Lightweight Semantic Web Oriented Reasoning in Prolog: Tableaux Inference for Description Logics

Thomas Herchenröder

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2006

Abstract

This work presents the reconstruction of a Tableaux-based reasoner for Description Logics in Prolog. Tableaux-based reasoning is a preferred style of reasoning about Semantic Web ontologies expressed in Description Logics and OWL. We will present the traditional Tableaux algorithm for Description Logics, discuss some of its properties and investigate options for its implementation. A refined version of the algorithm is developed and a concrete implementation in Prolog, *tableaux.pl*, is proposed and compared to other implementations, both in terms of design and performance.

Acknowledgements

My heartfelt gratitude goes to my supervisors Dave Robertson, Fiona McNeill and Stephen Potter, who all provided me with guidance, insights and cheerful support. To Stuart Aitken, who helped resolve questions concerning the Tableaux algorithm, provided his own implementation for comparison, and who taught me Description Logics and Tableaux in the first place. To Agnieszka Ciulkin, who undertook the effort of translating a Polish article about something she had never heard of before. To Adam Meissner, who helped in the translation of his article and provided me with a complete and updated version of his Tableaux implementation. To Racer Systems GmbH & Co, KG, who provided me with a fully functional, educational license for their RacerPro Description Logics reasoner for the time of my dissertation. To my current employer, Software AG in Germany, who allowed me a leave of absence to pursue the MSc programme without me quitting the job. A special acknowledgement goes to F.C.N. Pereira and S.M.Shieber. Their exceptional book on Prolog and natural language analysis ([1]) continues to be a source of inspiration and insight into the art of Prolog.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Thomas Herchenröder*)

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Background in a Nutshell: From Semantic Web to Tableaux Reasoning in Prolog

The promise of the *Semantic Web* is to allow machines, programs, to make sense of Web resources in ways this is only possible for humans today. In order to achieve that, Web resources like Web pages, images and so on, have to be enriched with *meta-data*, data that *describes* the particular resource. The general problem with meta-data today, like the HTML `META` tag, is that their use is too diverse to do automatic inference with. In order to make Web resources mechanically comparable, meta-data has to use *common terms and relations* to describe them. They need to draw on common vocabularies. And for many real-world domains, like medicine, biology, physics and law, such vocabularies exist, albeit again with variations. A common technique to organise and unify terms that pertain to a certain knowledge domain is to create *ontologies* for this domain, which organise these terms in hierarchical concept trees, together with attributes and additional relations. A well-established formalism to express ontologies are *Description Logics* and technologies built upon them, like OWL. Description Logics provide a logic-based foundation to build ontologies of high complexity and expressiveness. Once ontologies have been created and meta-data using them to describe different Web resources has been installed, standard procedures can be deployed to make *inferences* over the given meta-data, using the common ontology. One of those standard procedures to reason over Description Logics is the *Tableaux algorithm*, and this work is about re-creating Tableaux reasoning for Description Logics in *Prolog*.

1.1.2 A First Example

A simple illustrative example is the following. A person interested in buying a new car could charge an *agent* with this task. The agent would have to roam the Internet and come back with interesting pages. The agent would come across a Web page that details about cabriolets. Finding the term "cabriolet" in the page's meta-data, and knowing about a vehicle ontology, the agent would then be able to find out that cabriolets are a special kind of cars, and that therefore this page might be of interest for its human.

The formal task the agent has to solve here is to find out that the ontology term "car" (which it started with) and "cabriolet" (which it found for the Web page) are in a *subsumption relation*. Rather than trying to traverse a (possibly huge) taxonomy graph of the car ontology, trying to keep track of parent and child nodes, it uses inference to answers the query *"Are cabriolets a special kind of cars?"*; or more formally *"Cabriolet* $\sqsubseteq$ *Car?"*; or even more complex *"Cabriolet* $\sqcap \neg$*Car* $\sqsubseteq \bot$*?"* (all of which should become clearer later in this thesis). This is a typical scenario for using Description Logics reasoning on the Semantic Web.

1.1.3 Semantic Web Reasoning

Description Logics are a well-investigated, set-theoretical founded subset of First-Order Logic. They provide means to express concepts (or classes) and their relations among each other. The simplest representative of this class of logics is called $\mathcal{ALC}$[1] , but many more exist, each with varying degrees of additional language elements (They are easy to spot since they are conventionally written in script-like letters, such as $\mathcal{ALCN}$, $\mathcal{SHIF}$ or $\mathcal{SHOQ}(\mathcal{D})$). They are interesting for the Semantic Web since they provide enough expressiveness to be useful, yet retain good computational properties to be feasible in realistic applications.

There is more than one way to reason about Description Logics, but Tableaux-based reasoning has proven to be fast and efficient, with a lot of theoretical findings being applicable to Description Logics, mainly from modal logic. Therefore, Tableaux-based reasoners are state-of-the-art among available DL/OWL reasoners. Among the prominent systems are FaCT, Racer and Pellet, which all strive for high performance and a high coverage of DL language features. Implementation languages for those

[1] The acronym $\mathcal{ALC}$ stands for the rather unintuitive term "***A**ttributive Concept Description **L**anguage with **C**omplements*".

reasoners include Lisp, Java, C and C++.

1.1.4 Why Prolog?

A Prolog-based implementation explores the strengths of logical, declarative programming on the one hand, building on Prolog's proverbial strengths in implementing reasoners. On the other hand, such an implementation would be usable for other Prolog programmers who want to integrate Description Logics reasoning in their application. Particularly, implementers of Semantic Web reasoning agents might be interested in having a DL reasoner library ready to use. Alternatively, the library could be integrated in validation and reasoning services for particular ontologies, where a small footprint and easy access to the source code are desirable. Agents could make use of such a service to solve their reasoning tasks (which is probably more likely than the agents reasoning themselves).

Other beneficiaries might be end-users that look for an accessible, interactive interface to a standard DL reasoner, to quickly check ontologies and their properties. And taking it a step further, the reasoner could be integrated with ontology editors like OilEd and Protégé, providing an alternative to e.g. Racer.

1.2 Goal

The goal of this thesis is to re-construct a Tableaux reasoner for a basic Description Logic ($\mathcal{ALC}$) in Prolog, which is usable for end-users and other Prolog programmers and can be integrated in Semantic Web applications. We want to show the feasibility of such an implementation, the design options, and compare its performance and runtime behaviour.

1.3 Methods

The concrete steps to achieve a Prolog implementation of the Tableaux proof algorithm for Description Logics are:

- Define a program-internal representation for the logic language, to represent the logical expressions for knowledge bases and queries to the proof predicates. These will be dedicated Prolog terms.

- Implement the proof predicates that reason over the knowledge bases, answering the queries.
- Discuss and compare the implementation with regard to the literature and other implementations, considering design alternatives.
- Select and deploy a set of test ontologies and queries to measure and compare runtime performance.
- Supply necessary tools to convert ontology representations and run test suits.

Information concerning the design and implementation of the reasoner, and the chosen test cases and the reference systems for the comparison will be detailed in later chapters.

1.3.1 Features inside and outside the Scope of this Project

Ontology Size. No consideration will be given to issues that come with very large ontologies (many thousands of classes), which might make the transformation into a pure in-core representation infeasible on standard hardware.

Ontology Formats. Available ontologies come in various formats, OWL, Lisp-like or proprietary, to name just a few. The project will only cover filters that transform data so that it is possible to run the tests and benchmarks.

Query Capabilities. The query interface will only support satisfiability queries that cover

- a compound concept
- equivalence of two concepts
- subsumption between two concepts
- disjointness of two concepts
- unsatisfiability of a concept

Interfaces. There will be no particular focus on user interface considerations. The targeted interactive environment is the Prolog shell, which allows access to interface predicates.

1.4 Evaluation

A specification of the implementation will be given, to convince the reader about the correctness of the implementation. The implementation will be tested by reading in various ontology definitions and running qualified queries with known outcome against them. Other implementations will be used to compare runtime performance on these tests. The outcome of these comparisons will be analysed and commented.

1.5 Outputs

The output of the project is the Prolog code implementing the reasoner and all filters and interfaces developed alongside, available in the form of Prolog modules (libraries). Furthermore, there will be results of various tests and benchmarks, and the analyses thereof.

1.6 Structure of this Document

Chapter 2 continues to discuss the foundations of Description Logics. Chapter 3 introduces the Tableaux algorithm for reasoning with Description Logics. Chapter 4 introduces an implementation of this algorithm and discusses various issues and design alternatives. Chapter 5 presents a formal specification of the presented implementation. Chapter 6 continues to compare this implementation with reference systems, evaluating their runtime performance. Chapter 7 lists possible extensions and enhancements that could be applied to the current system, and chapter 8 closes with some conclusions.

Chapter 2

Description Logics

2.1 Description Logics

Description Logics [2] are, as the name suggests, a whole family of related logics, which have a set-theoretical foundation. The most basic one, $\mathcal{ALC}$, can represent primitive concepts, which are interpreted as sets. They are not further defined in the logic, but receive their contents through an interpretation (e.g. a model). The logic provides the default concepts "bottom" ($\bot$), the empty set, and "top" ($\top$), the universe. Moreover, it lets one name binary relations called roles that hold between concepts. In DL formulae, concepts and roles are usually denoted with uppercase letters ($A, B, C, \ldots$).

Role expressions and concept formation operators let you construct further concepts from primitive ones. The operators are negation ($\neg$), union ($\sqcup$), intersection ($\sqcap$), value restriction ($\forall$) and existential restriction ($\exists$). Negation, union and intersection take the usual intuitive meanings. Universals and existentials follow a slightly different notion from first-order logic. Basically, they describe concepts as sets of individuals that are characterised (or "restricted") by the individuals they relate to through a given relation. Value (or universal) restriction is formally defined as

$$\forall R.C = \{\ \mathrm{x} \mid \forall \mathrm{y}.\ \mathrm{R(x,y)} \Rightarrow \mathrm{y} \in \mathrm{C}\ \}.$$

This is the set of all individuals that, if they take part in the R relation at all, are related through R to only individuals of the concept C.[1]

[1] This gives rise to a somewhat counter-intuitive property of Description Logics, by which individuals that are not at all in the domain of R qualify for the universal restriction, due to an implication being true when its premise is false. As a consequence, $\forall R.C$ and $\forall R.\neg C$ are not necessarily contradictory, i.e. $\forall R.C \sqcap \forall R.\neg C$ might be non-empty.

Existential restriction is formally defined as

$$\exists R.C = \{ \text{ x} \mid \exists \text{y. R(x,y)} \wedge \text{y} \in \text{C} \}.$$

This is the set of all individuals that are related through R to at least one individual of the concept C. A member of this set has to be from the domain of R, and at least one of its relational values has to be a member of C.

Here are a few examples:

- $\neg C$ Everything that is outside of C; or in set-theoretical terms, the complement of C
- $C \sqcup D$ The concept represented by the union of C and D (C 'or' D logically)
- $C \sqcap D$ C intersected with D (C 'and' D logically)
- $\exists R.C$ The set of all individuals that are in relation R to at least one individual from concept C
- $\forall R.C$ The set of all individuals that are in relation R to only individuals from concept C

With these operators you can construct arbitrary complex concept expressions. To use such concept expressions for axioms, two further operators are introduced: subsumption ($\sqsubseteq$) and equivalence ($\equiv$). This allows you to relate concept expression to one another:

$$C \sqsubseteq D$$
$$C \equiv D$$

The interpretation of a subsumption expression like the one above is that D is a *necessary* condition for C (sometimes also expressed as "D $\rightarrow$ C"). Complex concept expressions can be used to define new concepts, using the equivalence operator ($\equiv$), e.g. $E \equiv C \sqcap D$. To contrast with complex expressions (i.e. expressions containing roles or constructors), C, D and E are called *atomic* concepts. Atomic concepts that have no defining axiom (i.e. they are simply "there") are called *primitive*. The interpretation of a defining expression like the one above is that the expression on the right hand side is *necessary and sufficient* for the left hand side.

Description Logics represent subsets of first-order logic, with decision procedures that are tractable and have known computational complexity. This makes them interesting for automatic reasoning, while the semantics of formulae is provided by a well-defined model theory.

2.1.1 Grammar

Here is the grammar for the various expressions ([3, lect.3], but also cf. e.g. [4, p. 51,55]):

Concept expression :: $\bot \mid \top \mid CN \mid \neg C \mid C \sqcup D \mid C \sqcap D \mid \exists R.C \mid \forall R.C$

Terminological axiom :: $C \equiv D \mid C \sqsubseteq D$

where CN is a primitive concept ("concept name"), C and D are arbitrary concepts, and R is a role.

2.1.2 Knowledge Bases

Knowledge bases expressed in Description Logics are usually divided into two sections

- The *TBox* ("terminology"), which contains the terminological axioms, i.e. statements that give the basic relations between concepts in terms of equivalence or subsumption.
- The *ABox* ("assertions"), which assigns concrete individuals to concepts (*Ben* $\in$ *Giraffe*), and lists relations between individuals (*loves(Yogi,Henriette)*). ABoxes as part of a given knowledge base are not further considered in this text[2].

Given such a knowledge base and a DL reasoner, the KB can then be queried, either using DL expressions representing concepts in the given logic, or by specific meta-queries such as whether the TBox as a whole is consistent, or to construct a complete subsumption hierarchy of the contained concepts. These query capabilities depend on the reasoner at hand.

2.2 ... and More

This basic Description Logic, $\mathcal{ALC}$, can be extended in various ways to add expressiveness while still retaining good computational properties. Common extensions are (cf. also [5, pp.39–41]):

- ***cardinality constraints*** – where the cardinality of the relation is constraint in a "not less than" (e.g. "$\geq$ 2 *hasChildren*" for people who have 2 or more children) or "no more than" (e.g. "$\leq$ 3 *hasCars*" for people who have no more

[2] But see section 3.1.1.

than 3 cars) way. Both constraints come in an unqualified and a qualified variant, where the range of the relation is further restricted to a given concept (e.g. "$\leq$ 3 *hasCars.Cabriolet*").

- ***transitive and inverse roles*** – where roles are closed under transitivity, and where you can specify a role to be the inverse of another role ("hasPart" – "isPartOf").
- ***functional roles*** – where each individual of the relation domain has at most one relation to an individual from the relation range ("hasMother").
- ***role hierarchies*** – where **roles** are in a subsumption hierarchy (e.g. *hasParent* $\sqsubseteq$ *hasAncestor*).
- ***nominals*** – where named individuals can occur in concepts ("PrimeMinister $\equiv$ $\{$ tony_blair $\}$").
- ***enumeration sets*** – where concepts can be enumerated in a finite set (Weekday $\equiv$ $\{$ monday, tuesday, wednesday, thursday, friday, saturday, sunday $\}$; often used in conjunction with nominals.
- ***data types*** – where data types such as integer and strings are added to the language, e.g. to express data type constraints on objects.

These extensions lead to Description Logics such as $\mathcal{ALC}(\mathcal{D})$, $\mathcal{SHIF}$, $\mathcal{SHOIN}$, $\mathcal{SHOQ}(\mathcal{D})$ and many others[3].

2.3 Relation to OWL

OWL [6] can be looked at as an encoding for certain Description Logics [7, p.4], mainly basing it on the $\mathcal{SH}$ family of Description Logics [7, p.6]. The W3C defines three levels of OWL:

- *OWL Light* – which represents $\mathcal{SHIF}$(**D**), which is $\mathcal{ALC}$ with transitive, functional and inverse roles, role hierarchies and data types; this is the language level current reasoners like FaCT and Racer work on. The language is of worst-case deterministic EXPTime complexity [7, p.19].

[3]The "Description Logic Complexity Navigator", http://www.cs.man.ac.uk/∼ezolin/logic/-complexity.html, is a nice and intuitive way to explore these mappings between language extensions and acronyms.

- *OWL DL* – which represents $\mathcal{SHOIN}(\mathbf{D})$, which is $\mathcal{ALC}$ with transitive and inverse roles, role hierarchies, unqualified cardinality constraints, nominals and data types. The language is of worst-case non-deterministic EXPTime complexity [7, p.19].

- *OWL Full* – which allows a maximum of freedom, with no computational guaranties; concepts can be seen as sets or individuals in their own right; the language can be extended by the user; close to FOL.

2.3.1 XML Encoding

From a syntactic point of view, OWL is built on top of XML and RDF(S). From XML it inherits the core syntactic structure, such as entities and the structure and nesting of opening and closing tags. From RDF(S) it inherits schema elements such as rdf:type and rdfs:subClassOf. I will not go into detail of these technicalities, but just to give you a flavour of what "real-life" OWL looks like, here is a brief example.

Suppose you wanted to model the set of people, at least one of who's parents is a physician, but who themselves are not physicians. In DL you would express this as

$$\exists \mathit{hasParent}.\mathit{Physician} \sqcap \neg \mathit{Physician}$$

In OWL, this would look somewhat like this:

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasParent" />
      <owl:someValuesFrom rdf:resource="#Physician" />
    </owl:Restriction>
    <owl:Class>
      <owl:complementOf>
        <owl:Class rdf:about="#Physician"/>
      </owl:complementOf>
    </owl:Class>
  </owl:intersectionOf>
</owl:Class>
```

This may give you an impression of how OWL encodes logical expressions.

Chapter 3

The Tableaux Algorithm

3.1 Tableaux – An Overview

The historical development of Description Logics languages and corresponding subsumption algorithms tried to tackle the fine line between expressiveness of the language and tractability of the algorithm [8, p. 6]. Early proposals were the structural subsumption algorithms (ibid.), which proved to be complete only for very inexpressive DLs. Baader and Sattler state that the CLASSIC system used "a restricted DL that still allowed for an (almost) complete polynomial structural subsumption algorithm" (ibid.).

The Tableaux (also "Tableau") algorithm is explained e.g. in Baader et al. [9]. Interesting hypotheses in DL include satisfiability (whether a concept is non-empty in the model), subsumption (whether one concept subsumes another), equivalence and disjointness of two concepts. All these relations can be reduced to either subsumption or unsatisfiability. The Tableaux algorithm is a procedure to decide unsatisfiability, i.e. the initial formula is re-written as a corresponding formula that makes it possible to test for satisfiability. For instance, the hypothesis $C \sqsubseteq D$ leads to the goal $C \sqcap \neg D$, its inverse, since a concept C is subsumed by another concept D *iff* the intersection of C with the complement of D is empty. But the second expression can be tested for satisfiability, i.e. it can be tested whether the intersection between the two concepts is empty or non-empty. This is not possible with the first expression (which would require to show that each individual in C is necessarily also in D, in order to prove the hypothesis). But falsifying (i.e. proving the non-satsifiability of) the second expression proves the satisfiability of the first.

Hypotheses ("queries") therefore undergo a set of transformational steps, before

the basic proof algorithm sets in. These steps are:

1. Recursively replacing all concepts by their definitions, until only primitive concepts are in the formula (also called “unfolding” or “TBox-elimination”).

2. Transforming the main relation into a corresponding expression that allows for satisfiability testing. The resulting formula is called the *goal*.

3. Putting the expression into Negation Normal Form, where all negations move “inwards” in subexpressions, until only primitive concepts are negated.

Now the Tableaux algorithm can start. In its most basic form it uses four rules to come to a decision: intersection elimination, union elimination, existential elimination and universal elimination. The goal is considered the label of the root node of a proof tree. Then, application of the proof rules transform the node, add elements to its label set, and spawn new nodes expanding the tree structure where the application of rules is repeated. This is continued until no more rules can be applied to any of the nodes in the tree. At this point, the proof tree (or the “tableaux”) is said to be *saturated*. If any of the branches of the tree contains an obvious contradiction, that is an expression C and its negation $\neg C$, this branch is said to be *closed*. Branches where this is not the case are called *open*. The initial goal of the proof is *satisfiable* iff the tree does not contain any closed branches.

The first two rules, intersection elimination and union elimination (also *and*- and *or*-elimination), operate directly on the formulae of the current node. The application of the intersection rule adds the two operands to the node’s label set, i.e. to the set of DL expressions that are associated with this node. Union connectives allow the addition of their first operand, and if the whole tree fails, the addition of their second operand is tried as an alternative. The following short hand shall give you a first impression of the two rules (all rules are introduced more formally in the next section):

- $x : \{...C \sqcap D...\} \longmapsto x : \{...C, D...\}$
- $x : \{...C \sqcup D...\} \longmapsto x : \{...C, ...\}$ or $x : \{...D, ...\}$

x identifies the current node, the list in braces denotes its node label, consisting of various Description Logics expressions. In the first example, the node label contains an intersection. Applying the *and*-rule transforms the node label to the expression to the right of the arrow: the two operands of the intersection are added as individual

members to the node label. Similarly, in the second example the *or*-rule adds its first or second operand to the node's label.

The existential elimination creates an edge and a new node, thereby extending the proof tree. The edge is labelled with the role name that was governed by the existential. The restricting concept of the existential is carried over to the new node. The universal elimination depends on an existing edge which is labelled with the name of the role governed by the universal. Then the restricting concept of the universal is carried over to the corresponding node, if this node does not contain it already. In short, for nodes x and y and an edge labelled with R, the result of the two operations would be:

- $x : \{...\exists R.C...\} \longmapsto x : \{...\}$ —R— $y : \{C\}$
- $x : \{...\forall R.C...\}$ —R— $y : \{...\} \longmapsto x : \{...\}$ —R— $y : \{...,C\}$

In the first example, the node *x* contains an existential expression in its node label. Applying existential elimination (shown to the right of the arrow), the node gets a child node *y*, connected by an edge which is labelled with the name of the existential's relation *R*. The existential's concept, *C*, is the initial contents of *y*'s node label. In the second example, *x* contains a universal expression, and is additionally connected to a child node *y* through an edge that has the same name as the universal's relation. Then universal elimination can be applied, leading to the situation to the right of the arrow, where the universal's concept expression *C* is added to the child's node label.

The process is carried on, until there are no more rules to apply (i.e. until the tableaux is saturated). If a contradiction shows up on any of the branches of the tree, listing both a concept and its complement in the same node, that closes the tableaux and proves the unsatisfiability of the goal. If this is not the case, then there is a satisfying model for the goal.

Appropriate conclusions have to be drawn to answer the initial hypothesis. For instance, in the above example, $C \sqsubseteq D$ would be true, if $C \sqcap \neg D$ is unsatisfiable. The complete list of rules is given in Table 3.1, a worked-out example of their application in section 5.7.

3.1.1 Constructive Proofs and ABoxes

The proof procedure is constructive, in that upon success the constructed tree represents a satisfying model. The nodes represent individuals (i.e. the constraints that characterise them), and the edges represent the relations among them. In DL terms,

$\sqcap - rule$	*if*	1.	$(C_1 \sqcap C_2) \in \mathcal{L}(x)$
		2.	$\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
	then		$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup - rule$	*if*	1.	$(C_1 \sqcup C_2) \in \mathcal{L}(x)$
		2.	$\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
	then		a. save **T**
			b. try $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1\}$
			If that leads to a clash then restore **T** and
			c. try $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_2\}$
$\exists - rule$	*if*	1.	$\exists R.C \in \mathcal{L}(x)$
		2.	there is no y s.t. $\mathcal{L}(\langle x, y \rangle) = R$ and $C \in \mathcal{L}(y)$
	then		create a new node y and edge $\langle x, y \rangle$
			with $\mathcal{L}(y) = \{C\}$ and $\mathcal{L}(\langle x, y \rangle) = R$
$\forall - rule$	*if*	1.	$\forall R.C \in \mathcal{L}(x)$
		2.	there is some y s.t. $\mathcal{L}(\langle x, y \rangle) = R$ and $C \notin \mathcal{L}(y)$
	then		$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$

Table 3.1: Tableaux Inference Rules for $\mathcal{ALC}$ [5, p. 48]

this would be constructing an ABox. Especially Baader uses this nomenclature in his publications to signify the various trees created during a proof (e.g. [10, 8, 4]). This can be a little confusing, because all we are doing here is *TBox-reasoning* (cf. p. 8). This means that the knowledge base we are reasoning over contains only a TBox, a set of terminology axioms. $\mathcal{ALC}$ has no means in the language to express an ABox, and the given algorithm does not handle ABox statements in the proof.

To illustrate this, here is an example. The knowledge base

$$\begin{aligned} Giraffe &\sqsubseteq Mammal \\ Ben &\in Giraffe \\ likes(Ben, Ben) & \end{aligned}$$

has a TBox, expressing that giraffes are subsumed by mammals, and an ABox, saying that the individual Ben is a giraffe (instance) and that he likes himself (relation). $\mathcal{ALC}$ cannot express the second and third statement, and the given Tableaux algorithm could not reason with them. Nevertheless the proof procedure does construct individuals and relations among them. The formalisation of the rules refers to those individuals with arbitrary (hence usually unintuitive) names (x, y, z, ...), and during the proof they are

just represented by their individual nodes in the tree and the contained node labels, which lists their defining constraints. The edges in the tree represent the relations between those individuals. Hence, this can be regarded an (constructed, rather than provided) ABox.

3.2 The Proof Rules

The rules of Table 3.1 represent the inference rules for the basic $\mathcal{ALC}$ Description Logic. Adding operators and constructors to the language leads to additional rules for the proof algorithm that take care of these additional elements (e.g. for cardinality constraints [4, p.85]). Other variants for proof rules exist that seek to optimise inference performance (cf. e.g. [5, chap.6]).

In Table 3.1, C, C_1 and C_2 are arbitrary DL concepts, R is a relation, **T** denotes the whole proof tree, while x and y denote specific nodes in the tree. Finally, $\mathcal{L}(x)$ signifies the node label of node x, which is the set of DL formulae associated with this node. Each rule has two preconditions (the if part) and an action (the *then* part). Preconditions test the applicability of the rule. The first condition in each rule simply ascertains the availability of an appropriate term to apply the rule to (which can be thought of as a term in the head of a clause that is used for pattern matching). The second rule adds more specific tests for each rule that usually check whether the results of applying the rule already exist in the given node. The action part describes the actual changes performed when applying the rule. Below I give an informal description of each rule.

The $\sqcap$-rule applies to intersection terms within a given node label. The second condition makes sure that both operands are not already available in the current node label. The action part adds both operands as new members to the set that constitutes the label of the current node.

The $\sqcup$-rule applies to union terms within the given node label. The second condition makes sure that none of the operands are already available in the current node label. The action part adds the first operand as a new member to the node label. If this tree eventually fails, the original tree at this point in the proof is restored and the second operand is added to the node label, to attempt satisfiability of this alternative tree.

The $\exists$-rule applies to existential restriction terms in a node label. The second condition tests for an identical edge to an existing child node, and an identical concept

within that child. If either part of the condition is false, the action part of the rule creates a new node and a new edge to this node labelled by the relation. The constraining concept of the existential becomes the initial concept in the new node's label.

The $\forall$-rule applies to universal restriction terms in a node label. The second condition searches for an existing child node with identical edge label, but without the constraining concept of the universal. If such a child node is found, the constraining concept is added to the child's node label.

3.3 Properties

The Tableaux algorithm given in Tab. 3.1 is sound and complete, and will always terminate using exponential time and space complexity (cf. e.g. [8, p.12]).

The *termination* property derives from the fact that the algorithm only adds subexpressions of the initial formula, which are strictly smaller than their parent expression. Since the initial expression is finite, there are only so many subexpressions to expand. Branching factor and depth of the proof tree depend on the number of existentials in the initial formula, which again is limited. Loops in expanding terms are prohibited by the rule conditions.

Soundness and *completeness* rely on the equivalence of the satisfiability of the initial formula and the consistency of the saturated proof trees, which can be shown in part by induction using a *canonical model* and exploiting the *finite tree model* property [8, p.13]. *Complexity* of the algorithm is exponential in time and space for certain initial formulae (worst-case, cf. again [8, p.13]).

3.4 Non-Deterministic Tree Expansion: AND- and OR-Rules

One thing to notice when looking at the Tableaux algorithm at work is that its expansion of the proof tree is not deterministic. It is not only not deterministic in the way the tree unfolds due to non-deterministic choices in the order of the rule applications, it is also non-deterministic in the set of trees it creates and the shape of the individual trees.

Consider the following example. Let $\{(P \sqcap \forall R.\neg P),\ (P \sqcup \exists R.P)\}$ be the label of the root node in our proof tree. The two available expressions are an intersection and a

union expression. There is no constraint on which to choose first for a rule application. But depending on whether you choose to eliminate the intersection or the union first, you get different proof trees (Fig. 3.1).

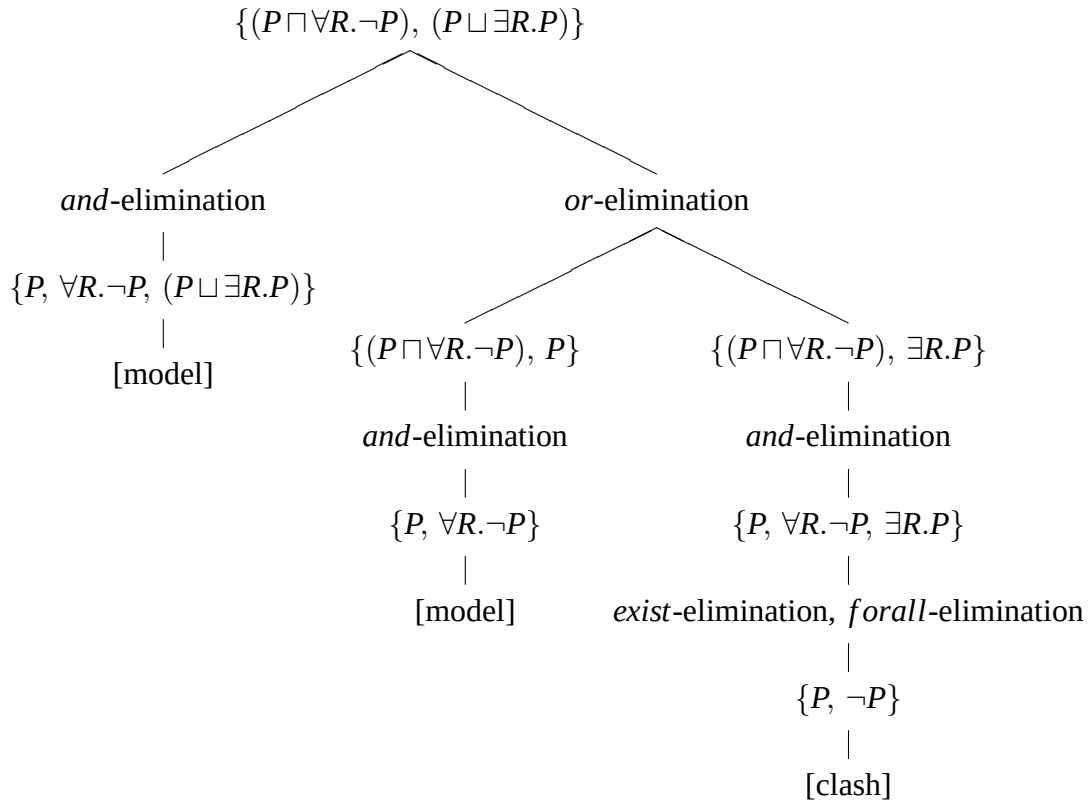

Figure 3.1: Search Space of a Proof, Depending on Rule Choice

Note: This does **not** represent the logical *proof tree*, as created by rule applications, but the *search tree* of the algorithm, showing fringe nodes interlaced with rules applied.

The important aspect is the choice of the *and*- or *or*-rule on the first level. One leads to a single tree (with a single branch), the other to two alternative trees (each with a single branch). As you can see, choosing *and*-elimination first leads to a node where no more rules can be applied (the *or* is blocked due to the presence of *P* in the node). Since there is no obvious contradiction, this constitutes a model: the initial goal is satisfiable.

Choosing the *or*-rule first gives you an entirely different picture. The proof tree is split into two, the left of which leads to a model and the right leads to a clash. Since a single open tree among alternative trees is enough for a successful proof, the result here is the same: the goal is satisfiable. But the difference in how the proof evolves is significant. Choosing the *and*-rule first leads to a single tree, with a single branch of

depth 1. Choosing the *or*-rule first leads to two alternative trees, both single-branched, the first of which has depth 1, and the other having depth 2.

Given the soundness and completeness results of the algorithm, that means that all those possible variants in proof evolution are insignificant for the proof result[1]. But seen as a rewrite system, the Tableaux rules are not confluent, i.e. exhaustive application of the rules on a given node does not necessarily always lead to the same saturated tree or set of trees. The proof tree evolves differently, depending on the rewrite rule you choose to apply. There is no "canonical form" to which all trees eventually converge, over possibly different intermediate stages. In other words: Even in proofs where there is no union elimination (and therefore no alternative trees to test), the proof rules do not construct a unique model. This is further explained in the next section.

However, since the constructed model is not significant for the decision procedure, which is only concerned with success or failure, these differences do not interfere with the proof result. Nevertheless, they might be interesting from a computational point of view, where one might be interested in finding e.g. a *minimal* model for performance reasons.

3.5 Non-Deterministic Tree Expansion: Exists-Rule

The the second condition on the $\exists$-rule deserves special attention. The condition, which requires that

$$\text{"there is no } y \text{ s.t. } \mathcal{L}(\langle x,y\rangle) = R \text{ and } C \in \mathcal{L}(y)\text{" [p. 14]}$$

concerns existing child nodes of the node under investigation. The first part looks at the relation that labels the relation between parent and child, and checks whether the relation is the same as that of the existential under investigation. If there is no such child, a new child node may be created.

If, on the other hand, a child node with such a relation exists, the next question is whether the existential's concept, C, is already contained in the child node's label. If so, the expansion of the existential is blocked. If not, again a new node may be created, connected to the current node by the same relation edge. The new node represents a different individual. Only through this interpretation of the rule, obviously satisfiable concepts produce a successful proof and a constructed model such as one would expect, as in Fig. 3.2.

[1]Try changing the initial expression in the example to get a clash in the *and* variant, and you get

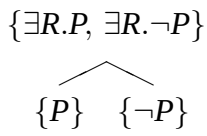

Figure 3.2: Proof Tree for two Existentials ranging over the same Relation

But since again the order of rule application is open, a child might contain a non-expanded expression that *contains* the relevant concept C, e.g. $\{P \sqcap C\}$. Now, the test $C \in \mathcal{L}(y)$ fails, and a new node is created (Fig. 3.3). A later expansion of the intersection in the child node would reveal the concept that then would block the expansion of the parent's existential. But at that point the second child is already created. Fig. 3.4 shows the situation, if the child node is expanded first, and therefore the expansion of the second existential in the parent is blocked.

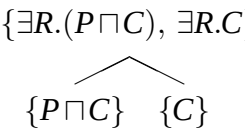

Figure 3.3: Proof Tree with Expanded Second Existential due to "hidden" C in Child

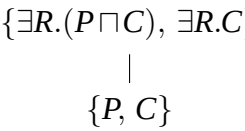

Figure 3.4: Proof Tree with Blocked Second Existential after *and*-expansion in the Child

The examples from this and the previous section show how differently a proof tree might evolve, depending on the choice and the order of rule applications. Significantly different proof trees emerge. The theoretical result for the Tableaux algorithm assure its soundness, but these observations reveal that there is a lot of variation going on "under the hood".

---

clashes in all trees of the *or* variant.

Chapter 4

Implementing Tableaux

4.1 Existing Implementations

Structural subsumption algorithms for Description Logics have been discussed since the 1980s, but the initial results were found to be incomplete for more expressive DLs. The first tableaux algorithm which successfully overcame these limitations was proposed by Schmidt-Schauß and Smolka [11, 1991]. A general overview of the field can be found in [8]. Various implementations have since then deployed tableaux reasoning, both in the commercial and non-commercial sector, such as Racer [12], Pellet[1] and FaCT [13]. These systems use Lisp, Java or C++ as their implementation language.

On the other hand, implementing tableaux-based reasoning in Prolog has been proposed, e.g. by Beckert and Posegga [14], without specifically applying it to Description Logics. An interesting approach is offered by SWI Prolog [15] from the University of Amsterdam that comes in its current version with an RDF and Semantic Web library. But the RDF implementation seems to focus on storing and querying of RDF triples, and the documentation of the SemWeb package states that there is only limited support for OWL reasoning, which is outside the standard distribution in the Triple20[2] ontology editor. So far, there is no hint to be found in the Triple20 documentation as to which inference algorithm it uses.

A single publication specifically addresses the implementation of a Description Logics variant, namely $\mathcal{ALCN}$, in Prolog, Adam Meissner's 2004 paper "An automated deduction system for description logic with ALCN language" [16]. He implemented Tableaux following the original definitions of the algorithm very closely and

[1] http://www.mindswap.org/2003/pellet/

[2] http://www.swi-prolog.org/packages/Triple20/

will be a reference for our work. Meissner also provided us with a complete and updated version of the code presented in his paper [17].

Stuart Aitken of the University of Edinburgh was kind enough to provide the author with an unpublished implementation of Tableaux in Prolog [18], which will also be used in the following discussions. To the best of our knowledge there has been no other work published that implements DL reasoning with Tableaux in Prolog.

4.2 Interpretation of Algorithm

Aitken [18] and Meissner [17] follow a traditional recursive approach and explore one particular tree (depth-first) before evaluating the alternative one, when hitting a disjunction. That allows them to be concerned with only one tree at any one time, relying on backtracking to unwind to the choicepoint and take the alternative route. Baader [10], in contrast, proposes a structure which keeps track of all the variant trees at the same time, leaving it to the specific implementation as to which of those trees are further unfolded.

This approach makes one aspect of the algorithm very clear: every single one of these trees provides a model if it is completely clash-free, i.e. the relation of the leaf nodes in a single tree is 'and' when it comes to evaluation. A proof tree might become arbitrarily deep, with multiple branches ending in a leaf node. Each leaf node either contains a clash, or cannot be expanded further. From each such leaf node a path leads back up to the root node. Each such path is considered a branch in the proof tree. A proof tree is considered "open" or clash-free, when there are only open branches. That means that the predicate "*open(Tree)*" for a saturated tree is given by "*open(Tree) = open(Branch1) and open(Branch2) and . . . and open(BranchN)*" for all N branches of the tree.

But looking at the set of the various trees that stem from resolving disjunctions in the node labels, it is enough that only one of them provides a model in order to achieve satisfiability for the initial DL formula, i.e. the relation among the various trees is 'or': "*satisfiable(Formula) = open(Tree1) or open(Tree2) or . . . or open(TreeK)*" for all K trees that were constructed during the proof.

This leaves room for implementation decisions. The recursive approach saves runtime memory, since it only concerns itself with a single tree at any one time. Only when the current tree has a clash is an alternative tree inspected. In terms of runtime, this is fine if a model is found early on in a "left-most", depth-first search of the search

space.

Maintaining the set of alternative trees at the same time allows for the application of heuristics. Which tree might be the biggest (fully expanded)? Since this will consume the most processing resources and a smaller tree might give you the necessary model faster, you might want to leave bigger trees for last. Which tree is the likeliest to provide a model? Since you can neglect all other trees once you have found an open tree, you might want to explore this first.

The most important thing to note for the algorithm here is that the tree being evolved is 'and'ed, but alternative trees are 'or'd. All branches of a single proof tree, resulting from the application of rules in the logic, have to be clash-free. The search space of the algorithm, consisting of possibly multiple trees, needs just one clash-free tree in order to succeed.

4.3 Ordered Application of Rules vs. Random Application

The original definition of the Tableaux algorithm (cf. [5, 4]) just lists the completion rules and makes no assumption about the order of their application. This has the nice theoretical property of keeping the algorithm as simple and as free of unessential preconditions as possible. But in a concrete implementation, ordering of the rule applications provides significant advantages. Baader and Sattler show that it reduces the worst-case space requirements of the algorithm to polynomial (PSPACE-complete) [8, pp.13f]. They summarise it by stating about the modified algorithm that

"... it starts with $C_0(x_0)$ and

1. applies the $\rightarrow \sqcap$- and $\rightarrow \sqcup$-rules as long as possible and checks for clashes;
2. generates all the necessary direct successors of x_0 using the $\rightarrow \exists$-rule and exhaustively applies the $\rightarrow \forall$-rule to the corresponding role assertions;
3. successively handles the successors in the same way." [8, p.14]

This section discusses this approach.

4.3.1 AND and OR Connectives

An exhaustive application of the $\sqcap$-rule and $\sqcup$-rule ensures that the maximum number of elements are available for later $\exists$- and $\forall$-expansions. Since existential and universal elimination only operate on new child nodes, that means that eliminating all possible 'and' and 'or' connectives first in a single node will allow you to concentrate on the subsequent quantifier elimination of that node. No surprises can happen in the form of new elements becoming available later in that node that might be significant for later proof steps (e.g. if a forall elimination is blocked due to a suitable child node and arc missing, which would become available later from an 'and' elimination that sets free the required existential).

As an example, the node

$$\{(\exists R.A \sqcap B) \sqcap \forall R.C\}$$

can be completed by first applying the and-rule (which is the only applicable rule at this stage), which gives you

$$\{(\exists R.A \sqcap B),\ \forall R.C\}$$

The $forall$-rule is blocked at this stage, although there is a suitable existential 'hidden' in the remaining *and*-term. Under the classical algorithm, the $forall$ just has to "wait", until eventually the *and*-rule is applied, and the existential is freed:

$$\{\exists R.A,\ B,\ \forall R.C\}$$

Now the existential can be expanded, and then finally the $forall$-term, giving you a new child node with two constraints

$$\{\exists R.A,\ B,\ \forall R.C\},\ \{A,\ C\}$$

This means it is advisable to expand all possible *and*- and *or*-terms in a node, so that all nested terms are available when proceeding to quantifier elimination.

4.3.2 Existentials

In the next stage of processing the elements of a node, an exhaustive application of the $\exists$-rule ensures that the maximum number of edges are available for subsequent $\forall$-expansions. In the above example, the forall-term remains blocked until the existential is expanded. Only then are the preconditions for the $forall$-rule met. It therefore makes sense to expand all existential terms in a node exhaustively, before proceeding to the $forall$-terms.

4.3.3 Universals

Since each ∀-expansion depends on the availability of a suitable edge, this rule should be applied last. If all existentials have been expanded, you can decide for each forall-term whether it is expandable or not. There is nothing that can happen in the further course of the proof that may invalidate this decision, specifically blocked universals of this node remain blocked.

This order of *and*/*or*-expansion, existential expansion, and then universal expansion allows for each possible element to be expanded in a single step, minimising on rule application attempts, without damaging the correctness of the proof. I will take advantage of this fact in the following sections.

4.4 Replacing Expressions by their Derivatives vs. Just Adding the Derivatives

Standard Tableaux just **adds** new elements to the proof tree, when applying its completion rules (cf. Tab 3.1 p. 14). These are either new elements for existing nodes (*and*-, *or*- and $forall$-rule), or new nodes that are children of existing ones (*exist*-rule). All previous elements are retained, at the node level as well as the tree level.

Here is another simple example to illustrate this. An initial DL expression will form the initial single member of the set, say

$$\{A \sqcap (B \sqcap (C \sqcap D))\}$$

Applying the *and*-rule gives you two new members of the set, the operands of the intersection, *in addition* to the initial member. That is, after applying the *and*-rule the set looks like

$$\{A,\ B \sqcap (C \sqcap D),\ A \sqcap (B \sqcap (C \sqcap D))\}$$

The *and*-rule can again be applied to the second member (Note that it **cannot** be applied again to the initial (now third) expression, due to the rule's precondition that checks for the presence of the operands). This again gives us the new members in addition to the old ones, namely the set

$$\{A,\ B,\ C \sqcap D,\ B \sqcap (C \sqcap D),\ A \sqcap (B \sqcap (C \sqcap D))\}$$

Continuously applying the *and*-rule to the new member of the set (no application on the same expression twice) eventually gives you the following set:

$$\{A,\ B,\ C,\ D,\ C \sqcap D,\ B \sqcap (C \sqcap D),\ A \sqcap (B \sqcap (C \sqcap D))\}$$

The proof tree is saturated, you cannot apply any more rules. As you can see, the single proof node contains the fully expanded members, A, B, C, and D, as you would expect. But it also contains the initial expression, as well as all intermediate expressions created during the completion process.

But the initial expression and all intermediate expressions (all the "parent" expressions so to speak) *do not contribute any further to the decision procedure*.

> If we apply an ordered application of rules, as described above, there is no need to retain a DL expression once we have expanded it; in fact as soon as we have evaluated it for expansion, whether the expansion was possible or not, it can be discarded.

It is therefore safe to discard them, and replace them by their derivatives. Moreover, if you remove evaluated terms from the proof tree, this prevents you from checking these expressions over and over again, making the second precondition for each derivation rule (cf. Tab. 3.1, p.14) essentially redundant. The second precondition of the $\sqcap$-, $\sqcup$- and $\exists$-rule and the second part of the second precondition of the $\forall$-rule all test for the presence of facts that their corresponding action will produce once it is executed. The *and*- and *or*-rules also test for the presence of their operands (to various degrees of completeness), the *exist*-rule condition checks for the presence of edges and child nodes, and the $forall$-rule checks for the presence of a concept in the child nodes. This all helps to avoid expanding the same compound term over and over again, which would invalidate the termination of the algorithm. "*The second condition in each rule constitutes a control strategy which ensures that the algorithm does not fail to terminate due to an infinite repetition of the same expansion.*" (Horrocks [5, p.48] - but see also section 4.8). Pruning the parent expression after its expansion simply avoids this problem altogether.

For the inference rules, this means rather than just adding new expressions derived from more complex ones, you replace the complex expression by its derivative(s). In particular, before adding the derivative of a rule application to some node label (using the set union operator "$\cup$") in the rule actions, the parent term is removed from the node label (expressed by the set minus operator "$\setminus$"). The modified consequent for the *and*-rule would look like $\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus \{C \sqcap D\}) \cup \{C, D\}$. The *or*-rule would

change into $\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus \{C \sqcup D\}) \cup \{C\}$ for the one alternative and into $\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus \{C \sqcup D\}) \cup \{D\}$ for the other. The existential rule would be augmented with $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \setminus \{\exists R.C\}$ in the consequent, and the $forall$-rule with an expression like $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \setminus \{\forall R.C\}$. A complete list of the modified rules is given in Tab. 4.1.

Since these modified Tableaux rules delete old expressions in the proof tree, I call them *parsimonious rules*. Aitken presented this variant [3] and used it in his implementation[18], while Meissner [17] chose the classical definition of the rules, monotonously adding to the tree.

The differences between the two algorithms have some implications:

- The set of expressions to maintain with the classical rules is much larger than if you discard evaluated expressions during the proof process. Your proof tree stays significantly smaller, saving on the memory footprint of the procedure.

- The classical algorithm has to maintain guards to avoid expanding the same expression twice during the process, spending computing power on the checks. The same expression is probably inspected multiple times before it might eventually get expanded, and after that it might again be inspected multiple times, with no effect. This shows a probably high degree of term inspections without effect, again drawing on computing time.

- The evaluated expressions do not add to the result of the proof any longer:

  - *Clashing*. The parent term will not be relevant for clashes, since due to the Negative Normal Form (NNF) of the initial expression, only atomic concepts are negated, never compound terms. That is, there can never be a situation where a complex term C will be relevant for a clash test with another element $\neg C$ in the same node[3].

  - *Branches/New Nodes*. Only existential terms can create new nodes. Once an existential has been evaluated, the new node exists and the existential in the parent node is no longer needed. The same node cannot be created twice. *and*-, *or*- and $forall$-terms cannot create new nodes, and $forall$-terms can only add to existing child nodes. If it is asserted that all possible child nodes are present at the time $forall$-terms are evaluated, existentials cannot add further.

[3] But this also shows the dependency of the modified rules on NNF, which depends on the previous concept unfolding. As a consequence, optimisation techniques like *lazy unfolding* [19, p.9] cannot be applied to the modified rules.

  - *Model*. An interesting question is whether deleting parent terms from a node label can lead to an empty node label. As for *and* $-$ */or*-terms this cannot happen, since they either produce new elements in the node label or these elements were already present (at least partly), as ascertained by the precondition. Dropping an *and* $-$ */or*-parent term can therefore never render a node empty.
    Removing quantified expressions after evaluation from the node might render it empty, but this can only happen after successfully creating child nodes, which are further inspected for a model. The decision procedure therefore continues unhindered.

- Parsimonious rules do, however, impact the constructive character of the proof, where in case of a saturated and open tree the classical algorithm constructs a satisfying model. Obviously, missing expressions in node labels might change the constructed model. But since we are only interested in a decision procedure, and never return the constructed model, this is no problem. The decision procedure remains intact.

4.5 The New Rules

Table 4.1 gives the new and modified set of inference rules, the *parsimonious rules*, that I am using in my implementation. To the best of my knowledge, these variants of the Tableaux rules have not been presented in academic publications before. The following sections discuss each of the new rules.

4.5.1 AND-Expansion

During the expansion of an *and*-expression, the operands get added to the node label. Since the semantics of the node label is that individuals of the represented concept have to satisfy all the contained concepts, i.e. every contained concepts acts as a constraint, all elements in the set are implicitly conjuncted. That is, the set $\{A,B\}$ is semantically equivalent to the set $\{A \sqcap B\}$. Moreover, in the set $\{A,B,A \sqcap B\}$ individuals must satisfy A and B (the first two members of the set). The third constraint, $A \sqcap B$ is re-stating this requirement, and is therefore redundant: it does not add a semantically new constraint.

It is also not possible to render the node “empty” by removing the complex *and*-term, since the child terms are necessarily present after the rule application.

$\sqcap - rule$	*if* 1.	$(C_1 \sqcap C_2) \in \mathcal{L}(x)$
	2.	$\{C_1, C\} \not\subseteq \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus (C_1 \sqcap C_2)) \cup \{C_1, C_2\}$
$\sqcup - rule$	*if* 1.	$(C_1 \sqcup C_2) \in \mathcal{L}(x)$
	2.	$\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
	then	a. save **T**
		b. try $\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus (C_1 \sqcup C_2)) \cup \{C_1\}$
		If that leads to a clash then restore **T** and
		c. try $\mathcal{L}(x) \longrightarrow (\mathcal{L}(x) \setminus (C_1 \sqcup C_2)) \cup \{C_2\}$
$\exists - rule$	*if* 1.	$\exists R.C \in \mathcal{L}(x)$
	2.	there is no y s.t. $\mathcal{L}(\langle x, y \rangle) = R$ and $C \in \mathcal{L}(y)$
	then	create a new node y and edge $\langle x, y \rangle$
		with $\mathcal{L}(y) = \{C\}$ and $\mathcal{L}(\langle x, y \rangle) = R$ and
		$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \setminus \exists R.C$
$\forall - rule$	*if* 1.	$\forall R.C \in \mathcal{L}(x)$
	2.	there is some y s.t. $\mathcal{L}(\langle x, y \rangle) = R$ and $C \notin \mathcal{L}(y)$
	then	$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$ for every applicable y and
		$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \setminus \forall R.C$

Table 4.1: Modified Tableaux Inference Rules for $\mathcal{ALC}$ (new parts are in blue)

4.5.2 OR-Expansion

The same basically holds true for *or*-expressions. The duplication of the whole tree with one ‘or’ operand in the node label in one tree, and the other operand in the node label of the other, preserves fully the semantics of the initial complex or-expression. There must either be a model fullfilling the the tree with the first operand, or a model for the tree with the second, for the initial expression to be satisfiable. The complex *or*-expression does not add a new constraint to the respective node label, and can therefore be discarded.

It is also not possible to render the node “empty”, since at least one of the operands is necessarily present after the rule application.

4.5.3 Exists-Expansion

Again, we face the same situation with the expansion of existentials. After (possibly) creating a new child node and a new edge to it, labelled with the name of the relation,

this fully represents the initial semantics of the existential. To keep the existential means to retain redundant information. Further expansion of the term is prohibited by the rule's guarding condition, so the term is effectively useless.

It is possible to render the node "empty" after a parsimonious rule application. However, this does not damage the proof, since the relevant information for the decision procedure are in the child nodes.

Therefore, the existential can be pruned from the node label.

4.5.4 Forall-Expansion

Forall-expressions ($\forall R.C$) can be evaluated multiple times, precisely once for each existing child node with matching relation edge and non-existent concept C. With a $forall$-term, all we have to make sure is that it is maximally applied, before we prune it from the node label. This pruning is then not damaging to the proof.

Pruning $forall$-expressions might render a node label empty, e.g. when the initial node label only consisted of *exist*- and $forall$-terms. However, this does not damage the proof, since the relevant information for the decision procedure is in the child nodes.

4.5.5 Emptying the Node by Pruning Parent Expressions

As stated above, it is possible to empty a node by applying parsimonious rules (in the existential and universal case), but this does not affect the proof result. The proof is continued with the the child nodes of the quantifier expansions. It is also not possible that the same term in its negated version could show up, causing a clash, since the expressions are in unfolded NNF, i.e. only basic concepts are negated.

4.5.6 Summary

This section introduced the notion of parsimonious Tableaux rules, where the rules not only add derivative terms to the proof tree, but also remove the parent terms. This is possible under the assumption of an ordered and exhaustive application of the inference rules to the DL expressions of a node label. The parsimonious rules also make some of the "guards" (preconditions) for the rules superfluous, which only protect against the repeated application of a rule to the same expression. In each proof step, only redundant information is removed from the node label, which is not relevant for

the remainder of the proof. Therefore, soundness and completeness properties of the classical algorithm are retained by parsimonious rules.

Termination is retained, since the set-minus operation ("\") used in the modified rules reduces the number of DL expression in a set. If the algorithm terminates (i.e. no more rules are applicable) with the larger number, it will also terminate with the reduced number of elements. Looping evaluation of the same term is prohibited, since every evaluated term is removed.

Even if the parsimonious rules changed the semantics of nodes, which represent individuals constructed by the proof, even if you interfered with the constructive character of the proof, it will still – if successful – construct a model that satisfies the initial formula, and you still have a decision procedure.

4.6 Fringe vs. Whole Tree

Once you are sure that a node has been exhaustively transformed and expanded, checked for obvious concept contradictions, and all possible child nodes have been derived with their labels fully expanded, you can simply neglect it. It does not contribute anymore to the result of the decision procedure. Analogous to complex DL expressions within a node to which a derivation rule has been applied, the whole node now can be discarded.

Therefore, for any given proof tree, it suffices only to keep track of the current set of leaf nodes, or *fringe* [20, p.70], in contrast to classical definitions of the proof expansion rules, which always maintain the whole tree.

Keeping only the fringe of the proof tree provides various advantages without incurring negative effects:

- Less memory is needed to keep the fringe rather than the whole tree. This is especially interesting for very large concept expressions.
- Non-leaf nodes in the tree do not add to the decision; therefore there is no need to keep track of them.
- Looking only at fringe nodes (and therefore potential subtrees of the proof tree) allows for heuristics on which branch a fast result (i.e. a clash) may be achieved. (A clash closes the branch and therefore the whole tree, which then is known not to provide a model for the proof goal).

This depends on the ordered execution of rules, as described above: restricting yourself to maintaining only the fringe is only possible if you can make sure that a

parent node has been **fully** expanded, and cannot be expanded any further – both in terms of child expansion and clash detection. Only then can you leave it behind and concern yourself only with the child nodes.

4.7 Handling OR Trees

During the expansion of the proof tree, every time you expand a $\sqcup$ connective, you have a non-deterministic choice: it is good enough if a tree with either of the two operands gives you a model, in order for the proof to succeed, therefore, you can choose any of them and try the other on failure.

This leaves room for implementation variants. In their algorithm description, Baader and Sattler (e.g. [10, p.6]) maintain a set of alternative trees that result from union elimination during the proof. Each time you encounter a union in a node, the whole proof tree will be replaced by two variants of itself, which are identical to the original tree with one union operand added to the first variant, and the second operand added to the second.

An alternative way, which is closer to Horrocks' Tableaux rule [5, p.48] and was chosen for the presented implementation, and also by Aitken [18] and Meissner [17], is to create a choice point, explore first the tree variant with the first union operand, and on failure return to this choice point and try the variant with the second union operand. This technique explores the different tree variants sequentially, in the order the union operands have been found, until one of the variant trees provides a model. Apart from saving runtime memory because you only have to keep a single tree in memory at any one time, this solution is very attractive for Prolog because it naturally fits into Prolog's backtracking strategy.

However, keeping a set of alternative trees over the proof's runtime also has its beauty. Beside the fact that it is more demanding to keep all these trees in memory simultaneously, considering that they can become quite large while having possibly a lot of overlap, this technique gives you a distinct advantage: since you can oversee all the trees you have encountered so far, it gives you maximum freedom in choosing which to pursue further in the proof and allows you to consider heuristics concerning which of the trees might lead to an answer (i.e. model) fastest, and pursue this one first.[4]

[4]The question of possible measures for such heuristics is not explored further here, but the least number of language tokens in leaf nodes and the least number of existentials come to mind. The aim

Yet also in the sequential version, you can apply heuristics at any choice point to choose the most promising variant first. It would suffice to measure the two union operands, since the resulting alternative trees are identical apart from these two expressions. But the disadvantage is that you can always only decide between the current possible two variant trees. It would be harder in this approach to take other choice points further up the proof tree into consideration as well.

4.8 The Second Condition of the OR Rule

An interesting question concerning alternative trees is posed by the second condition that guards application of the *or*-rule. This second condition effectively blocks *or*-expansion if only one of the operands is already available in the node (cf. Tab. 3.1). For example, in a tree with a single node like

$$\{A, A \sqcup B\}$$

the *or*-expression cannot be expanded, since A is already available. Horrocks' [5, p.48] basic rationale for this condition is to avoid multiple evaluations of the same complex term. If you allowed the expansion, it would lead to two alternative trees, namely

$$\{A\}, \{A, B\}$$

Blocking the *or*-expansion does not harm the first tree, since it is just the old tree without the *or*-expression (which is what you get with parsimonious rules as well). But obviously the whole second tree containing B is suppressed, which might be an arbitrary deep and complex DL expression. This raises the question of whether this second tree could contain a model, and with the first tree failing (through a clash), whether this would contribute to a successful proof, which you would otherwise miss.

But on closer inspection, this appears not to be possible (which increases the confidence in the rule condition). If the first and simpler tree clashes, there is no way the second can provide a model, since it contains all the constraints from the first tree (A) and just *adds* more constraints (namely B). If a simpler concept is not satisfiable, a more complex one never can be. The same holds true the other way round, when

---

is clear: you want to pick the tree with the least branching and the least depth before it is saturated. Branching and depth both depend on existential expansion, so the tree with the smallest fringe (number of leaf nodes) and the fewest number of existentials in its fringe would be the most promising for a fast saturation.

B was already present. You always end up with a subsumption relation between the alternative nodes, where the smaller set of constraints is enough to decide satisfiability.

For the proof that means that we can safely ignore alternative trees if one of the *or*-operands is already present. It will not damage the decision procedure, and has the nice computational advantage of saving us from exploring a potentially big alternative tree.

4.9 Summary

We have discussed various aspects of the Tableaux algorithm with regard to possible implementations, which led to design options that we will use in the proposed implementation, namely

- *rule ordering* and *exhaustive application of rules* on nodes,
- *replacing terms by their rule derivatives*, which led to the re-formulation of the classical rules as *parsimonious rules*, and
- keeping only the *fringe* of the proof tree during the proof procedure.

All these aspects show promise with regard to memory consumption and runtime of the proof.

Chapter 5

Specification of the Implementation

This chapter presents a specification of *tableaux.pl*, the proposed Prolog implementation of the Tableaux algorithm. A complete listing of *tableaux.pl* is given in Appendix A. In the following sections *A, B, C, ...* denote well-formed DL concept expressions. Expressions on the left of the left-pointing arrow ($\leftarrow$) are goals you want to satisfy, expressions on the right of it represent subgoals that, if satisfied, allow you to derive the main goal. Throughout this chapter, two-place predicates (like *expand_defs/2*) can be read as taking the first argument as an input parameter and constructing the second argument as an output parameter.

5.1 Goal Construction

The first step of the algorithm is the construction of the goal for the proof derived from the initial query. The top-level predicate *query/1* succeeds if the query is satisfiable, and fails if it is unsatisfiable.

$$
\begin{array}{ll}
query(A \equiv B) & \leftarrow \neg tableaux(A \sqcap \neg B) \wedge \neg tableaux(B \sqcap \neg A) \\
query(A \sqsubseteq B) & \leftarrow \neg tableaux(A \sqcap \neg B) \\
query(A \sqcap B \sqsubseteq \bot) & \leftarrow \neg tableaux(A \sqcap B) \\
query(unsatisfiable(A)) & \leftarrow \neg tableaux(A) \\
query(A) & \leftarrow tableaux(A)
\end{array}
$$

The first line says, if your question is whether A is equivalent to B, this is true if the tableaux proof can show that $A \sqcap \neg B$ is not satisfiable and $B \sqcap \neg A$ is not satisfiable. Likewise, goals for the prover are constructed and combined for queries for subsumption, disjointness, unsatisfiability and satisfiability. The last clause head matches all of the other clause heads, but always fails if invoked with one of the other arguments.

5.2 The Basic Tableaux Proof Predicate

The top-level proof goal, *tableaux/1*, is defined as follows:

$$\begin{aligned} tableaux(A) \leftarrow \quad & expand\_defs(A, A1) \wedge \\ & nnf(A1, A2) \wedge \\ & expand\_tree(A2) \end{aligned}$$

First, all complex concepts are replaced recursively by their definition, to obtain an expression that only contains atomic concept names (*expand_defs/2*). From this expression, the Negative Normal Form is derived, where only atomic concepts are negated (*nnf/2*). This formula is then fed into the actual proof algorithm that treats it as the root node of a proof tree. The initial node is subsequently transformed and expanded by well-defined proof steps (*expand_tree/2*). During the tree expansion, its leafs are inspected for possible clashes and the goal fails if one is found. If the tree is fully expanded without a clash, the goal succeeds.

5.3 Concept Unfolding

Complex DL concepts in the goal, i.e. named concepts that are defined in terms of other concepts, have to be replaced by their definitions to reach an expression that only contains atomic concepts. Formally,

$$\begin{aligned} expand\_defs(A, A1) \quad & \leftarrow atomic(A) \wedge ont(equiv(A, A2)) \wedge expand\_defs(A2, A1) \\ expand\_def(A, A) \quad & \leftarrow atomic(A) \wedge \neg ont(equiv(A, \_)) \end{aligned}$$

This says that every atomic concept name A, for which a definition is given in the ontology (through some $A \equiv f(X_i)$ equivalence expression, where $f(X_i)$ is a well-formed DL formula over some concepts X_i), is replaced by this definition. Concept names which have no defining expression in the ontology are returned as primitive. This rule has to be applied recursively to obtain the desired result.

5.4 Negative Normal Form

These are the transformation rules to convert an arbitrary DL formula into its Negative Normal Form:

$$
\begin{array}{ll}
nnf(\neg\neg C, C) & \\
nnf(\neg\forall R.C, \exists R.C1) & \leftarrow nnf(\neg C, C1) \\
nnf(\forall R.C, \forall R.C1) & \leftarrow nnf(C, C1) \\
nnf(\neg\exists R.C, \forall R.C1) & \leftarrow nnf(\neg C, C1) \\
nnf(\exists R.C, \exists R.C1) & \leftarrow nnf(C, C1) \\
nnf(\neg(A \sqcap B), A1 \sqcup B1) & \leftarrow nnf(\neg A, A1) \wedge nnf(\neg B, B1) \\
nnf(A \sqcap B, A1 \sqcap B1) & \leftarrow nnf(A, A1) \wedge nnf(B, B1) \\
nnf(\neg(A \sqcup B), A1 \sqcap B1) & \leftarrow nnf(\neg A, A1) \wedge nnf(\neg B, B1) \\
nnf(A \sqcup B, A1 \sqcup B1) & \leftarrow nnf(A, A1) \wedge nnf(B, B1) \\
nnf(\neg C, \neg C) & \leftarrow atomic(C) \\
nnf(C, C) & \leftarrow atomic(C)
\end{array}
$$

Again, the first argument to *nnf* represents some initial DL formula and the second argument its processed form, given that the subgoals on the right-hand side succeed. The rules have to be applied recursively until only atomic concepts are negated.

5.5 Expanding the Proof Tree

The actual proof starts with the goal derived by the transformations applied so far. This will be some kind of DL expression, with all concept names replaced by their most basic definitions, and only atomic concepts being negated. The basic control mechanism is a standard agenda-style tree expansion, where leaf nodes are examined, transformed or expanded, and the resulting leaf node(s) are put back into the list of the current fringe nodes (or agendas), for further examination.

I give here the interesting predicates that examine and transform/expand a single node. The basic predicate, *process_node/2*, transforms the $\sqcap$ and $\sqcup$ connectives using *transform_connectives/2*. The result is then examined for possible $\exists$ and $\forall$ quantifier expansions, which might result in new child-nodes being created (*expand_node/2*). Always only leaf nodes are kept in the fringe.

$$
\begin{array}{ll}
process\_node(A, B) \leftarrow & transform\_connectives(A, A1) \wedge \\
& expand\_node(A1, B)
\end{array}
$$

The resulting list of new and/or transformed nodes is put back to the agenda list, and the process starts over.

transform_connectives/2 is the first operation on each examined node. Intersection is resolved into the list of the two operands, union by replacing the original expression

with the first or the second operand. The new list of expressions replaces the original for further processing.

$$\begin{aligned} transform\_connectives([A \sqcap B|R], R1) \leftarrow\ & transform\_connectives([A, B|R], R1) \\ transform\_connectives([A \sqcup B|R], R1) \leftarrow\ & transform\_connectives([A|R], R1) \vee \\ & transform\_connectives([B|R], R1) \end{aligned}$$

expand_node/2 checks for clashes in the current node, i.e. obvious contradictions of a concept and its complement. But its core is the expansion of existential and universal quantifiers, achieved by *expand_exist/3* and *expand_univ/3*. Both take an extra argument as they have to keep track temporarily of child nodes.

$$\begin{aligned} expand\_node(N, \_) \quad &\leftarrow\ check\_clash(N) \\ expand\_node(N, NL) \quad &\leftarrow\ expand\_exist(N, [], E) \wedge \\ & \quad\ expand\_univ(N, E, NL) \end{aligned}$$

The inital list of child nodes to *expand_exist* is empty, and the resulting list is passed on in E. In addition to their node label, these nodes get attributed with a relation (which can be viewed as the named edge in the proof tree) and a unique identifier, to distinguish different child nodes that are connected to the parent through the same relation.

So input parameters are the current node as a list of DL expressions, and the current list of child nodes *E*. *expand_exist/3* returns the current list of child nodes, while *expand_univ* strips the attributes off at the end to return a list of plain leaf nodes.

$$\begin{aligned} expand\_exist([\exists R.C|L], E, EE) \quad &\leftarrow\ \neg member((R, \_, \_), E) \wedge id(I) \wedge \\ & \quad\ expand\_exist(L, [(R, I, [C])|E], EE) \\ expand\_exist([\exists R.C|L], E, EE) \quad &\leftarrow\ member((R, \_, L1), E) \wedge \neg member(C, L1) \wedge \\ & \quad\ id(I) \wedge expand\_exist(L, [(R, I, [C])|E], EE) \\ expand\_exist([], E, E) & \end{aligned}$$

In the case of an existential expression, the expansion is continued with a new node added to the list of child nodes, attributed with the name of the relation, provided such an attributed node with such a concept does not already exist.

If there is no existential expression at all, the current node constitutes a final leaf node and, since it is already checked for clashes, ascertains an open branch in the proof tree. Otherwise, the resulting list of attributed nodes is used as an input for *expand_univ/3*, together with the current node, to expand potential universal restrictions.

$$
\begin{aligned}
&expand\_univ([\forall R.C|L],E,N) \leftarrow expand\_univ\_exhaust((R,C),E,EE) \wedge \\
&\qquad expand\_univ(L,EE,N) \\
&expand\_univ([],E,N) \leftarrow extract\_nodes\_from\_edges(E,N) \\
&expand\_univ\_exhaust((R,C),E,EE) \leftarrow \\
&\qquad extract((R,I,L1),E,E1) \wedge \neg member(C,L1) \wedge \\
&\qquad expand\_univ\_exhaust((R,C),[(R,I,[C|L1])|E1],EE) \\
&expand\_univ\_exhaust(\_,E,E)
\end{aligned}
$$

In the case of a universal expression in the current node, a node with the same label R has to exist in the list of attributed nodes. Only then the universal can be transformed by way of its concept C being added to the corresponding child node. *expand_univ/3* does this through a helper predicate, *expand_univ_exhaust/3*, which applies the given universal **exhaustively** to the list of child nodes. It is therefore essential that the $\exists$ expansion takes place before the $\forall$ expansion, since only then all possible child nodes will be available to try the $\forall$ expansion on, and only after that the $\forall$ term may be discarded for the rest of the proof. *expand_univ_exhaust/3* uses *extract/3*, which is like member, but on success also returns the list without the found member. If the concept C is not already present, it is added to this child node, and the process recurs with the updated child list until no more suitable members in E can be found.

5.6 Testing the Fringe

These expansion rules are applied to the nodes in the fringe until a clash is found, in which case the proof of the current tree fails. If available, alternative trees will be tried via backtracking to the choice points created in the resolution of *or*-terms (in *transform_connectives/2*).

If no clash is found, the proof stops when no more rule application is possible and the tableaux is saturated. In this case, the tree is open and the proof succeeds.

5.7 Example

To illustrate the application of the various proof steps here is an example[1]. Consider the following ontology:

[1] From the KMM-A lecture. In order to distinguish from the rules, I use lower-case letters here to name relations and concepts, since these are concrete instances, not variables.

$$c \equiv \quad \forall r.\neg a \ \sqcap \ \forall r.(\exists s.\neg b)$$
$$d \equiv \quad \exists r.((\forall s.b) \sqcup a)$$

and the following query:

$$query(c \sqcap d \sqsubseteq \bot) \quad \% \text{ (Are c and d disjoint?)}$$

The proof goal then is to show that $c \sqcap d$ is unsatisfiable. The proof starts with this expression in the single node of the initial tree (nodes delimited by [], trees by {}). Here is how the proof evolves:

$\{[(\forall r.\neg a \sqcap \forall r.(\exists s.\neg b)) \sqcap \exists r.((\forall s.b) \sqcup a)]\}$	Concept unfolding	(1)
$\{[(\forall r.\neg a, \forall r.(\exists s.\neg b)), \exists r.((\forall s.b) \sqcup a)]\}$	Elimination of connectives	(2)
$\{[\forall r.\neg a, \forall r.(\exists s.\neg b), \exists r.((\forall s.b) \sqcup a)], [(r, 1, [((\forall s.b) \sqcup a)])]\}$	Existential expansion, creating a temporary list of attributed nodes	(3)
$\{[\forall r.\neg a, \forall r.(\exists s.\neg b), \exists r.((\forall s.b) \sqcup a)], [(r, 1, [\neg a, \exists s.\neg b, ((\forall s.b) \sqcup a)])]\}$	Universal expansion	(4)
$\{[\neg a, \exists s.\neg b, ((\forall s.b) \sqcup a)]\}$	Continuing with the new node, attributes stripped	(5)
$\{[\neg a, \exists s.\neg b, \forall s.b]\}, \{[\neg a, \exists s.\neg b, a]\}$	Union elimination, creating two alternative trees, with a single node in each, and with an immediate clash in the second	(6)
$\{[\neg b, b]\}, \{[clash]\}$	Existential and universal expansion in the first tree creates a new node, again with a clash	(7)
$\{[clash]\}, \{[clash]\}$	Result: both trees are closed	(8)

Here is more explanation for the proof steps:

- (1) The two concepts in the initial goal, *c* and *d*, are simply replaced by their definitions.
- (2) The connectives are eliminated; here, this only applies to intersection.
- (3) From the resulting three elements in the list, the existential can be expanded. This creates a new temporary list of (child-) nodes. In this case, there is only one existential to expand, so the list consists only of one child node, attributed with the name of the relation *r* and an identifier.

- (4) Now the universals can be expanded, since they scope over the same relation, *r*. Their concepts are added to the new attributed node *(r,1,[...])*.
- (5) Once we did all possible quantifier expansion, the list of child nodes has settled, and we continue the proof with these new leaf nodes. The attributes are no longer needed (they serve only to constrain the expansion of the universals). The parent node has been fully exploited and can be discarded.
- The third element contains a union, which can now be eliminated.
- (6) This results in two alternative trees with one node, each containing one of the disjuncts together with the rest of the initial node contents.
- The second of these lists shows an immediate clash ($\neg a$ and a), so this tree is closed. The node in the first tree can be further expanded.
- (7) As in steps (3) and (4), the existential $\exists s.\neg b$ creates a new child node, $(s, 2, [\neg b])$. In consequence, the universal $\forall s.b$ can be expanded, adding its concept b to the new node. Stripping the attributes and discarding the parent, we end up with the new node $[\neg b, b]$ (alongside the already closed second tree).
- (8) The left tree yields another clash, so both closed trees are shown. The tableaux is saturated and all trees are closed, so there is no model for the proof goal.

That means the answer to the initial query (disjointness of c and d) is “yes”. In the case of quantifications (over relations *r* and *s*), expansions of the universals in the corresponding steps were possible since existentials over the same relation were available.

Chapter 6

Benchmarking the System

Implementing an algorithm in a specific way always demands on evaluation of its runtime behaviour. For a Description Logics reasoner this means selecting suitable test ontologies and queries, and comparing the results against the correct solutions and other available systems. Choosing ontologies and queries provides some challenges of its own. The various language levels for DLs are reflected in the various ontologies, and there is not much sense in running an $\mathcal{ALC}$ reasoner against an ontology that uses e.g. $\mathcal{SHOIN}$ to express its concepts; the reasoner would simply not be able to cope with the language constructs that are unknown to it. Once you have settled for suitable ontologies and queries, it is important to know the correct outcome of the queries. Ontologies and queries quickly become complex, so that it becomes increasingly hard to convince yourself about their outcome "with pen and paper". The options that remain are either constructing test data yourself, e.g. by using an ontology editor like Protégé, and then run the test data through a reference system that you trust; or re-use existing examples and test data which have documented outcomes. This work follows the second approach[1].

[1] One advantage of this is also that you side-step a pitfall that is inherent in any software development. If you design the test cases for the software you have written yourself, chances are that you are duplicating errors and blind spots in both your implementation and your data. Independent tests avoid that. It also requires a fair amount of insight and experience to construct tests that are not trivial, and test across different parts of the proof process.

6.1 The Test Data

6.1.1 T98-sat

The T98-sat Description Logics benchmark test is part of a larger suite of benchmark tests that was issued in 1998, in preparation for the '98 International Workshop on Description Logics (DL'98) [2]. It contains a set of 2 times 9 files. Each file contains more or less complex, TBox-free $\mathcal{ALC}$ concept expressions. For each of the nine pairs, one file contains only satisfiable queries, the other only unsatisfiable ones. The task is to check the satisfiability of the expressions. Horrocks and Patel-Schneider write in their summary of the test: "*The test consists of 9 classes of concept (e.g.* `k_branch`*), in both coherent and incoherent forms. For each class of concept, 21 examples of supposedly exponentially increasing difficulty are automatically generated from a basic pattern which incorporates features intended to make the concept's coherence hard to compute.*" [21, p.2] As stated, within each file there are 21 queries, where the complexity of each expression (i.e. its sheer length and depth of nesting) increases. For example, the number of language tokens (i.e. number of concept names, role names and constructors) of the first query in `k_branch_n.alc` is 133, the last query in the same file contains 20353 tokens.

The organisation of the benchmark allows for both correctness and performance tests. But the original intent of the benchmark was to focus on performance. The idea was to run each query of a file and measure the time the reasoner needs to come back with the individual answers. The number of queries in the file that could be answered within a runtime below 100 seconds would be the "mark" of the reasoner and its benchmark result. E.g. the result for RacerPro 1.9 on a given machine could be 13, meaning RacerPro answered each of the first 13 queries in a time under 100 seconds.

For an overview, here is the complete list of files in the T98-sat test suite (*_n.alc contains the consistent queries, *_p.alc the inconsistent):

```
k_branch_n.alc
k_branch_p.alc
k_d4_n.alc
k_d4_p.alc
k_dum_n.alc
```

[2] http://dl.kr.org/dl98/comparison/data.html

```
k_dum_p.alc
k_grz_n.alc
k_grz_p.alc
k_lin_n.alc
k_lin_p.alc
k_path_n.alc
k_path_p.alc
k_ph_n.alc
k_ph_p.alc
k_poly_n.alc
k_poly_p.alc
k_t4p_n.alc
k_t4p_p.alc
```

The tests particularly considered in this work are the *k_dum_*.alc* files. The selection is partly due to the moderate demands they put on the reasoner, partly due to the various results that we obtained from running them, and partly due to time constraints for this work.

The T98-sat tests were adapted from a modal logic test suite developed by Balsiger, Heuerding and Schwendimann, a description of their work can be found in [22][3]. A complete description of the DL'98 test suite, together with pointers to relevant literature, is given in Horrocks and Patel-Schneider [21].

6.1.2 The Extended Mindswap Testsuite

The Maryland Information and Network Dynamics lab Semantic Web Agents Project, Mindswap[4], is a group within a special lab[5] of the Maryland Institute for Advanced Computer Studies[6] of the University of Maryland. The group focuses on Semantic Web research and has, among other things, developed the Description Logics reasoner Pellet. They also host the web page of the CMSC 498W "The Semantic Web" course[7] of the Computer Science department of the University of Maryland. As part of the

[3]This information was provided by Ian Horrocks in response to the author's question on the Description Logics mailing list, https://dl.kr.org/mailman/listinfo/dl

[4]http://www.mindswap.org

[5]The MIND Lab, http://www.mindlab.umd.edu/

[6]http://www.umiacs.umd.edu/

[7]http://www.mindswap.org/2004/cmsc498w/

course assignments the students had to develop an $\mathcal{ALC}$ reasoner in Python[8]. To provide for test cases the course maintainers collected a set of DL queries, together with necessary TBoxes. The examples come from various sources, from the course maintainers, the participants, or from relevant sources on the Internet. They are compact, they have known outcomes, and they stress various parts and potential pitfalls of the tested reasoner. Their total number is 29.

I have added another 15 test cases, some of which are based on the common "Family" example (used e.g. in the KMM lecture [3]). The others I have invented myself during my work; they are mostly very short and test specific features in the proof evaluation.

From this total of 44 test cases, 43 were used to validate and compare the various Prolog implementations. A complete list is given in table C.1 in Appendix C[9]. They have been translated from their original Python format to Prolog-style terms. All cases contain a prediction about the outcome of the query, *sat(F)*, where F is either the literal $true$ or $false$ depending on whether the query is satisfiable or not. A second predicate, *query/1*, holds the actual query, and then zero or more clauses of the *ont/1* predicate hold the corresponding TBox. (This is in contrast to the T98-sat tests, which are all TBox-free).

During the benchmark, each of the test files is *consulted* into the Prolog interpreter, the query is run, using the TBox if one is given, and the outcome of the proof is compared against the value of the *sat/1* predicate.

6.2 The Reference Systems

6.2.1 fact.pl

6.2.1.1 The Program

fact.pl is a Tableaux implementation in Prolog by Stuart Aitken [18]. The supported language level is $\mathcal{ALC}$. Besides implementing different design decisions, it uses a slightly different representation of concept constructors, e.g. *intersectionOf(A,B)* instead of *and(A,B)*.

In order to run the test examples, I had to apply a few changes that fixed bugs or

[8] http://www.mindswap.org/2004/cmsc498w/pa3.shtml

[9] One of the Mindswap tests, *ex3_2.pl*, has been omitted from the tests, since we could not obtain a measure for it from *fact.pl*, which would ran out of memory on it; *ex3_2.pl* is the last one in the list.

runtime behaviour. The larger code modifications concern mainly removing unneeded trace statements and test predicates, and slight corrections to the core predicates. I tried to be as unintrusive as possible. Despite these changes, *fact.pl* still causes a runaway recursion on one of the test cases (ex3_2.pl). For a complete listing of the *fact.pl* version used and a detailed list of the changes applied to the original code, refer to Appendix B.

6.2.1.2 Running the Tests

In order to run the test suite, the predicates from *fact.pl* were loaded into the Prolog interpreter. Since *fact.pl* does not provide concept expansion, *expand_defs/2* was imported from *tableaux.pl*. For any given test file, the file's content was loaded with *consult/1*, the goal was constructed from the *query/1* predicate obtained from the file, the goal was then expanded with any applicable TBox definitions using *expand_defs/2* and finally the expanded goal was translated into the DL representation used by *fact.pl*. With the translated expression, *fact.pl*'s top-level predicate *star/1* was called and its performance was measured.

To see the driver scripts that were used to glue all the various Prolog modules together, invoke the proof predicates and measure their execution, see Appendix F. The DCG rules to translate the ontology representation are given in Appendix E.

6.2.2 lpdl.pl

6.2.2.1 The Program

lpdl.pl [17] is also a Tableaux implementation in Prolog, and was developed by Adam Meissner of the Poznan University of Technology in Poland. It is largely based on his 2004 paper "An automated deduction system for description logic with ALCN language" [16]. The implementation tries to be close to the classical definition of the Tableaux algorithm (as given in Tab. 3.1) and emphasises algorithmic clarity over optimisation. The supported language level is $\mathcal{ALCN}$, which is $\mathcal{ALC}$ with unqualified cardinality constraints.

The program was originally developed in SICStus Prolog. In order to run it in SWI Prolog, a module prefix "*kb:*" on some of the predicates had to be removed. This does not change the program logic, just relaxes the (well considered) scoping of predicate names introduced into the name space. The file I/O and ontology managing

predicates of this implementation (which were affected by the module prefix) were not used anyway, since the prover was invoked only with already eliminated TBoxes.

6.2.2.2 Running the Tests

Analogous to *fact.pl*, the *lpdl.pl* predicates were loaded together with other predicates into the Prolog interpreter in order to prepare and run the tests. *lpdl.pl* does provide for concept expansion. But the translation to its own DL format made it necessary that, after loading the test case, construction of the goal and concept expansion (again by *expand_defs/2*) had to be done first. Then the language translation could take place, and *lpdl.pl*'s top-level proof goal *th/1* was called. This call was measured.

6.2.3 RacerPro

6.2.3.1 The Program

RacerPro[10] [12] is a commercial DL reasoner developed by Racer Systems[11]. The version deployed for the tests was 1.9. The system is available for Microsoft Windows, Linux and other Unix operating systems, such as Sun Solaris, both in 32- and 64-bit versions. It has been written in Lisp and optimised for speed and performance. The supported DL language level is $\mathcal{ALCQHI}_{\mathcal{R}^+}$ (aka $\mathcal{SHIQ}$). This is $\mathcal{ALC}$ enhanced with qualified cardinality constraints, role hierarchies, inverse roles, and transitive roles. Moreover, RacerPro supports different ontology formats (such as OWL), interfaces (such as DIG) and built-in functions (such as KRSS [23]).

6.2.3.2 Running the Tests

RacerPro supports various interfaces, but to be better comparable with the other programs the command line interface has been used for the test runs. The T98-sat tests are already in a format that RacerPro understands so that the general command for the program was simply

RacerPro -f [input-file]

Timing measures have been taken using the Unix command `time(1)`. From `time`'s three measurements, *real*, *user* and *sys*, the *user* value was chosen, since it represents

[10] http://www.racer-systems.com/products/racerpro/index.phtml
[11] Racer Systems GmbH & Co. KG, http://www.racer-systems.com/index.phtml

the time spent in the code of the program[12].

A similar scheme was devised in this case for *tableaux.pl*, so that the Prolog interpreter was started anew for every input file and timed, to have a comparable setup. The invocation looked like

pl -s reg_t98.pl -g 'get_files(*F*).' -t halt.[13]

where *F* was either the literal *satisfiable* or *unsatisfiable*, depending on the series to be run. The input data was converted from the Lisp-like format on the fly and put into a specific file before the Prolog invocation, where the driver predicate expected it (therefore no input file name appears on the command line).

6.3 Platform

All tests were run on a Intel Pentium 4 2.6GHz chip with 512MB RAM running on a Linux kernel 2.6.17.

Although the various Prolog programs would run with little or no change on the SICStus platform[14], SWI Prolog[15] was chosen for the tests, since it provides an excellent runtime environment, especially where the profiler is concerned. All profiling data has been gathered with SWI using its *time/1* predicate, and analyses were greatly supported by its *profile/1* predicate that collects runtime information for each predicate used during the proof.

6.4 Results

6.4.1 Running the Extended MindSwap Tests

The results of running the various Prolog implementations of Tableaux against the test cases of the extended Mindswap test suite are shown in Fig. 6.1. The x-axis simply uses the numbers from table C.1 to identify the test cases. Appendix C gives their details.

[12] *sys* denotes the time spent in kernel code, as triggered by calls to kernel functions from the program code. *real* gives the actual runtime of the program, which includes times in I/O wait state, and is therefore usually bigger than the sum of the other two.

[13] *pl* is the executable of SWI Prolog.

[14] http://www.sics.se/isl/sicstuswww/site/index.html

[15] http://www.swi-prolog.org

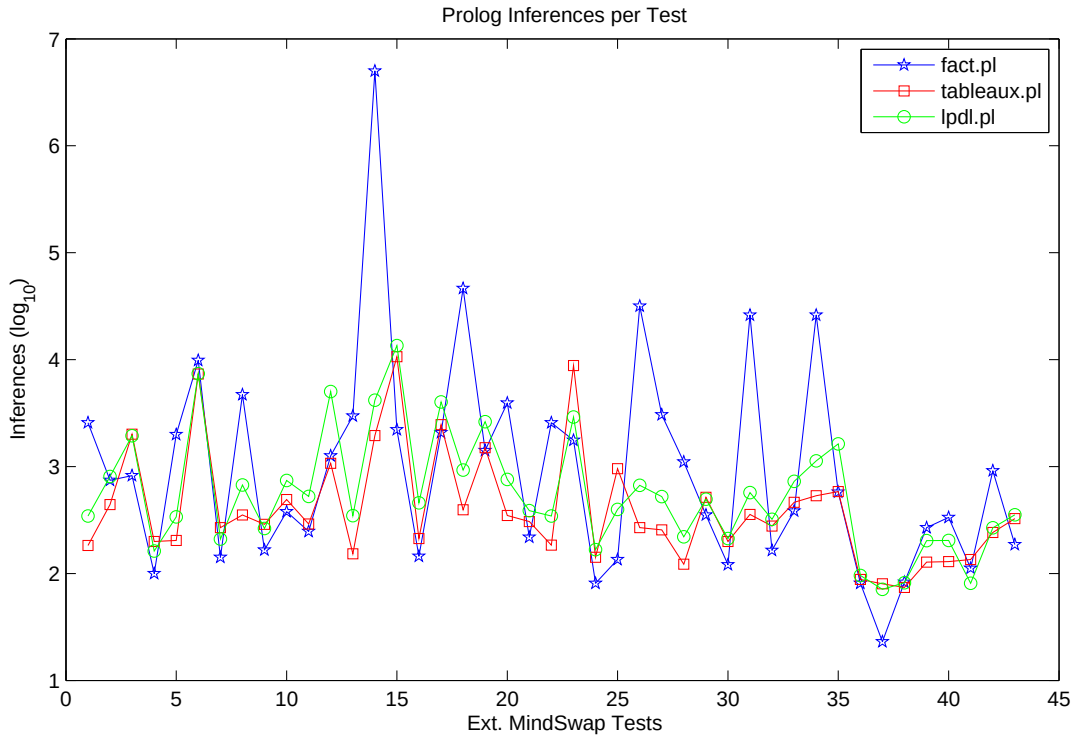

Figure 6.1: Comparison of Prolog Implementations (in number of inferences)

The evaluation metric chosen for the tests is the number of logical inferences made to come to a decision. Since the implementations performed in under 0.1 seconds on many of test examples, a comparison on the basis of CPU time would be of little expressiveness unless you used high-resolution timing. So rather than compare mere timing and CPU percentages used, the comparison is given in logical inferences made. This gives a better picture of how the various Prolog implementations fare. Due to a high variance, the values are put on an logarithmic scale, so the y-axis shows the log_{10} logarithm of the number of inferences made. The exact figures are given in Tab. 6.1.

fact.pl shows the greatest variance in results, both performing worst in many cases (e.g. 6, 14, 18, 26, 31, 34, 42), as well as coming out best in others (e.g.. 4, 7, 9, 16, 21, 24, 37). It holds both the record for best overall performance (37), as well as worst (14).

As an example, the exceptional high value of 5,007,280 inferences in test case 14 (*ex1_9.pl*) is mainly due to *fact.pl*'s use of the *member/2* built-in predicate (in which it spends around 67% of its runtime). Rather than using it mainly as a mere test applied to instantiated arguments, it relies heavily on the generative use of the predicate to iterate through lists, instantiating the first argument to a member and backtracking to

this choice point, if further analysis of the member renders it unsuitable for the main goal. *tableaux.pl* and *lpdl.pl* both use *member/2* more in its test role with instantiated arguments, *lpdl.pl* even resorting to *memberchk/2* in its place, to avoid choice points[16].

ex1_9.pl is one of the "Dragon" test cases the Mindswap course maintainers found on the Internet. Its TBox features various defining expressions for types of dragons, and the query asks "*whether something can be a Hydra and a Dragonet and a fire elemental*"[17]. The query is satisfiable and the unknown test author continues to state that "*it hits all of the basic constructs and rules, and should test useful features of the reasoner by having some or-branches that clash and some that don't, for example*" (ibid.).

The generative use of *member/2* in *fact.pl* allows for a beautiful declarative style in the *elim/3* predicate, stating the outcome of a clause application in the head (2nd/3rd argument) with initially completely unbound variables. These get bound successively in the body of each clause, where a first general call to *member/2* simply iterates through the nodes of the tree. A subsequent call to *member/2* then tries to pick a suitable expression from the selected node member list. In terms of the $\mathcal{ALC}$ rules, this call represents the first precondition of a rule. All these calls allow for backtracking.

Probably further member tests check the second rule condition. Only then the rule action is applied, by changing and re-constructing appropriate lists, eventually binding all variables in the clause head. Presumably, this use of *member/2* causes the high number of inferences in some of the other test cases too.

Test case 37 (*x_ex_ai.pl*), where *fact.pl* shows the overall best performance, is a very simple, unsatisfiable, TBox-free query that contains a single clash: $a \sqcap \neg a$. It is probably the simplest possible unsatisfiable query you can think of. *fact.pl* benefits mainly from the fact that it does not even try a concept expansion. The ordering of its *elim/3* clauses, early success of the *member/2* invocations and an early check for a clash add to the good performance. *tableaux.pl* spends alone around 25 inferences in the (unnecessary) attempt to expand the concepts in the query with ontology definitions. More inferences are then spent in housekeeping functions like maintaining a fringe, which all adds up to the nearly four-fold greater number of inferences (80).

Looking at the shape of their curves, *tableaux.pl* and *lpdl.pl* perform in remarkable unison, with *tableaux.pl* being more efficient by a factor of roughly two in many of the test cases (e.g. 1, 2, 8, 14, 16, 20, 27, 34), a few cases where the two perform very

[16] This would be a good optimisation option for *tableaux.pl* as well.

[17] From the *ex1_9.py* source code

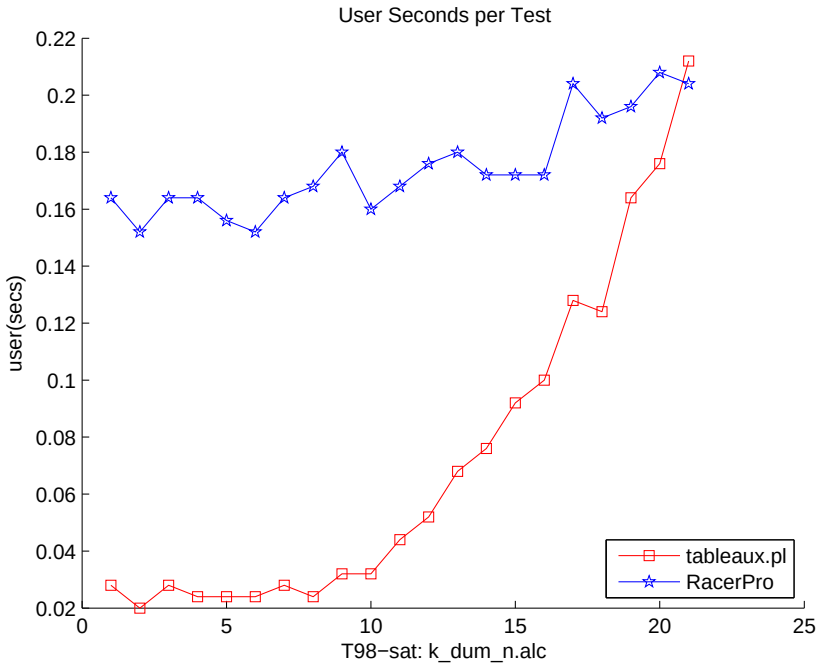

Figure 6.2: Comparing RacerPro and tableaux.pl on k_dum_n.alc

closely (e.g. 3, 6, 9, 29, 30), and some where *lpdl.pl* performs the better (e.g. 4, 7, 23, 25). Test case 23 (*ex3_1.pl*) deserves special attention, because the difference between the two is especially high, to the favour of *lpdl.pl* (*tableaux.pl*: 8,798; *lpdl.pl*: 2,912 inferences). It turns out that this is partly due to a profuse (and entirely unnecessary) use of Prolog's unification operator '=' in the fourth clause of the *transform_connect/3* predicate of *tableaux.pl*[18].

6.4.2 Running T98-sat:k_dum_n.alc

The results of running `k_dum_n.alc` with RacerPro and *tableaux.pl* are shown in Fig. 6.2. In this case, *tableaux.pl* compares fairly well against the reference system. But this is far from representative, and other tests from the `T98-sat` suite easily put *tableaux.pl* outside of measuring limits (many hours). As an example, Fig. 6.3 shows the performance of the two systems against `k_dum_p.alc` (Note that the scale of the values is logarithmic again). RacerPro is again nearly constant in its performance over the 21 test queries, as in the n series. *tableaux.pl*, while doing comparably well in the very first test, quickly soars to the times of about 15 to 20 minutes within the first six cases.

[18]By omitting '=' and constructing the result list directly in the head of the clause the number of inferences drops down to 5,258 for *tableaux.pl* in this test case.

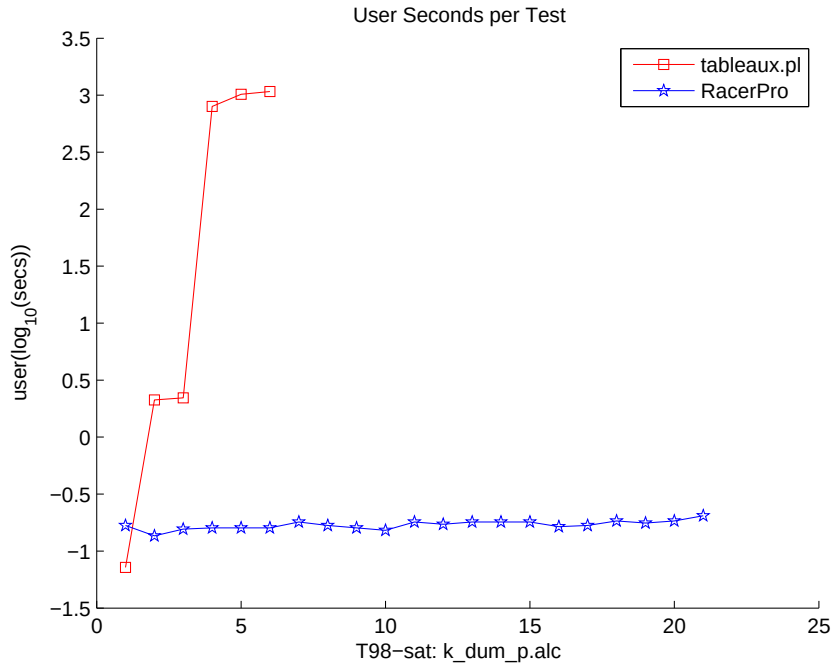

Figure 6.3: Comparing RacerPro and tableaux.pl on k_dum_p.alc

	Extended Mindswap Tests		
No.	fact.pl	tableaux.pl	lpdl.pl
1	2567	183	345
2	744	442	808
3	822	2005	1942
4	100	199	162
5	1995	204	339
6	9834	7337	7400
7	142	269	210
8	4710	353	672
9	166	288	264
10	378	492	740
11	247	290	525
12	1259	1072	5039
13	2975	153	348
14	5007280	1948	4177
15	2213	10674	13529
16	145	212	458

	Extended Mindswap Tests		
No.	fact.pl	tableaux.pl	lpdl.pl
17	2079	2470	4025
18	46334	395	929
19	1412	1503	2631
20	3927	349	757
21	219	306	387
22	2567	184	345
23	1765	8798	2912
24	81	142	167
25	135	957	397
26	31719	269	668
27	3055	256	524
28	1107	122	220
29	353	517	495
30	121	199	212
31	26136	357	571
32	165	278	321
33	384	463	728
34	26067	534	1129
35	578	584	1629
36	81	88	96
37	23	80	71
38	83	74	82
39	269	128	203
40	335	129	204
41	112	135	81
42	914	242	268
43	186	326	354

Table 6.1: Number of Inferences of Prolog Implementations

	k_dum_n.alc		*k_dum_p.alc*	
No.	*RacerPro*	*tableaux.pl*	*RacerPro*	*tableaux.pl*
1	0.164	0.028	0.168	0.072
2	0.152	0.02	0.136	2.116
3	0.164	0.028	0.156	2.208
4	0.164	0.024	0.16	797.49
5	0.156	0.024	0.16	1018.864
6	0.152	0.024	0.16	1076.867
7	0.164	0.028	0.18	
8	0.168	0.024	0.168	
9	0.18	0.032	0.16	
10	0.16	0.032	0.152	
11	0.168	0.044	0.18	
12	0.176	0.052	0.172	
13	0.18	0.068	0.18	
14	0.172	0.076	0.18	
15	0.172	0.092	0.18	
16	0.172	0.1	0.164	
17	0.204	0.128	0.168	
18	0.192	0.124	0.184	
19	0.196	0.164	0.176	
20	0.208	0.176	0.184	
21	0.204	0.212	0.204	

Table 6.2: Runtime Performance of RacerPro and tableaux.pl

6.5 Evaluation

Using Tableaux reasoning for Description Logics as implemented in *tableaux.pl* shows promise. Compared to other Prolog implementations, it fares very well, coming out fastest on average in the extended Mindswap test suite. Using ordered and exhaustive rule application on nodes, parsimonious rules and maintaining a fringe rather than the whole proof tree seem to offer a sound and efficient way of implementing Tableaux. Moreover, the initial implementation *tableaux.pl* still leaves room for obvious optimisations, like the elimination of '=' in *transform_connect/3*.

The comparison with a commercial and industrial-strength DL reasoner shows that

at least for certain problems, reasoning in Prolog can compete in speed, and even outperform the commercial product. The interesting issue here would be to explore the boundaries of this behaviour, and to analyse the characteristics – both of the test cases and the implementation / Prolog – that lead to it.

Chapter 7

Conclusions

The goal of this work was to re-construct a Tableaux reasoner for Description Logics in Prolog. The discussion of the classical Tableaux proof rules, as proposed e.g. by Horrocks [5] and Baader and Sattler [10] showed that there is a certain degree of non-determinism in the algorithm and generality in the proof rules that leave room for implementation decisions.

Taking existing ideas and implementations into account, we proposed three strategies how to concretise the general algorithm:

- *Ordered* and *exhaustive* application of rules on a give node of the proof tree. This approach was already suggested by Baader and Sattler (e.g. [10]).
- The use of *parsimonious* rules which delete compound expressions from the proof tree after they have been evaluated.
- Keeping a *fringe* of current leaf nodes, rather the whole tree, during a proof.

The latter two strategies depend on the first. You cannot delete compound expressions if you cannot be sure that by some future rule application this expression might gain relevance again; ordered and exhaustive application ensures this. And only by exhaustive node expansion can a node be discarded from the remainder of the proof.

These strategies have been put to work in a Prolog implementation, *tableaux.pl*. Comparisons with other Prolog implementations and even a commercial DL reasoner showed good results; and although an interpreted Prolog implementation might not challenge other optimised implementations on a general performance level, the declarative nature of the language makes it an interesting choice for rule-based proof systems like Tableaux. It facilitates extensions to the basic algorithm and the exploration of de-

sign alternatives. Combining and mixing the code of various implementations is easy, as the various tests showed.

Last but not least, a DL reasoner in Prolog provides opportunities for various kinds of users. As a Prolog module it makes it easy for other Prolog developers to take advantage of this facility in their own code, e.g. for the development of Semantic Web agents. Ontology developers can quickly check their ontologies and debug inconsistencies. Researchers can study and modify the source code to provide specialised services. It shows that DL reasoning, and therefore a corner stone of the Semantic Web, does not have to be an inaccessible terrain of heavyweight software. It can be lightweight.

7.1 Future Work

The presented work leaves room for a lot of extensions along various dimensions.

7.1.1 DL Language Extensions

The presented implementation only covers the most basic DL, $\mathcal{ALC}$. A natural and desirable way to extend it would be to support a more powerful language, the next step in the hierarchy being $\mathcal{ALCN}$. $\mathcal{ALCN}$ extends the basic language with number (cardinality) restrictions in their unqualified form ($\geq 3\, \mathit{hasChildren}$). Qualified number restrictions would give you $\mathcal{ALCQ}$ ($\geq 3\, \mathit{hasChildren}.\mathit{Person}$). $\mathcal{ALCN}$ is the language implemented by Meissner [16].

7.1.2 Integration with Rules

Initiatives are under way to add rules to ontologies, particularly in the context of the Semantic Web. These are among others the Rule Markup Language initiative, RuleML[1] , the W3C's Rule Interchange Format, RIF[2], and REWERSE[3], a European initiative that concerns itself with reasoning languages for the Web in general. Adding a "rulebox" to their definition means that ontologies enter a whole new area of expressiveness. Reasoning in Prolog over ontologies should make it easy to incorporate rules that come with them. A good starting point might be [25].

[1] http://www.ruleml.org/; Boley [24] especially discusses the relation to Prolog and XML.
[2] http://www.w3.org/2005/rules/
[3] http://rewerse.net/

7.1.3 Restricted Natural Language Interface

It should be straight-forward to define a controlled vocabulary to define and query ontologies. A corresponding interface to the reasoner could then be built, making the system more accessible for humans and less dependent on a formal syntax.

A fish is a kind of animal. $\{fish \sqsubseteq animal\}$
A brother is a sibling that is male. $\{brother \equiv sibling \sqcap male\}$
Every mother has a child. $\{mother \sqsubseteq \exists hasChild\}$
Koalas only eat eucalyptus leaves. $\{koala \sqsubseteq \forall eat.eucalyptusLeaf\}$

It could be easily translated into a variety of technical representations, like OWL, Prolog terms or Lisp terms[4].

7.1.4 Ontology Representations

The system could be enriched with more parsers for various ontology representations and formats. This would allow a greater variety of ontologies to be read in and reasoned with. Most prominently, a parser for OWL would allow the system to work on ontologies represented in OWL, as can be found on the Internet and produced by ontology editors such as Protégé. Prolog packages for parsing XML exist for example for SICStus (PiLLoW[5]) and SWI (SemWeb[6]), which also includes support for RDF.

7.1.5 Proof Explanation

A satisfiability reasoner is fed with an ontology and a query, and comes back with an answer, yes or no (meaning the query is satisfiable or not). Given the possible complexity of the ontology and the transformation steps necessary to derive a conclusion a result might be not at all obvious. Proof explanations that expose the crucial steps of the transformation could be deployed to e.g., convince the user or help in debugging the ontology. A possible starting point might be [26].

[4] See for example the work done in the "Attempto Controlled English" project, http://www.ifi.unizh.ch/attempto/, where translation from and into OWL DL is achieved.
[5] http://www.sics.se/sicstus/docs/3.12.5/html/sicstus/PiLLoW.html
[6] http://www.swi-prolog.org/packages/semweb.html

7.1.6 Optimisations

There is extensive literature about optimising Tableaux, starting with Ian Horrocks' PhD thesis [5]. Optimisation techniques that might be beneficial for even the basic $\mathcal{ALC}$ include *caching*[5, p.91], *lazy unfolding* [19, p.9][5, p.79] and *dependency directed backtracking* [19, p.9][5, p.89].

7.1.7 XSB Prolog

The implementation might benefit from an alternative Prolog system, such as XSB[7] [27] , to gain optimisation effects via caching using tabled resolution. This might be especially effective in proofs that contain equal sub-expressions in different parts of the proof tree, especially in connection with lazy unfolding.

7.1.8 Concurrent Implementation

The Tableaux algorithm lends itself well to concurrency. Branches in the proof tree and alternative trees that stem from union elimination could be pursued by independent reasoner threads. In conjunction with distributing the threads to multiple processors this could lead to a significant speed-up in reasoning throughput. Points to start from might be [28] and [29].

7.1.9 DIG Interface

The Description Logic Implementation Group (DIG)[8] has defined a network-based protocol to interact with a DL reasoner[9] [30]. DIG is a quasi-standard DL reasoner interface that is built on HTTP and XML. This allows arbitrary clients, e.g. ontology editors, to check their ontologies and run queries against them. The DIG interface is implementation-independent and therefore allows cross-reasoner interaction. DIG is supported on the reasoner side by FaCT and Racer, and on the client side by Protégé and OilEd.

For *tableaux.pl* the implementation could be done in a separate module that handles the communication protocol and formats, and interfaces to the reasoning predicates. In that sense, the DIG interface would be a first application using the reasoner library.

[7] http://xsb.sourceforge.net/
[8] http://dl.kr.org/dig/
[9] http://dig.sourceforge.net/

Zhisheng Huang from the Free University of Amsterdam has put together a promising framework called XDIG[10] [31]. XDIG is implemented in Prolog and is basically a network-based DIG proxy that sits between the DIG client and a backend DIG server like Pellet. Huang developed XDIG to intercept DIG requests and add additional functionality to the reasoning protocol, thereby extending DIG (hence the "X" in XDIG). But his framework leaves enough opportunity to integrate a complete DL reasoner and omit the backend server altogether. It comes with a well-documented API, so the reasoner library need not concern itself with the details of the network protocol, packing/unpacking XML etc.

[10] http://wasp.cs.vu.nl/sekt/dig/

Appendix A

Prolog Code: tableaux.pl

This is the code for *tableaux.pl*, the Tableaux reasoner that was developed for this dissertation.

```
% tableaux.pl -- tableaux reasoner for description logics
:- use_module(library(lists)).
:- op(100,fy,~).
:- dynamic ont/1.
:- dynamic id/1.

% Main Proof Goal
tableaux_proof(Exp) :-   % true/fals = satisfiable/unsatisfiable
        proof(Exp).

%construct_goal
proof(equiv(A,B)) :-
        \+ tabl(and(A,~B)),
        \+ tabl(and(B,~A)).
proof(subsum(A,B)) :-
        \+ tabl(and(A,~B)).
proof(disjoint(A,B)) :-
        \+ tabl(and(A,B)).
proof(unsat(A)) :-  % unsatisfiable A
        \+ tabl(A).
proof(A) :-         % try to satisfy everything else
        tabl(A).

% main worker
tabl(Exp) :-
        expand_defs(Exp,Exp1),     % expand expression into most basic
        negnormform(Exp1,Exp2),    % NNF transformation
        setID(0),
        !,
        search([[Exp2]],df,_).     % do the proof as an agenda search
```

```
negnormform(~ ~X,X1) :-
        negnormform(X,X1).
negnormform(~forall(R,C),exist(R,C1)) :-
        negnormform(~C,C1).
negnormform(forall(R,C),forall(R,C1)) :-
        negnormform(C,C1).
negnormform(~exist(R,C),forall(R,C1)) :-
        negnormform(~C,C1).
negnormform(exist(R,C),exist(R,C1)) :-
        negnormform(C,C1).
negnormform(~and(A,B),or(A1,B1)) :-
        negnormform(~A,A1),
        negnormform(~B,B1).
negnormform(and(A,B),and(A1,B1)) :-
        negnormform(A,A1),
        negnormform(B,B1).
negnormform(~or(A,B),and(A1,B1)) :-
        negnormform(~A,A1),
        negnormform(~B,B1).
negnormform(or(A,B),or(A1,B1)) :-
        negnormform(A,A1),
        negnormform(B,B1).
negnormform(~X,~X) :-
        atom(X).
negnormform(X,X) :-
        atom(X).

expand_defs(forall(R,C),forall(R,C1)) :-
        expand_defs(C,C1).
expand_defs(exist(R,C),exist(R,C1)) :-
        expand_defs(C,C1).
expand_defs(and(A,B),and(A1,B1)) :-
        expand_defs(A,A1),
        expand_defs(B,B1).
expand_defs(or(A,B),or(A1,B1)) :-
        expand_defs(A,A1),
        expand_defs(B,B1).
expand_defs(~A,~A1) :-
        expand_defs(A,A1).
expand_defs(X,Y) :-
        atom(X),
        ont(equiv(X,X1)),
        expand_defs(X1,Y).
expand_defs(X,X) :-
        atom(X),
        \+ ont(equiv(X,_)).

% search(+Goal,+Style,-ResultList)  -- agenda style search
%  -- transforms Goal into a list of [clash]/[model] elements
```

```
search([],_,[]).
search(Reduced, _, Reduced) :-
        Reduced = [H|_],
        H = [clash],         % a clash leaf fails the proof
        !,
        fail.
search([Node| T], Style, Reduced) :-
        process_node(Node,NewNodes),
        filter_nodes(NewNodes,NewNodes1),
        merge_agendas(NewNodes1, T, Style, New),
        search(New, Style, Reduced).

filter_nodes(NewNodes,NewNodes1) :-
        (   setof(X,(member(X,NewNodes),X\=[model]),NewNodes1);
            NewNodes1 = []),
        !.

merge_agendas(A1, A2, df, New) :-
        append(A1, A2, New),!.
merge_agendas(A1, A2, bf, New) :-
        append(A2, A1, New),!.

% reduce a node of the proof tree
process_node([clash],[]) :- !.
process_node([model],[]) :- !.
process_node(ListOfDLExps,ResultListOfLists) :-
        transform_connect(ListOfDLExps,_,LoL3),
        expand_nodes(LoL3,ResultListOfLists).

expand_nodes([],[]).
expand_nodes([H|R],RLoL) :-
        expand_node(H,RLoL1),
        expand_nodes(R,RLoL2),
        append(RLoL1,RLoL2,RLoL).

% process a single node
expand_node([],[]).
expand_node([model],[[model]]) :-!.
expand_node([clash],[[clash]]) :-!.
expand_node(Node,[N1]) :-
        check_clash(Node,N1),!.  % clash closes this branch
expand_node(Node,N1) :-
        expand_exist(Node,[],LoE),    % expand existential restrictions
        LoE \= [],                    % only continue with non-empty edges
        expand_forall(Node,LoE,LoE2), % try eliminate value restrictions
        extract_nodes(LoE2,N1).       % get the list of new fringe nodes
expand_node(Node,[N1]) :-        % model closes this branch
        expand_exist(Node,[],LoE),
        LoE = [],
```

```
        N1 = [model].

% extract list of nodes from list of edges
extract_nodes([],[]).
extract_nodes([edge(_,_,N)|R],[N|R1]) :-
        extract_nodes(R,R1).

% transform and/or connectives
transform_connect([],_,[[]]).
transform_connect([and(A,B)|R],N,R1) :-
        ( member(A,R) ->
            ( member(B,R) ->
                transform_connect(R,N,R1);
                transform_connect([B|R],N,R1));
            ( member(B,R) ->
                transform_connect([A|R],N,R1);
                transform_connect([A,B|R],N,R1))).
transform_connect([or(A,B)|R],N,LoL) :-
        ( \+ (member(A,R) ; member(B,R)) ->
            ( transform_connect([A|R],N,LoL);
              transform_connect([B|R],N,LoL));
            transform_connect(R,N,LoL)).
transform_connect([H|R],N,LoL) :-
        H \= and(_,_),
        H \= or(_,_),
        transform_connect(R,N,LL1),
        LL1 = [LL2],
        LoL = [[H|LL2]].

% transform forall/exist quantifiers
expand_exist([],L,L).
expand_exist([exist(R,C)|T],T1,L) :-
        (   (member(edge(R,_,X),T1), member(C,X) ) -> % existing R edge
            expand_exist(T,T1,L);
            getID(Id),
            expand_exist(T,[edge(R,Id,[C])|T1],L)
        ).
expand_exist([H|T],T1,L) :-
        H \= exist(_,_),
        expand_exist(T,T1,L).

expand_forall(_,[],[]).
% push concept into exist. node
expand_forall(Node,[edge(R,I,N)|RoE],[edge(R,I,N2)|R1]) :-
        setof(X,member(forall(R,X), Node),C1),
        append(C1,N,N1),
        remove_duplicates(N1,N2),
        expand_forall(Node,RoE,R1).
expand_forall(Node,[edge(R,I,N)|RoE],[edge(R,I,N)|LoE]) :-
```

```
        \+ member(forall(R,_),Node),
        expand_forall(Node,RoE,LoE).

check_clash(Exp,Exp1) :-
        member(A,Exp),
        member(~A,Exp),
        Exp1 = [clash].

remove_duplicates([],[]).
remove_duplicates([H|T],[H|R]) :-
        r_d(H,T,[],R1),
        remove_duplicates(R1,R),!.
r_d(_,[],T,T).
r_d(E,[E|T],TT,T1) :- r_d(E,T,TT,T1).
r_d(E,[X|T],TT,T1) :- X \= E, r_d(E,T,[X|TT],T1).

getID(I):-
        id(I1),
        I is I1 + 1,
        retract(id(I1)),
        asserta(id(I)),!.
getID(I):-
        \+ id(_),
        I is 0,
        asserta(id(I)),!.
setID(X):-
        ( id(Y) ->
            retract(id(Y));
            true
        ),
        asserta(id(X)).
```

Appendix B

Prolog Code: fact.pl

This is the *fact.pl* implementation of Tableaux in Prolog. It is a modified version of Stuart Aitken's *fact.pl* code [18].

```
% the FaCT algorithm for ALC
% - by Stuart Aitkens
% - some changes by Thomas Herchenroeder

:- dynamic count/1.
:- assert(count(1)).

inc(I):- count(I),retractall(count(_)),J is I + 1, assertz(count(J)).

star(G1) :- nnf(G1,G), start(G).

start(G):-
        elim([[a0,[G]]],R,S),
        !,rep_elim([[a0,[G]]],R),
        !,rep_elim([[a0,[G]]],S).

rep_elim(_,[]):-!.
rep_elim(_,R):- closed(R),!.
rep_elim(R,R):-!,fail.
rep_elim(_,R):-
        %write('Solving: '),
        %write(R),nl,
        elim(R,S,T),
        !,rep_elim(R,S),
        !,rep_elim(R,T).

rep_elim(_,R):-!,fail.

closed(T):-
```

```
        member([_,L],T),
        member(complementOf(A),L),
        member(A,L),!.

elim([],[],[]).
% inters elim
elim(T,[[N,[A,B|LL]]|U],[]):-
        member([N,L],T),
        member(intersectionOf(A,B),L),
        \+ (member(A,L), member(B,L)),
        delete(T,[N,L],U),
        delete(L,intersectionOf(A,B),LL).

% union elim
elim(T,[[N,[A|LL]]|U],[[N,[B|LL]]|U]):-
        member([N,L],T),
        member(unionOf(A,B),L),
        \+ (member(A,L); member(B,L)),
        delete(T,[N,L],U),
        delete(L,unionOf(A,B),LL).

% exists elim
elim(T,[[s(R,N,I),[A]],[N,LL]|U],[]):-
        member([N,L],T),
        member(exists(R,A),L),
        delete(T,[N,L],U),
        delete(L,exists(R,A),LL),
        inc(I).

% forall elim
elim(T,[[s(R,N,I),[A|M]]|V],[]):-
        member([N,L],T),
        member(forall(R,A),L),
        member([s(R,N,I),M],T),
        not(member(A,M)),
        delete(T,[s(R,N,I),M],V).

nnf(forall(R,B),forall(R,NB)):-
    nnf(B,NB).

nnf(exists(R,B),exists(R,NB)):-
    nnf(B,NB).

nnf(unionOf(A,B),unionOf(NA,NB)):-
    nnf(A,NA),nnf(B,NB).

nnf(intersectionOf(A,B),intersectionOf(NA,NB)):-
        nnf(A,NA),nnf(B,NB).
```

```
nnf(complementOf(A),NA):-
        neg_nnf(A,NA).

nnf(A,A).

neg_nnf(forall(R,B),exists(R,NB)):-
    neg_nnf(B,NB).

neg_nnf(exists(R,B),forall(R,NB)):-
    neg_nnf(B,NB).

neg_nnf(unionOf(A,B),intersectionOf(NA,NB)):-
        neg_nnf(A,NA),neg_nnf(B,NB).

neg_nnf(intersectionOf(A,B),unionOf(NA,NB)):-
        neg_nnf(A,NA),neg_nnf(B,NB).

neg_nnf(complementOf(A),NA):-
        nnf(A,NA).

neg_nnf(A,complementOf(A)).
```

B.1 List of Changes

This is a list of changes applied to the original code, in order to prepare the program for the tests:

- *count/1*. Add a "*:- dynamic count/1*" declaration.
- *inc/1*. Replaced the call to *retract(count(I))* with *retractall(count(_))*, since it caused multiple instances being left in the database and individuals getting the same number.
- *write/1*. Deleted a couple of *write/1* statements in various clauses that provided trace output, which was not necessary for the benchmarking and would only slow down performance.
- *star/1*. Added a top-level predicate *star/1* to invoke the NNF transformation before invoking the core reasoning predicate *start/1*.
- *rep_elim/2*. Re-ordered the cuts in the fourth clause of *rep_elim/2*, so they would appear *before* the recursive calls.

- *elim/3*. Add the standard Tableaux member checks for existing operands in the clauses for intersection elimination (\+ *(member(A,L), member(B,L))*) and union elimination (\+ *(member(A,L); member(B,L))*), to avoid looping (in connection with forall elimination).

Appendix C

Extended Mindswap Tests

Here is an overview of the extended Mindswap test suite used to benchmark the various Prolog implementations. Following that, the actual contents of the test files is given.

C.1 Table of Tests

All file names matchin the pattern *ex**.pl* are based on the corresponding files from the Mindswap Semantic Web course[1]. The other test cases, matching *x_ex_**.pl*, are from the KMM lecture or self-constructed.

No.	File Name	Satisfiable
1	ex1_10.pl	y
2	ex1_11.pl	n
3	ex1_1.pl	n
4	ex1_2_1.pl	n
5	ex1_2_2.pl	y
6	ex1_2_3.pl	n
7	ex1_2_4.pl	n
8	ex1_3.pl	y
9	ex1_4.pl	n
10	ex1_5.pl	n
11	ex1_6.pl	n
12	ex1_7.pl	n

[1] http://www.mindswap.org/2004/cmsc498w/. The original files have the same names with a "*.py*" extension.

No.	File Name	Satisfiable
13	ex1_8.pl	y
14	ex1_9.pl	y
15	ex2_1.pl	n
16	ex2_2.pl	n
17	ex2_3.pl	n
18	ex2_4.pl	y
19	ex2_5.pl	n
20	ex2_7.pl	y
21	ex2_8.pl	n
22	ex2_9.pl	y
23	ex3_1.pl	n
24	ex3_3.pl	n
25	ex3_4.pl	n
26	ex3_5.pl	y
27	ex3_7.pl	y
28	ex3_9.pl	y
29	x_ex_aa.pl	y
30	x_ex_ab.pl	y
31	x_ex_ac.pl	n
32	x_ex_ad.pl	y
33	x_ex_ae.pl	y
34	x_ex_af.pl	n
35	x_ex_ag.pl	y
36	x_ex_ah.pl	y
37	x_ex_ai.pl	n
38	x_ex_aj.pl	y
39	x_ex_ak.pl	y
40	x_ex_al.pl	y
41	x_ex_am.pl	y
42	x_ex_an.pl	y
43	x_ex_ao.pl	n
44	ex3_2.pl	y

No.	File Name	Satisfiable

Table C.1: The Extended Mindswap Test Suite

C.2 Contents of Tests

This is the contents of the test files containing the respective Prolog terms, including a file called *x_onto.pl*, the family ontology which is shared among several of the *x_ex_*.pl* files. For each section the file name precedes its contents.

```
ex1_10.pl:
sat(true).
query( and( veggiepizza, meatpizza)).
ont(equiv(veggiepizza,and( pizza, forall( hastopping, (~meat))))).
ont(equiv(meatpizza,and( pizza, forall( hastopping, (~veggie))))).
ont(equiv(veggie,or( mushroom, olive))).
ont(equiv(meat,or( pepperoni, sausage))).

ex1_11.pl:
sat(false).
query( and( exist( p, a), and(exist( p, b), and(and( c, d), (~exist( p, (~and(
(~e), f)))))))).
ont(equiv(a,and( h, and( i, (~d))))).
ont(equiv(j,( ~k))).
ont(equiv(b,( ~g))).
ont(equiv(d,forall( q, j))).
ont(equiv(g,( ~e))).

ex1_1.pl:
sat(false).
query( and( wine, beer)).
ont(equiv(beer,and( drink,and(exist( hasingr, water),and( exist( hasingr,
hops),and( exist( hasingr, malt), forall( hasingr, or( water, or(hops,
malt)))))))).
ont(equiv(grapes,and( (~hops),and( (~malt), (~water))))).
ont(equiv(wine,and( drink, exist( hasingr, grapes)))).

ex1_2_1.pl:
sat(false).
query( and( exist( r, b), forall( r, (~b)))).

ex1_2_2.pl:
sat(true).
query( and( exist( r, b), forall( r, or( a, (~b))))).

ex1_2_3.pl:
```

```
sat(false).
query( and( exist( r, b), forall( r, (~b)))).
ont(equiv(aa,or( a, (~a)))).
ont(equiv(a,exist( r, exist( r, exist( r, c))))).
ont(equiv(b,and( a, and(exist( r, aa), or( aa, (~aa)))))).

ex1_2_4.pl:
sat(false).
query( and( exist( r, b), forall( r, or( a, (~b))))).
ont(equiv(a,( ~b))).

ex1_3.pl:
sat(true).
query( and( werewolf, human)).
ont(equiv(werewolf,and( animal, and(exist( haspower, magical),forall( speaks,
language))))).
ont(equiv(acramantula,and( beast,and(or( male, female),exist( haspower,
magical))))).
ont(equiv(wizard,and( male,and(human,exist( haspower, magical))))).
ont(equiv(centaur,and( animal,and((~human),and(or( male, female),and(exist(
haspower, magical),forall( speaks, language)))))).
ont(equiv(vampire,and( beast,and(or( male, female),forall( haspower,
magical))))).
ont(equiv(beast,and( animal,(~human)))).
ont(equiv(muggle,and( human,and( or( male, female),forall( haspower,
(~magical)))))).
ont(equiv(witch,and( female,and( human, exist( haspower, magical))))).
ont(equiv(human,and( animal,exist( speaks, language)))).
ont(equiv(male,and( animal,(~female)))).
ont(equiv(merpeople,and( animal,and( (~human),and( forall( haspower, magical),
forall( speaks, language)))))).

ex1_4.pl:
sat(false).
query( and( roundoff, (~backhandspring))).
ont(equiv(salto,and( cartwheel, and( (~exist( hasposition, handsonfloor)),
exist( hasposition, twist))))).
ont(equiv(fronttuck,and( (~cartwheel),forall( hasposition, tuck)))).
ont(equiv(roundoff,and( cartwheel,and( handstand ,forall( hasposition,
pike))))).
ont(equiv(backwalkover,and( exist( hasposition, bridge),exist( hasposition,
handstand)))).
ont(equiv(forwardroll,or( exist( hasposition, pike),or( exist( hasposition,
straddle), exist( hasposition, tuck))))).
ont(equiv(backhandspring,exist( hasposition,bridge))).
ont(equiv(handstand,( ~forall( hasposition, pike)))).

ex1_5.pl:
sat(false).
```

```
query( and( werewolf, human)).
ont(equiv(werewolf,and( beast,and( exist( haspower, magical), forall( speaks,
language))))).
ont(equiv(acramantula,and( beast ,and(or( male, female), exist( haspower,
magical))))).
ont(equiv(wizard,and( male, and( human, exist( haspower, magical))))).
ont(equiv(centaur,and( animal,and( (~human), and(or( male, female), and( exist(
haspower, magical), forall( speaks, language)))))).
ont(equiv(vampire,and( beast,and( or( male, female), forall( haspower,
magical))))).
ont(equiv(beast,and( animal, (~human)))).
ont(equiv(muggle,and( human,and( or( male, female), forall( haspower,
(~magical)))))).
ont(equiv(witch,and( female,and( human, exist( haspower, magical))))).
ont(equiv(human,and( animal, exist( speaks, language)))).
ont(equiv(male,and( animal, (~female)))).
ont(equiv(merpeople,and( animal,and( (~human),and( forall( haspower, magical),
forall( speaks, language)))))).

ex1_6.pl:
sat(false).
query( and( lessergnupubliclicense, forall( hasrestriction,
cannotredistribute))).
ont(equiv(academicfreelicense,and( freesoftwarelicense, exist( hasrestriction,
and( mustkeepdisclaimer, and((~mustdistributemods), and((~licenseisviral),
(~cannotredistribute)))))))).
ont(equiv(commonpubliclicense,and( freesoftwarelicense, exist( hasrestriction,
and( mustkeepdisclaimer, and( mustdistributemods, and((~licenseisviral),
(~cannotredistribute)))))))).
ont(equiv(publicdomainlicense,and( freesoftwarelicense ,exist( hasrestriction,
(~or( mustkeepdisclaimer, or( mustdistributemods, or(licenseisviral,
cannotredistribute)))))))).
ont(equiv(lessergnupubliclicense,and( freesoftwarelicense, exist(
hasrestriction, and( mustkeepdisclaimer, and(mustdistributemods,
and((~licenseisviral), (~cannotredistribute)))))))).
ont(equiv(commercialsoftwarelicense,exist( hasrestriction,
cannotredistribute))).
ont(equiv(softwarelicense,or( freesoftwarelicense, commercialsoftwarelicense))).
ont(equiv(opensoftwarelicense,and( freesoftwarelicense, exist( hasrestriction,
and( mustkeepdisclaimer,and( mustdistributemods, and(licenseisviral,
(~cannotredistribute)))))))).
ont(equiv(gnugeneralpubliclicense,and( freesoftwarelicense, exist(
hasrestriction, and( mustkeepdisclaimer, and(mustdistributemods,
and(licenseisviral, (~cannotredistribute)))))))).

ex1_7.pl:
sat(false).
query( and( puppy, and(mamadog, wifedog))).
ont(equiv(parentdog,or( papadog, mamadog))).
```

```
ont(equiv(husbanddog,and( maledog, exist( haswife, femaledog)))).
ont(equiv(maledog,and( dog, (~female)))).
ont(equiv(papadog,and( maledog ,exist( haschild, dog)))).
ont(equiv(femaledog,and( dog, female))).
ont(equiv(wifedog,and( femaledog, exist( hashusband, maledog)))).
ont(equiv(puppy,and( or( maledog, femaledog), and( exist( hasmother,
mamadog),and( exist( hasfather, papadog) ,(~exist( haschild, dog)))))))).
ont(equiv(mamadog,and( femaledog, exist( haschild, dog)))).

ex1_8.pl:
sat(true).
query( or( and( or( a, b), or( c, d)), and( or( a, c), or( b, d)))).

ex1_9.pl:
sat(true).
query( and( hydra, and(dragonet, exist( elemental, fire)))).
ont(equiv(slitheringdragon,and( dragon, forall( transportmode, (~or( flying,
walking)))))).
ont(equiv(walkingdragon,and( dragon, exist( transportmode, walking)))).
ont(equiv(firedrake,and( drake, and(forall( elemental, fire), exist(
disposition, foe))))).
ont(equiv(icedrake,and( drake, and(forall( elemental, water), exist(
disposition, foe))))).
ont(equiv(orientaldragon,and( walkingdragon, and( exist( elemental, water),
forall( disposition, friend))))).
ont(equiv(drake,and( walkingdragon, and( exist( elemental, or( water, fire)),
forall( disposition, foe))))).
ont(equiv(hydra,and( or( slitheringdragon, flyingdragon), exist( disposition,
foe)))).
ont(equiv(westerndragon,and( flyingdragon,and( forall( elemental, or( earth,
water)), exist( disposition, foe))))).
ont(equiv(wyrm,and( slitheringdragon, exist( elemental, water)))).
ont(equiv(flyingdragon,and( dragon, exist( transportmode, flying)))).
ont(equiv(dragonet,and( forall( disposition, foe),and( or( walkingdragon,
flyingdragon), forall( elemental, (~or( earth, water)))))))).

ex2_1.pl:
sat(false).
query( and( a, and(d, and(g, and((~m), and((~n), and((~o), and((~p), and((~q),
and((~r), and((~s), and((~t),and( (~u), and((~v1), and((~w),
(~x)))))))))))))))).
ont(equiv(a,or( b, c))).
ont(equiv(c,or( o, p))).
ont(equiv(b,or( m, n))).
ont(equiv(e,or( q, r))).
ont(equiv(d,or( e ,f))).
ont(equiv(g,or( h ,i))).
ont(equiv(f,or( s ,t))).
ont(equiv(i,or( w ,x))).
```

```
ont(equiv(h,or( u ,v1))).

ex2_2.pl:
sat(false).
query( and( wifedog, husbanddog)).
ont(equiv(parentdog,or( papadog, mamadog))).
ont(equiv(husbanddog,and( maledog, exist( haswife, femaledog)))).
ont(equiv(maledog,and( dog, (~female)))).
ont(equiv(papadog,and( maledog, exist( haschild, dog)))).
ont(equiv(femaledog,and( dog, female))).
ont(equiv(wifedog,and( femaledog, exist( hashusband, maledog)))).
ont(equiv(puppy,and( or( maledog, femaledog), and(exist( hasmother, mamadog),
and(exist( hasfather, papadog), (~exist( haschild, dog))))))).
ont(equiv(mamadog,and( femaledog, exist( haschild, dog)))).

ex2_3.pl:
sat(false).
query( and( dragonet, (~and( forall( disposition, foe), and(or( walkingdragon,
flyingdragon), forall( elemental, (~or( earth, water))))))).
ont(equiv(slitheringdragon,and( dragon, forall( transportmode, (~or( flying,
walking)))))).
ont(equiv(walkingdragon,and( dragon, exist( transportmode, walking)))).
ont(equiv(firedrake,and( drake,and( forall( elemental, fire), exist(
disposition, foe))))).
ont(equiv(icedrake,and( drake, and(forall( elemental, water), exist(
disposition, foe))))).
ont(equiv(orientaldragon,and( walkingdragon, and(exist( elemental, water)
,forall( disposition, friend))))).
ont(equiv(drake,and( walkingdragon, and(exist( elemental, or( water, fire)),
forall( disposition, foe))))).
ont(equiv(hydra,and( or( slitheringdragon, flyingdragon), exist( disposition,
foe)))).
ont(equiv(westerndragon,and( flyingdragon,and( forall( elemental, or( earth,
water)), exist( disposition, foe))))).
ont(equiv(wyrm,and( slitheringdragon, exist( elemental, water)))).
ont(equiv(flyingdragon,and( dragon, exist( transportmode, flying)))).
ont(equiv(dragonet,and( forall( disposition, foe),and( or( walkingdragon,
flyingdragon), forall( elemental, (~or( earth, water))))))).

ex2_4.pl:
sat(true).
query( and( exist( p, or( a, or(b, c))), and(exist( q, or( d, or(e, f))),
forall( p, and( c, d))))).
ont(equiv(b,and( (~a), a))).
ont(equiv(f,exist( p, and( g, h)))).

ex2_5.pl:
sat(false).
query( and( nonfiction, childrensbook)).
```

```
ont(equiv(map,and( exist( contains, words), (˜or( pictures, chapters))))).
ont(equiv(nonfiction,and( (˜fiction),and( exist( contains, chapters) ,forall(
contains, words))))).
ont(equiv(childrensbook,forall( contains, and( pictures, (˜words))))).
ont(equiv(fiction,or( map, and( chapters, words)))).

ex2_7.pl:
sat(true).
query( and( b, and(c, and(exist( p, and( a, and(c, exist( r, (˜d))))), forall(
r, d))))).

ex2_8.pl:
sat(false).
query( and( forall( r, and( a, b)), and(exist( r, a), exist( r, b)))).
ont(equiv(b,( ˜a))).

ex2_9.pl:
sat(true).
query( and( veggiepizza, meatpizza)).
ont(equiv(veggiepizza,and( pizza, forall( hastopping, (˜meat))))).
ont(equiv(meatpizza,and( pizza, forall( hastopping, (˜veggie))))).
ont(equiv(veggie,or( mushroom, olive))).
ont(equiv(meat,or( pepperoni ,sausage))).

ex3_1.pl:
sat(false).
query( and( and(and( dalek, cyborg), and(and( cyberman, forall( hasweakness,
sonicscrewdriver)), and(and( silurian, monster), and(and( timelord, forall(
hasweakness, bullets)), and( quark, exist( hasweakness, gold)))))), (˜and( and(
timelord, monster), and(and( timelord, dalek), and( quark, humanoid)))))).
ont(equiv(dalek,and( biological, and(mechanical, forall( hasweakness,
blindness))))).
ont(equiv(monster,and( biological,and( (˜humanoid), exist( hasweakness,
bullets))))).
ont(equiv(humanoid,and( biological,and( (˜mechanical), forall( hasweakness,
bullets))))).
ont(equiv(cyberman,and( cyborg, forall( hasweakness, gold)))).
ont(equiv(timelord,and( biological, humanoid))).
ont(equiv(silurian,and( biological, forall( hasweakness, bullets)))).
ont(equiv(robot,and( mechanical, and( (˜biological),and( forall( hasweakness,
logicalparadox), exist( hasweakness, sonicscrewdriver)))))).
ont(equiv(quark,( robot))).
ont(equiv(cyborg,and( mechanical,and( biological,and( exist( hasweakness,
bullets), exist( hasweakness, sonicscrewdriver)))))).

ex3_3.pl:
sat(false).
query( and( maledog, femaledog)).
ont(equiv(parentdog,or( papadog,mamadog))).
```

```
ont(equiv(husbanddog,and( maledog, exist( haswife, femaledog)))).
ont(equiv(maledog,and( dog, (~female)))).
ont(equiv(papadog,and( maledog, exist( haschild, dog)))).
ont(equiv(femaledog,and( dog, female))).
ont(equiv(wifedog,and( femaledog, exist( hashusband, maledog)))).
ont(equiv(puppy,and( or( maledog, femaledog),and( exist( hasmother,
mamadog),and( exist( hasfather, papadog), (~exist( haschild, dog))))))).
ont(equiv(mamadog,and( femaledog, exist( haschild, dog)))).

ex3_4.pl:
sat(false).
query( and( (~forall( p, a)), (~exist( p, (~a))))).
ont(equiv(a,( ~and( b, and(c, d))))).
ont(equiv(c,( ~or( f, (~f))))).
ont(equiv(b,( ~or( e, (~e))))).
ont(equiv(d,( ~or( g, (~g))))).

ex3_5.pl:
sat(true).
query( and( g5, (~digslr))).
ont(equiv(g5,and( or( exist( uses, film), exist( has, zoom)), (~lcd)))).
ont(equiv(digslr,and( slr,and( digital, lcd)))).
ont(equiv(slr,and( film ,zoom))).
ont(equiv(digital,and( (~film) ,exist( has, zoom)))).

ex3_7.pl:
sat(true).
query( and( a, and(b, and(forall( p, c), and(forall( p, (~c)), exist( r,
d)))))).

ex3_9.pl:
sat(true).
query( or( (~wierdopizza), veggiepizza)).
ont(equiv(wierdopizza,and( pizza, exist( hastopping, alien)))).
ont(equiv(meatpizza,and( pizza, forall( hastopping, (~veggie))))).
ont(equiv(meat,or( pepperoni, sausage))).
ont(equiv(veggie,or( mushroom, olive))).
ont(equiv(alien,( anchovy))).
ont(equiv(veggiepizza,and( pizza, forall( hastopping, (~meat))))).

x_ex_aa.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(disjoint(brother,sister)).

x_ex_ab.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(disjoint(man,woman)).
```

```
x_ex_ac.pl:
:- ensure_loaded('x_onto.pl').
sat(false).
query(subsum(luckyBrother,brother)).

x_ex_ad.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(subsum(father,parent)).

x_ex_ae.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(subsum(grandfather,father)).

x_ex_af.pl:
:- ensure_loaded('x_onto.pl').
sat(false).
query(disjoint(grandfather,father)).

x_ex_ag.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(disjoint(grandfather,sister)).

x_ex_ah.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(and(a,b)).

x_ex_ai.pl:
:- ensure_loaded('x_onto.pl').
sat(false).
query(and(a,~a)).

x_ex_aj.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(or(a,b)).

x_ex_ak.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(and(a,and(b,or(a,b)))).

x_ex_al.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
```

```
query(and(a,and(b,or(a,~b)))).

x_ex_am.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(exist(r,a)).

x_ex_an.pl:
:- ensure_loaded('x_onto.pl').
sat(true).
query(and(exist(r,a),exist(r,~a))).

x_ex_ao.pl:
:- ensure_loaded('x_onto.pl').
sat(false).
query(and(forall(r,a),and(exist(r,a),exist(r,~a)))).

ex3_2.pl:
sat(true).
query( and( westerndragon, orientaldragon)).
ont(equiv(slitheringdragon,and( dragon, forall( transportmode, (~or( flying,
walking)))))).
ont(equiv(walkingdragon,and( dragon, exist( transportmode, walking)))).
ont(equiv(firedrake,and( drake,and( forall( elemental, fire), exist(
disposition, foe))))).
ont(equiv(icedrake,and( drake,and( forall( elemental, water), exist(
disposition, foe))))).
ont(equiv(orientaldragon,and( walkingdragon,and( exist( elemental, water)
,forall( disposition, and( friend, exist( towards, or( people, animals))))))))).
ont(equiv(drake,and( walkingdragon,and( exist( elemental, or( water, fire)),
forall( disposition, foe))))).
ont(equiv(hydra,and( or( slitheringdragon, flyingdragon), exist( disposition,
foe)))).
ont(equiv(westerndragon,and( flyingdragon,and( forall( elemental, or( earth,
water)), exist( disposition, and( foe, exist( towards, people)))))).
ont(equiv(wyrm,and( slitheringdragon, exist( elemental, water)))).
ont(equiv(flyingdragon,and( dragon, exist( transportmode, flying)))).
ont(equiv(dragonet,and( forall( disposition, foe),and( or( walkingdragon,
flyingdragon), forall( elemental, (~or( earth, water)))))).

x_onto.pl:
ont(equiv(man,and(person,male))).
ont(equiv(woman,and(person,~man))).
ont(equiv(mother,and(woman,exist(hasChild,person)))).
ont(equiv(father,and(man,exist(hasChild,person)))).
ont(equiv(parent,exist(hasChild,person))).
ont(equiv(grandfather,and(man,exist(hasChild,father)))).
ont(equiv(brother,and(man,exist(hasSibling,person)))).
ont(equiv(sister,and(person,and(~brother,exist(hasSibling,person))))).
```

```
ont(equiv(luckyBrother,and(man,forall(hasSibling,sister)))).
```

Appendix D

Supported Ontology Format

This is a simple grammar that basically explains in which Prolog terms $\mathcal{ALC}$ expressions have to be expressed in, in order to be understandable for *tableaux.pl*. This was also the format I used for the extended Mindswap tests, and from which I converted for the other Prolog implementations.

```
DLexpression :: DLconcept | DLaxiom | DLquery

DLquery      :: equiv(DLconcept, DLconcept)    |
                subsum(DLconcept, DLconcept)   |
                disjoint(DLconcept, DLconcept) |
                unsat(DLconcept)               |
                DLconcept

DLaxiom      :: ont(equiv(PrimitiveConcept, DLconcept))

DLconcept    :: PrimitiveConcept |
                and(DLconcept, DLconcept) |
                or(DLconcept,  DLconcept) |
                exist(Relation, DLconcept)|
                forall(Relation,DLconcept)|
                ~DLconcept

PrimitiveConcept :: PrologLiteral

Relation     :: PrologLiteral
```

Appendix E

Ontology Translation Grammars

The following grammar predicates have been used to translate between the various ontology representations. Their main use is in the *get_files/[1,2]* predicates (cf. F) to translate between T98-sat Lisp-style syntax and the alternative Prolog styles and my own (cf. D).

```
:- op(90,fy,~).                        % ~A (negation)

% parse Racer Lisp into my format
c(X) --> ['('], com(C), dl(X), [')'].
dl(X) --> ['('], dle(X), [')'].
dl(X) --> dle(X).
dle(~X) --> [not], dl(X).
dle(R) --> con(C), dl(X), dl(Y), {R =.. [C,X,Y]}.
dle(X) --> [X],{atom(X)}.
con(X) --> [Y],{con(Y,X)}.
con(and,and).
con(or,or).
con(some,exist).
con(all,forall).
com(X) --> [X], {com(X)}.
com('alc-concept-coherent').

% (experimental transformations)
i(X) --> ['('], com(C), dl1(X), [')'].
dl1(X) --> ['('], dle1(X), [')'].
dl1(X) --> dle1(X).
dle1(~X) --> [not], dl1(X).
dle1(R) --> andor(C), dl1(X), dl1(Y), {R =.. [C,X,Y]}.
dle1(R) --> allsome(C), dl1(X), dl1(Y), {R =.. [C,X,Y]}.
dle1(R) --> [R], {atom(R)}.
andor(X) --> [Y],{andor(Y,X)}.
andor(and,&).
andor(or,u).
```

```
allsome(X) --> [Y],{allsome(Y,X)}.
allsome(some,exist).
allsome(all,forall).


% transform my format into Stuart Aitkens's
f(exist(A,B),exists(A1,B1))       :- f(A,A1),f(B,B1).
f(forall(A,B),forall(A,B1))       :- f(B,B1).
f(and(A,B),intersectionOf(A1,B1)) :- f(A,A1),f(B,B1).
f(or(A,B),unionOf(A1,B1))         :- f(A,A1),f(B,B1).
f(~A,complementOf(A1))            :- f(A,A1).
f(X,X)                            :- atom(X).


% transfrom my format into Adam Meissner's
l(exist(A,B),ex(A1,B1))         :- l(A,A1),l(B,B1).
l(forall(A,B),all(A,B1)) :- l(B,B1).
l(and(A,B),'&'(A1,B1))   :- l(A,A1),l(B,B1).
l(or(A,B),'v'(A1,B1))    :- l(A,A1),l(B,B1).
l(~A,'-'(A1))            :- l(A,A1).
l(X,X)                   :- atom(X).
```

Appendix F

Test Driver Scripts

The following programs have been used as driver scripts for the benchmark tests. The first runs the various Prolog implementations against the extended Mindswap tests (or a subset thereof). The invocation of a test run looked like

pl -s reg_test.pl -g "get_files('*I*',*R*)." -t halt.

where *I* is the path to a file containing the list of test cases to run (e.g. `test_pl/index.pl`) and *R* is one of [*tabl*|*fact*|*lpdl*], depending on the system you want to run.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% reg_test.pl                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% currently, this has to be run from test_pl/.., where test_pl
% contains the extended Mindswap tests in Prolog
:- consult(['/home/s0564890/div/courses/diss/code/tableaux.pl',
            '/home/s0564890/div/courses/diss/code/grammar1.pl'
            ]).
syst(fact,'/home/s0564890/div/courses/diss/code/stuart/fact.pl').
syst(lpdl,'/home/s0564890/div/courses/diss/code/meissner/lpdl.pl').

% dir interface
get_files(Dir,Sys):-
        ( memberchk(Sys,['fact','lpdl']) ->
            syst(Sys,P),        % get system file name
            consult(P);         % load system to run
            true),
        consult(Dir), % get list of test files
        flist(Files), % into a variable
        eval_files(Files,Sys),
        true.

eval_files([F|T],S):-
        eval_file(F,S),
```

```
        eval_files(T,S).


eval_file(File1,tabl):-
        string_concat('test_pl/',File1,File),
        consult(File),
        query(Q),
        sat(S),
        time((   proof(Q) ->
            S1 = true;
            S1 = false )),
        (   S1 = S ->
            report(File,ok);
            report(File,err)).

eval_file(File1,fact):-
        string_concat('test_pl/',File1,File),
        consult(File),
        write('Testing file: '), write(File), nl,
        query(Q_1),
        trans(Q_1,Q,S),
        %sat(S),
        expand_defs(Q,Q1),
        f(Q1,Q2),          % translate
        !,
        time((   star(Q2) ->
            S1 = false;
            S1 = true )),
        (   S1 = S ->
            report(File,ok);
            report(File,err)),
        true.

eval_file(File1,lpdl):-
        string_concat('test_pl/',File1,File),
        consult(File),
        write('Testing file: '), write(File), nl,
        query(Q_1),
        trans(Q_1,Q,S),
        expand_defs(Q,Q1),
        l(Q1,Q2),          % translate
        !,
        time((   th(unsatisfiable(Q2)) ->
            S1 = false;
            S1 = true )),
        (   S1 = S ->
            report(File,ok);
            report(File,err)),
        true.
```

```
report(File,ok):-
        write(File),write(': ok'),nl.
report(File,err):-
        write(File), write(': err'),nl.

% creates the goal from the query (for fact.pl)
trans(subsum(A,B),and(A,~B),S1) :-
        sat(S),
        invers(S,S1),
        !.
trans(disjoint(A,B),and(A,B),S1) :-
        sat(S),
        invers(S,S1),
        !.
trans(X,X,S):- sat(S),!.

invers(true,false).
invers(false,true).
```

The second program is the Prolog script to invoke *tableaux.pl* on a single test from the T98-sat test suite. It is called by a Perl script (later in this chapter), and depends on its preparations.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% reg_t98.pl                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- consult(['/home/s0564890/div/courses/diss/code/tableaux.pl',
            '/home/s0564890/div/courses/diss/code/grammar1.pl',
            '/home/s0564890/tmp/t1.pl']).

get_files(Sat):-
        t(L),          % load the query
        c(E,L,[]),     % compile into my format
        !,
        (   proof(E) ->    % run the proof
            S1 = true;
            S1 = false),
        (   S1 = Sat ->
            write(' ok');
            write(' err')),nl,
        true.
```

The third one is a glue script in Perl that drives the previous Prolog program. It controls the conversion of the input files into Prolog terms using another Perl script (*tok.perl* further down). It then invokes the Prolog interpreter to run a single test file

and times its execution. The variable `$Satisfiable_p` has to be adapted to whether the test case is from a *_p.alc (‘*false*’) or *_n.alc (‘*true*’) file.

```
#!/usr/bin/perl
#####################################
# reg_t98.perl                      #
#####################################
# This is to run tableaux.pl on a list of files from the T98-sat automatically
$Satisfiable_p = 'false';

# command to start Prolog
$p_cmd = qq ?pl -s reg_t98.pl -g 'get_files($Satisfiable_p).' -t halt.?;

while (<>) { # read list of input files
   $file = $_;
   chomp($file);
   $file =~ m{^.*/(\S+)$}; print ">>$1: "; # print base name
   # convert syntax
   $cmd = qq ?perl -pe 's/\\n/ /g;' $file |tok.perl|? .
          qq ?perl -ne 'BEGIN{print "t(";};chomp;print;END{print ").\\n";}'? .
          qq ? > ~/tmp/t1.pl?;
   system($cmd) and die " Cannot convert file: $_"; # system(success)=0
   system("time $p_cmd") and die " Prolog not running: $_";
}
```

tok.perl is a little Perl script that tokenises a Lisp-like DL expression and does some sanitation on it so it can be properly parsed into a Prolog term by the grammar predicates (cf. App. E). All tokens are converted to lower case, and where necessary quoted.

```
#!/usr/bin/perl
#####################################
# tok.perl                          #
#####################################
# tokenise DL-Benchmark-T98-sat single queries
# maybe run:  perl -pe 's/\n/ /g;'  before (get rid of multiple lines)

while(<>) {
  @F=split(/(?:([()])|\s+)/);
  $L="[";
  for $i (@F){
      $i = lc($i);
      $i =~ /^\s*$/ and next;
      $i =~ /command=/i and next;
      $i =~ /^([()])$/ and do {$i = "\'$1\'";};
      $i =~ /-/ and $i = "\'$i\'";
      $i =~ /\*(?:bottom|top)\*/ and $i = "\'$i\'";
      $L .= "$i, ";
```

```
      }
  chop $L; chop $L;
  $L .= "]";
  print "$L\n";
}
```

Bibliography

[1] F. C. N. Pereira and S. M. Shieber, *Prolog and Natural Language Analysis*. Stanford: CSLI Publications, 1987.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, eds., *The Description Logic Handbook*. Cambridge, UK: Cambridge University Press, 2003.

[3] S. Aitken, "Knowledge Modelling and Management. Part A 'Ontologies'." Lecture Notes, 2006. URL http://www.inf.ed.ac.uk/teaching/courses/kmm/lectures.html.

[4] F. Baader and W. Nutt, "Basic Description Logics," in *The Description Logic Handbook*, pp. 43–95, Cambridge University Press, 2003.

[5] I. Horrocks, *Optimising Tableaux Decision Procedures For Description Logics*. PhD thesis, University of Manchester, 1997.

[6] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "OWL Web Ontology Language Reference." W3C Recommendation, 2004. URL http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[7] I. Horrocks, P. Patel-Schneider, and F. van Harmelen, "From SHIQ and RDF to OWL: The making of a web ontology language," *Journal of Web Semantics*, vol. 1(1), pp. 7–26, 2003.

[8] F. Baader and U. Sattler, "An overview of tableau algorithms for description logics," *Studia Logica*, vol. 69, no. 1, pp. 5 – 40, 2001/10/.

[9] F. Baader, J. Hladik, C. Lutz, and F. Wolter, "From Tableaux to Automata for Description Logics," *Fundamenta Informaticae*, vol. 57, no. 2-4, pp. 247 – 279, 2003.

[10] F. Baader and U. Sattler, “Tableau Algorithms for Description Logics,” in *Proceedings of the International Conference on Automated Reasoning with Tableaux and Related Methods (Tableaux 2000)* (R. Dyckhoff, ed.), vol. 1847, pp. 1–18, Springer, 2000.

[11] M. Schmidt-Schauss and G. Smolka, “Attributive concept descriptions with complements,” *Artificial Intelligence*, vol. 48, no. 1, pp. 1 – 26, 1991/02/.

[12] V. Haarslev and R. Möller, “RACER system description,” in *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR 2001)* (R. Goré, A. Leitsch, and T. Nipkow, eds.), no. 2083 in Lecture Notes in Computer Science, pp. 701–705, Springer-Verlag.

[13] I. Horrocks, “The FaCT system,” *Automated Reasoning with Analytic Tableaux and Related Methods. International Conference, TABLEAUX’98. Proceedings*, pp. 307 – 12, 1998.

[14] B. Beckert and J. Posegga, “leanTAP: Lean tableau-based deduction,” *Journal of Automated Reasoning*, vol. 15, no. 3, pp. 339 – 358, 1995.

[15] J. Wielemaker, “An Overview of the SWI-Prolog Programming Environment,” in *Proceedings of the 13th International Workshop on Logic Programming Environments* (F. Mesnard and A. Serebrenik, eds.), vol. CW371 of *Report*, pp. 1–16, Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee (Belgium), 2003.

[16] A. Meissner, “An automated deduction system for description logic with ALCN language,” *Studia z Automatyki i Informatyki*, vol. 28-29, pp. 91 – 110, 2004//.

[17] A. Meissner, “lpdl.pl.” A Prolog implementation of Tableaux based on [16]. unpublished, 2006.

[18] S. Aitken, “fact.pl.” A Prolog implementation of Tableaux. unpublished, 2006.

[19] I. Horrocks, “Reasoning with Expressive Description Logics: Theory and Practice,” in *Automated Deduction - CADE-18. 18th International Conference on Automated Deduction. Proceedings* (A. Voronkov, ed.), vol. 2392 of *Lecture Notes in Artificial Intelligence*, pp. 1–15, Springer, 2002.

[20] S. J. Russell and P. Norvig, *Artificial Intelligence. A Modern Approach*. Upper Saddle River, NJ: Pearson Education, 2nd ed., 2003.

[21] I. Horrocks and P. F. Patel-Schneider, "DL Systems Comparison (Summary Relation)," in *Proceedings of the 1998 International Workshop on Description Logics (DL'98), IRST, Povo - Trento, Italy, June 6-8, 1998* (E. Franconi, G. D. Giacomo, R. M. MacGregor, W. Nutt, and C. A. Welty, eds.), vol. 11 of *CEUR Workshop Proceedings*, CEUR-WS.org, 1998.

[22] P. Balsiger, A. Heuerding, and S. Schwendimann, "A benchmark method for the propositional modal logics K, KT, S4," *Journal of Automated Reasoning*, vol. 24, no. 3, pp. 297 – 317, 2000/04/.

[23] P. Patel-Schneider and B. Swartout, "Description Logic Knowledge Representation System Specification from the KRSS Group of the ARPA Knowledge Sharing Effort," tech. rep., AT&T Bell Laboratories, 1993. URL: http://www-db.research.bell-labs.com/user/pfps/publications/krss-spec.pdf.

[24] H. Boley, "The Rule Markup Language: RDF-XML data model, XML schema hierarchy, and XSL transformations," *Web Knowledge Management and Decision Support. 14th International Conference on Applications of Prolog, INAP 2001. Revised Papers (Lecture Notes in Artificial Intelligence Vol.2543)*, pp. 5 – 22, 2003//.

[25] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, "Description logic programs: combining logic programs with description logic," in *WWW '03: Proceedings of the 12th international conference on World Wide Web*, (New York, NY, USA), pp. 48–57, ACM Press, 2003.

[26] A. Felty and D. Miller, "Proof Explanation and Revision," tech. rep., University of Pennsylvania, 1987. URL http://www.site.uottawa.ca/ afelty/dist/proof87.ps.

[27] K. Sagonas, T. Swift, and D. Warren, "XSB as an efficient deductive database engine," *SIGMOD Record*, vol. 23, no. 2, pp. 442 – 453, 1994/06/.

[28] K. Konrad, "HOT: A Concurrent Automated Theorem Prover based on Higher-Order Tableaux," *Theorem Proving in Higher Order Logics. 11th International Conference, TPHOLs '98. Proceedings*, pp. 245 – 261, 1998.

[29] J. Cunningham, “Concurrent Tableaux.” First COMPULOG NET Workshop on Parallelism and Implementation Technologies; Madrid May 24–25, 1993. URL: http://clip.dia.fi.upm.es/Projects/COMPULOG/meeting93/c_tableaux.ps.Z.

[30] S. Bechhofer, R. Möller, and P. Crowther, “The DIG Description Logic Interface,” in *Proceedings of the 2003 International Workshop on Description Logics (DL2003)* (D. Calvanese, G. D. Giacomo, and E. Franconi, eds.), vol. 81, CEUR-WS.org, 2003. URL: http://ceur-ws.org/Vol-81/bechhofer.ps.

[31] Z. Huang and C. Visser, “Extended DIG Description Logic Interface Support for Prolog. SEKT Deliverable D3.4.1.2,” tech. rep., Free University of Amsterdam, as part of the SEKT project (www.sect-project.org), 2004. URL: http://wasp.cs.vu.nl/∼huang/papers/dig.pdf.