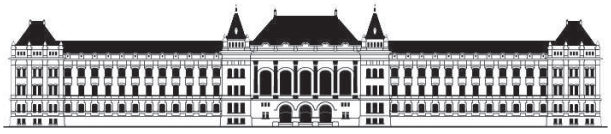

Budapest University of Technology and Economics

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 Dissertation summary

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

Budapest, Hungary
March, 2013

1 Introduction

Reasoning is the magic word that binds the parts 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 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 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.

1.1 Problem Formulation

In the following, I summarize my main research objectives and give some motivation for pursuing them.

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 transformation 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. Most PTTP implementations use an optimisation called loop elimination. However, its soundness 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 type errors 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 Booklet Overview

In Section 2 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. 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. In the end of this section, 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.

In Section 3 we present *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.

In Section 4 we present a reasoning algorithm that we developed to analyse programs written in the Q language. 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.

In Section 5 we present the DLog data reasoner system that can be used to query $\mathcal{RIQ}$ DL knowledge bases with really large data sets.

Finally, in Section 6 we present the `qtchk` type inference tool, that analyses Q programs and discovers type errors.

1.3 List of Publications

Journal Articles

1. [21] Zsolt Zombori: A Resolution Based Description Logic Calculus. In ACTA CYBERNETICA-SZEGED 19: pp. 571-588. (2010)

2. [30] 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. [19] Zsolt Zombori: Expressive Description Logic Reasoning Using First-Order Resolution. Submitted to JOURNAL OF LOGIC AND COMPUTATION.

Reviewed International Conference Proceedings

1. [20] Zsolt Zombori: Efficient Two-Phase Data Reasoning for Description Logics. In Artificial Intelligence in Theory and Practice II: World Computer Congress 2008. Milan, Italy, 2008.09.07-2008.09.10. Springer, pp. 393-402.(ISBN: 978-0-387-09694-0)

2. [27] 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. [23] 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. [22] 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. [31] 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. [24] 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. [4] 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. [26] 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. [5] 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. [28] Zsolt Zombori, Gergely Lukácsy, Péter Szeredi: Hatékony következtetés ontológiákon. In 17th Networkshop Conference 2008. Dunaújváros, Hungary, 2008.03.17-2008.03.19.

2. [25] 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. [29] Zsolt Zombori, Péter Szeredi: Szemantikus és deklaratív technológiák oktatási segédlet. Course handout. 2012. Budapest, pp. 1-32.

2 Resolution based Methods for Description Logic Reasoning

First, we present a resolution calculus in Subsection 2.1 that is a modified version of basic superposition and is specialised for $\mathcal{ALCHIQ}$ DL reasoning. In Subsection 2.2 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. Subsection 2.3 presents another resolution based calculus which, however, is defined directly on DL expressions, without recourse to first-order logic.

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

Motivated by [13], we cope with the complexity of reasoning by breaking it into two phases: a data independent phase on the general background knowledge (called the TBox, which is typically small and complex) and the real reasoning over the available data (called the ABox, which is typically large and simple). The two phases are called terminology and data reasoning, respectively. This separation has tremendous effect on efficiency. Besides being complex, 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 $\mathcal{SHIQ}$ DL knowledge base and we want to decide whether the knowledge base is satisfiable. It can be shown easily that this is sufficient for solving all other DL reasoning tasks.

In the first step we translate the knowledge base into first-order clauses. Doing this transformation carefully, it can be shown that the obtained clauses belong to a real subset of first-order logic. We call this set $\mathcal{ALCHIQ}$ *clauses*. Afterwards, the clause set is saturated using a modified version of basic superposition, called *modified calculus*. The main merit of this calculus is that it allows for performing all inferences related to function symbols before accessing the ABox.

The initial $\mathcal{SHIQ}$ DL knowledge base contains no functions. However, after translating TBox axioms to first-order logic, we have to eliminate existential quantifiers using skolemisation which introduces new function symbols. The ABox remains function-free, hence everything that is to know about the functions is contained in the TBox. This suggests that we should be able to perform all function-related reasoning before accessing the ABox. Indeed, this is what the modified calculus achieves: after saturating the TBox, we can eliminate clauses containing function symbols as they will play no further role. This makes the second phase of reasoning much simpler.

Proposition 1. *The modified calculus is sound and complete and the saturation of a set of $\mathcal{ALCHIQ}$ clauses using the modified calculus always terminates.*

2.1.1 Implementing Two-Phase Reasoning

We use the modified calculus to solve the reasoning task in two phases, 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 $\mathcal{SHIQ}$ TBox to a set of $\mathcal{ALCHIQ}$ clauses.
2. Saturate the TBox clauses with the modified calculus.
3. Eliminate all clauses containing function symbols.
4. Add the ABox clauses and saturate the set.

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

2.1.2 Benefits of Eliminating Functions

There are several advantages of eliminating function symbols before accessing the ABox. First, 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. Next, 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. Finally, ABox reasoning without functions is **qualitatively easier**. Some algorithms, such as those for Datalog reasoning are not available in the presence of function symbols.

2.2 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. In $\mathcal{SHIQ}$ we could express statements like $(\text{father} \sqsubseteq \text{relative})$, i.e., that a father is a relative. Complex role inclusion axioms allow for using composition of roles on the left side of such statements. For example, we can make the optimistic claim $(\text{spouse} \circ \text{mother} \sqsubseteq \text{friend})$, i.e., that a mother-in-law is a friend. 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. 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.

2.2.1 A Motivating Example

Before formally defining the transformation, we try to give an intuition through a small example. The example uses basic Description Logic syntax. Readers unfamiliar with DLs might prefer to skip this subsection.

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 concept inclusion axiom:

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

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

$$\begin{aligned} \forall R.C &\sqsubseteq X_1 \\ X_1 &\sqsubseteq \forall P.X_2 \\ X_2 &\sqsubseteq \forall Q.C \end{aligned}$$

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:

$$\begin{aligned} D &\sqsubseteq \forall R.C \\ \top &\sqsubseteq \forall R.D \end{aligned}$$

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:

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

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

2.2.2 From automata to concept inclusion axioms

Our results build on [9], which gives a tableau procedure for deciding $\mathcal{R}IQ$. For each role R, the authors define a non-deterministic finite automaton B_R 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. We show that the automata can be used to transform the initial $\mathcal{R}IQ$ knowledge base to an equisatisfiable $\mathcal{ALCH}IQ$ 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 $\mathcal{ALCH}IQ$ knowledge base can be used for satisfiability checking of a $\mathcal{R}IQ$ knowledge base. In particular, the two phase reasoning algorithm that we presented in Subsection 2.1 is applicable.

Proposition 2. *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 $\langle x,y\rangle \in S^I$.*

Proposition 2 states that two individuals are S-connected exactly when there is a role path w between them accepted by B_S.

In the following, we will make use of the notion of concept closure ($clos(KB)$), defined in [9], which contains all subconcepts appearing in KB.

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 1. *For any $\mathcal{R}IQ$ DL knowledge base KB, $\Omega(KB)$ is an $\mathcal{ALCH}IQ$ knowledge base constructed as follows:*

- *$\Omega(KB)_\mathcal{T}$ is obtained from $KB_\mathcal{T}$ by removing all role inclusion axioms $w \sqsubseteq R$ 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. $X_{(stop,R,C)} \sqsubseteq C$
- $\Omega(KB)_\mathcal{A} = KB_\mathcal{A}$

Theorem 2. *KB is satisfiable if and only if $\Omega(KB)$ is satisfiable. Furthermore, the size of $\Omega(KB)$ can be bounded exponentially in the size of KB.*

The transformation Ω maps an arbitrary $\mathcal{R}IQ$ knowledge base to an $\mathcal{ALCH}IQ$ knowledge base. Theorem 2 states that the transformation preserves satisfiability and increases the size of the TBox with at most an exponential factor, which is asymptotically optimal. Using this result, any algorithm that decides satisfiability for $\mathcal{ALCH}IQ$ can decide satisfiability for $\mathcal{R}IQ$. In particular, the modified calculus presented in Subsection 2.1 is applicable.

2.3 A Resolution Based Description Logic Calculus

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.

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 [1]. We prove that saturation terminates. Furthermore, we show that the knowledge base is inconsistent if and only if the saturated set contains the empty concept ($\bot$).

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 $\bot$. When the inference rules do not preserve disjunctive normal form, we use the explicit *dnf* operator.

We define a total ordering $\succ$ on DL expressions. Given a set N of concepts, concept $C \in N$ is *maximal* in N if C is greater (with respect to $\succ$) than any other concept in N. Since the ordering $\succ$ is total, for any finite set N there is always a unique maximal concept $C \in N$.

$\mathcal{SHQ}$-concepts The $\mathcal{SHQ}$ TBox is transformed into a set of *$\mathcal{SHQ}$-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 disjunctive normal form.

Inference Rules The inference rules are presented in Figure 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. 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 $\dfrac{C_1 \sqcup (D_1 \sqcap A) \qquad C_2 \sqcup (D_2 \sqcap \neg A)}{C_1 \sqcup C_2}$

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 $\dfrac{C \qquad W \sqcup (\geq nR.D)}{W \sqcup (\geq nR.dnf(D \sqcap E))}$

where E is obtained by using Rule1 on premises C and D

Rule3 $\dfrac{W_1 \sqcup (\leq nR.C) \qquad W_2 \sqcup (\geq kS.D)}{W_1 \sqcup W_2 \sqcup (\geq (k-n)S.dnf(D \sqcap \neg C))}$

$n < k, S \sqsubseteq^* R$, $(\leq nR.C)$ is maximal in $W_1 \sqcup (\leq nR.C)$
and $(\geq kS.D)$ is maximal in $W_2 \sqcup (\geq kS.D)$

Rule4 $\dfrac{W_1 \sqcup (\leq nR.C) \qquad W_2 \sqcup (\geq kS.D)}{W_1 \sqcup W_2 \sqcup (\leq (n-k)R.dnf(C \sqcap \neg D)) \sqcup (\geq 1S.dnf(D \sqcap \neg C))}$

$n \geq k, S \sqsubseteq^* R$, $(\leq nR.C)$ is maximal in $W_1 \sqcup (\leq nR.C)$
and $(\geq kS.D)$ is maximal in $W_2 \sqcup (\geq kS.D)$

Figure 1: TBox inference rules of the DL calculus

Along with the inference rules, we also use some *simplification rules* (not detailed in the booklet) that ensure that concepts always appear in their simplest form.

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 1 to deduce new concepts as long as possible. We claim that the consequent is always a $\mathcal{SHQ}$-concept (possibly after applying some simplification rules).

Proposition 3. *The set of $\mathcal{SHQ}$-concepts is closed under the inference rules in Figure 1 and only finitely many different $\mathcal{SHQ}$-concepts can be deduced from a finite TBox.*

Theorem 3. *The inference rules of the DL calculus are sound and complete.*

Let $\mathcal{T}$ be a $\mathcal{SHQ}$ TBox. Let $Sat_{\mathcal{T}}$ be the set of concepts obtained after transforming $\mathcal{T}$ into $\mathcal{SHQ}$-concepts and then saturating with the DL calculus. We have showed that saturation terminates. Furthermore, if $Sat_{\mathcal{T}}$ does not contain $\bot$ then it is possible to build a model for $\mathcal{T}$, i.e., it is satisfiable.

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

The Prolog Technology Theorem Prover approach (PTTP) [17] 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 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.

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* [11], which corresponds to the positive factoring inference rule.

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.

The DLog system [12] that will be presented in Section 5 is a specialisation of PTTP to Description Logic reasoning. DLog performs a two-phase reasoning, where the first phase uses the modified calculus of Subsection 2.1 and the second phase uses PTTP.

Loop elimination is an optimisation technique which prevents logic programs from trying to prove the same goal over and over again, thus avoiding certain types of infinite loops. Although both PTTP and DLog employ this optimisation, there has not yet been any rigorous proof of its soundness. My main contribution to this domain is providing such a proof.

Definition 2 (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. 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 turns out, 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.

Theorem 4. *For every complete PTTP proof containing loops there is a complete PTTP proof that is loop free.*

Theorem 4 justifies the use of loop elimination, which allows for reducing the search space, making both PTTP and DLog faster. Besides, loop elimination is sufficient to make the DLog reasoner terminating, thus allowing us to replace iterative deepening search with depth-first search, which further increases performance.

4 Type Inference for the Q Functional Language

We designed a type analysis tool for the Q vector processing language. 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 try to assign a type to each program expression in a consistent manner, 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 [6], [15].

4.1 Type Checking for the Q Language

Our type checking algorithm imposes some restrictions on Q programmers: they have to provide a type declaration for each variable and only ground declarations are allowed, i.e., type variables are forbidden. Both restrictions will be lifted in our type inference algorithm.

Algorithm 2 gives a summary of the type analysis component. We start out from an abstract syntax tree (AST) representation of the input program, constructed by the parser component. Our aim is to determine whether we can assign a type to each expression (each node in the AST) 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 2 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.

Constraints are handled using the Prolog CHR [15] 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 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.

4.2 Type Inference for the Q Language

In the second phase of the type analysis project, 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. We looked for a more flexible solution, where the analyser uses whatever information is available and infers as much as possible.

CSP We reformulate the task of type inference as a constraint satisfaction problem (CSP), which we solve using logic programming techniques. 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.

Constraints 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).

Constraint Reasoning Constraint reasoning is based on a *production system* [14], 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.

Our aim is to eventually eliminate all secondary constraints through the repeated firing of rules. 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.

In case some secondary constraints remain, we use *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 consistent type assignment that satisfies all constraints, or else we conclude the existence of a type error.

5 The DLog Description Logic Reasoner

The DLog system [12] 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 Section 2, 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 based on PTTP. 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`.

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.

Terminology Reasoning 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 2.1 and Section 2.2 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.

Future Work One of the most urgent tasks ahead of us is extending the system interface. Currently, we only support the DIG ([2]) format for the input knowledge base and query. We would like to provide the system with an OWL interface (see [8] and [7]). Moreover, we have already implemented the database support ([10]) 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 underpinning OWL2 ([7]).

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

We built a Prolog program called `qtchk` that implements the type analysis described in Section 4. The system 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.

More details of the system architecture are provided in Figure 2. 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.

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 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.

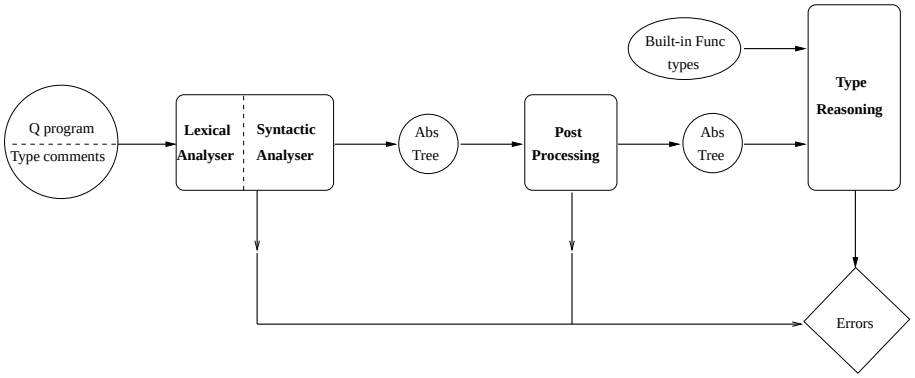

Figure 2: Architecture of the type analyser

`qtchk` runs both in SICStus Prolog 4.1 [16] and SWI Prolog 5.10.5 [18]. It consists of over 8000 lines of code.[1] 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 [3] that contains lots of examples along with the concrete syntax of the Q language.

7 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.

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. [21, 19, 20, 27, 22]

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. [19, 20, 22]

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. [19]

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 alter-

[1] We are happy to share the code over e-mail with anyone interested in it.

native to the tableau method. [21, 27]

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. [30, 31]

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. [30, 31]

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. [30, 31]

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. [24, 4, 26, 25, 5]

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. [24, 4]

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. [26, 25, 5]

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. [24, 4, 26, 25, 5]

References

[1] 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.

[2] 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`.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[12] 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.

[13] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany, January 2006.

[14] A. Newell and H.A. Simon. *Human Problem Solving*. Prentice Hall, Englewood Cliffs, 1972.

[15] 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.

[16] 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`.

[17] Mark E. Stickel. A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science*, 104(1):109–128, 1992.

[18] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *TPLP*, 12(1-2):67–96, 2012.

[19] Zsolt Zombori. Expressive description logic reasoning using first-order resolution. *Journal of Logic and Computation*. Submitted for publication.

[20] Zsolt Zombori. Efficient two-phase data reasoning for description logics. In *IFIP AI*, pages 393–402, 2008.

[21] Zsolt Zombori. A resolution based description logic calculus. *Acta Cybern.*, pages 571–588, 2010.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Zsolt Zombori and Gergely Lukácsy. A resolution based description logic calculus. In *Description Logics*, 2009.

[28] Zsolt Zombori, Gergely Lukácsy, and Péter Szeredi. Hatékony következtetés ontológiákon. In *17th Networkhop Conference 2008*, Budapest, Dunaújváros, 2008.

[29] Zsolt Zombori and Péter Szeredi. Szemantikus és deklaratív technológiák oktatási segédlet. Course handout.

[30] Zsolt Zombori and Péter Szeredi. Loop elimination, a sound optimisation technique for pttp related theorem proving. *Acta Cybernetica*, 20(3):441–458, 2012.

[31] 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.