{RIQ}$ knowledge base. In particular, the two phase reasoning algorithm that we presented in Section 3.2 is applicable. This result extends the input language of the DLog reasoner from $\mathcal{SHIQ}$ to $\mathcal{RIQ}$.

3.3.1 Building automata to represent RIAs

In this subsection we define a scheme for constructing finite automata to represent regular role hierarchies. We use the same construction as presented in [28].

Definition 8 (A_R, $\hat{A}_R$, B_R)**.** *Let $\mathcal{R}$ be a regular role hierarchy. For each role name R occurring in $\mathcal{R}$, the NFA A_R is defined as follows: A_R contains a single initial state i_R and a single final state f_R with the transition $i_R \xrightarrow{R} f_R$. Moreover, for each $w \sqsubseteq R \in \mathcal{R}$, A_R contains the following transitions:*

1. *if $w = RR$, then A_R contains $f_R \xrightarrow{\varepsilon} i_R$,*
2. *if $w = S_1 \dots S_n$ and $S_1 \neq R \neq S_n$, then A_R contains $i_R \xrightarrow{\varepsilon} i_w \xrightarrow{S_1} f_w^1 \xrightarrow{S_2} f_w^2 \dots \xrightarrow{S_n} f_w^n \xrightarrow{\varepsilon} f_R$*
3. *if $w = RS_1 \dots S_n$ then A_R contains $f_R \xrightarrow{\varepsilon} i_w \xrightarrow{S_1} f_w^1 \xrightarrow{S_2} f_w^2 \dots \xrightarrow{S_n} f_w^n \xrightarrow{\varepsilon} f_R$*
4. *if $w = S_1 \dots S_n R$ then A_R contains $i_R \xrightarrow{\varepsilon} i_w \xrightarrow{S_1} f_w^1 \xrightarrow{S_2} f_w^2 \dots \xrightarrow{S_n} f_w^n \xrightarrow{\varepsilon} i_R$*

where all f_w^i, i_w are assumed to be distinct.

Next, we introduce mirrored copies of automata, where all transitions go backwards and the initial and final states are switched. Formally, in the mirrored copy of an NFA we carry out the following modifications:

- *final states are made non-final but initial*
- *initial states are made non-initial but final*
- *each transition $p \xrightarrow{S} q$ is replaced with transition $q \xrightarrow{Inv(S)} p$*

- *each transition $p \xrightarrow{\varepsilon} q$ is replaced with transition $q \xrightarrow{\varepsilon} p$.*

We define NFAs $\hat{A_R}$ as follows:

- *if $R^- \sqsubseteq R \notin \mathcal{R}$ then $\hat{A_R} := A_R$.*
- *if $R^- \sqsubseteq R \in \mathcal{R}$ then $\hat{A_R}$ is obtained as follows: first, take the disjoint union of A_R with a mirrored copy of A_R. Second, make i_R the only initial state, f_R the only final state. Finally, for f'_R the copy of f_R and i'_R the copy of i_R, add transitions $i_R \xrightarrow{\varepsilon} f'_R$, $f'_R \xrightarrow{\varepsilon} i_R, i'_R \xrightarrow{\varepsilon} f_R$ and $f_R \xrightarrow{\varepsilon} i'_R$.*

Afterwards, the NFAs B_R are defined inductively over $\prec$:

- *if R is minimal w.r.t. $\prec$, then we set $B_R := \hat{A_R}$.*
- *otherwise, B_R is the disjoint union of $\hat{A_R}$ with a copy B'_S of B_S for each transition $p \xrightarrow{S} q$ in $\hat{A_R}$ with $S \neq R$. Moreover, for each such transition, we add ε-transitions from p to the initial state in B'_S and from the final state of B'_S to q, and we make i_R the only initial state and f_R the only final state in B_R.*

Finally, the automaton B_{R^-} is a mirrored copy of B_R.

Proposition 7. *For each role $R \in \mathcal{R}$ the size of B_R is bounded exponentially in the size of $\mathcal{R}$.*

Proof. See [28]. □

Definition 9 ($B_R(q,*)$, $B_R(*,q)$)**.** *We denote by $B_R(q,*)$ the automaton that differs from B_R only in its initial state, which is q. Analogously, $B_R(*,q)$ differs from B_R only in its final state, which is q.*

Proposition 8. *For a regular role hierarchy $\mathcal{R}$ and interpretation I, I is a model of $\mathcal{R}$ if and only if, for each (possibly inverse) role S occurring in $\mathcal{R}$, each word $w \in L(B_S)$ and each $\langle x,y \rangle \in w^I$, we have $\langle x,y \rangle \in S^I$.*

Proof. See [28]. □

Proposition 8 states that two individuals are S-connected exactly when there is a role path w between them accepted by B_S. This result gives us a key to handle value restrictions. Suppose individual x satisfies some S-restriction. If this is a maximum restriction ($\leq kS.C$), then S must be a simple role and the restriction effects only the immediate neighbours of x. This case is already treated in $\mathcal{SHIQ}$. If it is a minimum restriction ($\geq kS.C$), the restriction can be made true by adding some S-successors to x. The only problematic case is universal restriction ($\forall S.C$), because finding all S-successors might be rather difficult. However, Proposition 8 tells us that it is the role paths described by B_S that we need to check to look for S-successors.

3.3.2 A Motivating Example

Before formally defining the transformation of automata generated from the role hierarchy into axioms, we try to give an intuition through a small example. Suppose the role hierarchy of a knowledge base consists of the single axiom

$$PQ \sqsubseteq R$$

where R, P, Q are role names. One of the things that this axiom tells us is that in case an individual x satisfies $\forall R.C$ for some concept C, then the individuals connected to x through a $P \circ Q$ chain have to be in C. This consequence can be described easily by the following GCI:

$$\forall R.C \sqsubseteq \forall P.\forall Q.C$$

or equivalently, we can introduce new concept names to avoid too much nesting of complex concepts:

$$\forall R.C \sqsubseteq X_1$$
$$X_1 \sqsubseteq \forall P.X_2$$
$$X_2 \sqsubseteq \forall Q.C$$

Of course, these axioms only provide for the correct propagation of concept C and a new set of similar axioms is required for all other concepts. However, we only need to consider the universal restrictions that appear as subconcepts of some axiom in the knowledge base. These concepts can be determined by a quick scan of the initial knowledge base. For example, if the TBox contains the following GCIs:

$$D \sqsubseteq \forall R.C$$
$$\top \sqsubseteq \forall R.D$$

then, only concepts C and D appear in the scope of a universal R-restriction. Let us add a copy of the above GCIs for both C and D and eliminate the role hierarchy. We obtain the following TBox:

$$\begin{array}{ll} D \sqsubseteq \forall R.C & \top \sqsubseteq \forall R.D \\ \forall R.C \sqsubseteq X_1 & \forall R.D \sqsubseteq Y_1 \\ X_1 \sqsubseteq \forall P.X_2 & Y_1 \sqsubseteq \forall P.Y_2 \\ X_2 \sqsubseteq \forall Q.C & Y_2 \sqsubseteq \forall Q.D \end{array}$$

The two knowledge bases have different signatures and hence have different models, however they are equisatisfiable. We will prove this by showing that a model of one knowledge base can be constructed from a model of the other.

3.3.3 Translating automata to concept inclusion axioms

In this subsection we formally define the transformation of a regular role hierarchy into GCIs. In the end we obtain an $\mathcal{ALCHIQ}$ knowledge base. We make use of the notion of concept closure ($clos(KB)$) provided in Definition 6. The transformation itself is analogous to how transitivity axioms were eliminated from $\mathcal{SHIQ}$ (Definition 7). Here, the situation is more complex as we have to take into consideration more sophisticated role paths.

For each concept $\forall R.C \in clos(KB)$ and each automaton state s of B_R, we introduce a new concept name $X_{(s,R,C)}$. The concepts associated with the initial and final states of B_R are denoted with $X_{(start,R,C)}$ and $X_{(stop,R,C)}$, respectively.

Definition 10 ($\Omega(KB)$). *For any $\mathcal{RIQ}$ DL knowledge base KB, $\Omega(KB)$ is an $\mathcal{ALCHIQ}$ knowledge base constructed as follows:*

- *$\Omega(KB)_{\mathcal{T}}$ is obtained from $KB_{\mathcal{T}}$ by removing all RIAs $w \sqsubseteq R$ such that R is not simple and adding for each concept $\forall R.C \in clos(KB)$ the following axioms:*
  1. $\forall R.C \sqsubseteq X_{(start,R,C)}$
  2. $X_{(p,R,C)} \sqsubseteq X_{(q,R,C)}$ *for each* $p \xrightarrow{\varepsilon} q \in B_R$
  3. $X_{(p,R,C)} \sqsubseteq \forall S.X_{(q,R,C)}$ *for each* $p \xrightarrow{S} q \in B_R$
  4. $X_{(stop,R,C)} \sqsubseteq C$
- $\Omega(KB)_{\mathcal{A}} = KB_{\mathcal{A}}$

Proposition 9. *The size of $\Omega(KB)$ is bounded exponentially in the size of KB.*

Proof. We know from Proposition 7 that the size of each B_R is bounded exponentially in the size of $KB_{\mathcal{R}}$ and consequently in the size of KB. So for each concept $\forall R.C \in clos(KB)$ we introduce at most exponentially many new GCIs of type 1-4. The size of $clos(KB)$ is linear in KB, so the total number of GCIs introduced is at most exponential in the size of KB. □

The following proposition will be useful for proving that KB and $\Omega(KB)$ are equisatisfiable.

Proposition 10. *Let KB be some $\mathcal{RIQ}$ knowledge base and I be a model of $\Omega(KB)$. Assume that $\alpha \in (\forall R.C)^I$ and there is some β and role path $w \in L(B_R)$ such that $\langle \alpha, \beta \rangle \in w^I$. Then $\beta \in C^I$.*

Proof. Let $w = S_1 S_2 \dots S_n$, where S_i is possibly an ε transition. Let $start = b_0, b_1, \dots b_n = stop$ be states of B_R along the w path. Since $\langle \alpha, \beta \rangle \in w^I$, there are individuals $\alpha = a_0, a_1, \dots a_n = \beta$ such that $\langle a_{i-1}, a_i \rangle \in S^I$. Note that in case $S_i = \varepsilon$ then $a_{i-1} = a_i$.

We show inductively that $a_i \in X^I_{(b_i,R,C)}$ for all $0 \leq i \leq n$. For this we use the axioms added in the construction of $\Omega(KB)$. The axiom of type 1 ensures that the base case holds: $\alpha \in X^I_{(start,R,C)}$, i.e., $a_0 \in X^I_{(b_0,R,C)}$. For the inductive step, suppose first that S_i is an ε transition. Then $a_i = a_{i-1}$. By the inductive hypothesis $a_{i-1} \in X^I_{(b_{i-1},R,C)}$, and the corresponding axiom of type 2 ensures that $a_i \in X^I_{(b_i,R,C)}$. In the other case, when S_i is not an ε transition, the same argument referring to a corresponding axiom of type 3 ensures that $a_i \in X^I_{(b_i,R,C)}$.

Hence we know that $a_n \in X^I_{(b_n,R,C)}$, i.e., $\beta \in X^I_{(stop,R,C)}$. This, together with the axiom of type 4 ensures that $\beta \in C^I$. □

We are ready to formulate the main claim of this section:

Theorem 4. *KB is satisfiable if and only if $\Omega(KB)$ is satisfiable.*

Proof. ($\Rightarrow$) Let I be a model of KB. We extend this model to an interpretation I' of $\Omega(KB)$. I' differs from I only in the interpretation of the new concepts $X_{(s,R,C)}$:

$$X^{I'}_{(s,R,C)} = \left\{ y \mid \exists x (x \in (\forall R.C)^I \wedge (\exists w \in L(B_R(*,s))(\langle x,y \rangle \in w^I))) \right\}$$

We prove that I' is a model of $\Omega(KB)$, by showing that the axioms added in the definition of $\Omega(KB)$ are true. We consider the four cases separately:

1. $\forall R.C \sqsubseteq X_{(start,R,C)}$
   Suppose $y \in (\forall R.C)^{I'}$. Then, by choosing $x = y$ and $w = \varepsilon$, we can apply the above definition to show that $y \in X^{I'}_{(start,R,C)}$.

2. $X_{(p,R,C)} \sqsubseteq X_{(q,R,C)}$
   Suppose $y \in X^{I'}_{(p,R,C)}$. Then, there is some $x \in (\forall R.C)^{I'}$ and some $w \in L(B_R(*,p))$ such that $\langle x,y \rangle \in w^{I'}$. Since $p \xrightarrow{\varepsilon} q \in B_R$, it also holds that $w \in L(B_R(*,q))$. Hence, the same x and w testify that $y \in X^{I'}_{(q,R,C)}$.

3. $X_{(p,R,C)} \sqsubseteq \forall S.X_{(q,R,C)}$
   Suppose $y \in X^{I'}_{(p,R,C)}$. Then, there is some $x \in (\forall R.C)^{I'}$ and some $w \in L(B_R(*,p))$ such that $\langle x,y \rangle \in w^{I'}$. Let z be some $S^{I'}$-successor of y, i.e., $\langle y,z \rangle \in S^{I'}$. Since $p \xrightarrow{S} q \in B_R$, it also holds that $wS \in L(B_R(*,q))$. Hence, x and wS testify that $z \in X^{I'}_{(q,R,C)}$. This holds for all $S^{I'}$-successors of y, hence $y \in \forall S.X^{I'}_{(q,R,C)}$.

4. $X_{(stop,R,C)} \sqsubseteq C$
   Suppose $y \in X^{I'}_{(stop,R,C)}$. Then, there is some $x \in (\forall R.C)^{I'}$ and some $w \in L(B_R)$ such that $\langle x,y \rangle \in w^{I'}$. Since I and I' only differ in the extension of new concepts, we also have $x \in (\forall R.C)^I$ and $\langle x,y \rangle \in w^I$. From the latter, we infer using Proposition 8 that $\langle x,y \rangle \in R^I$. Since $x \in (\forall R.C)^I$, it follows that $y \in C^I$ and from that we conclude that $y \in C^{I'}$.

($\Leftarrow$) Let I be a model of $\Omega(KB)$ and I' an interpretation constructed from I as follows:

- $\Delta^{I'} = \Delta^I$;
- For each individual a, $a^{I'} = a^I$;
- For each atomic concept $A \in clos(KB)$, $A^{I'} = A^I$;
- For each role R, $R^{I'} = \left\{ \langle x,y \rangle \mid \exists w \in L(B_R)(\langle x,y \rangle \in w^I) \right\}$

By construction and referring to Proposition 8, I' satisfies the role hierarchy $KB_{\mathcal{R}}$. Since $R \in L(B_R)$, we have $R^I \subseteq R^{I'}$. Furthermore, if R is simple then $R^I = R^{I'}$.

For concepts in $clos(KB)$, we define the strict partial order $\lhd$: $C \lhd D$ if and only if C or NNF(C) occur in D. We will use induction on $\lhd$ to show that for each $D \in clos(KB)$, $D^I \subseteq D^{I'}$. For the base case, i.e., when D is an atomic concept or a negated atomic concept, this follows immediately from the definition of I'. We now turn to the inductive step:

- For $D = C_1 \sqcap C_2$, assume that $\alpha \in (C_1 \sqcap C_2)^I$ for some α. Then, $\alpha \in C_1^I$ and $\alpha \in C_2^I$. By the inductive hypothesis, $\alpha \in C_1^{I'}$ and $\alpha \in C_2^{I'}$, so $\alpha \in (C_1 \sqcap C_2)^{I'}$.
- For $D = C_1 \sqcup C_2$, assume that $\alpha \in (C_1 \sqcup C_2)^I$ for some α. If $\alpha \in C_1^I$, then by induction we also have $\alpha \in C_1^{I'}$; if $\alpha \in C_2^I$, then by induction we also have $\alpha \in C_2^{I'}$. Either way, $\alpha \in (C_1 \sqcup C_2)^{I'}$.
- For $D = \exists R.C$, assume that $\alpha \in (\exists R.C)^I$. Then, β exists such that $\langle \alpha, \beta \rangle \in R^I$ and $\beta \in C^I$. By induction, $\beta \in C^{I'}$. Since $R^I \subseteq R^{I'}$, we have $\langle \alpha, \beta \rangle \in R^{I'}$, so $\alpha \in (\exists R.C)^{I'}$.
- For $D = (\geq nR.C)$, assume that $\alpha \in (\geq nR.C)^I$. Then, there are at least n distinct domain elements β_i such that $\langle \alpha, \beta_i \rangle \in R^I$ and $\beta_i \in C^I$. By induction, $\beta_i \in C^{I'}$. Since $R^I \subseteq R^{I'}$, we have $\langle \alpha, \beta_i \rangle \in R^{I'}$, so $\alpha \in (\geq nR.C)^{I'}$.
- For $D = (\leq nR.C)$, we have $R^I = R^{I'}$ since R is simple. Let $E = \text{NNF}(\neg C)$. Assume that $\alpha \in (\geq nR.C)^I$, but $\alpha \notin (\geq nR.C)^{I'}$. Then, there exists β such that $\langle \alpha, \beta \rangle \in R^I, \beta \notin C^I, \beta \in C^{I'}$, i.e., $\beta \in E^I$ and $\beta \notin E^{I'}$. However, since $E \in clos(KB)$, by induction we have $\beta \in E^{I'}$, which is a contradiction. Hence, $\alpha \in (\leq nR.C)^{I'}$.
- For $D = \forall R.C$, assume that $\alpha \in (\forall R.C)^I$, but $\alpha \notin (\forall R.C)^{I'}$. Then some β exists such that $\langle \alpha, \beta \rangle \in R^{I'}$ and $\beta \notin C^{I'}$. By the definition of $R^{I'}$ there is some $w \in L(B_R)$ such that $\langle \alpha, \beta \rangle \in w^I$. Using Proposition 10, it follows that $\beta \in C^I$. By induction, $C^I \subseteq C^{I'}$, so $\beta \in C^{I'}$, which is a contradiction. Hence $\alpha \in (\forall R.C)^{I'}$.

□

3.3.4 Summary

In this section we defined a transformation Ω that maps an arbitrary $\mathcal{RIQ}$ knowledge base to an $\mathcal{ALCHIQ}$ knowledge base. Theorem 4 states that the transformation preserves satisfiability. We also showed that the transformation increases the size of the TBox with at most an exponential factor (Proposition 9). This is asymptotically optimal: $\mathcal{ALCHIQ}$ is known to be ExpTime-hard while $\mathcal{RIQ}$ is 2ExpTime-hard ([33]), so $\mathcal{RIQ}$ is indeed exponentially harder than $\mathcal{ALCHIQ}$.

Using this result, any algorithm that decides satisfiability for $\mathcal{ALCHIQ}$ can decide satisfiability for $\mathcal{RIQ}$. In particular, the modified calculus presented in Subsection 3.2.2 is applicable.

3.4 A Resolution Based Description Logic Calculus

In this section we present a reasoning algorithm, called *DL calculus*, which decides the consistency of a $\mathcal{SHQ}$ TBox. The novelty of this calculus is that it is defined directly on DL axioms. Working on this high level of abstraction provides an easier to grasp algorithm with less intermediary transformation steps and increased efficiency. As we showed in Theorem 2, such an algorithm can be used for solving all other TBox reasoning tasks as well.

In Subsection 3.4.1 we present the DL calculus that performs consistency check for a $\mathcal{SHQ}$ TBox. Afterwards, in Subsection 3.4.2 we prove termination of the algorithm. In Subsection 3.4.3 we prove the soundness of the DL calculus. In Subsection 3.4.4 we prove that the calculus is complete. Subsection 3.4.5 discusses the possibility of extending the DL calculus to ABox reasoning. Finally, Subsection 3.4.6 concludes by giving a brief summary of our results.

3.4.1 DL Calculus

The algorithm can be summarized as follows. We determine a set of concepts that have to be satisfied by each individual of an interpretation in order for the TBox to be true. Next, we introduce inference rules that derive a new concept from two concepts. Using the inference rules, we saturate the knowledge base, i.e., we apply the rules as long as possible and add the consequent to the knowledge base. We also apply redundancy elimination: whenever a concept extends another, it can be safely eliminated from the knowledge base [3]. It can be shown that saturation terminates. We claim that the knowledge base is inconsistent if and only if the saturated set contains the empty concept ($\bot$).

Preprocessing

We first eliminate transitivity from the knowledge base, as presented in Section 2.3. Next, we internalize the TBox, i.e., we transform all GCIs into a set of concepts that have to be satisfied by each individual. For instance, the axiom $C \sqsubseteq D$ is equivalent to the axiom $\top \sqsubseteq \neg C \sqcup D$, which amounts to saying that $\neg C \sqcup D$ has to be satisfied by all individuals.

Internalization is followed by structural transformation which eliminates the nesting of composite concepts into each other. A $\mathcal{SHQ}$ expression that appears in the TBox can be of arbitrary complexity, i.e., all sorts of composite concepts can appear within another concept. This makes reasoning very difficult. To solve this problem, we eliminate nesting composite concepts into each other by introducing new concept symbols that serve as names for embedded concepts. For details, see [41].

Finally, we make a small syntactic transformation: concepts $\forall R.C$ and $\exists R.D$ are replaced with equivalent concepts $(\leq 0R.\neg C)$ and $(\geq 1R.D)$, respectively. As a result, we obtain the following types of concepts, where L is a possibly negated atomic concept and R an arbitrary role:

$$L_1 \sqcup L_2 \sqcup \cdots \sqcup L_i$$
$$L_1 \sqcup (\geq kR.L_2)$$
$$L_1 \sqcup (\leq nR.L_2)$$

Notation

Before presenting the inference rules, we define some important notions. A *literal concept* (typically denoted with L) is a possibly negated atomic concept. A *bool concept* contains no role expressions (allowing only negation, union and intersection). We use capital letters from the beginning of the alphabet ($A, B, C \ldots$) to refer to bool concepts. In the following, we will always assume that a bool concept is presented in a simplest disjunctive normal form, i.e., it is the disjunction of conjunctions of literal concepts. So for example, instead of $A \sqcup A \sqcup (B \sqcap \neg B \sqcap C)$ we write A, and $A \sqcap \neg A$ is replaced with $\bot$. To achieve this, we apply eagerly some simplification rules, see later. When the inference rules do not preserve disjunctive normal form (DNF), we will use the explicit *dnf* operator:

$$dnf(A \sqcap B) = \begin{cases} dnf(A_1 \sqcap B) \sqcup dnf(A_2 \sqcap B) & \text{if } A = A_1 \sqcup A_2 \\ dnf(A \sqcap B_1) \sqcup dnf(A \sqcap B_2) & \text{if } B = B_1 \sqcup B_2 \\ (A \sqcap B) & \text{otherwise} \end{cases}$$

The *dnf* operator is defined only for concepts that are the intersection of two concepts. The bool concepts in the premises are always in DNF and the conclusion contains either the union or the intersection of such concepts. The union of two DNF concepts is also in DNF so we only need to apply the *dnf* operator to transform the intersection of two DNF concepts.

Ordering

Let $\succ$ be a total ordering, called a *precedence*, on the set of (atomic concept, atomic role, natural number, logic) symbols, such that $\geq \succ \leq \succ R \succ n \succ C \succ \neg \succ \sqcup \succ \sqcap \succ \top \succ \bot$ for any atomic concept C, atomic role name R and natural number n; furthermore for any two natural numbers $n_1 \succ n_2$ if and only if $n_1 > n_2$. We define a corresponding *lexicographic path ordering* $\succ_{lpo}$ (see [3]) as follows:

$s = f(s_1,\ldots,s_m) \succ_{lpo} g(t_1,\ldots,t_n) = t$ if and only if

1. $f \succ g$ and $s \succ_{lpo} t_i$, for all i with $1 \leq i \leq n$; or
2. $f = g$ and, for some j, we have $(s_1,\ldots,s_{j-1}) = (t_1,\ldots,t_{j-1}), s_j \succ_{lpo} t_j$, and $s \succ_{lpo} t_k$, for all k with $j < k \leq n$; or
3. $s_j \succeq_{lpo} t$, for some j with $1 \leq j \leq m$.

In order for the above definition to be applicable, we treat concept $(\geq kS.A)$ as $\geq(k,S,A)$ and concept $(\leq nR.D)$ as $\leq(n,R,D)$. If the precedence is total on the symbols of the language, then the lexicographic path ordering is total on DL expressions. For simplicity, we often write $\succ$ instead of $\succ_{lpo}$ when it does not lead to confusion. Note a couple properties of our ordering that will be useful later:

1. A $\geq$-concept is greater than any $\leq$-concept or any bool concept.
2. A $\leq$-concept is greater than any bool concept.
3. $C_1 = (\leq n_1 R_1.A_1)$ is greater than $C_2 = (\leq n_2 R_2.A_2)$ if and only if:
   - $R_1 \succ R_2$ or
   - $R_1 = R_2$ and $n_1 > n_2$ or
   - $R_1 = R_2$, $n_1 = n_2$ and $A_1 \succ A_2$

Definition 11 (maximal concept)**.** *Given a set N of concepts, concept $C \in N$ is* maximal *in N if C is greater than any other concept in N.*

Since the ordering $\succ_{lpo}$ is total, for any finite set N there is always a unique concept $C \in N$ that is maximal in N.

$\mathcal{SHQ}$-concepts

A derivation in the DL calculus generates concepts that are more general than the ones obtained after preprocessing. We call this broader set *$\mathcal{SHQ}$-concepts*, defined as follows (C, D, E stand for concepts containing no role expressions):

$$
\begin{array}{ll}
C & \text{(bool concepts)} \\
C \sqcup \bigsqcup(\leq nR.D) & (\leq\text{-max concepts}) \\
C \sqcup \left(\bigsqcup(\leq nR.D)\right) \sqcup (\geq kS.E) & (\geq\text{-max concepts})
\end{array}
$$

where bool concepts C, D, E are in DNF. Note two important properties of $\mathcal{SHQ}$-concepts:

1. A $\mathcal{SHQ}$-concept is a disjunction that contains at most one $\geq$-concept.
2. There are no nested concepts containing role expressions, i.e., a concept embedded into a $\geq$-concept or a $\leq$-concept is always a bool concept.

According to the ordering defined above, each $\leq$-concept is greater than any bool concept, so the maximal disjunct in a $\leq$-max concept is a $\leq$-concept. Similarly, any $\geq$-concept is greater than any $\leq$- or bool concept, so the maximal disjunct in a $\geq$-max concept is a $\geq$-concept. This is the rationale for naming these concepts $\leq$-max and $\geq$-max, respectively. Obviously, any concept obtained after preprocessing is a $\mathcal{SHQ}$-concept:

Proposition 11. *For any $\mathcal{SHQ}$ knowledge base KB, if we apply the preprocessing transformations described above on KB, we obtain a set of $\mathcal{SHQ}$-concepts.*

Inference Rules

The inference rules are presented in Figure 3.1, where C_i, D_i, E_i are possibly empty bool concepts. W_i stands for an arbitrary $\mathcal{SHQ}$-concept that can be empty as well. Some of the rules do not preserve the disjunctive normal form (DNF) of bool concepts. In such cases, we use the *dnf* operator as defined above. Note that two disjunctive concepts are resolved along their respective maximal disjuncts and the ordering that we imposed on the concepts yields a selection function. Since the ordering is total, we can always select the unique maximal disjunct to perform the inference step.

Rule1 $$\frac{C_1 \sqcup (D_1 \sqcap A) \qquad C_2 \sqcup (D_2 \sqcap \neg A)}{C_1 \sqcup C_2}$$
where $D_1 \sqcap A$ is maximal in $C_1 \sqcup (D_1 \sqcap A)$
and $D_2 \sqcap \neg A$ is maximal in $C_2 \sqcup (D_2 \sqcap \neg A)$

Rule2 $$\frac{C \qquad W \sqcup (\geq nR.D)}{W \sqcup (\geq nR.dnf(D \sqcap E))}$$
where E is obtained by using Rule1 on premises C and D

Rule3 $$\frac{W_1 \sqcup (\leq nR.C) \qquad W_2 \sqcup (\geq kS.D)}{W_1 \sqcup W_2 \sqcup (\geq (k-n)S.dnf(D \sqcap \neg C))}$$
$n < k, S \sqsubseteq^* R$, $(\leq nR.C)$ is maximal in $W_1 \sqcup (\leq nR.C)$
and $(\geq kS.D)$ is maximal in $W_2 \sqcup (\geq kS.D)$

Rule4 $$\frac{W_1 \sqcup (\leq nR.C) \qquad W_2 \sqcup (\geq kS.D)}{W_1 \sqcup W_2 \sqcup (\leq (n-k)R.dnf(C \sqcap \neg D)) \sqcup (\geq 1S.dnf(D \sqcap \neg C))}$$
$n \geq k, S \sqsubseteq^* R$, $(\leq nR.C)$ is maximal in $W_1 \sqcup (\leq nR.C)$
and $(\geq kS.D)$ is maximal in $W_2 \sqcup (\geq kS.D)$

Figure 3.1: TBox inference rules of the DL calculus

Along with the inference rules, we use a further set of rules that we call *simplification rules* and which are shown in Figure 3.2. These rules only have one premise which is redundant in the presence of the conclusion and hence can be eliminated. In other words, the simplification rules are used to simplify concepts and do not deduce new concepts. Simplification rules are applied not only to $\mathcal{SHQ}$-concepts, but also to subconcepts appearing in $\mathcal{SHQ}$-concepts. For example, S1 is used to replace the concept $C \sqcup A \sqcup A$ with $C \sqcup A$, but also to replace $(\geq nR.(C \sqcup A \sqcup A))$ with $(\geq nR.(C \sqcup A))$.

Rule1 corresponds to the classical resolution inference and Rule2 makes this same inference possible for entities whose existence is required by $\geq$-concepts. Rule3 and Rule4 are harder to understand. They address the interaction between $\geq$-concepts and $\leq$-concepts. Intuitively, if some entity satisfies $\leq nR.C$ and also satisfies $\geq kS.D$, then there is a potential for clash if concepts C and D are related, more precisely if D is subsumed by C. In such cases $D \sqcap \neg C$ is not satisfiable, which either leads to contradiction if $n < k$ (Rule3) or results in a tighter cardinality restriction on the entity (Rule4). If several $\geq$-concepts and a $\leq$-concept are inconsistent together, then each $\geq$-concept is used to deduce a $\leq$-concept with smaller cardinality (Rule4) until the $\leq$-concept completely disappears from the conclusion (Rule3) and we obtain the empty concept.

Saturation

We saturate the knowledge base, i.e., we apply the rules in Figure 3.1 to deduce new concepts as long as possible. Before adding the consequent to the concept set, we eagerly apply the simplification rules of Figure 3.2 to make the concept as simple as possible. We claim that the consequent is always a $\mathcal{SHQ}$-concept.

$$\textbf{S1}\quad \frac{C \sqcup L \sqcup \cdots \sqcup L}{C \sqcup L}$$

$$\textbf{S2}\quad \frac{C \sqcup D \sqcup (D \sqcap E)}{C \sqcup D}$$

$$\textbf{S3}\quad \frac{C \sqcup D \sqcup (\neg D \sqcap E)}{C \sqcup D \sqcup E}$$

$$\textbf{S4}\quad \frac{C \sqcup D \sqcup \neg D}{\top}$$

$$\textbf{S5}\quad \frac{C \sqcup (D \sqcap E \sqcap \neg E)}{C}$$

$$\textbf{S6}\quad \frac{W \sqcup (\geq nR.\bot)}{W}$$

$$\textbf{S7}\quad \frac{W \sqcup (\leq nR.\bot)}{\top}$$

Figure 3.2: TBox simplification rules of the DL calculus

Proposition 12. *The set of $\mathcal{SHQ}$-concepts is closed under the inference rules in Figure 3.1 and the simplification rules in Figure 3.2.*

Proof. Consider Rule1. $D_1 \sqcap A$ is maximal in $C_1 \sqcup (D_1 \sqcap A)$ which is only possible if C_1 does not contain any $\geq$- or $\leq$-concepts. Hence it is a bool concept. Analogously, the fact that $D_2 \sqcap \neg A$ is maximal in $C_2 \sqcup (D_2 \sqcap \neg A)$ ensures that C_2 is another bool concept. Bool concepts are in DNF. The conclusion is the disjunction of two bool concepts ($C_1 \sqcup C_2$) which is also in DNF and hence is a bool concept.

Rule2 resolves a bool concept with a $\geq$-max concept. We have just seen that resolving C and D by Rule1 yields a bool concept. We take the conjunction of this concept and another bool concept ($D \sqcap E$) which is not in DNF, but it yields a bool concept once we apply the *dnf* operator. Hence the conclusion is a $\geq$-max concept.

In Rule3, the maximal disjunct of the first premise is $(\leq nR.C)$, so it does not contain any $\geq$-concept. The second premise is a $\geq$-max concept and contains exactly one $\geq$-concept, namely $(\geq kS.D)$. The conclusion contains one $\geq$-concept and is a $\geq$-max concept. Again, the *dnf* operator is used to ensure that the bool concept appearing in the $\geq$-disjunct of the conclusion is in DNF.

In Rule4, the maximal disjunct of the first premise is $(\leq nR.C)$, so it is a $\leq$-max concept and does not contain any $\geq$-concept. The second premise contains exactly one $\geq$-concept, so W_2 contains no $\geq$-concept. Consequently, the conclusion will contain only one $\geq$-concept and all subconcepts inside $\geq$- and $\leq$-concepts are bool concepts. We obtain a $\geq$-max concept.

Simplification rules S1-S5 eliminate some disjuncts or conjuncts from bool concepts in DNF. The conclusion is always a simpler bool concept in DNF. S6 eliminates an unsatisfiable branch from a disjunction, turning a $\geq$-max concept either to a bool concept or to a $\leq$-max concept. In case of S7, the premise is a tautology and can be safely eliminated. □

3.4.2 Termination

The following proposition – along with Proposition 12 – ensures that the DL calculus terminates.

Proposition 13. *The set of all $\mathcal{SHQ}$-concepts that can be deduced from any finite TBox is finite.*

Proof. For any finite TBox, there can only be finitely many distinct role expressions and bool concepts. Furthermore, note that each inference rule either leaves the arity of a number restriction unaltered or reduces it. So in a $(\leq nR.C)$ or $(\geq nR.C)$ expression the number of possible values for n, R and C is finite for a fixed TBox. As all $\mathcal{SHQ}$-concepts are disjunctions of bool, $\leq$, and $\geq$-concepts, we have an upper limit for the set of deducible $\mathcal{SHQ}$-concepts. □

DL calculus deduces only $\mathcal{SHQ}$ concepts from $\mathcal{SHQ}$ concepts. Since there are finitely many $\mathcal{SHQ}$ concepts, even if we have to deduce every possible $\mathcal{SHQ}$-concept, it still requires finitely many steps, so the calculus is guaranteed to terminate.

3.4.3 Soundness

It is straightforward to show that the simplification rules are sound, i.e., if all individuals of an interpretation satisfy the premise then they also satisfy the conclusion. We leave this to the reader. The inference rules are slightly more complex.

Theorem 5. *The inference rules of the DL calculus are sound.*

Proof. Consider Rule1 and suppose that x satisfies both premises. Either A or $\neg A$ is true of x. If $A(x)$ is true, then x must satisfy C_2, due to the second premise. Analogously, if $\neg A(x)$ is true, then x must satisfy C_1. In either case, the conclusion holds for x.

We turn to Rule2. Let x be an individual. It satisfies the second premise, so either W or $(\geq nR.D)$ holds for x. In the first case the conclusion is satisfied by x, in the second case x has at least n R-successors that satisfy D. These successors also satisfy the first premise (C) and – given that Rule1 is sound – they satisfy E. If these R-successors satisfy both D and E, then they satisfy $D \sqcap E$ as well. So it holds for x that it has at least n R-successors that satisfy $D \sqcap E$, so the conclusion is again satisfied.

For Rule3, let x be an arbitrary individual. If x satisfies either W_1 or W_2, then it satisfies the conclusion. Otherwise, x satisfies $(\leq nR.C)$ and $(\geq kS.D)$, where $S \sqsubseteq R$. So, x has at least k distinct S-successors that satisfy D (that are R-successors as well). Of these, at most n successors can satisfy C, so there are at least $k-n$ S-successors that satisfy $\neg C$. From this it follows directly that the conclusion holds for x.

Finally, let us consider Rule4 and let again x denote an arbitrary individual. If x satisfies either W_1 or W_2, then it satisfies the conclusion. Otherwise, x satisfies $(\leq nR.C)$ and $(\geq kS.D)$, where $S \sqsubseteq R$. So, x has at least k distinct S-successors that satisfy D. If any of these successors satisfy $\neg C$ then the last disjunct of the conclusion holds. Otherwise, all the k S-successors satisfy C. Given that x can have no more than n successors that satisfy C, there cannot be more than $n-k$ successors that are not among those satisfying D, but they satisfy C. Hence the second to last disjunct of the conclusion holds for x. □

3.4.4 The Completeness of the DL Calculus

In this subsection we prove that the method presented in Subsection 3.4.1 is complete, i.e., whenever there is some inconsistency in a TBox $\mathcal{T}$, the empty concept is deduced. We prove completeness by showing that if a saturated set $Sat_{\mathcal{T}}$ does not contain $\bot$ then the axiom $\top \sqsubseteq \bigsqcap Sat_{\mathcal{T}}$ has a model. Instead of building the model itself, we will prove that the $\mathcal{ALCHQ}$ tableau method can find one such model. In order for the model to satisfy $\top \sqsubseteq \bigsqcap Sat_{\mathcal{T}}$, the concepts in $Sat_{\mathcal{T}}$ are added to the label of every newly created node in the tableau.

Although the tableau rules are fairly standard, there might be small variations. Hence, to avoid confusion, in Appendix A we provide the definition of the tableau rules that we assume in the following.

Building the Tableau Tree

In the previous sections, we replaced $\forall$- and $\exists$-concepts with $\leq$- and $\geq$-concepts to make the presentation of the inference rules simpler. As we turn to the tableau, however, the reader might be more familiar with the corresponding $\forall$-rule and $\exists$-rule. Hence, in the following, we will treat our $(\leq 0R.C)$ and $(\geq 1S.D)$ concepts as $(\forall R.\neg C)$ and $(\exists S.D)$, respectively.

Whenever we have several applicable tableau rules, we require the following ordering precedence: $\sqcup$-rules, $\sqcap$-rule, $\exists$-rule, $\geq$-rule, $\forall$-rule, $\bowtie$-rule and $\leq$-rule. When applying the $\sqcup$-rule we proceed with the branch[3] that adds the minimal possible concept to the label of a node. Given that the tableau method is don't care non-deterministic with respect to these choices, the completeness of the algorithm is preserved.

[3] Throughout this paper, "branch" refers to a branch of the meta-tableau tree, i.e., one of the tableaux resulting from the application of a non-deterministic rule.

Whenever a node n contains a disjunctive concept $W \sqcup C$, the branch where C is added to the label of n is only examined after each disjunct in W that is smaller than C has been proven unsatisfiable. A *clash* occurs in the tableau tree when an atomic concept name and its negation both appear in the label of some node. In this case we roll back and proceed with another branch. A *final clash* occurs when there are no branches left, i.e., the tableau proves the inconsistency of $Sat_{\mathcal{T}}$. We show that no final clash can be reached if $Sat_{\mathcal{T}}$ does not contain $\bot$.

Bool Concepts

Let us first consider the case when $Sat_{\mathcal{T}}$ contains only bool concepts.

Theorem 6. *If $Sat_{\mathcal{T}}$ contains only bool concepts and does not contain $\bot$, then no final clash is possible.*

Proof. To obtain contradiction, suppose that we reach a final clash. Hence, for some atomic concept A, both A and $\neg A$ appear in the label of some node. This is only possible if $Sat_{\mathcal{T}}$ contains concepts

$$W_1 = C_1 \sqcup (D_1 \sqcap A) \qquad W_2 = C_2 \sqcup (D_2 \sqcap \neg A)$$

The clash is final, so there are no more branches, i.e., $(D_1 \sqcap A)$ and $(D_2 \sqcap \neg A)$ are maximal in W_1 and W_2, respectively, and each disjunct in C_1 and C_2 leads to clash. W_1 and W_2 are resolvable using Rule1, so $Sat_{\mathcal{T}}$ also contains

$$W = C_1 \sqcup C_2$$

W cannot be empty because we assumed that $Sat_{\mathcal{T}}$ does not contain $\bot$. The simplification rules, and in particular S1 was eagerly applied on W_1 and W_2, so there are no other occurrences of $(D_1 \sqcap A)$ in C_1 and $(D_2 \sqcap \neg A)$ in C_2. So the maximal disjuncts in W_1 and W_2 are strictly maximal. Let X denote the greater concept of $(D_1 \sqcap A)$ and $(D_2 \sqcap \neg A)$. X is greater than any disjunct in either C_1 or C_2. This means that the branches corresponding to all disjuncts of W were examined before examining the branch corresponding to X (due to the ordering imposed on the application of the $\sqcup$-rule). But we know that all disjuncts in W lead to clash, so a final clash must have been obtained on W, even before introducing X to the label of the node, which contradicts our assumption that the final clash involved X. □

Corollary 1. *If $Sat_{\mathcal{T}}$ does not contain $\bot$, then the set of bool concepts in $\mathcal{T}$ is satisfiable.*

Notice that only Rule1 is used to detect the inconsistency of bool concepts. This observation will be useful for us later.

Corollary 2. *If a set N of bool concepts is unsatisfiable then there is a sequence of bool concepts $p_1 \dots p_n = \bot$ such that for each p_i, there is an instance of Rule1 with premises from $N \cup \{p_1, p_2 \dots p_{i-1}\}$ whose conclusion is p_i. We call this sequence a deduction of $\bot$.*

$\geq$-max Concepts

Let us now assume that $Sat_{\mathcal{T}}$ contains only bool concepts and $\geq$-max concepts.

Proposition 14. *Let $W = C \sqcup (\geq nR.D)$ be a $\geq$-max concept in $Sat_{\mathcal{T}}$. Then D is satisfiable.*

Proof. Suppose that D is unsatisfiable. Since it is in DNF, it is the disjunction of conjunctions such that each conjunction contains some atom together with its negation. However, the simplification rules are eagerly applied on all $\mathcal{SHQ}$-concepts and due to S5 all disjuncts of D were eliminated. Hence $D = \bot$ and $W = C \sqcup (\geq nR.\bot)$. S6 is applicable on W yielding C, so W was removed from $Sat_{\mathcal{T}}$ and replaced by C. This is a contradiction, so D must be satisfiable. □

Proposition 15. *Let $W = C \sqcup (\geq nR.D)$ be a $\geq$-max concept and $B = \{B_i\}$ a set of bool concepts. If $\{D\} \cup B$ is inconsistent, then there is a deduction of C using Rule1 and Rule2 and the simplification rules.*

Proof. We know from Corollary 2 that there is a deduction $p_1, p_2 \dots p_n = \bot$ from $\{D\} \cup B$ using Rule1. In this sequence each concept has a set of premises, either from the original concept set or from concepts that were deduced earlier. Let us define the *ancestor* relation as the transitive closure of the premise relation and let *descendant* be its inverse relation. For each p_i, let A_i denote the set of its ancestors that are either identical to D or are descendants of D. For each p_i such that A_i is not the empty set, replace p_i with $C \sqcup (\geq nR.(p_i \sqcap \bigsqcap A_i))$. We obtain a deduction in which each time the conclusion is a $\geq$-max concept, Rule2 is used instead of Rule1. In particular, $p_n = \bot$ is replaced with $C \sqcup (\geq nR.(\bot \sqcap \bigsqcap A_n))$, where the $\geq$-concept is unsatisfiable, so we can deduce C from this concept using the simplification rules (see Proposition 14). □

Corollary 3. *Let $W = C \sqcup (\geq nR.D)$ be a $\geq$-max concept in $Sat_{\mathcal{T}}$ and let $B = \{B_i\}$ be the set of bool concepts in $Sat_{\mathcal{T}}$. Then $\{D\} \cup B$ is consistent.*

Proof. Suppose $\{D\} \cup B$ is inconsistent. Then, from Proposition 15, $Sat_{\mathcal{T}}$ contains C. However, C makes W redundant, so W was eliminated from $Sat_{\mathcal{T}}$ when C was added to it. This contradicts our assumption that $W \in Sat_{\mathcal{T}}$. □

Theorem 7. *If $Sat_{\mathcal{T}}$ contains only bool concepts and $\geq$-max concepts and does not contain $\bot$, then it is consistent.*

Proof. We know from Corollary 1 that the bool concepts are satisfiable. As of the $\geq$-max concepts, at least one of their disjuncts, namely the $\geq$-disjunct can be satisfied: in each node we create separate successors for each $\geq$-concept, independent of each other (without $\leq$-concepts, these successors never need to be identified). The label of each successor is satisfiable (see Proposition 14 and Proposition 3), so the $\geq$-concept in the parent is satisfiable as well. □

$\leq$-max Concepts

We now consider a fully general saturated set $Sat_{\mathcal{T}}$, that might contain bool concepts, $\geq$-max concepts and $\leq$-max concepts. When we build the tableau tree, if a $\leq$-concept appears in the label of a node, we possibly have to add a new concept to the label of a node ($\forall$-rule) or identify two nodes ($\leq$-rule). We show that none of these rules will lead to final clash.

Each successor node is created with an initial concept in its label: for instance, if a new node is created due to concept $\geq 1R.A$, then we call A the *creator concept* of the node. Whatever other concept appears in its label (before performing any identification step), it is derived from $A \sqcap \bigsqcap B_i$, where $\{B_i\}$ is the set of bool concepts in $Sat_{\mathcal{T}}$. If a node with creator concept A has to be identified with another such that the second node contains A in its label, then identification cannot introduce new inconsistency and it can be seen as simply deleting the first node.

As previously, we are only interested in potential clashes that are final. This means that the (non-disjunctive) concepts that are involved in the clash can be assumed to be the maximal disjuncts of $\mathcal{SHQ}$-concepts from $Sat_{\mathcal{T}}$.

Proposition 16. *Let $Sat_{\mathcal{T}}$ be a saturated set of $\mathcal{SHQ}$-concepts that does not contain the empty concept $\bot$. Let us try to build a model for $\top \sqsubseteq \bigsqcap Sat_{\mathcal{T}}$ using the tableau method, observing the restrictions on the order of rules. Then we never obtain a final clash.*

Proof. We know from Theorem 7 that the set of bool concepts and $\geq$-max concepts is consistent. Hence, a final clash must involve a $(\leq nR.D)$ concept. We use induction on n, the arity of the $\leq$-concept to show that no final clash is possible. We first give a sketch of the proof:

1. In the base case of the inductive proof, we assume that we have a $(\leq 0R.D)$ concept in the label of a node, which is a $\forall$-concept. We show that no final clash is possible that would not have occurred in the absence of this $\forall$-concept.

2. In the inductive step, we assume that all $(\leq n'R.D)$ concepts that appear in the label of a node, no final clash is possible as long as $n' < n$. From this we prove that the same holds for all $(\leq nR.D)$ concepts.

A $\leq$-max concept can only lead to clash if the same label contains some $(\geq n_i S_i.A_i)$ concepts where $1 \leq i \leq l$, $S_i \sqsubseteq^* R$. We use a second, embedded inductive proof, on the number l of $\geq$-max concepts.

(a) In the base case we assume that $l = 0$ and show that no final clash is possible due to the $(\leq nR.D)$ concepts, as the examined node has no successors.

(b) In the inductive step, we assume that if a label contains $l' < l$ different $\geq$-max concepts, then the successor nodes created due to these concepts can be identified into some nodes such that at most n of them satisfies D. We show that this property holds if the label contains l different $\geq$-max concepts.

Now we fill in the details of the proof. The base case of the outer induction is when $n = 0$, that is, when we have a $\forall$-concept in the label of a node. The $\forall$-rule fires and a new concept is added to the label of some successors. To obtain contradiction, we assume that this leads to a final clash. Given a node x that has an S-successor y with creator concept A. This means that the label of x contains a concept $\geq kS.A$. Furthermore, the label of x also contains a $\forall$-concept, which is a $(\leq 0R.D)$ concept in our terminology. $S \sqsubseteq R$, so the $\forall$-rule is applicable and puts $\neg D$ in the label of y. We assumed that a clash is obtained, so $A \sqcap \neg D$ is not satisfiable. The $\geq$-concept and $\leq$-concept in the label of x originate from a $\geq$-max and a $\leq$-max concept, respectively, in $Sat_{\mathcal{T}}$, that is, $Sat_{\mathcal{T}}$ contains concepts

$$W = E \sqcup (\leq 0R.D) \qquad V = F \sqcup (\geq kS.A)$$

where $(\leq 0R.D)$ is maximal in W, $(\geq kS.A)$ is maximal in V and each disjunct in E and F leads to clash. W and V are resolvable using Rule3 and the conclusion is

$$E \sqcup F \sqcup (\geq kS.dnf(A \sqcap \neg D))$$

$A \sqcap \neg D$ is not satisfiable, so the DL calculus deduces $E \sqcup F$ as well (Proposition 15). However, we know that all disjuncts in E and F lead to clash, so we obtain a final clash without the $\leq$-concept in W. Contradiction.

We now turn to the inductive step. The inductive hypothesis is that a $\leq$-concept can never lead to final clash, i.e., a $(\leq n'R.D)$ concept in the label of a node that is derived from the maximal disjunct of a $\leq$-max concept of $Sat_{\mathcal{T}}$ can be satisfied for all $n' < n$. We show that this also holds for n.

Let some node x in the tableau tree contain concepts $(\leq nR.D)$ and $(\geq n_i S_i.A_i)$, where $1 \leq i \leq l$ and $S_i \sqsubseteq^* R$. Due to the $(\geq n_i S_i.A_i)$ concepts, we have already created $\Sigma_{i=1}^{l} n_i$ successors with creator concepts $A_1 \ldots A_l$, respectively. D appears in the label of each S_i-successor, so A_i, together with the bool concepts implies D. This means that $A_i \sqcap \neg D$ is unsatisfiable. Suppose that we have to perform identification which leads to final clash. $Sat_{\mathcal{T}}$ contains concepts

$$W = E \sqcup (\leq nR.D) \qquad W_i = F_i \sqcup (\geq n_i S_i.A_i) \quad 1 \leq i \leq l$$

where $(\leq nR.D)$ is maximal in W, $(\geq n_i S_i.A_i)$ is maximal in W_i and each disjunct of E and F_i leads to clash in x. By the time a $\leq$-rule is applied, we have already performed all possible $\bowtie$-rules, due to which the label of each S_i-successor contains either A_j or $\neg A_j$ for all $j \in \{1 \ldots l\}$. According to Corollary 3, each creator concept is satisfiable and hence will remain satisfiable by taking its conjunction with either A_i or $\neg A_i$.

We use induction on l, the number of $\geq$-concepts to show that the assumption that the $\leq$-concept gives rise to final clash leads to contradiction.

The base case (of the second, inner induction) is when $l = 0$. There are no $\geq$-concepts in the label of x, so there are no involved successors to be identified.

We now turn to the inductive step (of the inner induction). We assume that if the label of x contains only $l' < l$ different $\geq$-concepts then the resulting successors can be identified into n nodes without clash.

1. In case $n_l > n$ then Rule3 is applicable on W and W_l, resulting in:

$$E \sqcup F_l \sqcup (\geq (n_l - n)S_l.dnf(A_l \sqcap \neg D))$$

We know that $A_l \sqcap \neg D$ is unsatisfiable, so the DL calculus deduces $E \sqcup F_i$ (from Proposition 15). However, all disjuncts of E and F_i lead to clash in x, so we obtain a final clash even before introducing any $\leq$- and $\geq$-concept, contrary to our assumption.

2. If $n \geq n_l$, then concepts W and W_l are resolvable using Rule4, resulting in

$$E \sqcup F_l \sqcup (\leq (n-n_l)R.dnf(D \sqcap \neg A_l)) \sqcup (\geq 1S_l.dnf(A_l \sqcap \neg D))$$

Again, we know that $D \sqcap \neg A_l$ is unsatisfiable, so (from Proposition 15) the DL calculus deduces

$$E \sqcup F_l \sqcup (\leq (n-n_l)R.dnf(D \sqcap \neg A_l)) \tag{3.1}$$

Due to the $\bowtie$-rule, the label of every successor contains either A_l or $\neg A_l$. $n - n_l < n$, so the inductive hypothesis holds for (3.1), i.e., all the successors whose label contains both D and $\neg A_l$ can be identified into $n - n_l$ nodes by deleting some successors that are not necessary. Further to this, there are n_l successors with creator concept A_l, plus some k other successors such that the $\bowtie$-rule put A_l into their labels.

(a) If $k \leq n_l$ then we can eliminate $n_l - k$ nodes from those having A_l as their creator concept, leaving exactly n_l successors whose label contains A_l. Contrary to our assumption, we obtain no final clash.

(b) If $k > n_l$ then each of the nodes whose creator concept is A_l can be eliminated since there are more then n_l other nodes satisfying A_l. All remaining successors originate from the $\geq$-concepts in $W_1 \ldots W_{l-1}$. However, according to the inductive hypothesis (of the inner induction), these successors can be identified into n successors without clash.

This concludes the second inductive proof and the first one as well. We have showed that the assumption that a $\leq$-concept introduces inconsistency into the label of a node leads to contradiction. □

Let $\mathcal{T}$ be a $\mathcal{SHQ}$ TBox. Let $Sat_{\mathcal{T}}$ be the set of concepts obtained after performing preprocessing on $\mathcal{T}$ and then saturating it with the DL calculus. We have showed that if $Sat_{\mathcal{T}}$ does not contain $\bot$ then it is possible to build a model for $\mathcal{T}$ using the tableau algorithm. This concludes the proof of completeness for the DL calculus.

3.4.5 Towards a DL Calculus for ABox Reasoning

The DL calculus imitates the modified calculus that we presented in Subsection 3.2.2. Recall however, that the aim of that calculus was not to perform TBox reasoning, but to serve as a preprocessing phase for the ABox reasoning. The modified calculus was used to perform all inference steps that involve function symbols. Function symbols are derived via skolemisation when we translate $\geq$-max concepts to first-order clauses. The question naturally arises if the DL calculus can be used in a similar way to perform all inferences involving $\geq$-max concepts, which then can be eliminated before accessing the ABox.

Unfortunately, the answer seems to be negative, which we will illustrate through a small example. Consider the following knowledge base:

$$\begin{aligned}
&\top \sqsubseteq (\leq 1R_1)\\
&\top \sqsubseteq (\leq 1R_2)\\
&\top \sqsubseteq (\geq 1S)\\
&S \sqsubseteq R_1\\
&S \sqsubseteq R_2
\end{aligned}$$

$$\begin{aligned}
&R_1(a,b)\\
&R_2(a,c)\\
&X(b)\\
&\neg X(c)
\end{aligned}$$

The ABox satisfies the TBox as long as we neglect the only axiom in the TBox that yields a $\geq$-max concept: $\top \sqsubseteq (\geq 1S)$. In the presence of this axiom, however, b and c have to be identified into a single

S-successor of a, which leads to contradiction because b and c are not identifiable. Now, the question is: what kind of axiom(s) should the DL calculus derive from the TBox to ensure that the ABox remains unsatisfiable even if we eliminate the $\geq$-max concept? In this simple example, one could infer from the three number restrictions that $R_1 = R_2 = S$, which is sufficient to make the ABox inconsistent. Suppose however, that the axiom with the $\geq$-max concept is replaced with the following one:

$$\top \sqsubseteq C \sqcup (\geq 1S)$$

Since $(\geq 1S)(a)$ cannot be true, $C(a)$ must be true, i.e., if we query the knowledge base for concept C, then a should be returned. For this, however, we should infer an axiom expressing that *'For every individual, either it belongs to C or else all its R_1- and R_2-successors are also S-successors'*. We cannot formulate this using DL expressions.[4] The modified calculus does not suffer from this problem, because much more can be expressed using first-order clauses. Indeed, from the above TBox, the modified calculus infers the following two clauses:

$$C(x) \vee \neg R_1(x,y) \vee S(x,y)$$
$$C(x) \vee \neg R_2(x,y) \vee S(x,y)$$

which ensure the inconsistency of the ABox, even if we omit the axiom with the $\geq$-max concept.

It turns out that we need regular expressions on roles in order to be able to eliminate $\geq$-max concepts. In another example, the TBox

$$\top \sqsubseteq (\leq 2R_1)$$
$$\top \sqsubseteq (\leq 2R_2)$$
$$\top \sqsubseteq (\geq 1S)$$
$$S \sqsubseteq R_1$$
$$S \sqsubseteq R_2$$

is equisatisfiable to the following one:

$$\top \sqsubseteq (\leq 2R_1)$$
$$\top \sqsubseteq (\leq 2R_2)$$
$$S \sqsubseteq R_1$$
$$S \sqsubseteq R_2$$
$$\top \sqsubseteq (\leq 1(R_1 \sqcap \neg R_2)$$
$$\top \sqsubseteq (\leq 1(R_2 \sqcap \neg R_1)$$

Allowing regular expressions on roles, however, leads to the undecidability of the language in general. It seems very difficult to extend the DL calculus in this direction. Hence, we conclude that eliminating $\geq$-max concepts before accessing the ABox is not likely to succeed without recourse to first-order logic. The DL calculus can be used for TBox reasoning, however, it is not adequate for the two-phase data reasoning that we discussed in Section 3.2.

3.4.6 Summary

We have presented the DL calculus, a resolution based algorithm for deciding the consistency of a $\mathcal{SHQ}$ TBox. The novelty of this calculus is that it is defined directly on DL axioms. We showed that the algorithm is sound, complete and terminates. More work needs to be done to explore the real time complexity of the reasoning, as well as potential optimization techniques. We hope that further research will reveal that the DL calculus provides a reasonable alternative to the Tableau Method for certain reasoning tasks.

We have not been successful in extending the DL calculus for ABox reasoning in the way the modified calculus is used. In Subsection 3.4.5 we illustrated through some examples why we believe that this extension is not possible at all.

[4] At least not without a significant increase in the expressivity of the DL language.

Chapter 4

Loop Elimination, a Sound Optimisation Technique for PTTP Related Theorem Proving

In Section 2.1 we presented the Prolog Technology Theorem Prover (PTTP), which is a complete first-order theorem proving technique built on top of the Prolog language. The DLog system [38] that will be presented in Chapter 5 is a specialisation of PTTP to Description Logic reasoning. DLog performs a two-phase reasoning, where the first phase is that presented in Section 3.2 and the second phase uses PTTP. These systems exploit the backtracking mechanism of Prolog to search for a proof of the initial goal. Efficiency is crucial since these systems typically need to explore a huge search space.

Loop elimination is an optimisation technique which can make a tremendous impact on the speed of both of the aforementioned systems. This technique prevents logic programs from trying to prove the same goal over and over again, thus avoiding certain types of infinite loops. My main contribution to this domain is a rigorous proof of soundness of loop elimination.

Detecting loops to prune the search space for logic programs is not new, see for example [8]. However, the systems that we are interested in extend standard Prolog execution with a technique called *ancestor resolution*, that corresponds to the positive factoring inference rule. In the presence of ancestor resolution, the considerations that trivially justify loop elimination do not hold. It is easy to see that trying to prove a goal that is identical to some goal that we are already in the process of proving yields no useful solution and the corresponding proof attempt can be aborted. However, it is far from trivial that the same holds in case the two goals are identical only *modulo ancestor list*, i.e., they can be different in one of their arguments, namely in their list of ancestors. In this chapter we prove this stronger claim. We are not aware of any other work exploring the interaction between loop elimination and ancestor resolution.

In Section 4.1 we examine logic programs in terms of termination and identify the sources of infinite execution. Section 4.2 contains our main contribution: we define loop elimination and prove its soundness. We end the chapter with some concluding remarks in Section 4.3.

4.1 Termination of Logic Programs

Given that first-order logic is undecidable, it is not surprising that the Prolog Technology Theorem Prover is not guaranteed to terminate. In this section we review the ways in which a logic program can fall short of termination. Afterwards, we compare PTTP and DLog with respect to termination.

4.1.1 Sources of infinite execution

We identify three sources of infinite execution:

- If the program contains **function symbols**, then we might obtain terms of ever increasing depth. Consider, for example, the following simple program:

```
p(X)      :-  p(f(X)).
```

  If we attempt to prove $p(a)$ using the above rule, we will end up reducing it to the proof of $p(f(a))$, $p(f(f(a)))$ etc. and the program will never stop.

- A proof attempt might visit infinitely many goals if an unbounded number of **new variables** can be introduced during the proof. This happens with rules with a variable occurring in the clause body, but not in the head. For example, consider the transitivity rule:

```
r(X,Y)    :-  r(X,Z), r(Z,Y).
```

  It is easy to see that a proof attempt for the goal $r(a,b)$ using the above rule will generate infinitely many $r(a,V)$ subgoals, always with a fresh variable.

- Even if both the depth of terms and the number of variables can be bounded, the program might fall into a **loop** and attempt to prove the same goal over and over again. For example, the program consisting of the following rule

```
p(X)      :-  p(X).
```

  will never terminate, even though there are no function symbols and no new variables are introduced.

One can see easily that the above list is exhaustive. If the number of variables is bounded and there are no functions, then the total set of terms is that of the variables and the constants appearing in the program, i.e., it is finite. Since the set of predicate names is also finite, there can be finitely many different goals. If there are no loops, even if a proof attempt goes though all possible goals (the worst case), it will eventually terminate.

Hence, we conclude that infinite execution is due exactly to three aspects of logic programs: function symbols, the proliferation of new variables and loops.

4.1.2 Termination in DLog

In light of the preceding subsection, let us reexamine the input clause set of the second phase of the DLog data reasoner. We repeat this set here:

$$\neg R(x,y) \vee S(y,x) \tag{c11}$$
$$\neg R(x,y) \vee S(x,y) \tag{c12}$$
$$\mathbf{P}(x) \tag{c13}$$
$$\mathbf{P_1}(x) \vee \bigvee_i (\neg R(x,y_i)) \vee \bigvee_i \mathbf{P_2}(y_i) \vee \bigvee_{i,j} (y_i = y_j) \tag{c14}$$
$$(\neg) R(a,b) \tag{c15}$$
$$C(a) \tag{c16}$$
$$a = b \tag{c17}$$
$$a \neq b \tag{c18}$$

We see immediately that the absence of function symbols eliminates one of the three sources of infinite execution.

We shall see that new variables are not introduced, either. The second nice property of the input clause set is that the resulting contrapositives only contain a negative binary literal in the body in case the head is a negative binary literal. This means that we can only encounter negative binary subgoals if the initial query itself is a negative binary goal. In $\mathcal{SHIQ}$ DL reasoning, however, negative binary queries are

forbidden, so all contrapositives with a negative binary literal are unnecessary and can be disposed of. Consequently, in our logic program binary literals will only appear positively. For proving such binary goals only contrapositives from clauses of type c1 and c2 are available:

```
r(X,Y)     :-  s(X,Y).
r(X,Y)     :-  s(Y,X).
```

These rules do not introduce new variables. A proof of a binary goal consists of applying such rules possibly several times, until finally we obtain a matching data assertion `r(a,b)`, thanks to which the variables in the binary goal get instantiated. We know that in all rule bodies that contain binary literals every variable occurs in some binary literal (the third nice property of our input clause set). These are the rules that introduce new variables. If, however, we move the binary literals to the front of the body, i.e., we prove the binary goals first, by the time we reach the unary goals, they become ground. Hence, any unary goal in the body either contains the same variable as the one in the head – in case the rule contains no binary predicates – or else it is ground by the time it is called. New variables may appear only for a short time – until we prove the binary goals holding them. Hence, DLog will never encounter infinitely many new variables during a proof attempt.

If there are no terms of increasing depth and variables do not proliferate, then the only way a DLog program may not terminate is if it falls in an infinite loop and proves the same goal repeatedly.

4.1.3 Eliminating Loops

We have seen that there are three independent features that can make a PTTP execution non-terminating, of which only one, namely loops can occur in DLog programs. In Section 4.2 we shall show that proofs containing such loops are not necessary for completeness. This result yields an important optimization for both PTTP and DLog, called *loop elimination*. General PTTP still has to cope with infinite proof attempts (due to the other two sources) and hence has to use iterative deepening, i.e., build several proof attempts in parallel. However, even if loop elimination does not allow for changing the proof search strategy, but it still prunes the search space significantly. In DLog, loop elimination eliminates the only remaining source of infinite proofs. Accordingly, DLog always terminates and uses the standard depth-first search strategy of Prolog, which gives much better performance than iterative deepening.

4.2 Loop Elimination

In this section we present the optimization heuristic *loop elimination* for both PTTP and DLog. In the literature, loop elimination is often referred to as *identical ancestor pruning*, see for example [51] or [20]. Although both PTTP and DLog employ this optimisation, there has not yet been any rigorous proof of its soundness. In Subsection 4.2.1 we describe *proof trees* that can be used to represent Prolog execution. Afterwards, Subsection 4.2.2 contains the proof of soundness.

Definition 12 (Loop elimination)**.** *Let P be a Prolog program and G a Prolog goal. Executing G w.r.t. P using* loop elimination *means the Prolog execution of G extended in the following way: we stop the given execution branch with a failure whenever we encounter a goal H that is identical to an open subgoal (that we started, but have not yet finished proving). Two goals are identical only if they are syntactically the same.*

Loop elimination is very intuitive. If, for example, we want to prove goal *G* and at some point we realise that this involves proving the same goal *G*, then there is no point in going further, because 1) either we fall in an infinite loop and obtain no proof or 2) we manage to prove the second occurrence of *G* in some other way that can be directly used to prove the first occurrence of the goal *G*. This is the standard justification that we find in the literature. For example [20] says:

> Identical ancestor pruning (IAP) is a powerful pruning heuristic in a model elimination search. Imagine, in the course of expanding a ME proof space for a particular goal P, that one were to encounter that same goal P again. One of two situations must hold:

1. There are no proofs of P from this database (because it doesn't logically follow).
2. Whether or not there is a proof using this second occurrence of P, there must be another proof of the original P not using it. Also, the different proof occurs at a shallower depth.

This is true because the second occurrence must eventually be proven somehow, so this recursion must bottom out. And then, by whatever proof this second occurrence succeeds, an analogous proof path must exist below the first occurrence of P. In either case, it is justifiable to prune the space below the second occurrence of P.

Things get complicated, however, due to ancestor resolution. The two *G* goals have different ancestor lists and it can be the case that we only manage to prove the second *G* due to the ancestors that the first *G* does not have. As it will turn out in the rest of this section, while we can indeed construct a proof of the first *G* from that of the second, this proof might have to be very different from the original one.

4.2.1 Proof Trees

In this subsection we introduce *proof trees*, that are used to represent Prolog execution. We will only consider trees in the context of a PTTP like Prolog program, more precisely we will assume that the program contains all contrapositives. Each tree node has a unique name and is labelled with a goal: `(Name:Goal)` refers to a node called `Name` and labelled with goal `Goal`. The root is labelled with the initial goal to be proved. Suppose the current goal *G* is unified with the head of rule

$$\mathtt{G} :- \quad \mathtt{B_1}, \mathtt{B_2}, \ldots, \mathtt{B_k}.$$

In this case, the node labelled *G* will have *k* children, each labelled $B_1, B_2, \ldots, B_k$, respectively. In each inference step, the validity of a goal is reduced to the validity of a set of goals in the children. After a successful execution, we obtain a proof tree such that each of its leaves can be considered true without further proof. We formalise this in the following definitions.

Definition 13 (atomic proof tree)**.** *An* atomic proof tree *consists of a root node labelled* $A\sigma$ *with children labelled* $B_1\sigma$, $B_2\sigma$, …, $B_n\sigma$, *where* σ *is a variable substitution. We say that the atomic proof tree is* valid *if the corresponding Prolog program contains a rule*

$$\mathtt{A} :- \quad \mathtt{B_1}, \mathtt{B_2}, \ldots, \mathtt{B_n}.$$

A valid atomic proof tree can be seen as an instance of a rule. A proof tree *is built from atomic proof trees by matching nodes of identical labels. A proof tree is* valid *if all constituting atomic proof trees are valid.*

Remark 1. *The labels of proof trees are atomic predicates that can contain variables. Note that labels* $p(X)$ *and* $p(Y)$ *are not identical.*

Definition 14 (complete node)**.** *In a valid proof tree, a node labelled A is called* complete *if either 1) A can be unified with the head of a bodiless Prolog rule or 2) the node has an ancestor labelled* $\neg A$ *(ancestor resolution). A valid proof tree is complete if all its leafs are complete.*

To each successful Prolog execution that employs ancestor resolution, we can assign a complete proof tree.[1] In fact, the execution mechanism can be seen as a search in the space of complete proof trees. While standard Prolog will not necessarily traverse the whole space (because it might fall into an infinite loop), both PTTP and DLog are built so that they can enumerate all complete proof trees. This means that it is enough to show the existence of a complete proof tree to guarantee a successful PTTP or DLog execution.

Definition 15 (flipping along a child)**.** *For an arbitrary child b of an atomic proof tree, the transformation* flipping over along the *b* child *is defined as follows: the root node is switched with its child b and their labels are negated. The rest of the tree is unaltered. This transformation is illustrated in Figure 4.1.*

[1] In the Logic Programming community, it is customary to reserve the name proof tree only for complete proof trees. We introduce the notion of completeness because we will have to refer to trees that are not fully expanded.

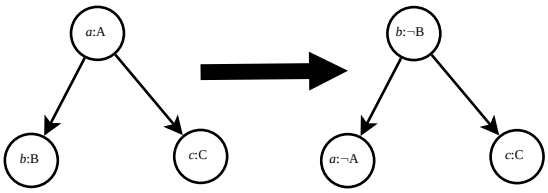

Figure 4.1: Flipping over along the b child

Lemma 1. *For every valid atomic proof tree, the atomic tree obtained after flipping over along a child results in a valid atomic proof tree.*

Proof. Let T be an atomic proof tree with the root node labelled $A\sigma$ and children labelled $B\sigma, C_1\sigma, \ldots, C_k\sigma$. T is an instance of the Prolog clause

$$\mathtt{A} :- \quad \mathtt{B}, \mathtt{C_1}, \ldots, \mathtt{C_k}.$$

which is a contrapositive of the first-order clause $A \vee \neg B \vee \neg C_1 \vee \cdots \vee C_k$. Since the Prolog program contains all contrapositives of this clause, we also have

$$\mathtt{not\_B} :- \quad \mathtt{not\_A}, \mathtt{C_1}, \ldots, \mathtt{C_k}.$$

an instance of which corresponds to the flipped over version of T. □

Note that flipping over allows us to move between contrapositives of the same first-order clause.

Definition 16 (flipping along a branch)**.** *The transformation* flipping over along the $a, \overline{a}$ branch *is defined on proof trees as follows: let F be a proof tree, with a node $(a : A)$ which has a leaf descendant $(\overline{a} : \neg A)$. The nodes on the path from a to $\overline{a}$ are $a = x_0, x_1, \ldots, x_{n-1}, x_n = \overline{a}$. To this tree we assign a tree F' which differs from F only in the subtree rooted at a. This subtree contains a branch $y_0 = x_n, y_1 = x_{n-1}, \ldots, y_i = x_{n-i}, \ldots, y_n = x_0$, and the label of each of these nodes is negated. Furthermore, each y_i in F' has the same siblings as x_{n-i+1} in F. The subtrees under the siblings are left unaltered. This transformation is illustrated in Figure 4.2.*

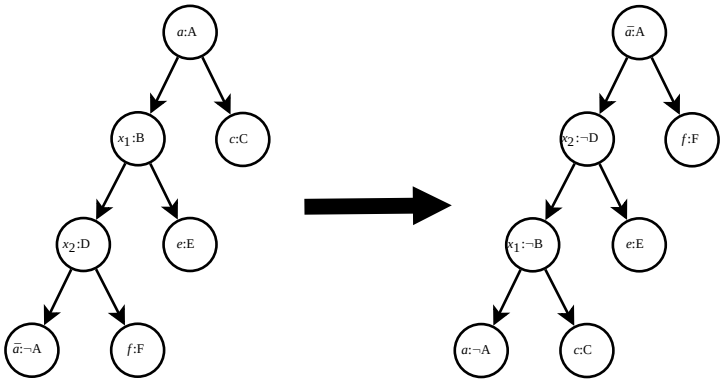

Figure 4.2: Flipping over along the $(a, \overline{a})$ branch

Lemma 2. *If we have a complete proof tree T that contains nodes $(a : A)$ and $(\overline{a} : \neg A)$ such that $\overline{a}$ is a leaf descendant of a, then the tree obtained after flipping T along the $(a, \overline{a})$ branch is a valid proof tree.*

Proof. The new downward path $\overline{a} \to a$ consists of atomic trees that are the flipped over versions of the atomic trees of the initial upward path $\overline{a} \to a$. For example, the atomic tree on the left side of Figure 4.2 that consists of parent node x_2 and children $\overline{a}$ and f turns into a flipped atomic tree with parent node $\overline{a}$ and children x_2 and f. We know from Lemma 1 that flipping over a valid atomic proof tree yields another valid atomic proof tree, hence the whole new proof tree is valid. □

Remark 2. *Although we obtained a valid proof tree after flipping over, the proof tree is not necessarily complete. This is because some ancestor lists change and branches that previously terminated in ancestor resolution might have to be expanded further (because the required ancestor disappeared).*

4.2.2 The Soundness of Loop Elimination

In this subsection we show that for every complete proof tree that contains loops, one can construct a complete proof tree that is loop free.

Definition 17 (loop in proof tree)**.** *A complete proof tree is said to contain a loop L if it contains a pair of nodes $(p_1 : P), (p_2 : P)$, for some label P, such that p_2 is a descendant of p_1. Node p_1 is called the* top node *and node p_2 the* bottom node *of the loop L. We define the* depth *of L to be the distance of p_1 from the root.*

Definition 18 (bad node)**.** *A node $n : N$ is said to be* eligible for ancestor resolution *if it has an ancestor with label $\neg N$. If an inner node is eligible for ancestor resolution, then it is called a* bad node.

Bad nodes are called bad, because they are unnecessarily expanded. There is no need to provide a proof tree under a bad node, since it is complete even if it remains a leaf.

Lemma 3. *If we have a complete proof tree that contains a bad node n, then the tree obtained after removing the subtree under n yields a complete proof tree in which n is not bad any more.*

Proof. Removing the subtree under n makes n a leaf node. However, n is complete due to ancestor resolution. The rest of the leaves are unaltered, so they remain complete. Hence, the new proof tree is complete. □

Definition 19 (loop-depth)**.** *We define the* loop-depth *of a tree T with a pair of integers $(-D, C)$, where D is the minimum depth of all loops in T and C is the number of nodes that are bottom nodes of some loop of depth D. If the tree contains no loops, then its loop-depth is $(-\infty, 0)$. Loop-depths are comparable using lexicographic ordering, i.e., loop-depth (A, B) is less than loop-depth (C, D) if and only if either $A < C$ or else $A = C$ and $B < D$.*

Lemma 4. *Let F be a complete proof tree with loop-depth LD that contains at least one loop. It is possible to find another complete proof tree F' for the same goal (i.e., with the same label in the root) such that the loop-depth of F' is strictly less than LD.*

Proof sketch. We pick a loop of greatest depth and try to get rid of it.

1. First, we eliminate bad nodes from the proof tree. If this eliminates the loop, we are ready.

2. Next, we try to replace the proof at the top of the root with the proof at the bottom of the loop. If this results in a valid proof tree, then we are again ready.

3. If the proof at the bottom cannot be moved to the top (due to ancestor resolution), then we flip the tree along the branch that connects the two ends of the loop. We obtain a valid proof tree which, however, is not necessarily complete.

4. If a node a becomes incomplete after flipping, this is because it loses an ancestor that previously allowed for ancestor resolution. In this case, however, we show that there is another node b in the tree with the same label, and the proof tree rooted at b can be copied under a to make it complete.

5. It can be shown that finitely many subtree copying results in a complete proof tree whose loop-depth is greater than that of the initial tree.

□

Proof. The loop-depth of F is LD=$(-D,C)$. This means that there is at least one loop of depth D and there are no loops with depth less than D. Let L be one such loop with top and bottom nodes $(p_1 : P)$ and $(p_2 : P)$, respectively. First, we eliminate all bad nodes by removing the subtrees rooted at the bad nodes. According to Lemma 3, the result is still a complete proof tree.

In case the elimination of the subtrees under bad nodes eliminates loop L, then the obtained complete proof tree has loop-depth $(-D_2,C_2)$. In case there were no other loops of depth D in F then $D_2 > D$. Otherwise, $D_2 = D$ and $C_2 = C - 1$. In either case $(-D_2,C_2) < (-D,C)$, so our lemma is satisfied.

Otherwise, in the obtained tree, all nodes that are eligible for ancestor resolution are leaf nodes. The ancestor list of p_2 contains the ancestors of p_1 plus the nodes on the path between p_1 and p_2. Let *ANC* denote the set of nodes between p_1 and p_2.

In case none of the nodes in *ANC* play any role in the proof of p_2 (i.e., they do not participate in ancestor resolution), the proof of p_1 can be directly replaced with that of p_2, eliminating loop L. This is illustrated in Figure 4.3. We obtained a complete proof tree F' and one of the loops at minimum depth was eliminated. The new loop-depth is less than the initial, so our lemma is satisfied.

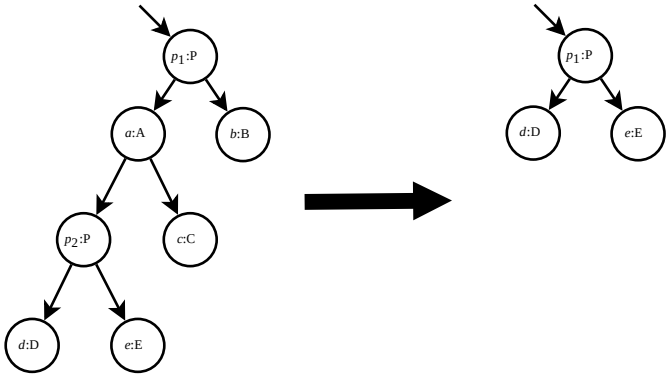

Figure 4.3: Replacing the proof of p_1 with that of p_2

The situation is more complicated when some nodes in *ANC* participate in ancestor resolution under p_2. Among these, let $(a : A)$ be the lowest one (i.e., the last one to enter the ancestor list). Somewhere under p_2 there is a leaf $(\overline{a} : \neg A)$ that is complete due to ancestor resolution. Let us flip over F along the branch $(a,\overline{a})$. In the flipped over branch the nodes between a and $\overline{a}$ will appear with negated labels and in inverse order. Afterwards, we once more eliminate all bad nodes by removing the subtrees under them. Node p_2 is on the path between a and $\overline{a}$, so its label will turn to $\neg P$, which makes p_2 eligible for ancestor resolution. Hence, when we eliminate badness, we eliminate the subtree under p_2. As a result, loop L disappears. An example of this is shown in Figure 4.4. We know that flipping a complete proof tree results in a valid proof tree, but it is not necessarily complete, because some goals that previously succeeded with ancestor resolution might loose the required ancestor (cf. Remark 2). This is the case when there is a node $(b : B)$ under a and somewhere underneath there is a leaf $(\overline{b} : \neg B)$. Node b has to be on the path between a and $\overline{a}$ otherwise b will continue to be an ancestor of $\overline{b}$ and their labels will not change. There are two possibilities:

1. As it is illustrated in Figure 4.5, b lies between a and p_2. Then, $\overline{b}$ cannot appear under p_2, because a was chosen to be the lowest node participating in ancestor resolution under p_2. Hence, b appears under b, but not under p_2. After flipping, both b and $\overline{b}$ will appear under p_2, so they will be eliminated when we eliminate the badness of p_2. Hence, this case will not yield any incomplete leaves.

2. We illustrate the second case, namely when b is under p_2 in Figure 4.6. We will treat all such nodes together, i.e., let $(b_1 : B_1),(b_2 : B_2),\ldots(b_k : B_k)$ be nodes on the path between a and $\overline{a}$ (nodes b,c on Figure 4.6), such that each b_i has at least one leaf descendant $(b_{il} : \neg B_i)$. The nodes are ordered so that b_1 is the closest to p_2 and b_k is the farthest. After flipping over, the labels of these nodes will be negated, i.e., turn to $\neg B_i$, respectively, and they will appear on the branch leading to p_2 in inverted order, i.e., b_k will be the topmost, while b_1 the lowest.

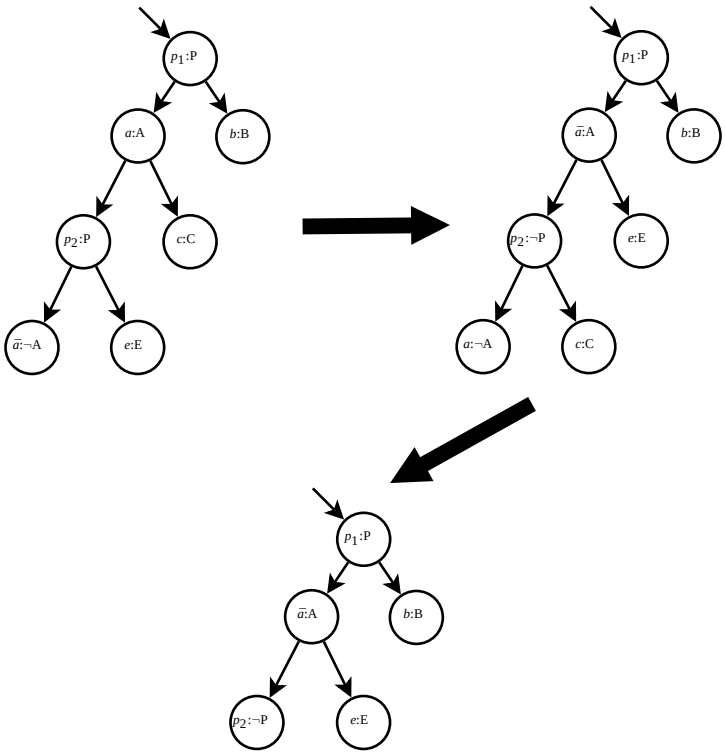

Figure 4.4: Flipping over along the $(a, \bar{a})$ branch, then bad node elimination

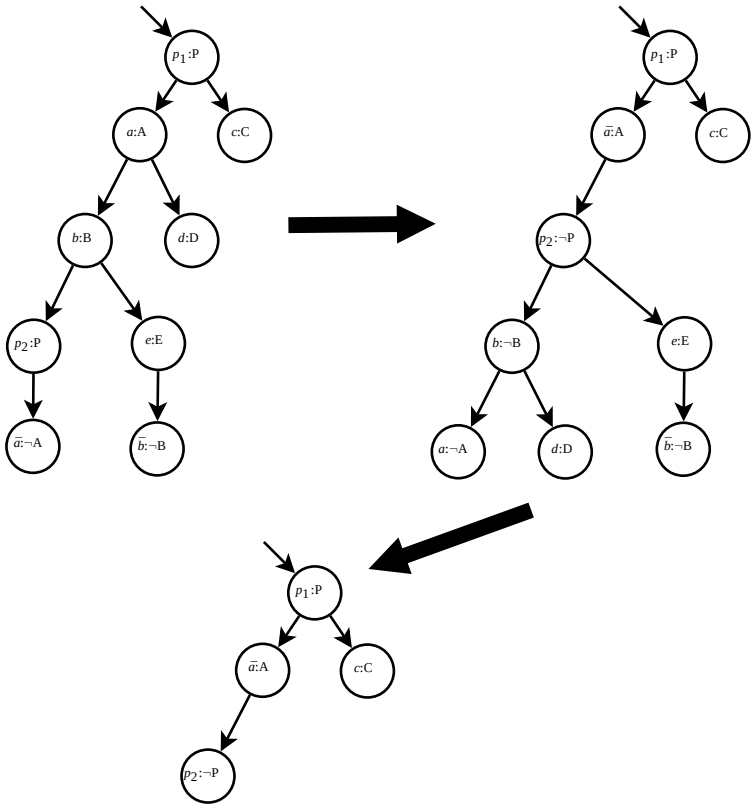

Figure 4.5: Ancestor resolution eliminates both b and $\bar{b}$

Let us consider b_1. Due to flipping over, it will lose all its previous descendants. Its new descendants will be its previous ancestors on the path between p_2 and b_1 along with their descendants towards other branches. We claim that none of the new descendants of b_1 can have lost an ancestor which previously allowed for ancestor resolution, i.e., none can be one of $\overline{b_{il}}$. This is because the lost ancestor would have been above b_1, however, b_1 was chosen to be the topmost one. Consequently, the subtree under b_1 after flipping has no incomplete leaves. This subtree in itself is not necessarily complete, because the ancestors of $\overline{a}$ might be needed for some ancestor resolution steps. We express this by saying that the subtree under b_1 is *complete in the context of the ancestors of* $\overline{a}$. In the following, we will always assume the same context (the ancestors of $\overline{a}$) and will omit specifying it whenever it leads to no misunderstanding. The label of b_1 is $\neg B_1$, so we have a complete proof for $\neg B_1$ (again in the context of the ancestors of $\overline{a}$). This means that we can copy the subtree under b_1 to any node $(\overline{b_{1l}} : \neg B_1)$, thus compensating such nodes for the lost ancestor. Note that we need to rename the copied nodes to ensure that each node has a unique name.

We next turn to b_2. Through analogous reasoning we can see that the new leaf descendants of b_2 are either complete or else are incomplete because they lost an ancestor labelled $\neg B_1$. However, by copying the subtree under b_1, we have already turned such leaves into complete trees. Hence, we have a complete proof tree under b_2 (in the context of $\overline{a}$), proving $\neg B_2$, which we copy to any incomplete leaf $(\overline{b_{2l}} : \neg B_2)$ (again assigning new names to the newly created nodes).

We continue the process. In the i^{th} step, we have a complete proof tree under b_i which we copy to any leaf $(\overline{b_{il}} : \neg B_i)$. By the end of the k^{th} step, we obtain a complete proof tree. Note that we make exactly one copying for each leaf $\overline{b_{il}}$ that lost its completeness after flipping over, so copying terminates.

We now obtained a new proof tree F'. Let us show that F' has the properties claimed by the lemma being proved. Flipping over turns the label of p_2 from P to $\neg P$, which makes loop L disappear. New loops can arise (some nodes were negated), however, no such loop can start above or at p_1. We show this by contradiction. Suppose a node $(n_1 : N)$ above or at p_1 obtains a descendant $(n_2 : N)$ after flipping. The labels of the nodes under n_1 in the new tree are either the same or the negated labels that appeared under n_1 before flipping. So, if a new loop appeared, it was either because the bottom node of an already existing loop L_2 was copied or because the label of a descendant of n_1, namely of n_2, changed from $\neg N$ to N. In the first case, the depth of loop L_2 is smaller than the depth of loop L, which is impossible because L was chosen to be a loop of minimum depth (cf. Definition 19. of loop-depth). In the second case, before flipping over, n_2 was eligible for ancestor resolution. Since we eliminated all bad nodes, n_2 was a leaf. However, flipping over does not negate the labels of leaf nodes, so we obtained a contradiction.

We conclude that the possibly arising loops are all of greater depth than the eliminated loop. Hence, the number of loops of depth D is reduced by one, i.e., the loop-depth of the new tree is strictly less than that of the original tree. □

Theorem 8. *For every complete proof tree containing loops there is a complete proof tree that is loop free.*

Proof. Using the transformation described in Lemma 4, we can create a series of proof trees of the same goal such that the loop-depth is always decreasing. The second component of the loop-depth is a positive integer (the number of loops at minimum depth) which cannot decrease infinitely, so eventually the first component will decrease as well. This means that the minimum depth of the loops increases, i.e. loops get deeper and deeper. There are two possibilities:

1. Eventually, we manage to eliminate each loop after a finite number of iterations. The resulting proof tree satisfies our theorem.

2. The elimination never terminates. Since the loops are getting farther from the root, it follows that the part of the proof tree that is loop free grows beyond any limit. Suppose the initial tree contains n distinct labels in its nodes. The transformation steps involve flipping over, copying subtrees and eliminating nodes, each of which either preserves node labels or introduces the negation of some

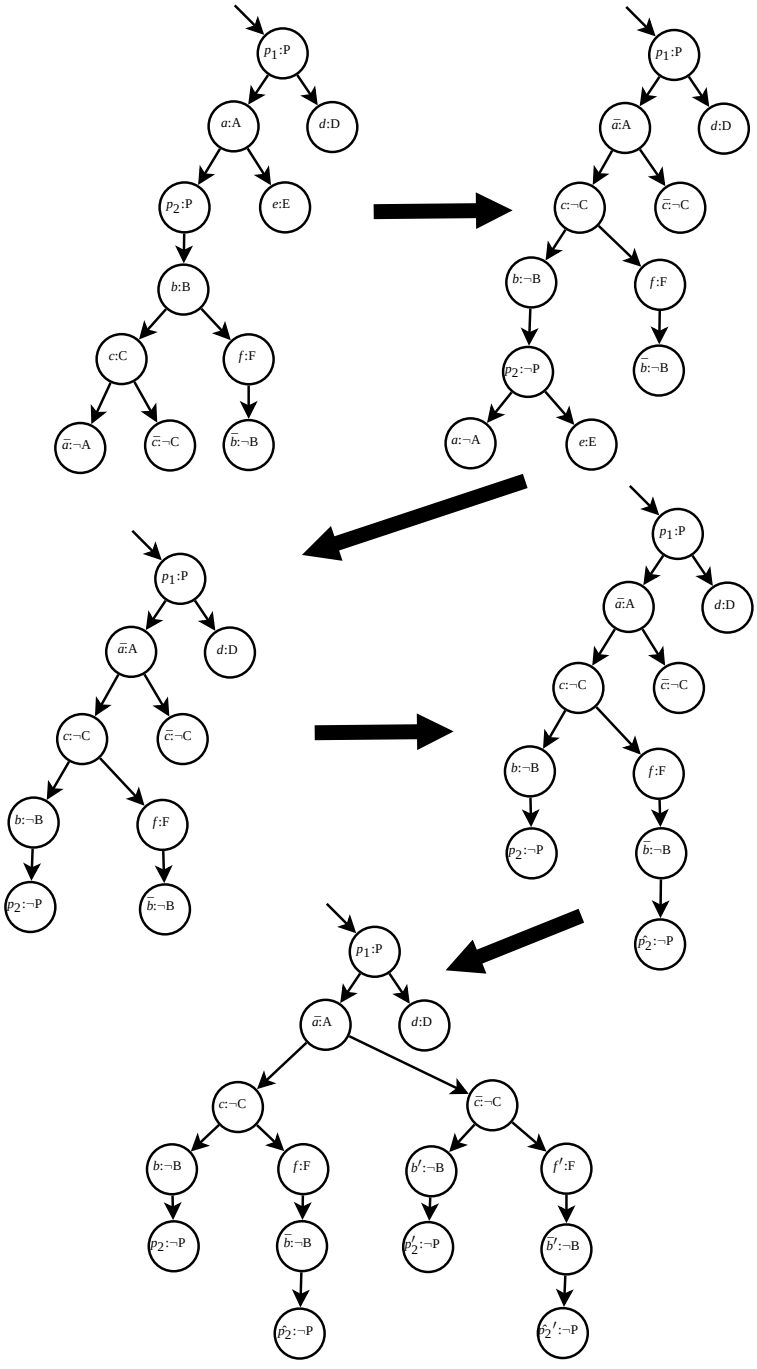

Figure 4.6: Copying makes first $\bar{b}$, then $\bar{c}$ complete

label to a node. Hence, there can be at most $2n$ distinct labels, i.e., any loop free path from the root node can be at most $2n$ long. This contradicts the assumption that the loop free part of the tree grows beyond any limit. Hence, all loops have to disappear after finitely many iterations.

□

4.3 Summary

Prolog based inference systems like PTTP and DLog can be used to prove a query goal. We have shown is Section 4.2 that these systems need not explore proof trees that contain loops, because in case there is a complete proof tree, there is one without loops (Theorem 8). This allows for reducing the search space, making both systems faster. Besides, loop elimination is sufficient to make the DLog reasoner terminating, thus allowing one to replace iterative deepening search with depth-first search, which further increases performance.

Chapter 5

The DLog Description Logic Reasoner

The DLog system [38] is a DL data reasoner, written in the Prolog language, which implements a two-phase reasoning algorithm based on first-order resolution, and it supports the $\mathcal{RIQ}$ language. As described in Chapter 3, the input knowledge base is first transformed into function-free clauses of first-order logic. The clauses obtained from the TBox after the first phase are used to build a Prolog program. It is the execution of this program – run with an adequate query – that performs the second phase, i.e., the data reasoning. The second phase is focused in that it starts out from the query and only accesses parts of the ABox that are relevant to answering the query. The relevant part is determined by the clauses derived from the TBox. Hence, the performance of DLog is not affected by the presence of irrelevant data. Furthermore, the ABox can be accessed through direct database queries and needs not be stored in memory. To our best knowledge, DLog is the only DL reasoner which does not need to scan through the whole ABox. Thanks to this, DLog can be used to reason over really large amounts of data stored in external databases. The last stable version of DLog that supports the $\mathcal{SHIQ}$ language is available at `http://dlog-reasoner.sourceforge.net`.

In Section 5.1 we give an overview of the architecture of the system. Afterwards, Section 5.2 discusses more in depth the implementation of the TBox saturation module, which performs the first phase of reasoning. In Section 5.3 we collect the most important tasks that still need to be done to make DLog usable in practical applications. Finally, Section 5.4 summarises our work in the DLog project.

5.1 Architecture of the DLog System

Figure 5.1 gives an overview of DLog. The system can be used both as a server and as a standalone application. It communicates through the DIG [7] interface, which is a standardised, XML based interface for Description Logic Reasoners. The input has three parts: the ABox which can be potentially huge, the TBox which is typically much smaller and the user queries. The ABox is left unmodified and is asserted into the Prolog module `abox`. The ABox can also be provided as a database, which is crucial for really large data sets. The content of the TBox is first transformed by the TBox saturation module into a set of function free clauses, which are next compiled into Prolog clauses using a specialised PTTP transformation, and are asserted into module `tbox`. The last part of the input contains the user queries. These are instance retrieval queries or their conjunctions. The generated Prolog program is run with the provided query as argument and returns all solutions through a backtracking search.

The first reasoning phase is independent from the ABox and from the query. Hence, as long as the TBox is unchanged, it is sufficient to perform the first phase only once, as a preprocessing step. For this reason, its speed is not critical as it does not affect the response time of the system when answering queries.

This dissertation only deals with the first reasoning phase, performed by the TBox saturation module. For a thorough description of the whole DLog system and in particular the Prolog code generation module, see [38].

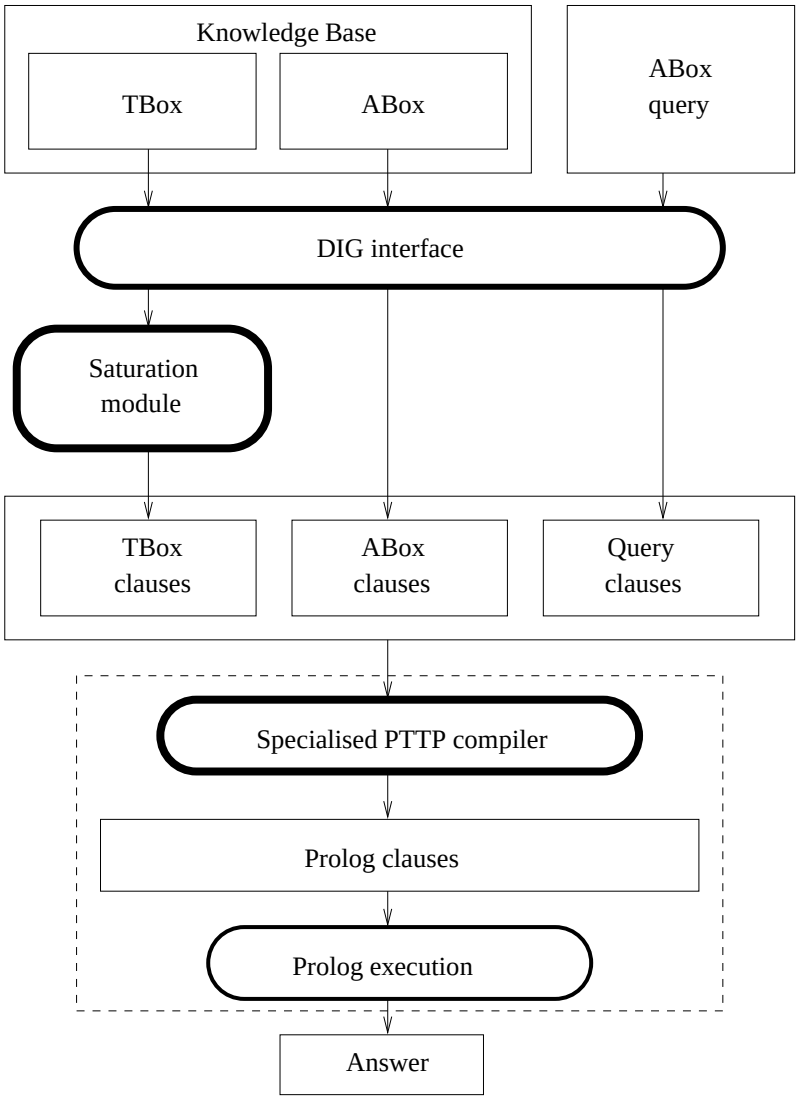

Figure 5.1: Architecture of the DLog system

5.2 Terminology Reasoning – the First Phase

The TBox saturation module takes the TBox part of the input and transforms it to first-order clauses of the following types:

$$\neg R(x,y) \vee S(y,x) \tag{c11}$$

$$\neg R(x,y) \vee S(x,y) \tag{c12}$$

$$\mathbf{P}(x) \tag{c13}$$

$$\mathbf{P_1}(x) \vee \bigvee_i (\neg R(x,y_i)) \vee \bigvee_i \mathbf{P_2}(y_i) \vee \bigvee_{i,j} (y_i = y_j) \tag{c14}$$

The transformation proceeds as described in Section 3.2 and Section 3.3 and this constitutes the first phase of reasoning. The output clauses have a rather simple syntax, which allows for using a highly optimised variant of PTTP in the subsequent data reasoning, where these clauses and the ABox are transformed into a Prolog program. The most important benefit of the TBox saturation is that there are no function symbols left in the knowledge base.

The first phase is implemented in the Prolog predicate `axioms_to_clauses/2`, which takes a $\mathcal{RIQ}$ knowledge base and generates clauses of types (c11) – (c14), through a series of transformation steps, as shown in Figure 5.2.

First, we eliminate from the TBox the complex role hierarchies, as described in Section 3.3 and obtain a set of $\mathcal{ALCHIQ}$ axioms. The predicate call

```
transitive:riq_to_alchiq(+RIQAxioms,-RBox,-ALCHIQGCIs)
```

results in a set of $\mathcal{ALCHIQ}$ GCIs and an RBox that contains neither transitivity axioms nor complex role inclusion axioms.

This is followed by internalisation and normalisation, yielding a set of $\mathcal{ALCHIQ}$ concepts.

```
dl_to_fol:axiomsToNNFConcepts(+ALCHIQGCIs,-NNF)
```

The semantics of these concepts is that all individuals of an interpretation have to satisfy all the concepts in order for the interpretation to be a model of the TBox.

Afterwards, we eliminate the nesting of composite concepts into each other, by introducing new concept names for embedded concepts.

```
dl_to_fol:defNormForms(+NNF,-Defs)
```

This is called structural transformation.

Next, we translate our concepts into first-order logic:

```
dl_to_fol:toFOLList(+Defs,-FOL1)

dl_to_fol:toFOLList(+RBox,-FOL2)

append(+FOL1,+FOL2,-FOL)
```

The first-order formulae are turned into first-order clauses:

```
dl_to_fol:list_cls(+FOL,-FOLClauses)
```

We obtain a set of $\mathcal{ALCHIQ}$ clauses (see Figure 2.4), i.e., they are of type (c1) – (c7).

This is followed by the real reasoning phase: the saturation of the $\mathcal{ALCHIQ}$ clauses by the modified calculus presented in Subsection 3.2.2.

```
saturate:saturate(+FOLClauses,-Saturated)
```

After saturation, no more inference steps can be performed using clauses containing function symbols, hence they can be eliminated.

```
eliminate_functions(+Saturated,-FunFree)
```

The remaining clauses are passed over to the Prolog translator module which builds a Prolog program from them based on PTTP.

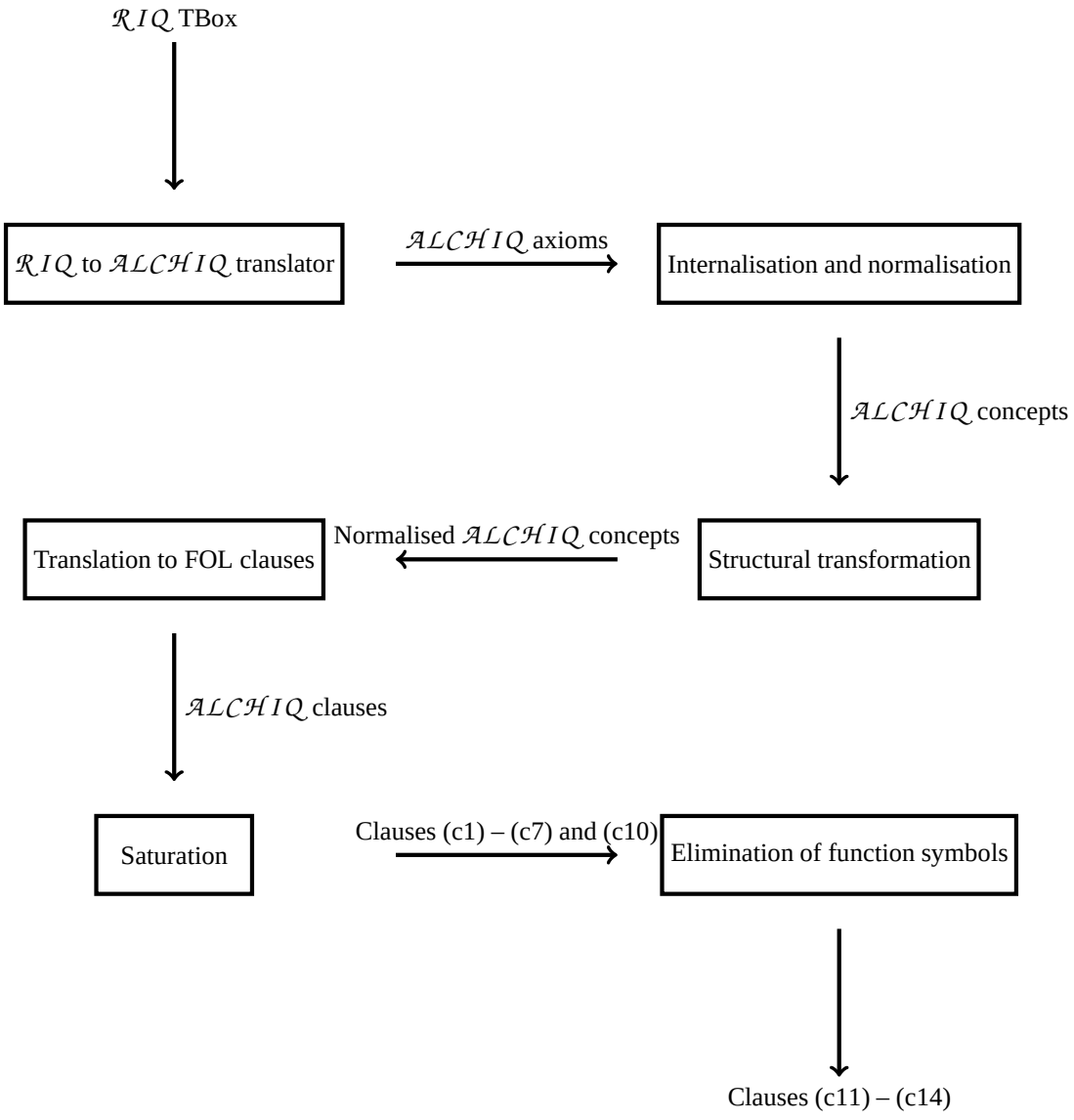

Figure 5.2: The TBox saturation module

5.2.1 Data Representation

The input of the TBox reasoner is a $\mathcal{RIQ}$ terminology, represented as a Prolog list $[GCIs, Hbox, Trbox]$ of three elements. *GCIs* is a list of concept inclusion axioms of the form $\texttt{implies}(C_1, C_2)$, where C_1, C_2 are concepts. *Hbox* is a list of complex role inclusion axioms of the form $\texttt{subrole}(Rs, R)$, where *Rs* is a list of roles and *R* is a role. *Trbox* is the list of transitive roles. The Prolog representation or roles and concepts is defined by the function $\cdot^p$ as follows:

DL expression		Prolog Representation
$R^p (R \in N_R)$	=	$\texttt{arole}(R)$
$(R^-)^p$	=	$\texttt{inv}(R^p)$
$C^p (C \in N_C)$	=	$\texttt{aconcept}(C)$
$\top^p$	=	$\texttt{top}$
$\bot^p$	=	$\texttt{bottom}$
$(\neg C)^p$	=	$\texttt{not}(C^p)$
$(C \sqcap D)^p$	=	$\texttt{and}([C^p, D^p])$
$(C \sqcup D)^p$	=	$\texttt{or}([C^p, D^p])$
$(\forall R.C)^p$	=	$\texttt{all}(R^p, C^p)$
$(\exists R.C)^p$	=	$\texttt{some}(R^p, C^p)$
$(\leq NR.C)^p$	=	$\texttt{atleast}(N, R^p, C^p)$
$(\geq NR.C)^p$	=	$\texttt{atmost}(N, R^p, C^p)$

For example, the DL axiom $(\geq 2 hasChild.Clever) \sqsubseteq (Rich \sqcap Happy)$ is represented with the following Prolog term:

```
implies( atmost(2,arole(hasChild),aconcept(clever)),
         and([aconcept(rich),aconcept(happy)]) )
```

After a series of transformation steps the TBox is translated into a set of first-order clauses, that are represented as lists of literals. We extend the $\cdot^p$ function to describe how terms and literals are represented:

FOL expression		Prolog Representation
FOL variable	=	Prolog variable
$(f(X))^p$	=	$\texttt{fun}(\texttt{f}, X^p, M)$
$(C(X))^p$	=	$\texttt{concept}(C^p, X^p)$
$(R(X,Y))^p$	=	$\texttt{role}(R, X^p, Y^p)$
$(\neg P)^p$	=	$\texttt{not}(P^p)$

The third argument of a functional term is used to indicate if the term is marked (see Subsection 2.1.1). If the term is marked, its value is the term `marked`, otherwise it is an uninstantiated variable. As an example, we give the Prolog representation of the FOL clause $C(x) \vee \neg R(x, f(x)) \vee S([g(x)], x)$:

```
[ concept(aconcept(c),X),
  not(role(arole(r),X,fun(f,X,_))),
  role(arole(s),fun(g,X,marked),X) ]
```

5.2.2 Saturation

The key part of the TBox saturation module is saturation itself, which performs all possible inference steps on the input clause set. A naive first implementation could be to non-deterministically select two clauses, try to resolve them and if it succeeds, then add the conclusion to the clause set. This is very inefficient, because (1) the same inference step might be performed more than once, (2) most of the time the selected clauses cannot together be premises of an inference and (3) it is hard to determine when to stop, i.e., when the clause set is saturated.

To make saturation more efficient, we separate the clauses into two sets: the clause set SAT is saturated, i.e., any inference with premises from SAT yields a conclusion that is either in SAT or is implied by some clause in SAT. The rest of the clauses constitute the set UNSAT. Initially, UNSAT contains all clauses and SAT is empty. We gradually add clauses from UNSAT to SAT and always collect all conclusions that can be drawn from the newly added clause and some clause already in SAT. These conclusions are added to UNSAT. Saturation terminates when UNSAT becomes empty.

Before adding a clause to SAT, it is very important to perform redundancy checking. If one clause is a consequence of another, then the first is said to be redundant and can be eliminated. For example, if we have clauses $C_1 = P(x)$ and $C_2 = P(x) \vee Q(x)$, then C_2 can be eliminated. Each clause C that is newly added to SAT has to be compared with every single clause already in SAT. If C turns out to be redundant, than it should not be added. If, on the other hand, the presence of C makes some other clauses redundant, then they should be eliminated from SAT.

Saturation, extended with redundancy checking is summarised in Algorithm 2.

Algorithm 2 Saturation of $\mathcal{ALCHIQ}$ clauses

SAT $= \emptyset$

UNSAT $=$ Input clause set

DO

IF UNSAT $= \emptyset$ THEN return SAT

ELSE LET $C \in$ UNSAT

remove C from UNSAT

IsRedundant = FALSE

FOREACH $C_2 \in$ SAT

IF C_2 is redundant due to C THEN remove C_2 from SAT

IF C is redundant due to C_2 THEN IsRedundant = TRUE

IF IsRedundant = FALSE THEN

Let RS be the set of R such that there is a clause $C_2 \in$ SAT and an inference rule with premises C and C_2 and conclusion R_2, where R_2 can be simplified into the logically equivalent R

add C to SAT

add Rs to UNSAT

5.2.3 Optimising saturation via indexing

Saturation can take a long time. The size of the sets SAT and UNSAT can grow exponential in the size of the initial clause set. Each time we add a clause C from UNSAT to SAT, we compare it with every clause in SAT to see if they can participate together in an inference and also to see if one is implied by the other. Performance can increase greatly if we manage to narrow down the set of clauses that are worth examining for possible inferences with C and also to narrow down the set of clauses that have the potential to make C redundant. We can achieve this through some index tables.

In our first implementation, SAT and UNSAT were stored in Prolog lists. However, a Prolog list does not allow for random access: if we want to find a particular element in the list, we have to go through all the preceding elements, so it takes linear time in the size of the list. Hence, we decided to use the dynamic predicate facility of Prolog for storing these sets. Each clause C is associated with a unique identifier ID_C and we use the following Prolog facts:

For each clause $C \in$ UNSAT, we assert `clause:clause0(`ID_C`,`C`)`

For each clause $D \in$ SAT, we assert `clause:clause1(`ID_D`,`D`)`

Most Prolog implementations perform indexing on the functor of their first argument, so if we have an identifier *ID*, then we can find the corresponding clause in constant time, regardless of the number of clauses asserted.

When looking for resolvent clauses with some clause C, we can use the maximal literal of C to focus our search. For example, if $C = A \vee B$ and literal A is greater than literal B, then C can only be resolved with a clause whose maximal literal is $\neg A$ (for a resolution step), or with a clause whose maximal literal is $A = w$ (for a superposition step). Hence, we maintain an index table, which allows us to look up the set of clauses associated with a particular maximal literal. This table is implemented using the Prolog fact

```
clause:starts_with(MaxLiteral,ID)
```

The following Prolog code collects all clauses *Cs* from SAT whose maximal literal is `L`:

```
is_maximal_literal(L,Cs):-
      findall(C, (
                  clause:starts_with(L,ID),
                  clause:clause1(ID,C)
                 ), Cs
             ).
```

The time that this predicate uses is linear in the size of *Cs*, but it is independent from the size of SAT.

Another aspect of saturation that can be a serious performance bottleneck is redundancy checking. In fact, it is a well known fact that modern theorem provers spend most of their reasoning time on redundancy checking. In return, this allows for avoiding repeated inferences and falling into infinite loops. Hence, any speedup in redundancy checking manifests directly in speedup in the whole reasoning process.

A clause C is made redundant by some clause D if there is a substitution σ such that the literals in $D\sigma$ are a subset of the literals in C. Consequently, when we want to check if clause C is redundant, it is enough to focus on clauses whose predicates are a subset of that of C. We maintain a lookup table (implemented as the Prolog fact `clause:is_contained(Pred,ID)`) which associates with each predicate the clauses that contain it and another table which associates with each clause the set of its predicates (`clause:all_predicates(ID,Preds)`). We first determine the set of predicates *CPreds* of C, collect the clauses that contain some of these predicates and then eliminate the ones that contain other predicates than those of C. The redundancy of C is checked only with respect to the remaining clauses. This is implemented in the following predicate:

```
narrower_predicate_set(CPreds,Ds):-
findall(ID, (
      member(P,CPreds),
      clause:is_contained(P,ID)
    ), IDs
       ),
sort(IDs,IDs2),
findall(D, (
     member(ID,IDs2),
     clause:all_predicates(ID,DPreds),
     ord_subset(DPreds,CPreds),
     clause:clause1(ID,D)
   ), Ds
      ).
```

On the other hand, if we want to see what clauses are made redundant by C, then it is enough to check those clauses whose predicate set is a superset of that of C. Hence we collect all the clauses that contain all the predicates of C:

```
broader_predicate_set([First|Preds],IDDs):-
        findall(ID, (
```

```
                    clause:is_contained(First,ID),
                    ( foreach(P,Preds), param(ID)
                    do is_contained_nochoice(P,ID)
                    )
                  ), IDs
              ),
        ( foreach(ID2,IDs), foreach(ID2-D,IDDs) do clause:clause1(ID2,D) ).
broader_predicate_set([],IDDs):- !, % The empty clause makes
                                    % everything redundant
        findall(ID-D, clause:clause1(ID,D), IDDs).


is_contained_nochoice(P,ID):-
        clause:is_contained(P,ID), !.
```

The optimisations described in this paragraph increased the overall speed of TBox saturation with two orders of magnitude.

5.3 Future Work

One of the most urgent tasks ahead of us is extending the system interface. Currently, we only support the DIG ([7]) format for the input knowledge base and query. We would like to provide the system with an OWL interface (see [27] and [21]). Moreover, we have already implemented the database support ([32]) which enables really large scale reasoning, however, it has not yet been incorporated into the reasoner. Once these tasks are done, we need to do more testing to evaluate DLog with respect to other DL reasoners such as RacerPro, Pellet, Hermit, KAON2.

On the theoretical side, we are curious to see how far we can extend the expressivity of DLog beyond $\mathcal{RIQ}$, approximating, as much as possible $\mathcal{SROIQ}(\mathcal{D})$, the language behind OWL2 ([21]).

5.4 Summary

The DLog program is in experimental stage. We implemented all the reasoning algorithms and we have prototype implementations for various further features, such as support for ABoxes stored in database. In the near future we plan to incorporate all our results in a reasoner that proves useful for the DL community.

Part II

Static Type Inference

Chapter 6

Introducing the Q Language and Constraint Logic Programming

In the following, we present some background knowledge that the user my find useful in the context of type inference for the Q functional programming language, in particular for understanding Chapters 7 and 8. In Section 6.1 we describe the Q functional programming language which was the target language for which we developed a type analysis tool. In Section 6.2 we briefly present the constraint satisfaction problem. In Chapter 7, we will rephrase the task of type inference as a constraint satisfaction problem. Finally, in Section 6.3 we present the Constraint Handling Rules language, which we used for implementing our type analyser.

6.1 The Q Programming Language

Q is a highly efficient vector processing functional language, which is well suited to performing complex calculations quickly on large volumes of data. Consequently, numerous investment banks (Morgan Stanley, Goldman Sachs, Deutsche Bank, Zurich Financial Group, etc.) use this language for storing and analysing financial time series [35]. The Q language first appeared in 2003 and is now (July 2012) so popular, that it is ranked among the top 50 programming languages by the TIOBE Programming Community [53].

Types Q is a strongly typed, dynamically checked language. This means that while each variable, at any point of time, is associated with a well defined type, the type of a variable is not declared explicitly, but stored along its value during execution. The most important types are as follows:

- **Atomic types** in Q correspond to those in SQL with some additional date and time related types that facilitate time series calculations. Q has the following 16 atomic types: `boolean`, `byte`, `short`, `int`, `long`, `real`, `float`, `char`, `symbol`, `date`, `datetime`, `minute`, `second`, `time`, `timespan`, `timestamp`.
- **Lists** are built from Q expressions of arbitrary types, e.g. `(1;2.2;‘abc)` is a list comprising two numbers and a symbol. However, if a variable is initialised to a list of atomic values of the same type, then certain operations, e.g. updating a certain element of the list, insist on keeping the list homogeneous.
- **Dictionaries** are a generalisation of lists and provide the foundation for tables. A dictionary is a mapping that is given by exhaustively enumerating all domain-range pairs. For example, `(‘a‘b ! 1 2)` is a dictionary that maps symbols `a,b` to integers `1,2`, respectively.
- **Tables** are lists of special dictionaries called **records**, that correspond to SQL records.
- **Functions** correspond to mathematical mappings specified by an algorithm.

Main Language Constructs Q being a functional language, functions form the basis of the language. A function is composed of an optional parameter list and a body comprising a sequence of expressions to be evaluated. Function application is the process of evaluating the sequence of expressions obtained after substituting actual arguments for formal parameters.

As an example, consider the expression

```
f: {[x] $[x>0;sqrt x;0]}
```

which defines a function of a single argument x, returning $\sqrt{x}$, if $x > 0$, and 0 otherwise. Note that the formal parameter specification `[x]` can be omitted from the above function, as Q assumes `x`, `y` and `z` to be implicit formal parameters. If a return value is specified, the function evaluates to its return value, otherwise it has no return value.

Input and return values of functions can also be functions: for example, a special group of functions, called *adverbs* take functions and return a modified version of the input. The whole Q program can be seen as a series of complex function evaluation steps.

Some built-in functions (dominantly mathematical functions) with one or two arguments have a special behaviour called *item-wise extension*. Normally, the built-in functions take atomic arguments and return an atomic result of some numerical calculation. However, these functions extend to list arguments item-wise. If a unary function is given a list argument, the result is the list of results obtained by evaluating each argument element. A binary function with an atom and a list argument evaluates the atom with each list element. When both arguments are lists, the function operates pair-wise on elements in corresponding positions. Item-wise extension applies recursively in case of deeper lists, e.g. `((1;2); (3;4)) + (0.1; 0.2) = ((1.1;2.1); (3.2;4.2))`

Although it is a functional language, Q also has imperative features, such as multiple assignment variables, loops, etc.

Q is often used for manipulating data stored in tables. Therefore, the language contains a sublanguage called Q-SQL, which extends the functionality of SQL, while preserving a very similar syntax.

Besides expressions to be evaluated, a Q program can contain so called *commands*. Commands control aspects of the Q environment. Among many other tasks, they are responsible for changing the current context (namespace), performing various O/S level operations, loading a file, etc.

Principles of evaluation In Q, expressions are always parsed from right to left. For example, the evaluation of the expression `a:2*3+4` begins with adding `4` to `3`, then the result is multiplied by `2` and finally, the obtained value is assigned to variable `a`. There is no operator precedence, one needs to use parentheses to change the built-in right-to-left evaluation order.

Flexibility Q is an extremely permissive language: for example, it is allowed to divide by zero and built-in functions accept extreme types without runtime error. This property of the language significantly increases the chance of program errors that are very difficult to explore once the program evaluation fails. Overcoming this difficulty by developing debugging tools for Q is likely to greatly enhance the usability of the language.

Type restrictions in Q The program code environment can impose various kinds of restrictions on types of expressions. In certain contexts, only one type is allowed. For example, in the do-loop `do[n;x*:2]`, the first argument specifies how many times `x` has to be multiplied by `2` and it is required to be an integer. In other cases we expect a polymorphic type. If, for example, function `f` takes arbitrary functions for argument, then its argument has to be of type `A -> B` (a function taking an argument of type `A` and returning a value of type `B`), where `A` and `B` are arbitrary types. In the most general case, there is a restriction involving the types of several expressions. For instance, in the expression `x : y + z`, the type of `x` depends on those of `y` and `z`. A type analyser for Q has to use a framework that allows for formulating all type restrictions that can appear in the program.

6.2 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) [24] can be described with a triple (X, D, C), where

- $X = \{x_1, \ldots, x_n\}$ is a series of variables,
- $D = \{D_1, \ldots, D_n\}$ is a series of finite sets called domains,
- variable x_i can only take values from domain D_i,
- $C = \{c_1, \ldots, c_k\}$ is a series of constraints, i.e., atomic relations whose arguments are variables from X.

A solution to a CSP is an assignment to each $x_i \in X$ a domain element $v_i \in D_i$, such that all constraints $c \in C$ are satisfied.

A value d_i of a variable x_i of a constraint c is *superfluous* in case there is no assignment to the rest of the variables of c along with $x_i = d_i$ that satisfies constraint c. Removing superfluous values from the corresponding domains yields an equivalent CSP.

There are two mechanisms that lead to a solution of a CSP. First, constraints constantly monitor the domains of their variables and remove superfluous values. Second, in case constraints fail to reduce some domain to a single value, we apply labeling: we choose a variable x_i and split its domain into two (or more) parts, creating a choice point where each branch corresponds to a reduced domain. Through a backtracking search we explore the branches. During labeling, constraints can wake up as the domains of their variables change and can further eliminate superfluous values. In case a domain becomes empty, we roll back to the last choice point. By the end of labeling, either we find a single value for each variable such that all constraints are satisfied, or else we conclude that the CSP is unsatisfiable.

6.3 Constraint Handling Rules (CHR)

Constraint Handling Rules (CHR) is a language embedded into a host language. Here we only give a brief introduction, a more detailed tutorial can be found in [47]. Most Prolog implementations contain a CHR extension, and CHR code is translated into Prolog code. In the following, we will as assume the host language to be Prolog.

CHR provides a very flexible tool, because arbitrary constraints can be formulated. However, there is no built-in constraint reasoning, it has to be provided by the programmer in the form of rewrite rules. A constraint can be any Prolog term except for variable.[1] A CHR program consists of a sequence of rules, that are simple if-then rules. Program execution is as follows:

1. There is a *constraint store* where constraints are accumulated. A constraint can appear anywhere in a Prolog program, instead of a predicate call. The constraint gets added to the store.
2. Each CHR rule monitors the constraint store and in case certain constraints are present, it can fire. The firing of a rule can result in the addition or removal of some constraints, along with the execution of some Prolog calls.
3. If the constraints in the store allow for no rule to fire any more, execution terminates and the user is shown the final state of the constraint store.

We illustrate the use of CHR with a simple example taken from [47]. The program describes how to mix colors. We will work with six different colors: `red`, `yellow`, `blue`, `green`, `purple`, `orange`. These colors are our constraints, declared at the beginning of the program:

```
:- chr_constraint red, yellow, blue.
:- chr_constraint green, purple, orange.
```

[1] Though, it can contain variables as subexpressions.

Of course, we know that three colors are sufficient for creating the other three. Red and blue yield purple, red and yellow yield orange, blue and yellow yield green. These are expressed using the following CHR rules:

```
red,  blue   <=> purple.
red,  yellow <=> orange.
blue, yellow <=> green.
```

The above rules are called *simplification rules*, because the constraints to the left of the `<=>` sign are simplified into the constraints to the right. The left part is called the *head* that contains all the constraints that need to be present in the store in order for the rule to fire. The right part is the *body* that holds the constraints to be added after firing. For example, the first rule can fire if we have constraints `red` and `blue` in the store. After firing, `red` and `blue` are removed and `purple` is added to the store.

We have a mixing bucket, which corresponds to the constraint store. What happens if we put `red` in the bucket?

```
?- red.
red
```

Nothing happens, `red` remains in the bucket, because the rules require two colors to fire. If, however, we also add yellow:

```
?- red, yellow.
orange
```

then the second rule fires and we obtain the color (constraint) `orange` in the bucket (store).

Now, let us add the color `brown` to our palette:

```
:- chr_constraint brown.
```

The particularity of `brown` is that it remains `brown`, no matter what color is added to it.

```
brown, orange <=> brown.
brown, purple <=> brown.
...
```

Notice that the constraint `brown` appears both in the head and in the body. For such rules, there is a simplified notation:

```
brown \ orange <=> true.
brown \ purple <=> true.
...
```

The head has two parts: constraints that remain after firing and those that are eliminated by the rule. Simplification rules are special cases of this rule, where the first part was empty. It is also possible that the second part is empty, i.e., nothing is removed from the store. For example, the color `yellow` might contain some constituent that leads to the corrosion of the mixing bucket. This is called a *propagation rule*:

```
yellow ==> corrosion
```

Rules where neither part of the head is empty can be seen as the combination of simplification and propagation rules. For this reason, they are called *simpagation rules*.

Until now, we only had atomic constraints. There is no reason for that, any Prolog term is allowed (except for variables). Let us add a saturation value to our colors. The arity of the constraints change, which has to be reflected in the constraint declaration:

```
:- chr_constraint red/1, yellow/1, blue/1.
:- chr_constraint green/1, purple/1, orange/1.
:- chr_constraint brown/1.
```

As we mix colors, the saturation values are added:

```
red(X),  blue(Y)   <=> Z is X+Y, purple(Z).
red(X),  yellow(Y) <=> Z is X+Y, orange(Z).
blue(X), yellow(Y) <=> Z is X+Y, green(Z).
```

What we see here is that arbitrary Prolog code can be inserted in the rule body. The code is executed, while the constraints are added to the store.

```
?- yellow(3), blue(4).
green(7)
```

Let us suppose that colors have a maximum saturation value, say 10. This means that if some color has maximum saturation, then it does not mix with any other color. This is a precondition for firing the rule, that can be placed in the so called *guard* part:

```
red(X),  blue(Y)   <=> X < 10, Y < 10 | Z is X+Y, purple(Z).
red(X),  yellow(Y) <=> X < 10, Y < 10 | Z is X+Y, orange(Z).
blue(X), yellow(Y) <=> X < 10, Y < 10 | Z is X+Y, green(Z).
```

The guard can contain arbitrary Prolog calls with the only restriction that it may not bind variables from the head. If the guard succeeds, the rule can fire and the body is executed.

Formal syntax After this informal introduction, we now present the precise syntax for the tree kinds of CHR rules:

- Simplification
  $H_1,\ldots,H_i <=> G_1,\ldots,G_j|B_1,\ldots,B_k.$
- Propagation
  $H_1,\ldots,H_i ==> G_1,\ldots,G_j|B_1,\ldots,B_k.$
- Simpagation $H_1,\ldots,H_l\backslash H_{l+1},\ldots,H_i <=> G_1,\ldots G_j|B_1,\ldots,B_k.$

The rules consist of the following parts:

- Head: $H_1,\ldots,H_i$, where H_m is a CHR constraint
- Guard: $G_1,\ldots,G_j$, where G_m is a host constraint[2]
- Body: $B_1,\ldots,B_k$, where B_m is either a CHR or a host constraint

The semantics and execution of the rules:

- Simplification: In case the guard is true, the head and the body are equivalent. The constraints in the store that match the head are removed and the body is executed. This might involve adding new constraints to the store.
- Propagation: In case the guard is true, the head implies the body. The body is executed.
- Simpagation: In case the guard is true, the head is equivalent to the body along with the first part of the head. The constraints in the store that match the second part of the head are removed and the body is executed. Note that simpagation can be expressed as a simplification, since the following two rules are equivalent:[3]
  `Head1 \ Head2 <=> Body`
  `Head1, Head2 <=> Head1, Body.`

[2] In the case of Prolog a host constraint can be arbitrary predicate call.

[3] However, the rules are different in terms of efficiency, since the constraints in `Head1` are removed and then re-added in the second rule.

Example We demonstrate the usefulness of CHR through a small example program. The program computes the prime numbers in the range $\{0 \ldots N\}$, implementing the sieve of Eratosthenes:

```
:- chr_constraint primes/1, prime/1.
primes(1) <=> true.
primes(N) <=> ground(N), N>1 | M is N-1, primes(M), prime(N).
prime(X) \ prime(Y) <=> Y mod X =:= 0 | true.
```

The code is remarkably short. Let us see what happens if we add the constraint `primes(10)` to the store. The second rule generates constraints `prime(`I`)` for all $I \in \{2 \ldots 10\}$. Afterwards, the first rule removes the `primes/1` constraint. Finally, the third rule fires as long as it finds two constraints `prime(`X`)` and `prime(`Y`)` in the store, such that Y is divisible by X, in which case it eliminates Y. Only the primes remain.

```
?- primes(10).
prime(2)
prime(3)
prime(5)
prime(7)
```

Chapter 7

Type Inference for the Q Functional Language

In this chapter we present our work on designing a type analysis tool for the Q vector processing language, see Section 6.1. This work was carried out in the framework of a collaborative project between Budapest University of Technology and Economics and Morgan Stanley Business and Technology Centre, Budapest. We emphasize two merits of our work: 1) we provide a type language that allows for adding type declarations to Q programs, making the code better documented and easier to maintain and 2) our tool checks the type correctness of Q programs and detects type errors that can be inferred from the code before execution.

The type analysis tool has been developed in two phases. In the first phase we built a *type checker*: the programmer was expected to provide type annotations for all variables (in the form of appropriate Q comments) and our task was to verify the correctness of the annotations. In the second phase we moved from type checking towards *type inference*: we devised an algorithm for inferring the possible types of all program expressions, without relying on user provided type information. Although we no longer require type annotations, we allow them as they provide documentation and improve maintenance and code reuse.

The main goal of the type analysis tool is to detect type errors and provide detailed error messages explaining the reason of the inconsistency. Our tool can help detect program errors that would otherwise stay unnoticed, thanks to which it has the potential to greatly enhance program development.

We perform type inference using constraint logic programming: the initial task is mapped into a constraint satisfaction problem (CSP), which is solved using the Constraint Handling Rules extension of Prolog [19], [48].

First, in Section 7.1, we give an overview of previous work done in the field of static type analysis. In Section 7.2, we present some restrictions that we had to impose on the Q language in order to make type analysis feasible. Afterwards, in Section 7.3 we present the type language that we designed in order to enable Q programmers to add type annotations to their programs. The following two sections describe the type analysis itself. Section 7.4 shows how to check Q programs for type correctness in case there is a ground type declaration for each variable. The algorithm discussed in Section 7.5 lifts this restriction and allows for inferring the possible types for each program expression without any type information provided by the user.

7.1 Work Related to Type Inference

Static analysis of computer programs is a very broad concept and encompasses numerous techniques. These techniques analyse the code in compile time and try to predict the runtime behaviour. Often they aim to optimise resource consumption through better memory management, reuse of previously computed results etc. Furthermore, they can be used to automatically predict properties of the program that hold for all possible execution paths.

Static type analysis aims to ensure that program execution will never cause an error. This is not possible with full generality as errors may depend on particular input values of the program, but a large class of

errors may be discovered based on the types of the involved expressions and these types are often known already in compile time. A type represents a set of expressions and working with types as opposed to values is a useful abstraction that enables the early discovery of many programming errors.

A successful method for static analysis and in particular static type analysis is *abstract interpretation* [12]. In order to demonstrate a certain property of the program, we approximate the program with a simpler, more abstract one that shares the property to be demonstrated. This involves mapping concrete values to abstract values (types) and mapping concrete operations to abstract operations. The benefit of the mapping is that instead of considering all the possible execution branches of the initial program, we only need to consider groups of execution branches, such that the various executions within a group cannot be distinguished on the abstract level. Abstract interpretation can be very fine grained or very abstract, depending on the complexity of the property to be demonstrated.

A very different approach to type analysis is to generate constraints from the program to ensure that it is well typed. One of the first such algorithms used for type inference is the Hindley-Milner type system [25]. It associates the program to be analysed with a set of equations which can be solved by unification. It supports parametric polymorphism, i.e., allows for using type variables. The type inferred by the algorithm for an expression is guaranteed to be the most general possible type, the *principal type*. Most type systems for statically typed functional languages can be seen as extensions of the Hindley-Milner system. Some of the best known examples are the ML family [44] and Haskell [31]. We also find several examples of dynamically typed languages extended with a type system allowing for type checking and type inference. These attempts aim to combine the safeness of static typing with the flexibility of dynamic typing. [42] describes a polymorphic type system for Prolog, which is essentially the same as that of ML. Here, the only addition to the language are type declarations, and it is guaranteed that any well-typed program will behave identically with or without type analysis.

A major limitation of the Hindley-Milner system is that it requires disjoint types. In such a system one cannot have, for example, a *numeric* and an *integer* type since they are not disjoint. Another approach to type inference which does not suffer from this limitation is based on subtyping [10]. Here, the input program is mapped into type constraints of the form $U \subseteq V$ where U and V are types, as opposed to Hindley-Milner systems where we obtain constraints of the form $U = V$. Subtyping systems can be seen as generalisations of Hindley-Milner systems. [40] presents a type checker for Erlang, a dynamically typed functional language, based on subtyping. Several of the shortcomings of this system were addressed in [36]. Their tool aims to automatically discover hidden type information, without requiring any alteration of the code. The inferred types enhance program maintenance and reuse by helping programmers understand code written long ago. They introduce the notion of success typing: in case of potential type errors (for example, because a variable can have two possible types during execution and one leads to abnormal behaviour), they assume that the programmer knows what he wants. They only reject programs where the type error is certain, i.e., when there is no way the program can run correctly.

The Q language is similar to Erlang in that they are both dynamically typed functional languages. The usage of the language naturally yields many constraints of the form $U \subseteq V$ for types U, V. Still, a type system based on subtyping is not sufficient. Due to built-in functions being highly overloaded (ad-hoc polymorphism), we need tools to formulate and handle versatile and complex constraints. Constraint logic programming seems ideal for this task.

[16] reports on using constraints in type checking and inference for Prolog. They transform the input logic program with type annotations into another logic program over types, whose execution performs the type checking. They give an elegant solution to the problem of handling infinite variable domains by not explicitly representing the domain on unconstrained variables. The way variable domains are represented in the Q type inference tool was motivated by their work. [52] describe a generic type inference system for a generalisation of the Hindley-Milner approach using constraints, and also report on an implementation using Constraint Handling Rules. The CLP($\mathcal{SET}$) [17] framework provides constraint logic reasoning over sets. Our solution has many similarities to CLP($\mathcal{SET}$) as types can be easily seen as sets of expressions. The main difference is that we have to handle infinite sets.

7.2 Necessary Restrictions of the Q Language for Type Reasoning

Q is a very permissive language. In consultation with experts at Morgan Stanley we decided to impose some restrictions on the language supported by our tool, in order to promote good coding practice and make the type analysis more efficient.

With multiple assignment variables and dynamic typing, Q allows for setting a variable to a value of type different from that of the current value. However, this is not the usual practice and it defies the very goal of type checking. Hence we agreed that each variable should have a single type in a program, otherwise the type analyser gives an error message.

Other restrictions concern the type of the built-in functions. Most built-in functions in Q are highly overloaded, thanks to which some functions do not raise errors for certain "strange" arguments. For example, the built-in function `last` takes a list as argument and returns the last element of the list. However, this function works on atomic arguments as well: it simply returns the input argument. To increase the efficiency of the type reasoner we decided to ignore some special meanings of some built-in functions. For example, we neglected this special meaning of the `last` function. Consequently, we infer that the argument of the `last` function is a list, which is not necessarily true in general.

7.3 Extending Q with a Type Language

In order to allow the users to annotate their programs with type declarations, we had to devise a type language that could be comfortably integrated into a Q program. Our type language supports type polymorphism, i.e., the usage of type variables. Type expressions are built from atomic types and variables using type constructors. The concrete syntax is provided in Appendix B. The abstract syntax of the type language – which is at the same time the Prolog representation of types – is as follows:

```
TypeExpr =
    AtomicTypes    | TypeVar    | symbol(Name)    | any
  | list(TypeExpr) | tuple([TypeExpr,...,TypeExpr])
  | dict(TypeExpr, TypeExpr)    | func(TypeExpr, TypeExpr)
```

AtomicTypes This is shorthand for the 16 atomic types of Q. Furthermore, the `numeric` keyword is used to denote a type consisting of all numeric values.

TypeVar represents an arbitrary type expression with the restriction that the same variables stand for the same type expression. Type variables make it possible to define polymorphic type expressions, such as `list(A) -> A` (a function mapping a list of a certain type to a value of the given type) and `tuple([A,A,B])`.

`symbol(`*`Name`*`)` The named symbol type is a degenerate type, as it has a single instance only, namely the provided symbol. Nevertheless, it is important because in order to support certain table operations, the type reasoner needs to know what exactly the involved symbols are. For example, when we insert a new record into a table, it is not sufficient to know that the record maps symbols to the adequate types (that of the column values), we also have to check that the column names match.

`any` This is a generic type description, which denotes all data structures allowed by the Q language.

`list(`*`TE`*`)` The set of all lists with elements from the set represented by *TE*.

`tuple([`*`TE`*$_1$`, ..., `*`TE`*$_k$`])` The set of all lists of length k, such that the i^{th} element is from the set represented by TE_i.

`dict(`*`TE`*$_1$`,`*`TE`*$_2$`)` The set of all dictionaries, defined by an explicit association between domain list (TE_1) and range list(TE_2) via positional correspondence. For example, the dictionary `(‘name;‘date) ! (‘Joe; 1962)` has type `dict(tuple([symbol(name),symbol(date)]),tuple([symbol(Joe),int]))`[1].

[1] To facilitate type inference for tables, we include detailed information on the domain/range of a dictionary in its type. (A record is a dictionary with the domain being a list of column names.)

`func(`*`TE`*$_1$`,` *`TE`*$_2$`)` The set of all functions, such that the domain and range are from the sets represented by TE_1 and TE_2, respectively.[2]

While some type expressions correspond directly to Q language constructs (such as `list`, `dict` or `func`), others were "discovered" in the process of trying to describe Q expressions. Such are the `tuple(...)` and `symbol(...)` type expressions. Some built-in functions require list arguments with fixed length. These lists might also have to be non-homogeneous, with well specified type for each list member. To be able to describe the type of such functions (and that of their argument), we introduced the `tuple` type. Using the `tuple` type, we can for example easily describe a function that takes a list consisting of an integer and a symbol and returns another list consisting of two integers and a float: it has type `func(tuple([int,symbol]),tuple([int,int,float]))`.

The `symbol(`*Name*`)` type was introducted to enable type checking table operations. For example, it allows for deciding whether a given record to be inserted into a given table has matching column names. A record is a dictionary that maps column names to values. By using `symbol(`*Name*`)`, we can represent the domain type of dictionaries in such a way that contains the names of all columns. Hence, instead of treating dictionary `(`‘`name`‘`age)!(`‘`jim;2)`, as `dict(tuple([symbol,symbol]),tuple([symbol,int]))`, we represent its type as `dict(tuple([symbol(name),symbol(age)]),tuple([symbol,int]))`.

Note that our type system contains non-disjoint types: for example, `int` is a subtype of `numeric` and `tuple([int,int])` is a subtype of `list(int)`. As we shall see later, this greatly complicates the type analysis.

7.3.1 Type Declarations

Type annotations appear as Q comments and hence do not interfere with the Q compiler. A type declaration can appear anywhere in the program and it will be attached to the smallest expression that it follows immediately. For example, in the code

```
x + y //$: int
```

variable `y` is declared to be an integer.

Type declarations can be of two kinds, having slightly different semantics: *imperative* (believe me that the type of expression E is T) or *interrogative* (I think the type of E is T, but please do check). To understand the difference, suppose the value of `x` is loaded from a file. This means that both the value and the type is determined in runtime and the type checker will treat the type of `x` as `any`. If the user gives an imperative type declaration that `x` is a list of integers, then the type analyser will believe this and treat `x` as a list of integers. If, however, the type declaration is interrogative, then the type analyser will issue a warning, because there is no guarantee that `x` will indeed be a list of integers (it can be anything). Interrogative declarations are used to check that a piece of code works the way the programmer intended. Imperative declarations provide extra information for the type analyser.

Different comment tags have to be used for introducing the two kinds of declarations. We give an example for each:

```
f //$: int -> boolean      interrogative
g //!: int -> int          imperative
```

7.4 Type Checking for the Q Language

In this section we give an outline of the data structures and algorithms developed for the first version of our type analyser tool: the type checker. There are two requirements towards Q programmers: they have to provide a type declaration for all variables and only ground declarations are allowed, i.e., type variables are not allowed. Both restrictions will be lifted in the type inference algorithm to be described in Section 7.5.

We only discuss type analysis proper: details about parsing Q programs can be found in [62]. Hence, we assume that the input of this phase is the abstract syntax tree (AST), constructed by the parser. Its output is a (possibly empty) list of type errors.

[2] To help readability, we often use the notation `A -> B` instead of `func(A,B)`.

7.4.1 Type Analysis Proper

Algorithm 3 gives a summary of the type analysis component. Our aim is to determine whether we can assign a type to each expression of the program in a coherent manner. Some types are known from the start: the types of variables are provided by the programmer, furthermore, we know the types of atomic expressions and built-in functions. The analyser infers the types of the other expressions and checks for consistency.

Algorithm 3 Algorithm of the type analysis component

1. To each node of the abstract syntax tree, we assign a type variable.
2. We traverse the tree and formulate type constraints. For each program expression there is a constraint that can be used to determine its type based on the types of its subexpressions. In terms of the abstract syntax tree, these constraints specify the type of a node based on the types of its child nodes.
3. Constraint reasoning is used to automatically
   - propagate constraints,
   - deduce unknown types
   - detect and store clashes, i.e., type errors.

   From the types of the leaf nodes, we infer the types of their immediate parents. This wakes up new constraints, so in the next step we can determine the types of nodes that are at most two steps away from all their leaf descendants. Continuing this process, we eventually find all types.
4. If there is a type mismatch, we mark the erroneous node. All the parent nodes will also be marked erroneous – however, we only show the smallest erroneous expressions to the user, i.e., those that have no erroneous subexpression.
5. By the end of the traversal, each node that corresponds to a type correct expression is assigned a type. The types satisfy all constraints.

Each expression in the concrete syntax corresponds to a subtree of the AST. Hence, we maintain a variable (in mathematical sense) for each node of the tree, that stands for the type of the subtree rooted at the node. The task of the type checker is to instantiate the variables to proper ground types, as described in the type language in Section 7.3. During reasoning, there may be situations where we can only partially instantiate a variable, for example, we might first infer that a certain expression is a list and only later narrow it to be a list of floats. To handle these situations, we allow type variables in the inner representation of types, despite the fact that the programmers are not allowed to use them in the declarations.

We traverse the tree and formulate context specific constraints on the type of the current node and those of its children. For instance, in the example in Figure 7.1, when we reach the `app` node, we know it is a function application, so the left child has to be of type `a -> b`, the right child of type `a` and the whole subtree of type `b`. In some cases the constraint determines the type of some node, but in many others it only narrows down the range of possible values. In case of clash between the restrictions, there is a type error in the program.

The type checker also detects hazardous code that contains potential type error. This is the case when the expected type of some expression is a subtype of the inferred one. An example for this is when a function is declared to expect an integer argument and all we know about the argument is that it is numeric. We cannot determine the runtime behaviour of such a code, since the type error depends on what sort of numeric argument will be provided. Instead of an error, we give a warning in such cases that the user can decide to suppress.

Constraints are handled using the Prolog CHR [48] library. For each constraint, the program contains a set of constraint handling rules. Once the arguments are sufficiently instantiated (what this means differs from constraint to constraint), an adequate rule wakes up. The rule might instantiate some type variable, it

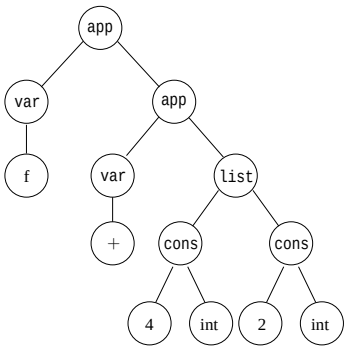

Figure 7.1: The abstract tree format of the expression `f (4+2)`

might invoke further constraints or else it infers a type error. In the latter case we mark the location of the error, along with the clashing constraint.

In case all variables are provided with a type declaration, we start the analysis with the knowledge of the types of all leaves of the abstract syntax tree. This is because a leaf is either an atomic expression or a variable. Once the leaf types are known, propagation of types from the leaves upwards is immediate, because we can infer the type of an expression from those of its subexpressions. Constraints wake up immediately when their arguments are instantiated, as a result of which the type variables of the inner nodes become instantiated.

7.4.2 Constraints

The constraints that can be used for type inference come from two sources. First, we know the types of atomic expressions and built-in functions. For example, `2.2` is immediately known to be a float. Similarly, we know that the function `count` is of type `any -> int`. Such knowledge allows us to set – or at least constrain – the types of certain leaves of the abstract syntax tree. The other source of constraints is the language syntax. This can be used to propagate constraints, because the language syntax imposes restrictions on the types of neighbouring nodes.

Besides these type constraints, there can be type information provided by the user at any level of the abstract syntax tree.

Constraint Handling Rules To handle type constraints, we use constraint logic programming. More precisely, we use the Prolog CHR (Constraint Handling Rules) library [48], which provides a general framework for defining constraints and describing how they interact with each other. The advantage of CHR is that the constraint variables can take values from arbitrary Prolog structures, so we can comfortably represent all values that a type expression can have.

An Example Constraint We illustrate constraint handling with a small example. Consider the expression `x in y`, where the types of `x,y` are `X,Y`, respectively. The `in` function checks if the first argument is a member of the second. The second argument is either a list or a dictionary. The type of the whole expression is boolean and the restriction on `X,Y` is expressed using the constraint `dict_list_c(Y,_,X)`, which can be defined by the following constraint handling rules:

```
% dict_list_c(X,A,B):-
% either X is a list of type B and A is integer
% or X is a dictionary with domain type A and range type B
dict_list_c(dict(X,Y),A,B) <=> A = X, B = Y.
dict_list_c(list(X),A,B)   <=> A = int, B = X.
```

The rules remain suspended until the first argument gets instantiated to a `dict/2` or `list/1` structure. The constraint fails if the adequate types cannot be unified.

However, these rules are incomplete in two ways. First, as we have seen in Section 7.3, a list can also be represented as a tuple. Hence, we have to add the following rule:

```
dict_list_c(tuple(Xs),A,B)     <=>
          A = int, ( foreach(B,Xs), param(B) do true ).
```

The second problem is the lack of error handling. If the constraint fails, the whole program fails. Thus, instead of telling the user where the type error occurred, we only indicate that there is a type error, which is not useful at all. We address this by assigning an identifier to each expression. Each time an error occurs, we store the identifier and the kind of error. After all constraints exited, we retrieve the identifiers of the erroneous expressions and the relevant location in the program code. With these we can give an error message that explains the problem. The final version of the constraint handling rules for `dict_list_c`:

```
% dict_list_c(X,A,B,ID):-
% either X is a list of type B and A is integer
% or X is a dictionary with domain type A and range type B
dict_list_c(X,A,B,ID) <=> nonvar(X) |
          ( X = dict(A,B)
          ; X = list(B), A = int
          ; X = tuple(Xs),  A = int,
            ( foreach(B,Xs), param(B) do true )
          ; assert(q:error(type,ID,wrong_dict_list))
          ), !.
```

The constraint wakes up as soon as the first argument is instantiated. Then, if it is a dictionary or a list, we can enforce the constraint by unifying some terms. If the unification succeeds, the constraint exits successfully. Otherwise, we mark that an error occurred.

7.4.3 Issues about Type Declarations

We require programmers to provide every variable with a ground type declaration. In this subsection we give reasons for this requirement.

The immediate benefit is that the types of all leaves of the abstract syntax tree are known at the beginning of the analysis. Without type declarations, some constraints might remain suspended and lots of types unknown. In this case we would have to use some sort of labeling to assign a type to each expression.

Furthermore, if the arguments of constraints are ground, we do not have to worry about the interaction of constraints. Consider, for example the following two constraints:

```
int_or_float(X) <=> (X == int ; X == float) | true.
int_or_long(X)  <=> (X == int ; X == long)  | true.
```

If these two constraints apply to `T`, then they will not do anything as long as `T` is a variable, even though there is only one solution, namely `T=int`. In order for the type analyser to infer this, we have to add a new rule that describes the interaction of the two constraints, such as

```
int_or_float(X), int_or_long(X) <=> X = int.
```

More complex constraints can interact in many different ways and the number of constraint handling rules necessary for capturing all interactions can be exponential in the number of constraints. Given that we work with more than 60 different constraints, it is not realistic to exhaustively write up all rules. If, on the other hand, the arguments are sufficiently instantiated that the constraints can wake up individually (not knowing about the others), then we only need to provide a couple of rules for each constraint. In the above example, if `X` is instantiated, then either `X=int` and both constraints exit successfully or else at least one constraint indicates an error.

When we have a variable in a Q program, we have to copy its type from its defining occurrence to all its applied occurrences. If the type is ground, copying is simple since we unify the type expressions. This, however, does not work if the type of the variable contains variables that are possibly constrained. Let

the type of `f` be `X -> int` where there is a constraint on `X` ensuring that it is from the set `{int, float}`. Consider the following code:

```
x:f 2
y:f 3.1
```

If we unify the type variables for each occurrence of `f` with `X -> int`, then from the first line `X` will be instantiated to `int`, which will make the type checker indicate a type error in the second line, since it will try to unify `int` with `float`. What we need is separate instances of the type of `f` with distinct variables, while holding the same constraints, which is quite complicated. Fortunately, this problem does not arise if all variables are provided with a ground type declaration.

7.5 Type Inference for the Q Language

In the second phase of the development of our type analyser tool, we set out to eliminate the two main restrictions of the type checker: sometimes it is too burdensome for the programmers to have to provide type declarations and sometimes it is too restrictive that the declarations have to be ground. In order to find a more flexible solution, where the analyser uses whatever information is available and infers as much as possible, we looked for a more solid theoretical foundation. In the following, we will show how to reformulate the task of type inference as a constraint satisfaction problem (CSP) and then provide a solution through this reformulation, based on logic programming.

7.5.1 Type Inference as a Constraint Satisfaction Problem

Type reasoning starts from a program code that can be seen as a complex expression built from simpler expressions. Our aim is to assign a type to each expression appearing in the program in a coherent manner. The types of some expressions are known immediately (atomic expressions, certain built-in functions), besides, the program syntax imposes restrictions between the types of certain expressions. The aim of the reasoner is to assign a type to each expression that satisfies all the restrictions.

We associate a CSP variable with each subexpression of the program. Each variable has a domain, which initially is the set of all possible types. Different type restrictions can be interpreted as constraints that restrict the domains of some variables. In this terminology, the task of the reasoner is to assign a value to each variable from the associated domain that satisfies all the constraints.

Domains Type expressions can be embedded into each other (e.g. `list(int)`, `list(list(int))`, etc.), and tuples can be of arbitrary length, consequently we have infinitely many types, which makes representing domains more difficult than in a classical CSP. Furthermore, the types determined by the type language are not disjoint. For example `1.1f` might have type `float` or `numeric` as well. It is evident that every expression which satisfies type `float` also satisfies type `numeric`, i.e., `float` is a *subtype* of `numeric`. We will use the subtype relation to represent infinite domains finitely: a domain will be represented with an upper and a lower bound.

We say that type expression T_1 is a subtype of type expression T_2 ($T_1 \leq T_2$) if and only if, all expressions that satisfy T_1 also satisfy T_2. The subtype relation determines a partial ordering over type expressions. For example, consider the `tuple([int,int])` type which represents lists of length two, where both elements are integers. Every expression that satisfies `tuple([int,int])` also satisfies `list(int)`, i.e., `tuple([int,int])` is a subtype of `list(int)`. For atomic expressions it is trivial to check if one type is the subtype of another. Complex type expressions can be checked using some simple recursive rules. In the following, we provide these rules:

- `list(`A`)` is a subtype of `list(`B`)` exactly if A is subtype of B.
- `tuple([`$A_1,\ldots,A_k$`])` is a subtype of `tuple([`$B_1,\ldots,B_k$`])` exactly if A_i is a subtype of B_i for all $1 \leq i \leq k$.
- `tuple([`$A_1,\ldots,A_k$`])` is a subtype of `list(`B`)` exactly if A_i is a subtype of B for all $1 \leq i \leq k$.

- `func(`D_1`,`R_1`)` is a subtype of `func(`D_2`,`R_2`)` exactly if D_2 is a subtype of D_1 and R_1 is a subtype of R_2.
- `dict(`D_1`,`R_1`)` is a subtype of `dict(`D_2`,`R_2`)` exactly if D_2 is a subtype of D_1 and R_1 is a subtype of R_2.
- `symbol(`*Name*`)` is a subtype of `symbol`.
- Every type is a subtype of `any`.

The domain of a variable is initially the set of all types, which can be constrained with different upper and lower bounds.

An upper bound restriction for variable X is a list $A = [A_1, \ldots, A_k]$, meaning that the upper bound of X is $\bigcup_{j=1}^{k} A_j$, i.e., X is a subtype of some element of A. Disjunctive upper bounds are very common and natural in Q, for example, the type of an expression might have to be either `list` or `dict`. The conjunction of upper bounds is easily described by having multiple upper bounds. If we have two upper bounds $A = [A_1, \ldots, A_k]$ and $B = [B_1, \ldots, B_l]$ on the same variable X, this means the value of X has to be in $\bigcup(A_i \bigcap B_j)$, for all $1 \leq i \leq k$ and $1 \leq j \leq l$.

A lower bound restriction for variable X is a single type expression A, meaning that A is a subtype of X. For lower bounds, it is their union which is naturally represented by having multiple constraints: if X has two lower bounds A and B, then $A \cup B$ has to be subtype of X. We do not use lists for lower bounds and hence cannot represent the intersection of lower bounds. We chose this representation because no language construct in Q yields a conjunctive lower bound.

With the following example we demonstrate that lower and upper bounds are natural restrictions in Q: In the code `a:f[b]` function `f` is applied to `b` and the result is assigned to `a`. Suppose the type of `f` turns out to be a map from `numeric` to `tuple([int, int])`. We can infer that the type of `b` must be at most `numeric`, which can be expressed with an upper bound. The result of `f[b]` has the type `tuple([int,int])`, which means, that the type of `a` must be at least `tuple([int,int])`, which can be expressed with a lower bound. If later the type of `a` turns out to be `list(int)` (a list of integers) and the type of `b` to be e.g. `float`, then the above expression is type correct.

Constraints After parsing – where we build an abstract syntax tree representation of the input program – the type analyser traverses the abstract syntax tree and imposes constraints on the types of the subexpressions. The constraints describing the domain of a variable are particularly important, we call them *primary constraints*. These are the upper and lower bound constraints. We will refer to the rest of the constraints as *secondary constraints*. Secondary constraints eventually restrict domains by generating primary constraints, when their arguments are sufficiently instantiated (i.e., domains are sufficiently narrow). Constraints that can be used for type inference can originate from the following sources in a Q program:

Type declarations If the user gives a type declaration, the expression will be treated as having the declared type.

Built-in functions For every built-in function, there is a well-defined relationship between the types of its arguments and the type of the result. These relations are expressed by adequate – sometimes quite complicated – constraints.

Atomic expressions The types of atomic expressions are revealed already by the parser, so for example, `2.2f` is immediately known to be a `float`.

Variables Local variables are made globally unique by the parser, so variables with the same name must have the same type. We ensure this by equating their corresponding domains. However, care has to be taken with polymoprhic functions. If, for example, there is a function `f` that maps arbitrary input to an integer, then its various applied occurrences might have different types: in `f[2]` and `f[‘jack]` the function will have types `int -> int` and `symbol -> int`, respectively. In such cases, instead of equality, we impose the *specialised* relation on the defining and the various applied occurrences of the function symbol. We will discuss this later in more detail.

Program syntax Most syntactic constructs impose constraints on the types of their constituent constructs. For example, the first argument of an `if-then-else` construct must be `int` or `boolean`. Another example is the assignment construct. The type of the left side has to be at least as "broad" as the type of the right side. It means the type of the right side is subtype of the type of the left side.

Constraint Reasoning Constraint reasoning is based on a *production system* [43], i.e., a set of IF-THEN rules. We maintain a *constraint store* which holds the constraints to be satisfied for the program to be type correct. We start out with an initial set of constraints. A production rule fires when certain constraints appear in the store and results in adding or removing some constraints. We also say (with the terminology of CHR) that each rule has a head part that holds the constraints necessary for firing and a body containing the constraints to be added. The constraints to be removed are a subset of the head constraints. One can also provide a guard part to specify more refined firing conditions.

The semantics of the constraints is given by describing their consequences and their interactions with other constraints. At each step we systematically check for rules that can fire. The more rules we provide the more reasoning can be performed.

Primary constraints represent variable domains. If a domain turns out to be empty, this indicates a type error and we expect the reasoner to detect this. Hence, it is very important for the constraint system to handle primary constraints as "cleverly" as possible. For this, we formulated rules to describe the following interactions on primary constraints:

- Two upper bounds on a variable should be replaced with their intersection.
- Two lower bounds on a variable should be replaced with their union.
- If a variable has an upper and a lower bound such that no type satisfies both, then the clash should be made explicit by setting the upper bound to the empty set.
- Upper and lower bounds can be polymorphic, i.e., they might contain other variables. From the fact that the lower bound must be a subtype of the upper bound, we can propagate constraints to the variables appearing in the bounds.

Secondary constraints connect different variables and restrict several domains. There are two approaches for reasoning over such constraints: 1) We can use multi-headed rules to capture the interactions of several constraints or 2) we only provide single headed rules, in which case constraints interact only through the narrowing of domains. Unfortunately, it is not realistic to capture all interactions of secondary constraints as that would require exponentially many rules in the number of constraints. Hence, we only describe (fully) the interaction of secondary constraints with primary constraints, i.e., we formulate rules of the form: if certain arguments of the constraints are within a certain domain, then some other argument can be restricted. E.g., if there is an expression `x+y` and we know that the arguments are numeric values, then the result must be either integer or float. If the second argument later turns out to be float, then the result must be float. At this point, there is nothing more to be inferred and the constraint can be eliminated from the store.

Our aim is to eventually eliminate all secondary constraints. If we manage to do this, the domains described by the primary constraints constitute the set of possible type assignments to each expression. In case some domain is the empty set, we have a type error. Otherwise, we consider the program type correct.

If the upper and lower bounds on a variable determine a singleton set, then we say that it is *instantiated*. If all arguments of a secondary constraint are instantiated, then there are two possibilities. If the instantiation satisfies the constraint, then the latter can be removed from the store. Otherwise, the constraint fails.

Error Handling As we parse the input program, we generate constraints and add them to the constraint store. The production rules automatically fire whenever they can. If some domain gets restricted to the empty set, this means that the corresponding expression cannot be assigned any type, i.e., we have a type error. At this point we mark the erroneous expression, as well as the primary constraints whose interaction resulted in the empty domain. This information – along with the position of the expression – is used to

generate an error message. The primary constraints are meant to justify the error. Once the error has been detected and noted, we roll back to the addition of the last constraint and simply proceed by skipping the constraint. This way, the type analyser can detect more than one error during a single run.

Labeling Eventually, after all constraints have been added, we obtain a constraint store such that none of the rules can fire any more. There are three possibilities:

- There were some discovered errors. Then we display the collected error messages and terminate the type inference algorithm.
- There were no type errors found and only primary constraints remain. In this case the domains described by the primary constraints all contain at least one element. Any type assignment from the respective domains satisfies all constraints, so the type analyser stops with success.
- No type errors were found, however, some secondary constraints remain. In order to decide if the constraints are consistent, we do *labeling*.

Labeling is the process of systematically assigning values to variables from within their domains. The assignments wake up production rules. We might obtain a failure, in which case we roll back until the last assignment and try the next value. Eventually, either we find a type assignment to all variables that satisfies all constraints or we find that there is no consistent assignment. In the first case we indicate that there is no type error. In the second case, however, we showed that the type constraints are inconsistent, so an error message to this effect is displayed. Due to the potentially large size of the search space traversed in labeling, it looks very difficult to provide the user with a concise description of the error.

7.6 Summary

In this chapter we presented our methods developed for checking Q programs for type correctness. This work involved the design of a type language with which programmers can add type annotations to their programs. Our first algorithm is capable of analysing a Q program that contains a ground type declaration for each variable and discover any type mismatches. Afterwards, we designed a more involved method that can infer the possible types of all program expressions without any information provided by the programmer. This method proceeds by transforming the initial task of type inference into a constraint satisfaction problem, which is solved using a production system.

All our algorithms have been implemented in a tool called `qtchk`, based on the Constraint Handling Rules extension of the Prolog language. A detailed description of our tool will be provided in Chapter 8.

Chapter 8

The `qtchk` Static Type Inference Tool for the Q Functional Language

In this chapter we present a Prolog program called `qtchk` that implements the type analysis described in Chapter 7. In Section 8.1 we give an overview of the system architecture. Afterwards, in Section 8.2, we discuss the implementation of the constraint satisfaction problem. Section 8.3 presents how we implemented error handling. Section 8.4 discusses labeling. In Section 8.5 we summarize the major difficulties that we came across with during the development of `qtchk`. In Section 8.6 we briefly evaluate our tool, based on test results.

8.1 Architecture

The type analysis can be divided into three parts:

- Pass 1: lexical and syntactic analysis
  The Q program is parsed into an abstract syntax tree structure.
- Pass 2: post processing
  Some further transformations make the abstract syntax tree easier to work with.
- Pass 3: type checking proper
  The types of all expressions are processed, type errors are detected.

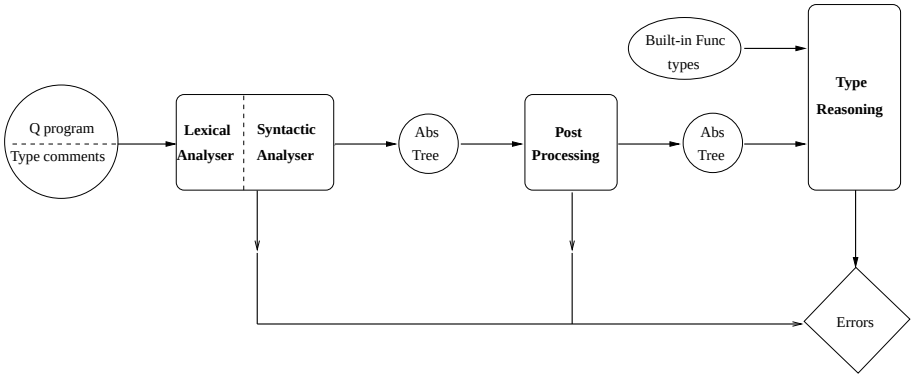

Figure 8.1: Architecture of the type analyser

The algorithm is illustrated in Figure 8.1. The analyser receives the Q program along with the user provided type declarations. The lexical analyser breaks the text into tokens. The tokeniser recognises

constants and hence their types are revealed at this early stage. Afterwards, the syntactic analyser parses the tokens into an abstract syntax tree representation of the Q program. Parsing is followed by a post processing phase that encompasses various small transformation tasks.

In the post processing phase some context sensitive transformations are carried out, such as filling in the omitted formal parameter parts in function definitions; and finding for each variable occurrence the declaration the given occurrence refers to.

Finally, in pass 3, the type analysis component traverses the abstract syntax tree and imposes constraints on the types of the subexpressions of the program. This phase builds on the user provided type declarations and the types of built-in functions. The latter are listed in a separate text file, that is parsed just like any Q program. The predefined constraint handling rules trigger automatic constraint reasoning, by the end of which each expression is assigned a type that satisfies all the constraints.

Each phase of the type analyser detects and stores errors. At the end of the analysis, the user is presented with a list of errors, indicating the location and the kind of error. In case of type errors, the analyser also gives some justification, in the form of conflicting constraints.

8.2 Representing variables and constraint reasoning

All subexpressions of the program are associated with CSP variables. In case some constraint fails, we need to know which expression is erroneous in order to generate a useful error message. If the arguments of the constraints are variables, we do not have this information at hand. Hence, instead of variables we use identifiers `ID = id(N,Type,Error)` which consist of three parts: an integer `N` which uniquely identifies the corresponding expression, the type proper `Type` (which is a Prolog variable before the type is known) and an error flag `Error` which is used for error propagation. We use the same representation for type variables in polymorphic types, e.g. the type `list(X)` may be represented by `list(id(2)`[1]`)`.

Constraint reasoning is performed using the Constraint Handling Rules library of Prolog. CHR has proved to be a good choice as it is a very flexible tool for describing the behaviour of constraints. Any constraint involving arbitrary Prolog structures could be formulated. We illustrate our use of CHR by presenting some rules that describe the interaction of primary constraints. Our two primary constraints are

- `subTypeOf(ID,L)`: The type of identifier `ID` is a subtype of some type in `L`, where `L` is a list of polymorphic type expressions.
- `superTypeOf(ID,T)`: The type of identifier `ID` is a supertype of type `T`, a polymorphic type expression.

With polymorphic types we can restrict the domain by a type expression containing the – not yet known – type of another identifier. If the type of such an identifier becomes known, the latter is replaced with the type in the constraint. For example, consider the following two constraints:

```
subTypeOf(id(1),[float,list(id(2))])
superTypeOf(id(1),tuple([id(3),int])
```

Suppose the types of `id(2)` and `id(3)` both turn out to be `int`. Then the above two constraints are automatically replaced with constraints:

```
subTypeOf(id(1),[float,list(int)])
superTypeOf(id(1),tuple([int,int])
```

Due to the lower bound, `float` can be eliminated from the upper bound. This is performed by the following CHR rule:

```
superTypeOf(X,A) \ subTypeOf(X,B0) <=> eliminate_sub(A, B0, B) |
      create_log_entry(eliminate_sub(X,A,B0,B)), subTypeOf(X, B).
```

[1] In order to make this and the following examples easier to read, we will write `id(N)` instead of `id(N,Type,Error)`.

Here we make use of the following Prolog predicates:

- `eliminate_sub(A,B0,B)`: The list of upper bounds `B0` can be reduced to a proper subset `B` based on lower bound `A`.
- `create_log_entry(X)`: We assert a log entry used for creating error messages.

Consequently, we obtain:

```
subTypeOf(id(1),[list(int)])
superTypeOf(id(1),tuple([int,int])
```

In another example, we show how two upper bounds on the same identifier are handled. Suppose we have the following constraints:

```
subTypeOf(id(1),[float,list(int)])
subTypeOf(id(1),[tuple([int,int]),func(int,float)])
```

The upper bounds trigger the following CHR rule:

```
subTypeOf(X,T1), subTypeOf(X,T2) <=> type_intersection(T1,T2,T) |
      create_log_entry(intersection(X,T1,T2,T)),
      subTypeOf(X,T).
```

The predicate `type_intersection(T1,T2,T)` imposes the constraint that `T` is the intersection of `T1` and `T2`. We obtain a single upper bound:

- `subTypeOf(id(1),[tuple([int,int])])`

8.3 Error Handling

During constraint reasoning, a failure of Prolog execution indicates some type conflict. In such situations, before we roll back to the last choice point, we remember the details of the error. We maintain a log that contains entries on how various domains change during the reasoning and what constraints were added to the store. Furthermore, to make error handling more uniform, whenever secondary constraints are found violated, they do not lead to failure, but they reduce the domain of a variable contributing to the failure of the constraint to the empty set. Hence, we only need to handle errors for primary constraints. Whenever a domain gets empty, we mark the expression associated with the domain and we look up the log to find the domain restrictions that contributed to the clash. We create and assert an error message and let Prolog fail. For example, the following message

```
Expected to be broader than (int -> numeric) and
              narrower than (int -> int)
  file:samples/s1.q  line:13  character:4
   {[x] f[x]}
   ^^^^^^^^^^
```

indicates that the underlined function definition is erroneous: the return value is numeric or broader (inferred from the type of `f`), although it is supposed to be narrower than integer (inferred from a type declaration).

8.4 Labeling

After all constraints are added to the constraint store, we use labeling to find a type assignment to each program expression (i.e., to each identifier associated with a node of the abstract syntax tree) that satisfies

the constraints. This involves another traversal of the abstract syntax tree to make sure no program expression is left without a type assignment. We select the next identifier `X` to be labelled and set its domain to a singleton set, based on its current domain. We implemented this by adding a new constraint `label(X)`. This constraint triggers the narrowing of the domain of `X` through the following CHR rules:

```
label(X) <=> id_known_type(X,_) | true.
label(X), superTypeOf(X,A), subTypeOf(X,L) <=>
        label_upwards(X,A,L,Type),
        hasType(X,Type).
label(X), superTypeOf(X,A) <=>
        label_upwards(X,A,[any],Type),
        hasType(X,Type).
label(X), subTypeOf(X,L) <=>
        label_downwards(X,L,Type),
        hasType(X,Type).
label(X) <=>
        label_downwards(X,[any],Type),
        hasType(X,Type).
```

First, we check if the type of `X` is already known. If so, we do nothing. Otherwise, we have four cases based on the presence or absence of a lower and upper bound:

- If we have a lower and an upper bound, we nondeterministically select a type from the domain. We start from the lower bound and successively try the broader types. This directionality is comfortable for implementation, because while a type might have many subtypes (e.g. any tuple of integers is a subtype of the type 'list of integers'), it has only few supertypes.
- If only a lower bound is present, we set the upper bound to `any` and proceed as in the previous case.
- If only an upper bound is present, we start from that type and go successively to its subtypes.
- If there is neither a lower, nor an upper bound, then we assume an implicit upper bound `any` and proceed as above.

Note that the `hasType/2` constraint, used above in the labeling code, translates to an upper and a lower bound:

```
hasType(X,Y):- subTypeOf(X,[Y]), superTypeOf(X,Y).
```

8.5 Difficulties

In this section, we discuss some difficulties that we had to overcome during the implementation of the type inference tool. These problems arose on the one hand from some special features of the Q language, and on the other hand from some limitations of the CHR library used.

8.5.1 Handling Meta-Constraints

As we described earlier, several built-in functions of Q have a special behaviour, called item-wise extension. We discuss the implementation of this feature now.

Let us consider, for example, the constraint `sum` which captures the relation between the arguments and the result of the built-in function '+'. If some of the arguments turn out to be lists, then the relation should be applied to the types of the list elements. We could capture this by adding adequate rules to the `sum` constraint. However, the rules describing the list extension behaviour would have to be repeated for each built-in function, which is counter-productive. Instead, we introduced a meta-constraint `list_extension/3`.

Consider a binary built-in function f, which extends item-wise to lists in both arguments and which imposes constraints `Cs` on its atomic arguments and result. Suppose that f has arguments identified by `X`, `Y`

and result identified by Z. We cannot add the constraints of Cs to the constraint store until we know that the arguments are all of atomic type. Instead, we use the meta-constraint list_extension(Dir,Args,Fun), where Dir specifies which arguments can be extended item-wise to lists, Args is the list of arguments on which the list of constraints[2] imposed by function Fun, will have to be formulated.

Hence, the constraint list_extension(both,[X,Y,Z],+) is added in our example. If later the input arguments are inferred to be atomic, then the meta-constraint list_extension/3 adds the atomic constraints Cs and removes itself:

```
subTypeOf(X,Ux), subTypeOf(Y,Uy) \
  list_extension(both,[X,Y,Z],Fun) <=> nonlist(Ux), nonlist(Uy) |
  list_ext_constraints(Fun,[X,Y,Z],Cs), ( foreach(C,Cs) do C ).
```

Here, the complicated part is to find the arguments of the proper constraints imposed by the given built-in function. We solved this by asserting the relevant information in the list_ext_constraints predicate. E.g. in the case of the Q function '+' we have the following fact:

```
list_ext_constraints(+, [A,B,C], [sum(A,B,C)]).
```

If, on the other hand, some argument turns out to be a list, the meta-constraint is replaced by another one. For example, if we know that the types of X and Y are list(A) and list(B), then the type of Z must be a list as well and we replace the list_extension constraint with the following two constraints: list_extension(both,[A,B,C],+) and hasType(Z,list(C)).

In fact, the list_extension meta-constraint could have been avoided, had CHR been more flexible: the difficulty arose from the fact that it is not possible to refer to a constraint in a CHR rule head by supplying a variable holding its name and a list of its arguments (cf. the call/N built-in predicate group of Prolog).

To express item-wise extension, it would be more convenient to write generic rules where the name of the involved constraint can also be a variable (this is in fact what the list_extension meta-constraint simulates).

For example, in the case of unary functions, where the corresponding constraint has two arguments (the identifiers of the input and the output), item-wise extension could be implemented using the following, quite natural "meta-rule"[3]:

```
call(Cons,A,B) <=> is_list(A,X), is_list_extensible(Cons) |
      call(Cons,X,Y), hasType(B,list(Y)).
```

where is_list_extensible(Cons) succeeds exactly when Cons has the list-extension behaviour and is_list(A,X) means that the type of A is list(X).

8.5.2 Copying Constraints over Variables

Local variables are made globally unique by the parser. This means, that variables with the same name have the same value, so we can constrain their types to be the same. However, each occurrence of a variable that holds a polymorphic function can have a different type assigned. Consider, for example, the following three lines of a program:

```
f:{[x] x+2}                    (1)
...
f [2]                          (2)
...
f [1.1f]                       (3)
```

In the first line, f is defined to be a function having a single argument x which returns x+2. This means that the type of f is a (polymorphic) function which maps A to B (A -> B), where a secondary constraint

[2] Note that there are several built-in functions, whose type is described using more than one constraint.

[3] Here we assume that CHR supports meta-constraints in rule heads using the call/N formalism of Prolog.

`sum(A, int, B)` holds between the argument and the result. In (2) and (3) there are two different applied occurrences of `f`, which specialise this `sum` constraint in two independent ways. In these examples `f` is applied to an integer and to a float, therefore the types of the second and third occurrence of `f` are `int -> int` and `float -> float`.

The above example shows that if a variable holds a (polymorphic) function then we cannot assume that the type of an applied occurrence is the same as that of the defining occurrence. We introduced the "specialisation" relationship to capture the connection between the defining and an applied occurrence of the same variable. A type is a specialisation of another if it is obtained by substituting zero or more type variables with (possibly polymorphic) types.

Specialisation could be seen as yet another constraint between two types, for which the natural implementation would be as follows:

1. At the defining occurrence of a variable, post the relevant type constraints.
2. At the applied occurrence of a variable, make a replica of the type constraints posted for the variable and apply them with fresh type variables.

This approach requires that the CHR library provide means for accessing the constraints that involve a specific argument, a feature similar to the `frozen(X, Goals)` built-in predicate of SICStus Prolog. Unfortunately, the CHR implementations we used do not have this feature. This means that a Q variable holding a polymorphic function has to be treated specially: the constraints involving its type have to be collected and remembered, so that they can be accessed at the applied occurrences of the given Q variable.

8.5.3 Handling Equivalence Classes of Variables

The constraint system yields lots of equalities. For example, two occurrences of the same (non-function-valued) variable give rise to an equality constraint. One way to handle this is to propagate all primary constraints between equal variables, i.e. whenever $X = Y$, Y inherits all primary constraints of X and the other way round. For example, a simple implementation of propagating the upper bounds in the equality constraint (`eq/2`) would be the following:

```
eq(X,Y), subTypeOf(X, T) ==> subTypeOf(Y,T).          (1)
eq(X,Y), subTypeOf(Y, T) ==> subTypeOf(X,T).          (2)
```

Unfortunately, this solution is rather inefficient, since all reasoning is repeated at each variable occurrence. Moreover, we have found cases which lead to an infinite propagation of CHR constraints. In the following paragraph we outline an example of this.

As we have seen in Section 8.2, two upper bounds on a variable are replaced with their intersection. Let us suppose that variable `A` has two upper bounds `list(X)` and `list(Y)`. There is an intersection rule which replaces these two with the upper bound `list(Z)`, where `Z` is a new variable and $\texttt{Z} \leq \texttt{X}$ and $\texttt{Z} \leq \texttt{Y}$ also have to be satisfied. Consider the following state of the constraint store:

```
eq(id(1), id(2)),
subTypeOf(id(1), [list(id(3))]),
subTypeOf(id(2), [list(id(4))]).
```

First, the equality rule can fire, yielding two new upper bounds on `id(1)` and `id(2)`. Now, the intersection rule can merge upper bounds on the same variable to create a new one, which can be propagated to the other variable by the equality rule again.

It is easy to see that the above constraint store results in an infinite firing sequence. Consider the following condition C: either one of the variables has two upper bounds that have not yet been merged by the intersection rule, or else there is an upper bound on one variable that has not yet been propagated to the other by the equality rule. If C holds, then at least one rule can fire (intersection or equality). However, C is an invariant condition, it remains true when any of these rules fire if it was true before. Since C holds in the initial store, the rules will never stop firing (regardless of the rule execution order).

The problem is caused by repeating the reasoning at each equal identifier. We solved this by introducing a directionality to the constraint propagation: we take a strict total order on identifiers and only propagate

constraints towards the smaller identifier. The smallest in a set of equal identifiers thus represents the whole set in the sense that it accumulates all constraints.[4] Once the type of the smallest identifier becomes known, it gets propagated back to the other identifiers. Hence, instead of `eq(X,Y)` we introduced the constraint `represented_by(X,Y)`, where `Y` $\leq$ `X` holds. Furthermore, for all constraints C we have a new rule, which states that if `X` is represented by `Y` and `X` occurs in C, then it should be substituted with `Y`. As we could not formulate meta-constraints with CHR, we had to provide propagation rules for every single constraint. For example, in case of the constraint `sum` we needed the following code:

```
represented_by(A,B) \ sum(A,C,D) <=> sum(B,C,D).
represented_by(A,B) \ sum(C,A,D) <=> sum(C,B,D).
represented_by(A,B) \ sum(C,D,A) <=> sum(C,D,B).
```

This yielded lots of new rules, however, it was easy to generate them automatically, using a small Prolog program.

There are efficiency problems even with this solution. Suppose we have the following constraint: `c(id(2))` and a propagation rule R, whose head matches the above constraint (possibly involving other constraints) and the body of the rule contains a new CHR constraint: `d(id(2))`. If `id(2)` later turns out to be equivalent to `id(1)`, then we substitute `id(2)` with `id(1)` in every constraint that contains `id(2)`. This yields a store with constraints:

```
c(id(1))
d(id(1))
```

The propagation rule R might fire now, which can infer the second constraint `d(id(1))` again. In order to avoid further cumulation of repeated inferences, we added idempotency rules for every constraint, i.e., some rules constantly monitor the store for duplicate constraints to be eliminated.

Unfortunately, idempotency rules do not fully solve the problem. Duplicate constraints might still yield redundant inferences in case these inference rules fire before the idempotency rules eliminate their premises. Consider, for example, the following constraint store: C_i for all $1 \leq i \leq n$. Furthermore, let us suppose that we have propagation rules R_j: $C_j => C_{j+1}$ for all $1 \leq j \leq n-1$. Suppose that constraint C_1 is inferred redundantly (twice). If the rules R_j fire before the idempotency rules, then it is possible that we infer all constraints C_i twice, before eliminating the duplicates. This results in $2n$ inference steps instead of the optimal 1 (in case the duplicate C_1 is eliminated immediately when it appears). This problem occurs because programmers have no control over the firing order of CHR rules with different heads. We identify this as a major shortcoming of CHR and believe that giving the programmers more elaborate tools to specify firing priorities would often help in improving the efficiency of CHR applications.

8.5.4 Labeling

The implementation of labeling posed several challenges. We noticed that the order in which identifiers are selected is crucial for efficiency. For example, it is important to label subexpressions first and then find the type of a complex expression. Another example is function application, where labeling should first assign a type to the input and then the type of the output is typically automatically inferred by the constraints. Consequently, labeling involves a traversal of the abstract syntax tree, and at each node we decide the order in which expressions are labelled based on the syntactic construct involved. Often we had to rely on heuristics as it was hard to guess what order would work best in practice.

The next difficulty arises when we already know which identifier to label, and we have to choose a value. The set of all types is infinite, so we cannot try all values for a variable during labeling, hence we made some restrictions. First, if there is an unbounded variable X, we only try terms of depth at most two. Hence, we do *not* replace it with `list(list(int))`. We can make this restriction because our constraints do not distinguish between deeply embedded types. If the unbounded X appears in constraint c and there is a substitution of X with depth greater than two that satisfies c, then we can generalise it to a substitution of depth two that also satisfies c. For example, if $X =$`tuple([int,list(float)])` satisfies c then $X =$`tuple([int,any])` is also a solution. If X does not satisfy this property with respect to c, then

[4] This is similar to how Prolog handles the unification of two variables.

as soon as c is added to the store, we add a bound to X. If, for example, c forces X to be a list of lists, then we add the upper bound `subTypeOf(X,[list(list(A))])`, and here A will be the unbounded variable which needs not be labelled with deep types. During labeling, we first assign a type T to A, and then use the ground upper bound `list(list(`T`))` to make a finite choice for the type of X.

Unfortunately, even if we have a ground bound on X, we might still have an infinite search space. This is because tuples can have arbitrarily many arguments. Suppose we know that X is a subtype of `list(int)`. Then X can be `tuple([int])`, `tuple([int,int])`, `tuple([int,int,int])` etc. Labeling through all the different kinds of tuples only makes sense if it can happen that some tuple types satisfy a constraint, while others do not. For this reason, we made sure that for all such constraints explicit bounds indicate the possible tuple types. Hence, if not all lists of integers are accepted, the constraint will generate either a lower bound (such as `superTypeOf(X,tuple([int,int]))`) or an upper bound (such as `subTypeOf(X,[tuple([int]),tuple([int,int,int])])`). Consequently, if neither the lower nor the upper bound of X contains a tuple type, then we do not assign a tuple type to it, we only try `list(int)`. If, however, X also has a lower bound `tuple([int,int])`, then we try both `tuple([int,int])` and `list(int)`.

The main challenge of labeling comes from the fact that it aims to traverse a huge search space. The abstract syntax tree can have many nodes even for moderately long programs, hence we have many identifiers. Besides, Q programs are typically full of ambiguous expressions (in terms of type), so without labeling, very few types are known for sure. All this amounts to labeling being the bottleneck of type inference.

A solution to this problem would be to find a good partitioning of the program, such that not all the tree is labelled together, but in smaller portions. Consider, for example, two function definitions. The first definition contains an expression E_1 that allows many different types. Labeling assigns one possible type to E_1 and then starts labeling the second function definition. Suppose the second definition contains a type error at expression E_2 which leads labeling to failure. Hence, we backtrack to the choice point at E_1 and assign another possible type to E_1. However, this type has nothing to do with the type mismatch – since it occurs in a different function definition, – and we get failure again at E_2. This cycle is repeated until all possible types for E_1 are tried and only then do we conclude that the program contains a type error. This procedure could be made more efficient by placing a cut after labeling the first function definition, thus eliminating the irrelevant choice point. Realizing that the types of expressions in one piece of code are independent from those of another can lead to much smaller fragments to be labelled, which has the potential to drastically reduce the time spent on labeling. Dependency analysis ([1]) could be used to find a code partitioning. Also, some kind of intelligent backtracking ([6]) algorithm could be used to avoid unnecessary choice points. However, adapting these techniques to the Q language requires further work.

8.6 Evaluation

`qtchk` runs both in SICStus Prolog 4.1 [49] and SWI Prolog 5.10.5 [54]. It consists of over 8000 lines of code.[5] Q has many irregularities and lots of built-in functions (over 160), due to which a complex system of constraints had to be implemented using over 60 constraints. The detailed user manual for `qtchk` can be found in [13] that contains lots of examples along with the concrete syntax of the Q language.

The best way to evaluate our tool would be on Q programs developed by Morgan Stanley, our project partner. However, we could not obtain such programs due to the security policy of the company. Instead, we used user contributed Q examples, publicly available at the homepage of Kx-System [9]. This test set contains several (extended) examples from the Q tutorial and other more complex programs. Table 8.1 summarizes our findings.

Table 8.1: Test results.

All	*Type correct*	*Restrictions*	*Labeling timeout*	*Type error*	*Analyser error*
128	43 (33.6%)	43 (33.6%)	32 (25%)	5 (3.9%)	5 (3.9%)

[5] We are happy to share the code over e-mail with anyone interested in it.

We used 128 publicly available Q programs. Of this 43 were found type correct. As explained in Subsection 7.2, we made some restrictions on the Q language, following the requirements of Morgan Stanley. 43 programs were found erroneous due to not fulfilling these restrictions. Most of the error messages arose from the same variable used with different types. Unfortunately, there is no way for the type analyser to lift this restriction as it defies the very goal of type checking.

The test set that we used often contained code fragments, instead of full programs: we found several cases that a function is called but defined in another file that was not included among the examples. In such programs the lack of information often resulted in an extremely large search space to be traversed during labeling. In 32 programs labeling could not find any solution within the given time limit (1000 sec), partly for the former reason. We believe, however, that on full programs that actually contain all necessary type information, `qtchk` can perform type analysis in acceptable time. Unfortunately, the available test set has not yet allowed us to ascertain about this.

We were happy to find 5 genuine errors in the test set. These are in the following programs: `run.q`[6]. `mserve.q`[7]. `oop.q`[8] `quant.q`[9] and `dgauss.q`[10]. We have found 5 programs containing some language element that our tool cannot handle well. We are in the process of eliminating these problems.

8.7 Summary

In this chapter we presented the `qtchk` type inference tool, developed to detect type errors in Q programs. The tool takes an input Q program, parses it into an abstract tree representation and then tries to assign a type to each subexpression in a coherent manner. This is achieved by rephrasing the task as a constraint satisfaction problem, which is solved using constraint logic programming and in particular the Constraint Handling Rules extension of Prolog.

CHR has proved to be a good choice as it is a very flexible tool for describing the behaviour of constraints. In CHR, arbitrary Prolog structures can be used as constraint arguments, therefore it was natural to handle the special domain defined by the type language.

However, we also had negative experiences with CHR. As described in Section 8.5, it often would be more convenient if we could write “meta-rules” in CHR. The need to access the constraint store also arose in some situations. For efficiency reasons, we believe it would often be useful to be able to influence the firing order of rules with different heads. Furthermore, the long and tiring process of debugging our CHR programs was seriously hampered by the lack of a tracing tool.

We have found that our program is a useful tool for finding type errors, as long as the programmers adhere to some coding practices, negotiated with Morgan Stanley, our project partner. Unfortunately, it turns out that many publicly available programs do not respect these practices. Nonetheless, we believe that the restrictions that we impose on the use of the Q language are reasonable enough for other programmers as well, and our tool will find users in the broader Q community.

[6] `http://code.kx.com/wsvn/code/contrib/cburke/qreference/source/run.q`
[7] `http://code.kx.com/wsvn/code/kx/kdb+/e/mserve.q`
[8] `http://code.kx.com/wsvn/code/contrib/azholos/oop.q`
[9] `http://code.kx.com/wsvn/code/contrib/gbaker/common/quant.q`
[10] `http://code.kx.com/wsvn/code/contrib/gbaker/deprecated/dgauss.q`

Summary and List of Contributions

I have presented our results in the fields of Description Logic data reasoning and static type inference. Although these domains are different in many ways, they both require some sort of automated reasoning. Our algorithms exploit and extend a variety of techniques of logic programming, and hence it was very natural to choose Prolog as an implementation language. The implemented systems – DLog and `qtchk` – demonstrate that the built-in inference mechanism of Prolog can be extended to solve various reasoning tasks.

In the following, I summarise my personal contributions to our results. I also indicate my relevant publications.

Thesis 1. I designed a transformation scheme from Description Logic axioms to first-order clauses that are function-free. I implemented all methods in the DLog Description Logic reasoner. [58, 55, 56, 65, 59]

Thesis 1.A. I designed a first-order resolution calculus called *modified calculus*, which is a modified version of basic superposition. I proved that the calculus is sound, complete and terminating for $\mathcal{ALCHIQ}$ clauses, which constitute a sublanguage of first-order logic. This result is what makes the two-phase reasoning algorithm of the DLog system possible: the complex reasoning over the TBox becomes independent of the potentially large ABox. [55, 56, 59]

Thesis 1.B. I designed a transformation that maps a $\mathcal{RIQ}$ knowledge base into an $\mathcal{ALCHIQ}$ knowledge base by eliminating complex role hierarchies. I proved that the transformation is sound, i.e., the initial knowledge base is satisfiable if and only if the transformed knowledge base is. Thanks to this transformation, any of the numerous techniques that were designed for reasoning over the $\mathcal{ALCHIQ}$ language became available for the more expressive $\mathcal{RIQ}$ language as well. [55]

Thesis 1.C. I designed the DL calculus, which decides the consistency of a $\mathcal{SHQ}$ terminology. I proved that the calculus is sound, complete and always terminates. The DL calculus provides an interesting alternative to the tableau method. [58, 65]

Thesis 1.D. I implemented the modified calculus, along with the transformation from $\mathcal{RIQ}$ to $\mathcal{ALCHIQ}$ in the TBox saturation module of the DLog data reasoner. This constitutes the first phase of our reasoning algorithm.

Thesis 2. I proved the soundness of loop elimination, a crucial optimisation technique for PTTP related theorem proving. [68, 69]

Thesis 2.A. I identified the three features in logic programs that can lead to infinite execution: function symbols, proliferation of variables and loops. I showed that from these only loops can occur in DLog programs. From this follows that the loop elimination optimisation makes DLog reasoning terminating.

[68, 69]

Thesis 2.B. I gave a rigorous proof of the soundness of loop elimination, based on a novel technique called flipping, which identifies alternative proofs of the same goal in PTTP programs. From this result follows that for any statement that can be proved by PTTP, there is a proof that contains no loops. [68, 69]

Thesis 3. I designed a static type analysis algorithm to check programs written in the Q language for type correctness. I also implemented this algorithm in the type analysis module of the `qtchk` system. [61, 14, 64, 63, 15]

Thesis 3.A. I designed a method for type checking: based on type annotations of program variables provided by the user, the algorithm determines the types of more complex expressions. [61, 14]

Thesis 3.B. I designed a method to move from type checking to type inference: no type annotations are required and the algorithm tries to infer the possible types of all expressions. This is achieved by transforming the task of type inference into a constraint satisfaction problem. [64, 63, 15]

Thesis 3.C. I implemented both type checking and type inference in the type analysis component of the `qtchk` system. The implementation uses constraint logic programming and in particular the Constraint Handling Rules extension of Prolog. [61, 14, 64, 63, 15]

Bibliography

[1] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. *SIGARCH Comput. Archit. News*, 20(2):342–351, April 1992.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2004.

[3] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 2, pages 19–100. North Holland, 2001.

[4] Leo Bachmair and Harald Ganzinger. Strict basic superposition. *Lecture Notes in Computer Science*, 1421:160–174, 1998.

[5] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001.

[6] Andrew B. Baker. Intelligent backtracking on constraint satisfaction problems: Experimental and theoretical results, 1995.

[7] S. Bechhofer, R. Moller, and P. Crowther. The dig description logic interface. In *In Proc. of International Workshop on Description Logics*, 2003. `citeseer.ist.psu.edu/690556.html`.

[8] Roland N. Bol, Krzysztof R. Apt, and Jan Willem Klop. An analysis of loop checking mechanisms for logic programs. *Theor. Comput. Sci.*, 86:35–79, August 1991.

[9] Jeffry A. Borror. *Q For Mortals: A Tutorial In Q Programming*. CreateSpace, Paramount, CA, 2008.

[10] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM COMPUTING SURVEYS*, 17(4):471–522, 1985.

[11] Alain Colmerauer and Philippe Roussel. *The birth of Prolog*. ACM, New York, NY, USA, 1996.

[12] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, June 1996.

[13] János Csorba, Péter Szeredi, and Zsolt Zombori. *Static Type Checker for Q Programs (Reference Manual)*, 2011. http://www.cs.bme.hu/~zombori/q/qtchk_reference.pdf.

[14] János Csorba, Zsolt Zombori, and Péter Szeredi. Using constraint handling rules to provide static type analysis for the q functional language. In *Proceedings of the 11th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2011)*, 2011.

[15] János Csorba, Zsolt Zombori, and Péter Szeredi. Pros and cons of using CHR for type inference. In Jon Sneyers and Thom Frühwirth, editors, *Proceedings of the 9th workhop on Constraint Handling Rules (CHR 2012)*, pages 16–31, September 2012.

[16] Bart Demoen, M. García de la Banda, and P. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In *Proceedings of Australian Workshop on Constraints*, pages 1–12, 1998.

[17] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, September 2000.

[18] Melvin Fitting. *First-order logic and automated theorem proving*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[19] Th. Fruehwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Journal of Logic Programming*, volume 37(1–3), pages 95–138, October 1998.

[20] Donald F. Geddis. *Caching and non-Horn Inference in Model Elimination Theorem Provers*. PhD thesis, Stanford University, USA, June 1995.

[21] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. OWL 2: The next step for OWL. *Web Semant.*, 6:309–322, November 2008.

[22] V. Haarslev and R. Möller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In *Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5*, pages 163–173, 2004.

[23] V. Haarslev, R. Möller, R. van der Straeten, and M. Wessel. Extended Query Facilities for Racer and an Application to Software-Engineering Problems. In *Proceedings of the 2004 International Workshop on Description Logics (DL-2004), Whistler, BC, Canada, June 6-8*, pages 148–157, 2004.

[24] Pascal Van Hentenryck. Incremental constraint satisfaction in logic programming. In *ICLP*, pages 189–202, 1990.

[25] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:pp. 29–60, 1969.

[26] Ian Horrocks. Reasoning with expressive description logics : Theory and practice. *Language*, pages 1–15, 2002.

[27] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: the making of a web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7 – 26, 2003.

[28] Ian Horrocks and Ulrike Sattler. Decidability of SHIQ with complex role inclusion axioms. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 343–348. Morgan Kaufmann, 2003.

[29] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. A description logic with transitive and converse roles, role hierarchies and qualifying number restrictions. LTCS-Report 99-08, LuFg Theoretical Computer Science, RWTH Aachen, Germany, 1999.

[30] Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reasoning for Description Logics around SHIQ in a resolution framework. Technical report, FZI, Karlsruhe, 2004.

[31] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

[32] Balázs Kádár, Gergely Lukácsy, and Péter Szeredi. Large scale semantic web reasoning. In *Proceedings of the 3rd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS2008), Udine, Italy*, pages 57–70, December 2008.

[33] Yevgeny Kazakov. RIQ and SROIQ are harder than SHOIQ. In *In Proc. KR08*, 2008.

[34] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[35] Kx-Systems. Representative customers . `http://kx.com/Customers/end-user-customers.php`.

[36] Tobias Lindahl and Konstantinos F. Sagonas. Practical type inference based on success typings. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 167–178. ACM, 2006.

[37] Gergely Lukácsy and Péter Szeredi. Efficient description logic reasoning in Prolog: the DLog system. *Theory and Practice of Logic Programming (in press)*, April 2009. http://arxiv.org/abs/0904.0578.

[38] Gergely Lukácsy and Péter Szeredi. Efficient Description Logic reasoning in Prolog: The DLog system. *Theory and Practice of Logic Programming*, 9(03):343–414, 2009.

[39] Gergely Lukácsy, Péter Szeredi, and Balázs Kádár. Prolog based description logic reasoning. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Proceedings of 24th International Conference on Logic Programming (ICLP'08),Udine, Italy*, pages 455–469, December 2008.

[40] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. *SIGPLAN Not.*, 32:136–149, August 1997.

[41] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany, January 2006.

[42] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.

[43] A. Newell and H.A. Simon. *Human Problem Solving*. Prentice Hall, Englewood Cliffs, 1972.

[44] Francois Pottier and Didier Remy. The essence of ML type inference. *Advanced Topics in Types and Programming Languages*, pages 389–489, 2005.

[45] ISO Prolog standard, 1995. ISO/IEC 13211-1.

[46] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[47] Tom Schrijvers. Constraint handling rules. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 9–10. Springer, 2008.

[48] Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, 2004.

[49] SICS. *SICStus Prolog Manual version 4.1.3*. Swedish Institute of Computer Science, September 2010.
`http://www.sics.se/sicstus/docs/latest4/html/sicstus.html`.

[50] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semant.*, 5(2):51–53, 2007.

[51] Mark E. Stickel. A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science*, 104(1):109–128, 1992.

[52] Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *Journal of Functional Programming*, 18:251–283, March 2008.

[53] TIOBE. TIOBE programming-community, TIOBE index, 2012. `http://www.tiobe.com`.

[54] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *TPLP*, 12(1-2):67–96, 2012.

[55] Zsolt Zombori. Expressive description logic reasoning using first-order resolution. *Journal of Logic and Computation*. Submitted for publication.

[56] Zsolt Zombori. Efficient two-phase data reasoning for description logics. In *IFIP AI*, pages 393–402, 2008.

[57] Zsolt Zombori. Efficient two-phase data reasoning for description logics. In Max Bramer, editor, *IFIP AI*, volume 276 of *IFIP*, pages 393–402. Springer, 2008.

[58] Zsolt Zombori. A resolution based description logic calculus. *Acta Cybern.*, pages 571–588, 2010.

[59] Zsolt Zombori. Two phase description logic reasoning for efficient information retrieval. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *ESWC (2)*, volume 6089 of *Lecture Notes in Computer Science*, pages 498–502. Springer, 2010.

[60] Zsolt Zombori. Two Phase Description Logic Reasoning for Efficient Information Retrieval . In John Gallagher and Michael Gelfond, editors, *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 296–300, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[61] Zsolt Zombori, János Csorba, and Péter Szeredi. Static Type Checking for the Q Functional Language in Prolog. In John Gallagher and Michael Gelfond, editors, *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 62–72, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[62] Zsolt Zombori, János Csorba, and Péter Szeredi. Static type checking for the q functional language in prolog. In John P. Gallagher and Michael Gelfond, editors, *ICLP (Technical Communications)*, volume 11 of *LIPIcs*, pages 62–72. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[63] Zsolt Zombori, János Csorba, and Péter Szeredi. Static Type Inference as a Constraint Satisfaction Problem. In *Proceedings of the TAMOP PhD Workshop: TAMOP-4.2.2/B-10/1-2010-0009*, Leibniz International Proceedings in Informatics (LIPIcs), Budapest, Hungary, 2012.

[64] Zsolt Zombori, János Csorba, and Péter Szeredi. Static Type Inference for the Q language using Constraint Logic Programming. In Agostino Dovier and Vítor Santos Costa, editors, *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 119–129, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[65] Zsolt Zombori and Gergely Lukácsy. A resolution based description logic calculus. In *Description Logics*, 2009.

[66] Zsolt Zombori, Gergely Lukácsy, and Péter Szeredi. Hatékony következtetés ontológiákon. In *17th Networkhop Conference 2008*, Budapest, Dunaújváros, 2008.

[67] Zsolt Zombori and Péter Szeredi. Szemantikus és deklaratív technológiák oktatási segédlet. Course handout.

[68] Zsolt Zombori and Péter Szeredi. Loop elimination, a sound optimisation technique for pttp related theorem proving. *Acta Cybernetica*, 20(3):441–458, 2012.

[69] Zsolt Zombori, Péter Szeredi, and Gergely Lukácsy. Loop elimination, a sound optimisation technique for pttp related theorem proving. In *Hungarian Japanese Symposium on Discrete Mathematics and Its Applications*, pages 503–512, Kyoto, Japan, 2011.

Appendix A

ALCHQ tableau rules

In this appendix we provide the rules of the tableau method. Even though the TBox reasoning starts out from an $\mathcal{SHQ}$ knowledge base, we quickly eliminate transitivity axioms during preprocessing and obtain an $\mathcal{ALCHQ}$ knowledge base. Accordingly, the rules provided in Figures A.1 and A.2 are those for the $\mathcal{ALCHQ}$ language. This appendix is not meant to explain how the tableau works. Instead, we provide it to make explicit what sorts of tableau rules we assume. For a comprehensive treatment of $\mathcal{SHIQ}$-tableau, we refer the reader to [29].

$\sqcap$-**rule**	
Condition:	$(C_1 \sqcap C_2) \in \mathcal{L}(x)$, x is not indirectly blocked and $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$.
New state $\mathbf{T}'$***:***	$\mathcal{L}'(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$.
$\sqcup$-**rule**	
Condition:	$(C_1 \sqcup C_2) \in \mathcal{L}(x)$, x is not indirectly blocked and $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$.
New state $\mathbf{T}_1$***:***	$\mathcal{L}'(x) = \mathcal{L}(x) \cup \{C_1\}$.
New state $\mathbf{T}_2$***:***	$\mathcal{L}'(x) = \mathcal{L}(x) \cup \{C_2\}$.
$\exists$-**rule**	
Condition:	$(\exists R.C) \in \mathcal{L}(x)$, x is not blocked and x has no R-neighbour y for which $C \in \mathcal{L}(y)$.
New state $\mathbf{T}'$***:***	$V' = V \cup \{y\}$ ($y \notin V$ is a new node), $E' = E \cup \{(x,y,\})$, $\mathcal{L}'((x,y,)) = \{R\}$, $\mathcal{L}'(y) = \{C\}$.
$\forall$-**rule**	
Condition:	$(\forall R.C) \in \mathcal{L}(x)$, x is not indirectly blocked, and x has an R-neighbour y for which $C \notin \mathcal{L}(y)$.
New state $\mathbf{T}'$***:***	$\mathcal{L}'(y) = \mathcal{L}(y) \cup \{C\}$.

Figure A.1: The transformation rules of the $\mathcal{ALCHQ}$ tableau algorithm, part 1.

$\bowtie$- **rule**	
Condition:	$(\bowtie nR.C) \in \mathcal{L}(x)$, where $\bowtie$ is one of the symbols $\geq$ or $\leq$, x is not indirectly blocked, and x has an R-neighbour y for which $\{C, \sim C\} \cap \mathcal{L}(y) = \emptyset$.
New state $\mathbf{T}_1$***:***	$\mathcal{L}'(y) = \mathcal{L}(y) \cup \{C\}$.
New state $\mathbf{T}_2$***:***	$\mathcal{L}'(y) = \mathcal{L}(y) \cup \{\sim C\}$.
$\geq$**-rule**	
Condition:	$(\geq nR.C) \in \mathcal{L}(x)$, x is not blocked, and it is not the case that there exist nodes $y_1, \ldots, y_n$ such that no two of them are identifiable, and for every i, y_i is an R-neighbour of x, and $C \in \mathcal{L}(y_i)$ holds.
New state $\mathbf{T}'$***:***	$V' = V \cup \{y_1, \ldots, y_n\}$ ($y_i \notin V$ new nodes), $E' = E \cup \{(x, y_1, ,) \ldots, (x, y_n, \})$, $\mathcal{L}'((x, y_i,)) = \{R\}$, $\mathcal{L}'(y_i) = \{C\}$, for every $i = 1 \leq i \leq n$, $I' = I \cup \{y_i \not\doteq y_j \mid 1 \leq i < j \leq n\}$.
$\leq$**-rule**	
Condition:	$(\leq nR.C) \in \mathcal{L}(x)$, x is not indirectly blocked, x has $n+1$ R-neighbours $y_0, \ldots, y_n$ such that $C \in \mathcal{L}(y_i)$ holds for every i, and there exist y_i and y_j that are identifiable.
For every $(0 \leq i < j \leq n)$, where y_i and y_j are identifiable, let $\{y, z\} = \{y_i, y_j\}$ so that x is not a successor of y:	
New state $\mathbf{T}_{ij}$***:***	$\mathcal{L}'(z) = \mathcal{L}(z) \cup \mathcal{L}(y)$, $\mathcal{L}'((x, y,)) = \emptyset$, $\mathcal{L}'((z, x,)) = \mathcal{L}((z, x,)) \cup \mathsf{Inv}(\mathcal{L}((x, y,)))$ if x is a successor of z, $\mathcal{L}'((x, z,)) = \mathcal{L}((x, z,)) \cup \mathcal{L}((x, y,))$ if x is not a successor of z, $I' = I[y \to z]$ (each occurrence of y is replaced by z).

Figure A.2: The transformation rules of the $\mathcal{ALCHQ}$ tableau algorithm, part 2.

Appendix B

Syntax of the Q Type Language

We provide the concrete syntax of the type language that we designed for the Q language. Note that all type expressions can be enclosed in parentheses, to improve readability, or to specify precedence (in the case of functions). Formally, this means that each syntactic rule whose head is of the form '*xxx* `type`' should be implicitly extended with a first alternative '`"(", `*xxx* `type, ")" |`'. To illustrate the point we have carried out this extension for the first rule, `type expr`, as shown by the text in italics.

```
type expr =
        "(", type expr, ")"
       | atom type
      | type variable
      | list type
      | tuple type
      | stuple type
      | dictionary type
      | record type
      | table type
      | function type
      | "any"
      | "cond"
      | "id"
      | "stuple" ;

atom type =
        "boolean" | "byte" | "short" | "int" | "long"
      | "real" | "float" | "numeric"
      | "char" | "string" | "symbol" | "hsymbol"
      | "month" | "date" | "datetime" | "minute" | "second" | "time"
      | "timestamp" | "timespan" ;

type variable =
      ? any Q identifier starting with a capital letter ?;

list type =
        "list", "(", type expr , ")" ;

tuple type =
        "tuple", "(", type expr,  ";", type expr , ")" ;
```

```
stuple type =
        "stuple", "(", symbol,  ";", symbol , ")" ;

dictionary type =
        "dict", "(", type expr, ";", type expr, ")"

record type =
        "record", "(", column names types, ")" ;

table type =
        "table", "(", "[", [column names types], "]", column names types, ")" ;

function type =
        type expr, "->", type expr ;

column names types =
        column name type, ";", column name type ;

column name type =
        column name, ":", type expr ;
```