Artificial Intelligence Institute
Department of Computer Science
Dresden University of Technology

Diploma Thesis

Concept Learning in Description Logics

Jens Lehmann

September 12, 2006

Supervisors: Prof. Dr. Steffen Hölldobler
Dr. habil. Pascal Hitzler, AIFB, Univ. of Karlsruhe

Abstract

The problem of learning logic programs has been researched extensively, but other knowledge representations formalisms like Description Logics are also an interesting target language. The importance of inductive reasoning in Description Logics has increased with the rise of the Semantic Web, because the learning algorithms can be used as a means for the computer aided building of ontologies. Ontology construction is a burdensome task and powerful tools are needed to support knowledge engineers.

The thesis focuses on learning $\mathcal{ALC}$ concept definitions, although many ideas apply to concept learning in general. It deeply researches the properties of $\mathcal{ALC}$ refinement operators, which are an efficient way to traverse the space of concepts ordered by subsumption. We give a full theoretical analysis of interesting properties of such operators. Based on this analysis, we propose a suitable concrete refinement operator and research its properties. We show that it is not possible to define better operators with respect to the properties we are investigating and establish a complete learning algorithm by adding an intelligent search heuristic.

As a second approach we investigate the use of Genetic Programming to solve the learning problem in Description Logics. We discuss the characteristics of Genetic Programming in this context and show a way to incorporate refinement operators in the Genetic Programming framework. Again, we define a suitable operator and analyse it. Some further extensions like learning from uncertain data and concept invention are also proposed.

Besides the analysis of the two learning approaches mentioned above, we will also briefly investigate current problems in evaluating concepts and describe possible solutions.

Contents

1 Introduction

The field of Machine Learning (Mitchell, 1997) is an important area of Artificial Intelligence (Russel and Norvig, 2003). Since logic is an important foundation of Artificial Intelligence, it is natural that learning logic programs (also called *induction*) plays a significant role in Machine Learning. While the induction of logic programs has been studied extensively in Inductive Logic Programming (ILP) (Nienhuys-Cheng and de Wolf, 1997) other knowledge representation languages also deserve attention. Description Logics (Baader et al., 2003) are an interesting fragment of first order logic with decidable inference methods. They have an easy to understand and readable syntax without function symbols and variables. Horn logics and Description Logics are incomparable (none includes the other as a fragment), which is another argument for considering learning in Description Logics worthwhile. Description Logics are also used for the Semantic Web (Berners-Lee et al., 2001) as underlying knowledge representation formalism for ontologies. The World Wide Web Consortium and many KRR (knowledge representation and reasoning) research groups consider it the most suitable choice for representing machine processable knowledge in the web. The Web Ontology Language (OWL), which is used to formally define terminologies, is based on Description Logics.

When we investigate learning methods in Description Logics we have two application areas in mind:

First, we can consider classical ILP application areas (see Bratko and Muggleton, 1995). There is a variety of existing ILP systems and ILP has been applied successfully in many areas, e.g. learning drug-structure activity rules for Alzheimer's disease, predicting mutagenesis, and many more. While Description Logics are different from logic programs, it may be the case that for some of the tasks, where ILP is used, Description Logics are also suited. For some tasks it may even be a better target language. (The used target language is an important bias of a learning algorithm, so the performance of a learning algorithm greatly depends on whether the target language is appropriate.)

The second application area of our work is ontology engineering in the Semantic Web. A lot of research effort has been spent to develop knowledge representation formalisms for the Semantic Web and standardise them. However, the creation of ontolgies is still a bottleneck. Building up an ontology is a difficult and error-prone task, so there is currently a great need for tool support in this area. Our approach helps to overcome this problem by automatically inducing concept definitions (concepts are called classes in the OWL context) from examples. This means that the knowledge engineer can select positive and negative examples for the class he wants to define and will automatically receive a correct class definition. The first advantage of this approach is that the obtained definition is guaranteed not to contradict other knowledge in the ontology. Furthermore the induced definition is likely to cover all important aspects of the class, which the knowledge engineer may have missed or does not know about.

Learning in Description Logics has attracted some attention during the 90s, e.g. (Cohen and Hirsh, 1994). It has recently gained momentum (Esposito et al., 2004) due to the rise of the Semantic Web. So far only a few learning systems exist in practise with YINYANG being the most recent one (see Iannone and Palmisano, 2005, for the theory underlying the YINYANG system) .

Refinement operators have been shown to be sucessful in Inductive Logic Programming, so a large part of the thesis will be devoted to a thorough analysis of such operators. For the description language $\mathcal{ALER}$ some theoretical issues have been researched in (Badea and Nienhuys-Cheng, 2000). In this thesis we will give a full analysis of interesting properties of refinement operators for the description language $\mathcal{ALC}$. To the best of our knowledge, such a complete analysis has not been done before for a description language. Using this theoretical analysis as a starting point, we design a new refinement operator for $\mathcal{ALC}$ and extend this step-by-step to a complete learning algorithm. We believe that the algorithm we invent offers a number of advantages over existing approaches. Later in the thesis, we will introduce the novel idea of using Genetic Programming (see Koza et al., 2003, for a general overview) for concept learning. In particular, we will show how to usefully combine refinement operators and genetic programming. We think that these contributions will be useful for practical applications and the future research in the field of learning in Description Logics.

Outline In Section 2 we will give a brief introduction in Description Logics. Of course, it is impossible to cover all important aspects of Description Logics, but we will introduce all necessary notions, which are needed to understand the content of the thesis. We will also show the connection between Description Logics and ontology languages, in particular OWL. Section 3 introduces the learning problem in Description Logics. In this section we will describe the goal we want to achieve and give an overview of a general framework for solving it. We propose two solution approaches in Section 4 and 5.

In Section 4 we will introduce the notion of refinement operators and define interesting properties of such operators like completeness, weak completeness, properness, non-redundancy, and finiteness. After that we do a full analysis of these properties, i.e. we look which of these properties can be combined and which properties are impossible to combine. This serves as a useful guide for designing concrete operators. We will design such an operator and show that it is complete. Then we will extend it to a complete and proper operator. We will give reasons why weakly complete, proper, and non-redundant operators are impractical (although they exist). We show that redundancy can be avoided for a complete and proper operator by using a redundancy-eliminating heuristic. We show how such a heuristic works and which steps have to be done to implement it efficiently. After mentioning further optimisations, we formally define a full learning algorithm, prove its correctness and analyse its characteristics. Finally an example run is presented.

Section 5 starts with an overview of Evolutionary Computing and then presents Genetic Programming as an evolutionary algorithm in more detail. It is more appropriate to view Genetic Programming as a framework instead of looking at it as a specific algorithm. What we introduce first is the standard approach, which is widely used. We then discuss the characteristics of this approach and conclude that modifications are necessary to create an efficient learning algorithm. As main modification we introduce so called genetic refinement operators, which provide a way to use refinement operators as genetic operators. We show how this works in general for all finite refinement operators and then define a concrete refinement operator, which is suitable for learning within the Genetic Programming framework. In this section we will also show how the algorithm can handle noise, learn from uncertain data and invent new helper concepts.

Section 6 looks at the quality measurement of concepts. Quality measurement means to measure how good a concept is, i.e. to compute which examples are covered (follow logically) by a concept. We show that the Open World Assumption in Description Logics can be a problem in learning. To solve this problem, we propose a simple retrieval algorithm, which can reason under a fixed domain (so the Open World Assumption is weakenend) and is sound and incomplete. We show that with some restrictions the algorithm is also complete and we give reasons why it makes sense to use the algorithm within the learning framework.

Finally in Section 7, we summarise the results of the thesis and look at future work and research directions.

2 Description Logics and Ontology Languages

Description Logics is the name of a family of knowledge representation (KR) formalisms. They emerged from earlier KR formalisms like semantic networks and frames. The origin is a work of Bachman on structured inheritance networks (Brachman, 1978). Since then Description Logics have enjoyed increasing popularity. They can essentially be understood as fragments of first order predicate logic. They have less expressive power, but usually decidable inference problems and a user-friendly variable free syntax.

Description Logics represent knowledge in terms of *objects*, *concepts*, and *roles*. Concepts formally describe notions in an application domain, e.g. we could define the concept of being a father as "a man having a child". Objects are members of entities in the application domain and roles are binary relations between objects. Objects correspond to constants, concepts to unary predicates, and roles to binary predicates in first order logic.

In Description Logic systems information is stored in a *knowledge base*. It is divided in two parts: *TBox* and *ABox*. The ABox contains *assertions* about objects. It relates objects to concepts and roles. The TBox describes the *terminology* by relating concepts and roles. (For some expressive description logics this clear separation does not exist.)

As mentioned before, DLs are a family of KR formalisms. In order to simplify the introduction we will introduce the $\mathcal{ALC}$ Description Logic as a prototypical example and then briefly describe other description logics.

2.1 The Description Language $\mathcal{ALC}$

$\mathcal{ALC}$ stands for *attribute language with complement*. It allows to construct complex concepts from simpler ones using various language constructs. The next definition shows how such concepts can be build.

Definition 2.1 (syntax of $\mathcal{ALC}$ concepts)
Let N_R be a set of *role names* and N_C be a set of concept names ($N_R \cap N_C = \emptyset$). The elements of the latter set are called *atomic concepts*. The set of $\mathcal{ALC}$ concepts is inductively defined as follows:

1. Each atomic concept is a concept.
2. If C and D are $\mathcal{ALC}$ concepts and $r \in N_R$ a role, then the following are also $\mathcal{ALC}$ concepts:
   - $\top$ (top), $\bot$ (bottom)
   - $C \sqcup D$ (disjunction), $C \sqcap D$ (conjunction), $\neg C$ (negation)
   - $\forall r.C$ (value/universal restriction), $\exists r.C$ (existential restriction) □

The following rule is a more succinct description of $\mathcal{ALC}$ syntax:

$$C, D \rightarrow A \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \forall r.C \mid \exists r.C$$

We now know how to syntactically build $\mathcal{ALC}$ concepts. We still have to define their semantics. As usual in logic this is done by interpretations.

Definition 2.2 (interpretation)
An *interpretation* $\mathcal{I}$ consists of a non-empty *interpretation domain* $\Delta^\mathcal{I}$ and an *interpretation function* $\cdot^\mathcal{I}$, which assigns to each $A \in N_C$ a set $A^\mathcal{I} \subseteq \Delta^\mathcal{I}$ and to each $r \in N_R$ a binary relation $r^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$. □

As we can see from Definition 2.2 atomic concepts are interpreted as sets of objects and roles are interpreted as binary relations. We will now extend the interpretation function to complex $\mathcal{ALC}$ terms. This is done as shown in Table 1. Given an interpretation this allows us to interpret any $\mathcal{ALC}$ concept.

construct	syntax	semantics
atomic concept	A	$A^\mathcal{I} \subseteq \Delta^\mathcal{I}$
role	r	$r^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$
top concept	$\top$	$\Delta^\mathcal{I}$
bottom concept	$\bot$	$\emptyset$
conjunction	$C \sqcap D$	$(C \sqcap D)^\mathcal{I} = C^\mathcal{I} \cap D^\mathcal{I}$
disjunction	$C \sqcup D$	$(C \sqcup D)^\mathcal{I} = C^\mathcal{I} \cup D^\mathcal{I}$
negation	$\neg C$	$(\neg C)^\mathcal{I} = \Delta^\mathcal{I} \setminus C^\mathcal{I}$
exists restriction	$\exists r.C$	$(\exists r.C)^\mathcal{I} = \{a \mid \exists b.(a,b) \in r^\mathcal{I} \text{ and } b \in C^\mathcal{I}\}$
value restriction	$\forall r.C$	$(\forall r.C)^\mathcal{I} = \{a \mid \forall b.(a,b) \in r^\mathcal{I} \text{ implies } b \in C^\mathcal{I}\}$

Table 1: $\mathcal{ALC}$ semantics

Example 2.3 (interpreting concepts)
Let the interpretation $\mathcal{I}$ be given by:

$$\begin{aligned}\Delta^\mathcal{I} &= \{\texttt{MONICA}, \texttt{JESSICA}, \texttt{STEPHEN}\}\\ \texttt{Woman}^\mathcal{I} &= \{\texttt{MONICA}, \texttt{JESSICA}\}\\ \texttt{hasChild}^\mathcal{I} &= \{(\texttt{MONICA}, \texttt{STEPHEN}), (\texttt{STEPHEN}, \texttt{JESSICA})\}\end{aligned}$$

We then have:

$$(\texttt{Woman} \sqcap \exists \texttt{hasChild}.\top)^\mathcal{I} = \{\texttt{MONICA}\}$$ □

Previously we mentioned that a DL knowledge base consists of TBox and ABox. We will first introduce TBoxes and then ABoxes.

Definition 2.4 (terminological axiom)
If C and D are concepts, then $C \sqsubseteq D$ and $C \equiv D$ are *terminological axioms.* The former axioms are called *inclusions* and the latter *equalities.* □

We can define the semantics of terminological axioms in a straightforward way. An interpretation $\mathcal{I}$ satisfies an inclusion $C \sqsubseteq D$ if $C^\mathcal{I} \subseteq D^\mathcal{I}$ and it satisfies the equality $C \equiv D$ if $C^\mathcal{I} = D^\mathcal{I}$. $\mathcal{I}$ satisfies a set of terminological axioms if it satisfies all axioms in the set.

Definition 2.5 (model of terminological axioms, equivalence)
An interpretation, which satisfies a (set of) terminological axioms is called a *model* of this (set of) axioms. Two (sets of) axioms are *equivalent* if they have the same models. □

Definition 2.6 (concept definition)
An equality whose left hand side is an atomic concept is a *concept definition.* □

Definition 2.7 (TBox)
A finite set $\mathcal{T}$ of concept definitions is called *TBox* if it does not contain multiple definitions (a concept name does not occur more than once on the left hand side). □

Often a different definition of a TBox is used, which is sometimes called general TBox. In this thesis we refer to general TBoxes unless mentioned otherwise.

Definition 2.8 ((general) TBox)
A finite set $\mathcal{T}$ of terminological axioms is called a *(general) TBox.* □

TBoxes are used to describe the relationship between concepts. The next step is to look at ABoxes, which relate objects to concepts and roles.

Definition 2.9 (assertion)
Let N_I be the set of object names (disjoint with N_R and N_C). An *assertion* has the form $C(a)$ (*concept assertion*) or $r(a, b)$ (*role assertion*), where a, b are object names, C is a concept, and r is a role. □

Definition 2.10 (ABox)
An *ABox* $\mathcal{A}$ is a finite set of assertions. □

Objects are also called individuals. To allow interpreting ABoxes we extend the definition of an interpretation. Additionally to mapping concepts to subsets of our domain and roles to binary relations, an interpretation has to assign to each individual name $a \in N_I$ an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. In particular different individual names must not be mapped to the same domain element. This is required in order to satisfy the *unique names assumption* (UNA), which states that individuals with different names have to be interpreted as different individuals.

Definition 2.11 (model of an ABox)
An interpretation $\mathcal{I}$ is a model of an ABox $\mathcal{A}$ (written $\mathcal{I} \models \mathcal{A}$) if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all $C(a) \in \mathcal{A}$ and $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ for all $r(a, b) \in \mathcal{A}$. □

Definition 2.12 (model of a knowledge base)
An interpretation $\mathcal{I}$ is a model of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ (written $\mathcal{I} \models \mathcal{K}$) iff it is a model of $\mathcal{T}$ and $\mathcal{A}$. □

Example 2.13 (models of a knowledgebase)
We have introduced basic notions relating to DL knowledge bases and interpretations. An example is in order. Let the knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be given by:

TBox $\mathcal{T}$:

$$\begin{aligned}\texttt{Man} &\equiv \neg\texttt{Woman}\\ \texttt{Mother} &\equiv \texttt{Woman} \sqcap \exists\texttt{hasChild}.\top\end{aligned}$$

ABox $\mathcal{A}$:

$$\begin{aligned}&\texttt{Man}(\texttt{STEPHEN}).\\ &\neg\texttt{Man}(\texttt{MONICA}).\\ &\texttt{Woman}(\texttt{JESSICA}).\\ &\texttt{hasChild}(\texttt{STEPHEN}, \texttt{JESSICA}).\end{aligned}$$

We will now look at some interpretations and determine whether or not they are a model of $\mathcal{K}$. For all interpretations we use the domain $\{\texttt{MONICA}, \texttt{JESSICA}, \texttt{STEPHEN}\}$ and all object names are interpreted in the obvious way (STEPHEN is interpreted as STEPHEN etc.).

Let the interpretation $\mathcal{I}_1$ be given by:

$$\begin{aligned}\texttt{Man}^{\mathcal{I}_1} &= \{\texttt{JESSICA}, \texttt{STEPHEN}\}\\ \texttt{Woman}^{\mathcal{I}_1} &= \{\texttt{MONICA}, \texttt{JESSICA}\}\\ \texttt{Mother}^{\mathcal{I}_1} &= \emptyset\\ \texttt{hasChild}^{\mathcal{I}_1} &= \{(\texttt{STEPHEN}, \texttt{JESSICA})\}\end{aligned}$$

Clearly this does not satisfy $\mathcal{T}$, because the definition $\texttt{Man} \equiv \neg\texttt{Woman}$ is not satisfied. We have $\texttt{Man}^{\mathcal{I}_1} = \{\texttt{JESSICA}, \texttt{STEPHEN}\}$ and $(\neg\texttt{Woman})^{\mathcal{I}_1} = \{\texttt{STEPHEN}\}$, which are not equal. However, $\mathcal{I}_1$ satisfies $\mathcal{A}$.

Let the interpretation $\mathcal{I}_2$ be given by:

$$\begin{aligned}\texttt{Man}^{\mathcal{I}_2} &= \{\texttt{STEPHEN}\}\\ \texttt{Woman}^{\mathcal{I}_2} &= \{\texttt{JESSICA}, \texttt{MONICA}\}\\ \texttt{Mother}^{\mathcal{I}_2} &= \emptyset\\ \texttt{hasChild}^{\mathcal{I}_2} &= \emptyset\end{aligned}$$

$\mathcal{I}_2$ satisfies $\mathcal{T}$, but does not satisfy $\mathcal{A}$, because we have $\texttt{hasChild}(\texttt{STEPHEN}, \texttt{JESSICA}) \in \mathcal{A}$ but $(\texttt{STEPHEN}^{\mathcal{I}_2}, \texttt{JESSICA}^{\mathcal{I}_2}) \notin \texttt{hasChild}^{\mathcal{I}_2}$.

Let the interpretation $\mathcal{I}_3$ be given by:

$$\begin{aligned}\texttt{Man}^{\mathcal{I}_3} &= \{\texttt{STEPHEN}\}\\ \texttt{Woman}^{\mathcal{I}_3} &= \{\texttt{JESSICA}, \texttt{MONICA}\}\\ \texttt{Mother}^{\mathcal{I}_3} &= \{\texttt{MONICA}\}\\ \texttt{hasChild}^{\mathcal{I}_3} &= \{(\texttt{MONICA}, \texttt{STEPHEN}), (\texttt{STEPHEN}, \texttt{JESSICA})\}\end{aligned}$$

$\mathcal{I}_3$ is a model of $\mathcal{T}$ and $\mathcal{A}$, so it is a model of $\mathcal{K}$. One may argue that nothing in our knowledgebase justifies the fact that we interpret MONICA as mother. However, in DLs we usually have the *open world assumption.* This means that the given knowledge is viewed as incomplete. There is nothing, which tells us that MONICA is not a mother. In databases one usually uses the *closed world assumption*, i.e. all facts, which are not explicitly stored, are assumed to be false. □

2.2 Reasoning in Description Logics

As we have seen a knowledge base can be used to store the information we have about the application domain. Besides this *explicit* knowledge we can also deduce *implicit* knowledge from a knowledge base, e.g. from the fact that JESSICA belongs to the concept Woman and the axiom $\texttt{Man} \equiv \neg\texttt{Woman}$ we can deduce that JESSICA does not belong to the concept Man. It is the aim of *inference algorithms* to extract such implicit knowledge. There are some standard reasoning tasks in Description Logics, which we will briefly describe.

In *terminological reasoning* we reason about concepts. The standard problems are *satisfiability* and *subsumption.* Intuitively satisfiability determines if a concept can be satisified, i.e. it is free of contradictions. Subsumption of two concepts detects if one of the concepts is more general than the other.

Definition 2.14 (satisfiability)
Let C be a concept and $\mathcal{T}$ a TBox. C is *satisfiable* iff there is an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$. C is *satisfiable with respect to* $\mathcal{T}$ iff there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$. □

Example 2.15 (satisfiability)
$\texttt{Male} \sqcap \texttt{Female}$ is satisfiable. However, it is not satisfiable with respect to the TBox in Example 2.13. □

Definition 2.16 (subsumption, equivalence)
Let C, D be concepts and $\mathcal{T}$ a TBox. C *is subsumed by* D, denoted by $C \sqsubseteq D$, iff for any interpretation $\mathcal{I}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. C *is subsumed by* D *with respect to* $\mathcal{T}$ (denoted by $C \sqsubseteq_{\mathcal{T}} D$) iff for any model $\mathcal{I}$ of $\mathcal{T}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

C *is equivalent to* D *(with respect to* $\mathcal{T}$*)*, denoted by $C \equiv D$ ($C \equiv_{\mathcal{T}} D$), iff $C \sqsubseteq D$ ($C \sqsubseteq_{\mathcal{T}} D$) and $D \sqsubseteq C$ ($D \sqsubseteq_{\mathcal{T}} C$).

C *is strictly subsumed by* D *(with respect to* $\mathcal{T}$*)*, denoted by $C \sqsubset D$ ($C \sqsubset_{\mathcal{T}} D$), iff $C \sqsubseteq D$ ($C \sqsubseteq_{\mathcal{T}} D$) and not $C \equiv D$ ($C \equiv_{\mathcal{T}} D$). □

Example 2.17 (subsumption)
Mother is not subsumed by Woman. However, Mother is subsumed by Woman with respect to the TBox in Example 2.13. □

In *assertional reasoning* we reason about objects. The *consistency problem* is the question whether an ABox has a model. The *instance problem* is to find out whether an object is an instance of a concept, i.e. belongs to it. *Retrieval* is the problem of finding all instances of a given concept.

Definition 2.18 (consistency)
An Abox $\mathcal{A}$ is *consistent* iff it has a model. An ABox A is *consistent with respect to a TBox* $\mathcal{T}$ iff $\mathcal{A}$ and $\mathcal{T}$ have a common model. $\square$

Example 2.19 (consistency)
The ABox $\{\texttt{Male} \sqcap \neg\texttt{Male}(\texttt{STEPHEN})\}$ is inconsistent. $\square$

Definition 2.20 (instance)
Let $\mathcal{A}$ be an ABox, $\mathcal{T}$ a TBox, $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a knowledge base, C a concept, and $a \in N_I$ an object. *a is an instance of C with respect to $\mathcal{A}$* , denoted by $\mathcal{A} \models C(a)$, iff in any model $\mathcal{I}$ of $\mathcal{A}$ we have $a^{\mathcal{I}} \in C^{\mathcal{I}}$. *$a$ is an instance of C with respect to $\mathcal{K}$* , denoted by $\mathcal{K} \models C(a)$, iff in any model $\mathcal{I}$ of $\mathcal{K}$ we have $a^{\mathcal{I}} \in C^{\mathcal{I}}$.

To denote that a is not an instance of C with respect to $\mathcal{A}$ ($\mathcal{K}$) we write $\mathcal{A} \not\models C(a)$ ($\mathcal{K} \not\models C(a)$). $\square$

Definition 2.21 (retrieval)
Let $\mathcal{A}$ be an ABox, $\mathcal{T}$ a TBox, $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a knowledge base, C a concept. The *retrieval $R_{\mathcal{A}}(C)$ of a concept C with respect to $\mathcal{A}$* is the set of all instances of C: $R_{\mathcal{A}}(C) = \{a \mid a \in N_I \text{ and } \mathcal{A} \models C(a)\}$. Similarly the *retrieval $R_{\mathcal{A}}(C)$ of a concept C with respect to $\mathcal{K}$* is: $R_{\mathcal{K}}(C) = \{a \mid a \in N_I \text{ and } \mathcal{K} \models C(a)\}$ $\square$

Example 2.22 (instance, retrieval)
In Example 2.13 we have $R_{\mathcal{K}}(\texttt{Woman}) = \{\texttt{JESSICA}, \texttt{MONICA}\}$. JESSICA and MONICA are instances of Woman, because in any model $\mathcal{I}$ of $\mathcal{K}$ we have $\texttt{JESSICA}^{\mathcal{I}} \in \texttt{Woman}^{\mathcal{I}}$ and $\texttt{MONICA}^{\mathcal{I}} \in \texttt{Woman}^{\mathcal{I}}$. $\square$

2.3 Normal Forms

In this section we will introduce normal forms of $\mathcal{ALC}$ concepts, which will be of interest later on. A normal form of a given concept is an equivalent concept, i.e. it has the same meaning, but differs in its syntactical representation. The most well known normal form of $\mathcal{ALC}$ concepts is the negation normal form.

Definition 2.23 (negation normal form)
An $\mathcal{ALC}$ concept is in *negation normal form* if negation only occurs in front of concept names. $\square$

It is usually the case that we can produce a normal form of an arbitrary concept with the help of rewriting rules. An $\mathcal{ALC}$ normal form of a concept can be obtained by using the following rewrite rules (which have to be applied exhaustively):

$$
\begin{aligned}
\neg\bot &\rightarrow \top \\
\neg\top &\rightarrow \bot \\
\neg\neg C &\rightarrow C \\
\neg(C \sqcup D) &\rightarrow \neg C \sqcap \neg D \\
\neg(C \sqcap D) &\rightarrow \neg C \sqcup \neg D \\
\neg(\forall r.C) &\rightarrow \exists r.\neg C \\
\neg(\exists r.C) &\rightarrow \forall r.\neg C
\end{aligned}
$$

Example 2.24 (negation normal form)
The negation normal form of the concept $\neg(\exists r.\neg A \sqcap (A \sqcup \forall r.A))$ is $\forall r.A \sqcup (\neg A \sqcap \exists r.\neg A)$.□

We can also define concepts in negation normal form inductively. In contrast to the definition of $\mathcal{ALC}$ concepts (Definition 2.1, page 8) we will omit parentheses in the case of nested disjunctions and conjunctions, e.g. $((C_1 \sqcap C_2) \sqcap C_3) \sqcap C_4$ is written as $C_1 \sqcap C_2 \sqcap C_3 \sqcap C_4$.

Definition 2.25 (inductive definition of negation normal form)
The set of $\mathcal{ALC}$ concepts in negation normal form is inductively defined as follows:

1. If $A \in N_C$ then A and $\neg A$ are $\mathcal{ALC}$ concepts in negation normal form.

2. If $C, C_1, \ldots C_n$ are $\mathcal{ALC}$ concepts in negation normal form and $r \in N_R$, then the following are also $\mathcal{ALC}$ concepts in negation normal form:

   - $\top$
   - $\bot$
   - $C_1 \sqcap \cdots \sqcap C_n$
   - $C_1 \sqcup \cdots \sqcup C_n$
   - $\forall r.C$
   - $\exists r.C$ □

Definitions 2.23 and 2.25 are equivalent. Obviously every concept, which satisifies Definition 2.25, has the negation symbol only in front of atomic concepts. And conversely a concept, which satisfies Definition 2.23, can be build like in Definition 2.25, because it corresponds to the definition of $\mathcal{ALC}$ concepts with the exception that negation is only allowed in front of atomic concepts.

A more restricted version than negation normal form is $\mathcal{ALC}$ normal form. $\mathcal{ALC}$ normal form is not as widely used as negation normal form. It has been defined in a similar way like we will do it here in (Brandt et al., 2002).

Definition 2.26 ($\mathcal{ALC}$ normal form)
A concept C is in $\mathcal{ALC}$ *normal* form iff $C \equiv \top$ or $C \equiv \bot$ or $C = C_1 \sqcup \cdots \sqcup C_n$ with

$$C_i = \bigsqcap_{A \in \text{pos}(C_i)} A \sqcap \bigsqcap_{A \in \text{neg}(C_i)} \neg A \sqcap \bigsqcap_{r \in N_R} \left(\bigsqcap_{C' \in \text{ex}_r(C_i)} \exists r.C' \sqcap \forall r.\text{val}_r(C_i) \right)$$

with

- pos(C) is the set of all atomic concepts occuring on the top level conjunction of C
- neg(C) is the set of all negated atomic concepts occuring on the top level conjunction of C
- val$_r$(C) is

  C' if there exists a value restriction of the form $\forall r.C'$ on the top level conjunction of C (due to the equivalence $\forall r.(C_1 \sqcap C_2) \equiv \forall r.C_1 \sqcap \forall r.C_2$ we can assume without loss of generality that only one such value restriction exists on the top level conjunction of C)

  $\top$ otherwise

- ex$_r$(C) $= \{C' \mid$ there exists $\exists r.C'$ on the top level conunction of $C\}$ □

An $\mathcal{ALC}$ normal form of a given $\mathcal{ALC}$ concept can be reached by moving negations inside, placing disjunctions over conjunctions, e.g. $(A_1 \sqcup A_2) \sqcap A_3 \equiv (A_1 \sqcap A_3) \sqcup (A_2 \sqcap A_3)$, and by using the equality $\forall r.(C_1 \sqcap C_2) \equiv \forall r.C_1 \sqcap \forall r.C_2$.

Unfortunately a transformation of a concept to $\mathcal{ALC}$ normal form can take exponential time (Brandt et al., 2002). The size of the $\mathcal{ALC}$ normal form of the concept $(A_1 \sqcup A_2) \sqcap \cdots \sqcap (A_{2n-1} \sqcup A_{2n})$ is exponential (with respect to n). In contrast the negation normal form of a given concept C can at most add a negation symbol in front of all concept names in C, so this transformation at most doubles the length (later we will define the notion of the length of a concept formally).

2.4 Other Description Languages

In the area of Description Logics a variety of different Description Languages has evolved. In principle many of the notions and ideas we introduced for $\mathcal{ALC}$ can be transfered to other Description Languages.

While first order predicate logic is undecidable Description Logics are usually decidable fragments of first order logic. Description Languages mainly differ in their expressive power and syntactic structures.

There is a naming scheme for Description Logics. As mentioned before $\mathcal{ALC}$ stands for *attribute language with complement.* The letters in the name of a Description Language describe its syntactic constructs. Some of these letters and their corresponding (informal) meaning are:

construct	syntax	semantics
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
role	r	$r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
nominals	$\{o\}$	$\{o\}^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}, \lvert\{o\}\rvert^{\mathcal{I}} = 1$
top concept	$\top$	$\Delta^{\mathcal{I}}$
bottom concept	$\bot$	$\emptyset$
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
negation	$\neg C$	$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
exists restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} = \{a \mid \exists b.(a,b) \in r^{\mathcal{I}} \text{ and } b \in C^{\mathcal{I}}\}$
value restriction	$\forall r.C$	$(\forall r.C)^{\mathcal{I}} = \{a \mid \forall b.(a,b) \in r^{\mathcal{I}} \text{ implies } b \in C^{\mathcal{I}}\}$
atleast restriction	$\geq n\ r.C$	$(\geq n\ r)^{\mathcal{I}} = \{a \mid \lvert(\{b \mid (a,b) \in r^{\mathcal{I}}\}\rvert \geq n\}$
atmost restriction	$\leq n\ r.C$	$(\leq n\ r)^{\mathcal{I}} = \{a \mid \lvert(\{b \mid (a,b) \in r^{\mathcal{I}}\}\rvert \leq n\}$

Table 2: syntax and semantics for concepts in $\mathcal{SHOIN}$

$\boxed{\mathcal{S}}$ $\mathcal{ALC}$ + transivity: Transitivity allows to express that a role is transitive.

$\boxed{\mathcal{H}}$ subroles: $r \sqsubseteq s$ says that r is a subrole of s, i.e. $r^{\mathcal{I}} \subseteq s^{\mathcal{I}}$.

$\boxed{\mathcal{I}}$ inverse roles: r^- denotes the inverse role of r, i.e. $(a,b) \in r^{\mathcal{I}}$ iff $(b,a) \in r^{-\mathcal{I}}$.

$\boxed{\mathcal{O}}$ nominals: Sets of objects can be used to construct concepts, e.g. {`MONICA`} denotes the singleton set, which only contains `MONICA`. This can be used as a concept constructor. Nominals are useful in cases where the members of a concept should be enumerated, e.g. the members of the European Union.

$\boxed{\mathcal{N}}$ number restrictions: Allows constructs of the form $\geq n\ r$ and $\leq n\ r$ to build concepts. This is useful if you want to define a concept like "mother of at least three children" (`Woman`$\sqcap \geq 3$ `hasChild`).

$\boxed{\mathcal{Q}}$ qualified number restrictions: Concept constructors of the form $\geq n\ r.C$ and $\leq n\ r.C$ can be used. If C is the top concept, this is equivalent to (unqualified) number restrictions. This is useful to define a concept like "mother of at least three male children" (`Woman`$\sqcap \geq 3$ `hasChild.Male`).

$\boxed{\mathcal{F}}$ functional roles: Allows to express that a role r is functional, which is equivalent to the axiom $\top \sqsubseteq \leq 1\ r$.

$\boxed{\mathcal{D}}$ data types: Data types are used to incorporate different types of data e.g. numbers or strings in Description Logics. This allows for instance to define the concept of an old person as a person of age 65 or higher. Techniques for integrating data types in Description Logics have been developed, but will not be discussed further.

An interesting Description Language in context of the Semantic Web (Berners-Lee et al., 2001) is $\mathcal{SHOIN}(D)$. Table 2 shows how concepts can be constructed in this

class constructor	$\mathcal{SHOIN}$ syntax
Thing	$\top$
Nothing	$\bot$
intersectionOf	$C_1 \sqcap \cdots \sqcap C_n$
unionOf	$C_1 \sqcup \cdots \sqcup C_n$
complementOf	$\neg C$
oneOf	$\{x_1\} \sqcup \cdots \sqcup \{x_n\}$
allValuesFrom	$\forall r.C$
someValuesFrom	$\exists r.C$
maxCardinality	$\leq n\ r$
minCardinality	$\geq n\ r$
cardinality	$\leq n\ r \sqcap \geq n\ r$

Table 3: class constructors in OWL-DL

language. We can see from its name that $\mathcal{SHOIN}(D)$ also allows for different statements about roles. The set of all role statements is sometimes called RBox in analogy to TBox and ABox.

2.5 The OWL Ontology Language

OWL is an acronym for *Web Ontology Language*. The specification of OWL is maintained by the World Wide Web Consoritum (W3C). It allows the construction of ontologies and currently comes in three flavors: OWL-Lite, OWL-DL, and OWL-Full. OWL-DL is currently based on the description language $\mathcal{SHOIN}(D)$ and OWL-Lite is based on $\mathcal{SHIF}(D)$. In the OWL 1.1 specification OWL-DL may be based on $\mathcal{SROIQ}(D)$, which is more expressive than $\mathcal{SHOIN}(D)$. OWL-Lite is less expressive than OWL-DL. OWL-Full is more expressive, but undecidable (see e.g. Horrocks et al., 2003). This is not hard to show, because there are restrictions, which were necessary to keep OWL-DL decidable, and do not apply to OWL Full. OWL-DL is of great importance within the Semantic Web, because it has been established as a standard for creating ontologies. Many ontology editors like Protegé (Gennari et al., 2003) and reasoners like KAON2 (see Motik, 2006), Racer (Haarslev and Möller, 2003), Pellet (Sirin and Parsia, 2004), and many more exist for OWL-DL.

OWL-DL *classes* correspond to concepts in Description Logics and *properties* correspond to roles. OWL-DL offers more convenience constructs than $\mathcal{SHOIN}(D)$, but does not extend its expressivity. It should be noted that OWL-DL does not make the unique name assumption, so different individuals can be mapped to the same domain element. However, it allows to express equality and inequality between individuals ($a = b$, $a \neq b$). This is not a real extension of $\mathcal{SHOIN}(D)$, because there are reasoning algorithms, which do not make the unique name assumption, and also allow for expressing equality and inequality between individuals (actually most algorithms support this). Tables 3 and 4 show how constructs in OWL can be mapped to Description Logics. Description Logics are a well understood formalism. By basing OWL-DL on Description Logics,

axiom	$\mathcal{SHOIN}$ syntax
subClassOf	$C_1 \sqsubseteq C_2$
equivalentClass	$C_1 \equiv C_2$
disjointWith	$C_1 \equiv \neg C_2$
sameIndividualAs	$\{x_1\} \equiv \{x_2\}$
differentFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$
domain	$\forall r.\top \sqsubseteq C$
range	$\top \sqsubseteq \forall r.C$
subPropertyOf	$r_1 \sqsubseteq r_2$
equivalentProperty	$r_1 \equiv r_2$
inverseOf	$r_1 \equiv r_2^-$
transitiveProperty	$r^+ \sqsubseteq r$
functionalProperty	$\top \sqsubseteq \leq 1\ r$
inverseFunctionalProperty	$\top \sqsubseteq \leq 1\ r^-$

Table 4: axioms in OWL-DL

it can make use of the theory developed for DLs in particular sophisticated reasoning algorithms.

The most important reason for mentioning OWL-DL as ontology language for the Semantic Web in this thesis and showing that its foundation is the Description Logic $\mathcal{SHOIN}(D)$, is that all of the learning methods for Description Logics can also be used for OWL-DL. As a consequence the methods presented in the following sections are strongly linked to ontologies in the Semantic Web. One application of learning concepts in Description Logics is to use it as a helper method for building new classes in an existing ontology.

3 The Learning Problem

In this section we will briefly describe the learning problem in Description Logics. The process of learning in logics, i.e. finding logical explanations for given data, is also called *inductive reasoning*. In a very general setting this means that we have a logical formulation of background knowledge and some observations. We are then looking for ways to extend the background knowledge such that we can explain the observations, i.e. they can be deduced from the modified knowledge. More formally we are given background knowledge B, positive examples E^+, negative examples E^- and want to find a hypothesis H such that from H together with B the positive examples follow and the negative examples do not follow. It is not required that the same logical formalism is used for background knowledge, examples, and hypothesis, but often this is the case.

The most prominent research area within inductive reasoning is the induction of logic programs (Nienhuys-Cheng and de Wolf, 1997). In many cases Prolog (a popular logic programming language) programs are induced. In this thesis we are not concerned with inducing logic programs, but instead we want to find concept definitions in Description Logics.

For learning in Description Logics we can give a more specific description of the learning problem. The background knowledge is a knowledge base $\mathcal{K}$. The goal is to find a definition for a concept we want to call `Target`. Hence the examples are of the form $\texttt{Target}(a)$ where $a \in N_I$ is an individual. We are then looking for an acyclic concept definition of the form $\texttt{Target} \equiv C$ such that we can extend our knowledge base by this definition. Let $K' = \mathcal{K} \cup \{\texttt{Target} \equiv C\}$ be this extended knowledge base. Then we want that the positive examples follow from it, i.e. $\mathcal{K}' \models E^+$, and the negative examples do not follow, i.e. $\mathcal{K}' \not\models E^-$. Please note that the description language of the background knowledge can be more expressive than the language of the concept C we want to learn. Later on we will focus on learning $\mathcal{ALC}$ concepts, but language of the background knowledge can be e.g. $\mathcal{SHIQ}$ or $\mathcal{SHOIN}(D)$.

Definition 3.1 (learning problem in Description Logics)
Let a concept name `Target`, a knowledge base $\mathcal{K}$, and sets E^+ and E^- with elements of the form $\texttt{Target}(a)$ $(a \in N_I)$ be given. The learning problem is to find a concept C such that $\texttt{Target} \equiv C$ is an acyclic definition and for $\mathcal{K}' = \mathcal{K} \cup \{\texttt{Target} \equiv C\}$ we have $\mathcal{K}' \models E^+$ and $\mathcal{K}' \not\models E^-$. □

Some remarks about the learning problem are in order. First of all there are slightly different formulations of the learning problem in the literature. A possible different setting is that the negative examples are of the form $\neg\texttt{Target}(a)$ $(a \in N_I)$ and also have to follow, i.e. $\mathcal{K}' \models E^-$. Due to the open world assumption in Description Logics there is a difference between facts logically following from a knowledge base and the negation of a fact following from the knowledge base. Both ways of formulating the learning problem are meaningful and actually most of the content of this thesis can be used for both formulations of the problem. We decided to focus on one formulation of the learning problem to keep the presentation of the solution methods more compact.

The acyclicity restriction in the definition of the learning problem is mostly made for reasons of efficiency. If the definition is acyclic, then we check whether a concept C is a solution of the learning problem by posing C as a (retrieval or instance) query to the knowledge base, i.e. we do not need to modify the knowledge base. For an example $\texttt{Target}(a)$ instead of checking $\mathcal{K}' \models \texttt{Target}(a)$ we check $\mathcal{K} \models C(a)$. This allows the use of more efficient algorithms and in particular an extensive pre-processing of the knowledge base (see Section 6).

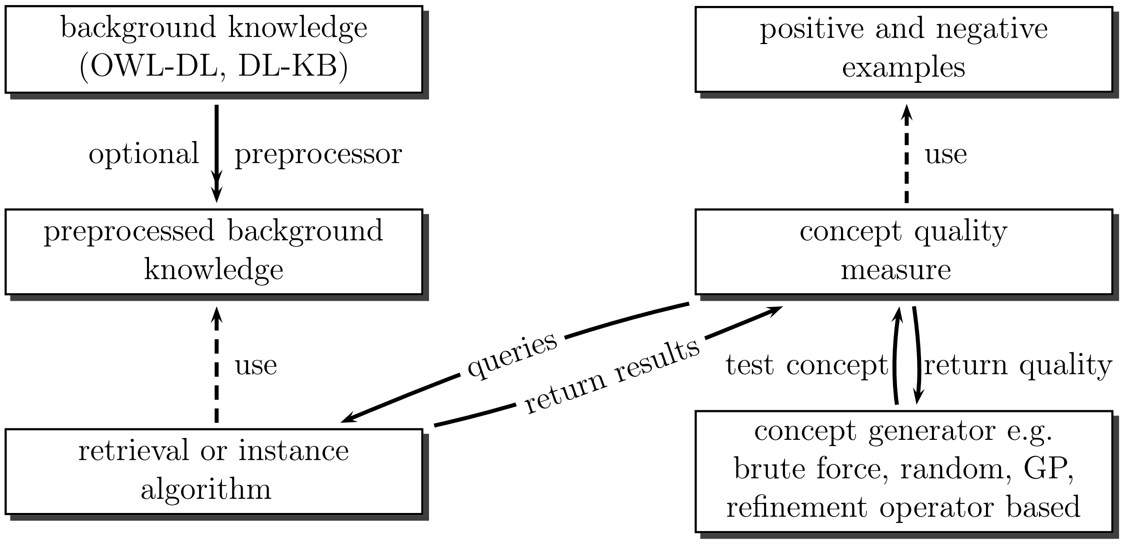

background knowledge (OWL-DL, DL-KB)
optional
preprocessor
preprocessed background knowledge
use
retrieval or instance algorithm
queries
return results
positive and negative examples
use
concept quality measure
test concept
return quality
concept generator e.g. brute force, random, GP, refinement operator based

Figure 1: overview of the learning framework

Figure 1 gives an overview how the learning problem can be solved. In this section we do not want to focus on a particular solution method, but introduce a general framework. Concept learning can be seen as a search process, so we need to have a method to generate concepts (see bottom right in Figure 1). This can for instance be an algorithm, which randomly creates concepts or an algorithm which generates the set of all concepts, e.g. from smaller to larger concepts, until a solution is found. However, it should be clear that such approaches are not very efficient. Instead we want to look at other approaches like Genetic Programming (GP) and combinations of refinement operators (to be defined in Section 4) and heuristics.

After a concept is generated we measure its quality. In the simplest case we could just measure whether it is a solution. Often we will use the number of positive and negative examples, which are *covered* by the generated concept. (We say that a concept covers an example e if $K' \models e$, where K' is defined as above.) To see if an example is covered we have to call a Description Logic reasoner, which works on the (possibly pre-processed) background knowledge. While the main topic of this thesis is to create an efficient concept generator we will also briefly look at reasoning in Section 6. This is useful since there are special requirements for reasoning algorithms in the learning framework: We make a large number of queries on the same knowledge base (so we should perform extensive pre-processing) and all the queries are instance or retrieval queries, i.e. we need assertional reasoning.

In the remainder of this section we will introduce some necessary notions. The main purpose of this section is to define a common problem description for the solution methods presented in Section 4 and 5.

When we speak about concepts as possible problem solutions it is useful to introduce some shortcuts for the two main criteria: covering all positive examples and not covering negative examples.

Definition 3.2 (complete, consistent, correct)
Let C be a concept, $\mathcal{K}$ the background knowledge base, $\texttt{Target}$ the target concept, $\mathcal{K}' = \mathcal{K} \cup \{\texttt{Target} \equiv C\}$ the extended knowledge base, and E^+ and E^- the positive and negative examples.

C is *complete* with respect to E^+ if for any $e \in E^+$ we have $\mathcal{K}' \models e$. C is *consistent* with respect to E^- if for any $e \in E^-$ we have $\mathcal{K}' \not\models e$. C is *correct* with respect to E^+ and E^- if C is complete with respect to E^+ and consistent with respect to E^-. $\square$

Definition 3.3 (too strong, too weak, overly general, overly special)
Let C be a concept, $\mathcal{K}$ the background knowledge base, $\texttt{Target}$ the target concept, $\mathcal{K}' = \mathcal{K} \cup \{\texttt{Target} \equiv C\}$ the extended knowledge base, and E^+ and E^- the positive and negative examples.

C is *too strong* with respect to E^- if C is not consistent with respect to E^-. C is *too weak* with respect to E^+ if C is not complete with respect to E^+.

C is *overly general* with respect to E^+ and E^- if C is complete with respect to E^+, but not consistent with respect to E^-. C is *overly specific* with respect to E^+ and E^- if C is consistent with respect to E^- but not complete with respect to E^+. $\square$

So far we have only looked at how a possible solution classifies the examples. Another important criteria is the length of a concept. By the well-known Occam's razor principle (Blumer et al., 1990), we should choose the simplest hypothesis for explaining the data. For our learning problem this means that from two concepts we should prefer the simpler one if both classify the examples equally good. The reason is that smaller concepts usually generalise better to unseen examples, i.e. their predictive quality is better. If smaller concepts classify the examples correctly this is less likely to be a coincidence.

We measure simplicity as the length of a concept, which is defined below in a straightforward way.

Definition 3.4 (length of a concept)
The *length* $|C|$ *of a concept* C is defined inductively (A stands for an atomic concept):

$$\begin{aligned}
|A| = |\top| = |\bot| &= 1 \\
|\neg D| &= |D| + 1 \\
|D \sqcap E| = |D \sqcup E| &= 1 + |D| + |E| \\
|\exists r.D| = |\forall r.D| &= 2 + |D|
\end{aligned}$$ $\square$

4 Refinement Operators for $\mathcal{ALC}$ Concepts

4.1 Introduction

The goal of learning as defined in Section 3 is to find a correct concept with respect to the examples. This can be seen as a search process in the space of concepts. A natural idea is to impose an ordering on this search space and use operators to traverse it. This idea is well-known in Inductive Logic Programming (Nienhuys-Cheng and de Wolf, 1997), where refinement operators are widely used to find hypotheses. Intuitively downward (upward) refinement operators construct specialisations (generalisations) of hypotheses.

Definition 4.1 (refinement operator)
A *quasi-ordering* is a reflexive and transitive relation. In a quasi-ordered space $(S, \preceq)$ a *downward (upward) refinement operator* ρ is a mapping from S to 2^S, such that for any $C \in S$ we have that $C' \in \rho(C)$ implies $C' \preceq C$ $(C \preceq C')$. C' is called a *specialisation* (*generalisation*) of C. □

This idea can be used for searching in the space of concepts. As ordering we can use subsumption. (Note that the subsumption relation $\sqsubseteq_{\mathcal{T}}$ is a quasi-ordering.) Often it makes sense to take the TBox of our background knowledge base into account. So we will, unless explicitly mentioned otherwise, consider subsumption with respect to this TBox.

If a concept C subsumes a concept D $(D \sqsubseteq_{\mathcal{T}} C)$, then C will cover all examples, which are covered by D. This makes subsumption a suitable order for searching in concepts. In this section we will analyse refinement operators for $\mathcal{ALC}$ concepts with respect to subsumption, which we will call $\mathcal{ALC}$ refinement operators in the sequel.

Definition 4.2 ($\mathcal{ALC}$ refinement operator)
A refinement operator in the quasi-ordered space $(\mathcal{ALC}, \sqsubseteq_{\mathcal{T}})$ is called an $\mathcal{ALC}$ *refinement operator*. □

We need to introduce some notions for refinement operators.

Definition 4.3 (refinement chain)
A *refinement chain of an* $\mathcal{ALC}$ *refinement operator* ρ *of length* n from a concept C to a concept D is a finite sequence $C_0, C_1, \dots, C_n$ of concepts, such that $C = C_0, C_1 \in \rho(C_0), C_2 \in \rho(C_1), \dots, C_n \in \rho(C_{n-1}), D = C_n$. This refinement chain *goes through* E iff there is an i $(1 \leq i \leq n)$ such that $E = C_i$. We say that D can be reached from C by ρ if there exists a refinement chain from C to D. $\rho^*(C)$ denotes the set of all concepts, which can be reached from C by ρ. $\rho^m(C)$ denotes the set of all concepts, which can be reached from C by a refinement chain of ρ of length m. □

Definition 4.4 (downward and upward cover)
A concept C is a *downward cover* of a concept D iff $C \sqsubset_{\mathcal{T}} D$ and there does not exist a concept E with $C \sqsubset_{\mathcal{T}} E \sqsubset_{\mathcal{T}} D$. A concept C is an *upward cover* of a concept D iff $D \sqsubset_{\mathcal{T}} C$ and there does not exist a concept E with $D \sqsubset_{\mathcal{T}} E \sqsubset_{\mathcal{T}} C$. □

If we look at refinements of an operator ρ we will often write $C \rightsquigarrow_\rho D$ instead of $D \in \rho(C)$. If the used operator is clear from the context it is usually omitted, i.e. we write $C \rightsquigarrow D$.

We will introduce the concept of weak equality of concepts, which is similar to equality of concepts, but takes into account that the order of elements in conjunctions and disjunctions is not important. By equality of two concepts we mean that the concepts are syntactically equal. Equivalence of two concepts mean that the concepts have the same meaning (see Definition 2.16 on page 12). Weak equality of concepts is coarser than equality and finer than equivalence (viewing the equivalence, equality, and weak equality of concepts as equivalence classes).

Definition 4.5 (weak syntactic equality)
We say that the concepts C and D are *weakly (syntactically) equal*, denoted by $C \simeq D$ iff one of the following conditions hold:

- $C = \top$ and $D = \top$
- $C = \bot$ and $D = \bot$
- $C = A$ and $D = A$ ($A \in N_C$)
- $C = \neg C'$ and $D = \neg D'$ and $C' \simeq D'$
- $C = \exists r.C'$ and $D = \exists r.D'$ and $C' \simeq D'$
- $C = \forall r.C'$ and $D = \forall r.D'$ and $C' \simeq D'$
- $C = C_1 \sqcap \cdots \sqcap C_n$ and $D = D_1 \sqcap \cdots \sqcap D_n$ and there is a permutation $\psi : N \mapsto N$ with $N = \{1, \dots, n\}$ such that for all $i \in N$ we have $C_i \simeq D_{\psi(i)}$
- $C = C_1 \sqcup \cdots \sqcup C_n$ and $D = D_1 \sqcup \cdots \sqcup D_n$ and there is a permutation $\psi : N \mapsto N$ with $N = \{1, \dots, n\}$ such that for all $i \in N$ we have $C_i \simeq D_{\psi(i)}$

Two sets S_1 and S_2 of concepts are weakly equal if for any $C_1 \in S_1$ there is a $C_1' \in S_2$ such that $C_1 \simeq C_1'$ and vice versa. □

Refinement operators can have certain properties, which can be used to evaluate their usefulness for learning hypothesis.

Definition 4.6 (properties of $\mathcal{ALC}$ refinement operators)
An $\mathcal{ALC}$ refinement operator ρ is called

- *(locally) finite* iff $\rho(C)$ is finite for any concept C.

- *(syntactically) redundant* iff there exists a refinement chain from a concept C to a concept D, which does not go through a concept E and a refinement chain from C to a concept weakly equal to D, which does go through E.[1]
- *proper* iff for any concepts C and D, $D \in \rho(C)$ implies $C \not\equiv_{\mathcal{T}} D$.
- *ideal* iff it is finite, complete, and proper.

An $\mathcal{ALC}$ downward refinement operator ρ is called

- *complete* iff for any concepts C and D with $C \sqsubset_{\mathcal{T}} D$ we can reach a concept E with $E \equiv_{\mathcal{T}} C$ from D by ρ.
- *weakly complete* iff for any concept C with $C \sqsubset_{\mathcal{T}} \top$ we can reach a concept E with $E \equiv_{\mathcal{T}} C$ from $\top$ by ρ.
- *minimal* iff for any C, $\rho(C)$ contains only downward covers and all its elements are incomparable with respect to $\sqsubseteq_{\mathcal{T}}$.

An $\mathcal{ALC}$ upward refinement operator ρ is called

- *complete* iff for any concepts C and D with $D \sqsubset_{\mathcal{T}} C$ we can reach a concept E with $E \equiv_{\mathcal{T}} C$ from D by ρ.
- *weakly complete* iff for any concept C with $\bot \sqsubset_{\mathcal{T}} C$ we can reach a concept E with $E \equiv_{\mathcal{T}} C$ from $\bot$ by ρ.
- *minimal* iff for any C, $\rho(C)$ contains only upward covers and all its elements are incomparable with respect to $\sqsubseteq_{\mathcal{T}}$.

All statements must hold for all TBoxes, i.e. an $\mathcal{ALC}$ refinement operator has one of the mentioned properties only if the prerequisite is satisfied for all TBoxes. □

For technical reasons we will also assume that for a complete downward (upward) operator we always have $\bot \in \rho^*(C)$ for any $C \not\equiv \bot$ ($\top \in \rho^*(C)$ for any $C \not\equiv \top$). Analogously for a weakly complete downward (upward) refinement operator we assume $\bot \in \rho^*(\top)$ ($\top \in \rho^*(\bot)$). This does not follow from the definition, e.g. for a weakly complete downard refinement operator if $\bot \sqcap \bot \in \rho^*(\top)$ then we do not require $\bot \in \rho^*(\top)$. It will become clear why we make these assumptions when we analyse the combination of completeness and non-redundancy.

[1]Another way to define redundancy is to consider equivalence ($\equiv_{\mathcal{T}}$) instead of weak equality. This is a stronger condition, but in description logics like $\mathcal{ALC}$, for which no structural subsumption algorithms exist (see Chapter 2 in Baader et al., 2003), such a semantic non-redundancy is very hard to achieve. Structural subsumption algorithms decide subsumption between two concepts by comparing their syntatictic structure. Since refinement operators usually provide syntactic rewriting rules, the non-existance of such an algorithm is problematic if we want to achieve semantic non-redundancy.

4.2 Analysing the Properties of $\mathcal{ALC}$ Refinement Operators

In this section we will analyse the properties of $\mathcal{ALC}$ refinement operators. The need for such an analysis was expressed in (Esposito et al., 2004, section 5). In particular we are interested in seeing which desired properties can be combined in a refinement operator and which properties are impossible to combine. This is interesting for two reasons: The first one is that this gives us a good impression of how hard (or easy) it is to learn $\mathcal{ALC}$ concepts. The second reason is that this can also serve as a practical guide for designing $\mathcal{ALC}$ refinement operators. Knowing the theoretical limits allows the designer of an $\mathcal{ALC}$ refinement operator to focus on achieving the best possible properties.

$\mathcal{ALC}$ refinement operators have been designed in (Esposito et al., 2004; Iannone and Palmisano, 2005). However, a full theoretical analysis for $\mathcal{ALC}$ has not been done to the best of our knowledge. Therefore all propositions in this section are new unless explicitly mentioned otherwise. Some properties for $\mathcal{ALER}$ refinement operators were shown in (Badea and Nienhuys-Cheng, 2000). $\mathcal{ALER}$ is not closed under boolean operations, so $\mathcal{ALC}$ is usually more difficult to handle in this context.

As a first property we will briefly analyse minimality of $\mathcal{ALC}$ refinement operators, in particular the existence of upward and downward covers in $\mathcal{ALC}$. It is not immediately obvious that e.g. downward covers exist in $\mathcal{ALC}$, because it could be the case that for any concept C and D with $C \sqsubset D$ one can always construct a concept E with $C \sqsubset E \sqsubset D$. However, the next proposition shows that downward covers do exist.

Proposition 4.7 (existence of covers in $\mathcal{ALC}$)
Downward (upward) covers exist in $\mathcal{ALC}$.

Proof Let $N_R = \{r\}$ and $N_C = \{A\}$. We look at subsumption with respect to an empty TBox. We want to show that $C = \exists r.\top \sqcup A$ is a downward cover of $\top$. We have to show that there is no concept D with $C \sqsubset D \sqsubset \top$. We will prove that such a concept cannot exist from a semantical point of view. By contradiction, we assume that we have found such a concept D.

Since C is strictly subsumed by D, there is an interpretation $\mathcal{I}_1$ and an object a_1 such that $a_1^{\mathcal{I}_1} \notin C^{\mathcal{I}}$ and $a_1^{\mathcal{I}_1} \in D^{\mathcal{I}_1}$. a_1 cannot have an r-filler, because then we would immediately get $a_1^{\mathcal{I}_1} \in C^{\mathcal{I}}$ (due to the $\exists r.\top$ in C). Similarly we get that $a_1^{\mathcal{I}_1} \notin A^{\mathcal{I}_1}$.

Since D is strictly subsumed by $\top$, there is an interpretation $\mathcal{I}_2$ and an object a_2 such that $a_2^{\mathcal{I}_2} \notin D^{\mathcal{I}_2}$. Because of $C \sqsubset D$, we know $a_2^{\mathcal{I}_2} \notin C^{\mathcal{I}_2}$. By the same arguments as above we can deduce that a_2 does not have an r-filler in $\mathcal{I}_2$ and $a_2^{\mathcal{I}_2} \notin A^{\mathcal{I}_2}$.

In both interpretations $\mathcal{I}_j$ ($j \in \{1, 2\}$), the associated objects a_j do not have an r-filler and do not belong to A. Additionally there are no other concept names and roles, so both interpretations have to interpret concepts - in particular D - in the same way with respect to the associated objects, i.e. we either have ($a_1^{\mathcal{I}_1} \in D^{\mathcal{I}_1}$ and $a_2^{\mathcal{I}_2} \in D^{\mathcal{I}_2}$) or we have ($a_1^{\mathcal{I}_1} \notin D^{\mathcal{I}_1}$ and $a_2^{\mathcal{I}_2} \notin D^{\mathcal{I}_2}$). (The reason is that to determine, whether $a^{\mathcal{I}} \in E^{\mathcal{I}}$ holds for an arbitrary $\mathcal{ALC}$ concept E and an interpretation $\mathcal{I}$, such that a does not have a role filler in $\mathcal{I}$ and does not belong to any atomic concept, it is not important how objects different from a are interpreted by $\mathcal{I}$.) This is a contradiction, because we assumed $a_1^{\mathcal{I}_1} \in D^{\mathcal{I}_1}$ and $a_2^{\mathcal{I}_2} \notin D^{\mathcal{I}_2}$.

Upward covers can be handled analogously, i.e. $\forall r.\bot \sqcap A$ is an upward cover of $\bot$. ■

The idea in the proof Proposition 4.7 can be extended to situations with more than one role and concept name. In this case we obtain the following concept as a downward cover of $\top$ (we do not prove this explicitly, because we do not use this result later on):

$$\bigsqcup_{r \in N_R} \exists r.\top \sqcup \bigsqcup_{A \in N_C} A$$

The observations show that non-trivial minimal operators, i.e. operators which do not map every concept to the empty set, can be constructed. However, minimality of refinement steps is not a directly desired goal in general. Minimal operators are in some languages more likely to lead to overfitting, because they may not produce sufficient generalisation leaps. This is true in a language like $\mathcal{ALC}$, which is closed under boolean operations.

In the sequel we will analyse desired properties of $\mathcal{ALC}$ refinement operators: completeness, properness, finiteness, and non-redundancy. We will show several positive and negative results, which together yield a full analysis of these properties.

Proposition 4.8 (complete and finite $\mathcal{ALC}$ refinement operators)
There exists a complete and finite $\mathcal{ALC}$ refinement operator.

Proof Consider the downward refinement operator ρ defined by:

$$\rho(C) = \{C \sqcap \top\} \cup \{D \mid |D| \leq (\text{number of } \top \text{ occurences in } C) \text{ and } D \sqsubset_{\mathcal{T}} C\}$$

The operator can do one of two things:

- add a $\top$ symbol
- generate the set of all concepts up to a certain length, which are subsumed by C

The operator is finite, because the set of all concepts up to a given length is finite (and the singleton set $\{C \sqcap \top\}$ is finite).

The operator is complete, because given a concept C we can reach an arbitrary concept D with $D \sqsubset_{\mathcal{T}} C$. This is obvious, because we only need to add $\top$-symbols until there are $|D|$ occurences of $\top$. Within the next step we can then be sure to reach D.

For upward refinement operators we can use an analogous operator φ, which is also complete and finite:

$$\varphi(C) = \{C \sqcap \top\} \cup \{D \mid |D| \leq (\text{number of } \top \text{ occurences in } C) \text{ and } C \sqsubset_{\mathcal{T}} D\}$$ ■

Remark 4.9 (complete and finite refinement operators)
In (Badea and Nienhuys-Cheng, 2000) it was stated that there can be no complete and finite $\mathcal{ALER}$ refinement operator. However, this is not correct, because by using the same technique as in the proof of Proposition 4.8 one can construct a finite and complete $\mathcal{ALER}$ refinement operator. □

Of course, it is obvious that the operator used to prove Proposition 4.8 is not useful in practice, since it merely generates concepts without paying attention to efficiency. However, in this section we are interested in theoretical limits of refinement operators, so it is a valid method to define impractical operators. It is indeed difficult to design a good complete and finite refinement operator. The reason is that finiteness can only be achieved by using non-proper refinement steps (in the operator this was done by adding $\top$ symbols). In Section 5 we will define a complete and finite operator in the context of the Genetic Programming framework. We will now show that it is impossible to define a complete, finite, and proper refinement operator. Such operators are known as ideal and their non-existance indiciates that learning $\mathcal{ALC}$ concepts is not easy.

Proposition 4.10 (ideal $\mathcal{ALC}$ refinement operators)
There exists no ideal $\mathcal{ALC}$ refinement operator.

Proof By contradiction, we assume that there exists an ideal downward refinement operator ρ. We further assume an empty TBox, $N_C = \emptyset$, and $N_R = \{r\}$. Let $\rho(\top) = \{C_1, \dots, C_n\}$ be a set of refinements of the $\top$ concept. (This set has to be finite.) Let m be a natural number larger than the maximum of the *quantor depths* (depth of the nesting of quantifications) of the concepts in $\rho(\top)$. We construct a concept D as follows:

$$D = \underbrace{\forall r. \dots . \forall r}_{m-\text{times}} . \bot \sqcup \underbrace{\exists r. \dots . \exists r}_{(m+1)-\text{times}} . \top$$

What is the meaning of D? As we can see, the semantics of $\forall r.\bot$ is that an object must not have an r-filler and the semantics of $\exists r.\top$ is that an object must have an r-filler. We say that an object $a \in N_I$ has an *r-successor at distance n in $\mathcal{I}$* if there is a set of objects $\{a_0, \dots, a_n\} \subseteq N_I$ with $a = a_0, (a_0^{\mathcal{I}}, a_1^{\mathcal{I}}) \in r^{\mathcal{I}}, \dots, (a_{n-1}^{\mathcal{I}}, a_n^{\mathcal{I}}) \in r^{\mathcal{I}}$. The meaning of D is that an object does not have an r-successor at distance m in $\mathcal{I}$ or it has an r-successor at distance $m+1$ in $\mathcal{I}$. Formally for an arbitrary object a and an interpretation $\mathcal{I}$ we have $a^{\mathcal{I}} \in D^{\mathcal{I}}$ iff a does not have an r-successor at distance m in $\mathcal{I}$ or a has an r-successor at distance $m+1$ in $\mathcal{I}$.

D is not equivalent to $\top$. For the interpretation $\mathcal{I}$ with $r^{\mathcal{I}} = \{r(a,b) \mid a = a_i, b = a_{i+1}, 0 \le i < m\}$, illustrated by

$$a_0 \xrightarrow{r} a_1 \xrightarrow{r} \dots \xrightarrow{r} a_m$$

we have $a_0^{\mathcal{I}} \notin D^{\mathcal{I}}$.

As a prerequisite for proving the proposition, we want to show that there exists no concept with a quantifier depth smaller than m, which strictly subsumes D and is not equivalent to $\top$. By contradiction, we assume such a concept E with $D \sqsubset E \sqsubset \top$ exists. Since we have $E \not\equiv \top$, there exists an interpretation $\mathcal{I}$ and an object a such that $a^\mathcal{I} \notin E^\mathcal{I}$. By $D \sqsubset E$, this also implies $a^\mathcal{I} \notin D^\mathcal{I}$. We do a case distinction on a and $\mathcal{I}$:

1. a does not have an r-successor at distance m in $\mathcal{I}$: By the semantics of D, as discussed above, this means $a^\mathcal{I} \in D^\mathcal{I}$, which contradicts $a^\mathcal{I} \notin D^\mathcal{I}$.

2. a has an r-successor at distance m in $\mathcal{I}$:

   We can view the interpretation $\mathcal{I}$ as a directed graph in a straightforward way. The set of nodes is $\{b^\mathcal{I} \mid b \in N_I\}$ and the edges are defined by $\{(b, c) \mid (b, c) \in r^\mathcal{I}\}$. This graph does not contain a cycle, which is reachable from $a^\mathcal{I}$, because then we would have $a^\mathcal{I} \in D^\mathcal{I}$ due to the $\exists r. \ldots . \exists r.\top$ part of D (if we have a reachable cycle, there always exists a successor). Hence, $a^\mathcal{I}$ spans an acyclic graph, i.e. a tree, of successors in $\mathcal{I}$.

   Because of $a^\mathcal{I} \notin D^\mathcal{I}$, we know that there is a path of length m starting from $a^\mathcal{I}$ in this tree (due to the $\forall r. \ldots . \forall r.\bot$ part of D). This means we have pairwise different objects $a_0, \ldots, a_m$ such that $a = a_0$ and $(a_0^\mathcal{I}, a_1^\mathcal{I}) \in r^\mathcal{I}, \ldots, (a_{m-1}^\mathcal{I}, a_m^\mathcal{I}) \in r^\mathcal{I}$.

   We create a new interpretation $\mathcal{I}'$ from $\mathcal{I}$ by adding a new object a_{m+1} and changing $r^\mathcal{I}$ to $r^{\mathcal{I}'} = r^\mathcal{I} \cup \{(a_m^\mathcal{I}, a_{m+1}^\mathcal{I})\}$. Since E has a quantifier depth smaller than m, we know that a_{m+1} is out of the scope of these quantifiers, so we can deduce $a^{\mathcal{I}'} \notin E^{\mathcal{I}'}$ from $a^\mathcal{I} \notin E^\mathcal{I}$. However, a has a successor at distance m in $\mathcal{I}'$, so by the semantics of D we have $a^{\mathcal{I}'} \in D^{\mathcal{I}'}$. This implies $a^{\mathcal{I}'} \in E^{\mathcal{I}'}$ due to $D \sqsubset E$, which is contradiction to $a^{\mathcal{I}'} \notin E^{\mathcal{I}'}$.

Hence we have shown that there does not exist a more general concept than D, which is not equal to $\top$, with a quantifier depth smaller than m. This means that $C_1, \ldots, C_n$ do not subsume D (note that the properness of ρ implies that $C_1, \ldots, C_n$ are not equivalent to $\top$), so D cannot be reached by further refinements from any of these concepts. Since $C_1, \ldots, C_n$ are the only refinements of $\top$, it is impossible to reach D from $\top$. Thus ρ is not complete. ■

Proposition 4.11 (complete and proper $\mathcal{ALC}$ refinement operators)
There exists a complete and proper $\mathcal{ALC}$ refinement operator.

Proof The proof is trivial, because we can just use $\rho(C) = \{D \mid D \sqsubset C\}$ as downward refinement operator, which is obviously complete and proper. For upward refinement we can analogously consider $\rho(C) = \{D \mid C \sqsubset D\}$. ■

We have shown that the combination of completeness and properness is possible. Propositions 4.8, 4.10, and 4.11 state that for complete refinement operators, which are usually desirable, one has to sacrifice properness or finiteness. We will now look at non-redundancy.

Proposition 4.12 (compelete, non-redundant $\mathcal{ALC}$ refinement operators)
There exists no complete and non-redundant $\mathcal{ALC}$ refinement operator.

PROOF Again, we look at subsumption with respect to an empty TBox. Let A_1 and A_2 ($A_1, A_2 \not\equiv \top$ and $A_1, A_2 \not\equiv \bot$) be two atomic concepts and ρ a complete downward refinement operator. By completeness of ρ, we know there exists a concept C_1 with $C_1 \equiv A_1$ and $C_1 \in \rho^*(\top)$. Analogously there exists a concept C_2 with $C_2 \equiv A_2$ and $C_2 \in \rho^*(\top)$. As a requirement on complete operators we stated before on page 24, that for a complete downward refinement operator $\bot \in \rho^*(C)$ for any $C \not\equiv \bot$ must hold. In particular we have $\bot \in \rho^*(C_1)$ and $\bot \in \rho^*(C_2)$. Because of $C_1 \not\sqsubseteq C_2$ and $C_2 \not\sqsubseteq C_1$ we know $C_1 \notin \rho^*(C_2)$ and $C_2 \notin \rho^*(C_1)$.

Hence there exists a refinement chain from $\top$ to $\bot$ through C_1 and a refinement chain from $\top$ to $\bot$, which goes through C_2 and not through C_1. Thus ρ is redundant.

Note that without the assumption $\bot \in \rho^*(C)$ for $C \not\equiv \bot$, the result does not necessarily hold. However, it would require that for incomparable concepts D_1, D_2, and a concept D_3 with $D_3 \sqsubset D_1$ and $D_3 \sqsubset D_2$ we would have to reach different syntactic representations of D_3 (i.e. concepts, which are equivalent to D_3 but not weakly equal to D_3). This means that the only way to avoid redundancy is to encode the (usually infinitely many) possible paths to a concept in syntactic constructs, e.g. from A_1 we can reach $\bot \sqcap \bot$, but from A_2 we reach $\bot \sqcap \bot \sqcap \bot$ and not $\bot \sqcap \bot$ etc. This is clearly not desirable and the resulting operator would not be meaningful. For this reason we have chosen to make the additional assumption $\bot \in \rho(C)$ for any $C \not\equiv \bot$ in $\mathcal{ALC}$ downward refinement operators.

Again, the proof for upward refinement operators is analogous. ■

As a consequence, completeness and non-redundancy cannot be combined. Usually it is desirable to have (weakly) complete operators, but in order to have a full analysis of $\mathcal{ALC}$ refinement operators we will now also investigate incomplete operators.

Proposition 4.13 (incomplete $\mathcal{ALC}$ refinement operators)
There exists a finite, proper, and non-redundant $\mathcal{ALC}$ refinement operator.

PROOF The following operator has the desired properties:

$$\rho(C) = \begin{cases} \{\bot\} & \text{if } C \not\equiv \bot \\ \emptyset & \text{otherwise} \end{cases}$$

It is obviously finite, because it maps concepts to sets of cardinality at most 1. It is non-redundant, because it only reaches the bottom concept and there exists no refinement chain of length greater than 2. It is proper, because all concepts, which are not equivalent to the bottom concept strictly subsume the bottom concept.

The corresponding upward operator is:

$$\varphi(C) = \begin{cases} \{\top\} & \text{if } C \not\equiv \top \\ \emptyset & \text{otherwise} \end{cases}$$

The arguments for its finiteness, properness, and non-redundancy are analogous to the downward case. ■

We can now summarise the results we have obtained so far.

Theorem 4.14 (properties of $\mathcal{ALC}$ refinement operators (I))
Considering the properties completeness, properness, finiteness, and non-redundancy the following are maximal sets of properties (in the sense that no other of the mentioned properties can be added) of $\mathcal{ALC}$ refinement operators:

1. *{complete, finite}*
2. *{complete, proper}*
3. *{non-redundant, finite, proper}*

Proof The theorem is a consequence of the previous results. We have seen that downward and upward operators allow the same combinations of properties, so it is not necessary to distinguish between them. To be sure to cover all combinations of properties we make a case distinction.

1. The operator is complete. In this case we cannot add non-redundancy (Proposition 4.12). Finiteness (Proposition 4.8) and properness (Proposition 4.11) can be added, but not both (Proposition 4.10).
2. The operator is not complete. In this case we can add all other properties (Proposition 4.13). ■

A property we have not yet considered is weak completeness. Usually weak completeness is sufficient, because it allows to search for a good concept starting from $\top$ downwards (top-down approach) or from $\bot$ upwards (bottom-up approach).

We will see that we get different results when considering weak completeness instead of completeness. As a first observation we see that the arguments in the proof of Proposition 4.12, which have shown that an $\mathcal{ALC}$ refinement operator cannot be complete and non-redundant, do no longer apply if we consider weak completeness and non-redundancy.

The reason is that there is no longer a guarantee that $\bot$ can be reached from both, A_1 and A_2. Indeed it turns out that there are weakly complete and non-redundant operators and this set of properties is not even maximal.

Proposition 4.15 (weakly complete, non-redundant, and proper operators) *There exists a weakly complete, non-redundant, and proper $\mathcal{ALC}$ refinement operator.*

Proof The following operator is weakly complete, non-redundant, and proper:
Let S be a maximal subset of $\{C \mid C \not\equiv_{\mathcal{T}} \top\}$ with $C_1, C_2 \in S \implies C_1 \not\simeq C_2$.

$$\rho(C) = \begin{cases} S & \text{if } C = \top \\ \emptyset & \text{otherwise} \end{cases}$$

Such a set S as used in the definition of the operator indeed exists. It contains one representative of each equivalence class with respect to weak equality of the set $\{C \mid C \not\equiv \top\}$. The operator is proper, since it contains only mappings of the top concept to concepts, which are not equivalent to top. It is non-redundant, because there is no refinement chain of length greater than 1 and all concepts we reach are pairwise not weakly equal. It is weakly complete, because for every concept, which is not equivalent to $\top$, we can reach an equivalent concept from $\top$ by ρ.

The corresponding upward refinement operator is: Let S be a maximal subset of $\{C \mid C \not\equiv_{\mathcal{T}} \bot\}$ with $C_1, C_2 \in S \implies C_1 \not\simeq C_2$.

$$\rho(C) = \begin{cases} S & \text{if } C = \bot \\ \emptyset & \text{otherwise} \end{cases}$$

■

Proposition 4.16 (weakly complete, non-redundant, and finite operators) *There exists a weakly complete, non-redundant, and finite $\mathcal{ALC}$ refinement operator.*

Proof The following operator is weakly complete, non-redundant, and finite: For an arbitrary concept C, let S_C be a maximal subset of $\{D \mid D \sqsubset_{\mathcal{T}} \top \text{ and } |D| = \text{number of } \top \text{ occurences in } C\}$ with $C_1, C_2 \in S_C \implies C_1 \not\simeq C_2$.

$$\rho(C) = \begin{cases} \{\underbrace{\top \sqcap \dots \sqcap \top}_{n+1 \text{ times } \top}\} \cup S_C & \text{if } C = \underbrace{\top \sqcap \dots \sqcap \top}_{n \text{ times } \top} \\ \emptyset & \text{otherwise} \end{cases}$$

The operator is finite, because S_C is finite for any concept C (the number of concepts with a fixed length is finite). It is weakly complete, because every concept C with $C \sqsubset \top$

can be reached from $\top$. This is done by accumulating $\top$ symbols until we have $|C|$ such symbols and then generate C.

We will show the non-redundany of the operator by a simple case distinction. By contradiction, we assume that ρ is redundant, i.e. there exist concepts C, D, and E with $C \neq D$, such that there is a refinement chain from C to D through E and a different refinement chain from C to D, which does not go through E.

1. $C \not\equiv \top$: In this case a refinement chain to D cannot exist. ($D \sqsubseteq_{\mathcal{T}} C$ so $D \not\equiv_{\mathcal{T}} \top$, but there is only one element not equivalent to $\top$ in each refinement chain.)

2. $C \equiv \top$ and $D \equiv \top$: In this case there is exactly one refinement chain from C to D of the form:
$$\underbrace{\top \sqcap \cdots \sqcap \top}_{=C} \rightsquigarrow_\rho \ldots \rightsquigarrow_\rho \underbrace{\top \sqcap \cdots \sqcap \top}_{=D}$$
This contradicts the redundancy of ρ in this case, because for ρ to be redundant at least two different refinement chains from C to D must exist.

3. $C \equiv \top$ and $D \not\equiv \top$: Again, there is exactly one refinement chain:
$$\underbrace{\top \sqcap \cdots \sqcap \top}_{=C} \rightsquigarrow_\rho \ldots \rightsquigarrow_\rho \underbrace{\top \sqcap \cdots \sqcap \top}_{|D| \text{ times}} \to D$$
Note that we ensured that there cannot be a weakly equal concept of D in other refinement chains by definition of S_C.

The corresponding upward operator is analogous. It works by accumulating $\bot$ symbols instead of $\top$ symbols and generates concepts, which are strictly more general than $\bot$. ■

Corollary 4.17 (weakly complete, proper, and finite operators)
There exists a weakly complete, finite, and proper $\mathcal{ALC}$ refinement operator.

Proof To show this we can use the proof of Proposition 4.10. There we have shown that in a finite and proper $\mathcal{ALC}$ refinement operator there exists a concept, which cannot be reached from the $\top$ concept. This means that such an operator cannot be weakly complete. ■

The result of the previous observations is that, when requiring only weak completeness instead of completeness, non-redundant operators are possible. The following theorem is the result of the full analysis of the desired properties of $\mathcal{ALC}$ refinement operators.

Theorem 4.18 (properties of $\mathcal{ALC}$ refinement operators (II))
Considering the properties completeness, weak completeness, properness, finiteness, and non-redundancy the following are maximal sets of properties (in the sense that no other of the mentioned properties can be added) of $\mathcal{ALC}$ refinement operators:

1. $\{$*weakly complete*, *complete*, *finite*$\}$
2. $\{$*weakly complete*, *complete*, *proper*$\}$
3. $\{$*weakly complete*, *non-redundant*, *finite*$\}$
4. $\{$*weakly complete*, *non-redundant*, *proper*$\}$
5. $\{$*non-redundant*, *finite*, *proper*$\}$

Proof We can do a similar case distinction like in Theorem 4.14. The first case (complete operator) is analogous except that obviously a complete operator is also weakly complete. For the second case (operator is not complete) we can make a simple case distinction again:

1. The operator is weakly complete. Propositions 4.15 and 4.16 have shown that weakly complete operators can be non-redundant and proper as well as non-redundant and finite. Proposition 4.17 shows that finiteness and properness cannot be combined, so these sets of properties are maximal.

2. The operator is not weakly complete. In this case we can add all remaining properties (Proposition 4.13), i.e. non-redundany, finiteness, and properness. ■

Theorem 4.18 summarizes the analysis of $\mathcal{ALC}$ refinement operators and is an important theoretical result of this thesis. As the reader has probably noticed the operators used as positive examples are very odd. It still remains to define a useful $\mathcal{ALC}$ refinement operator, which we will do in the next section.

4.3 A Refinement Operator for $\mathcal{ALC}$ Concepts

In the sequel we will analyse the refinement operator $\rho_\downarrow$ given by:

$$\rho_\downarrow(C) = \begin{cases} \{\bot\} \cup \rho'_\downarrow(C) & \text{if } C = \bot \\ \rho'_\downarrow(C) & \text{otherwise} \end{cases}$$

$$\rho'_\downarrow(C) = \begin{cases} \{C_1 \sqcap \dots \sqcap C_{i-1} \sqcap D \sqcap C_{i+1} \sqcap \dots \sqcap C_n & \text{if } C = C_1 \sqcap \dots \sqcap C_n \ (n \geq 2) \\ \quad \mid D \in \rho'_\downarrow(C_i), 1 \leq i \leq n\} & \\ \{C_1 \sqcup \dots \sqcup C_{i-1} \sqcup D \sqcup C_{i+1} \sqcup \dots \sqcup C_n & \text{if } C = C_1 \sqcup \dots \sqcup C_n \ (n \geq 2) \\ \quad \mid D \in \rho'_\downarrow(C_i), 1 \leq i \leq n\} & \\ \cup \{C \sqcap D \mid D \in \rho'_\downarrow(\top)\} & \\ \{A' \mid A' \sqsubset_\mathcal{T} A, A' \in N_C, & \text{if } C = A \ (A \in N_C) \\ \ \text{there is no } A'' \in N_C \text{ with } A' \sqsubset_\mathcal{T} A'' \sqsubset_\mathcal{T} A\} & \\ \cup \{C \sqcap D \mid D \in \rho'_\downarrow(\top)\} & \\ \{\neg A' \mid A \sqsubset_\mathcal{T} A', A' \in N_C, & \text{if } C = \neg A \ (A \in N_C) \\ \ \text{there is no } A'' \in N_C \text{ with } A \sqsubset_\mathcal{T} A'' \sqsubset_\mathcal{T} A'\} & \\ \cup \{C \sqcap D \mid D \in \rho'_\downarrow(\top)\} & \\ \{\exists r.E \mid E \in \rho'_\downarrow(D)\} \cup \ \{C \sqcap D \mid D \in \rho'_\downarrow(\top)\} & \text{if } C = \exists r.D \\ \{\forall r.E \mid E \in \rho'_\downarrow(D)\} \cup \ \{C \sqcap D \mid D \in \rho'_\downarrow(\top)\} & \text{if } C = \forall r.D \\ \cup \{\forall r.\bot \mid D = A \in N_C & \\ \ \text{and there is no } A' \in N_C \text{ with } \bot \sqsubset A' \sqsubset A\} & \\ \emptyset & \text{if } C = \bot \\ \{D \mid D \in M\} & \text{if } C = \top \\ \cup \{D \sqcup E \mid D \in M, E \in \rho'_\downarrow(\top)\} & \end{cases}$$

The set M used in the definition of $\rho'_\downarrow$ is defined as:

$$\begin{aligned} M = & \{A \mid A \in N_C, \text{ there is no } A' \in N_C \text{ with } A \sqsubset_\mathcal{T} A' \sqsubset_\mathcal{T} \top\} \\ & \cup \ \{\neg A \mid A \in N_C, \text{ there is no } A' \in N_C \text{ with } \bot \sqsubset_\mathcal{T} A' \sqsubset_\mathcal{T} A\} \\ & \cup \ \{\exists r.\top \mid r \in N_R\} \\ & \cup \ \{\forall r.C \mid r \in N_R, C \in \rho'_\downarrow(\top)\} \end{aligned}$$

Proposition 4.19 (downward refinement of $\boldsymbol{\rho_\downarrow}$)
$\rho_\downarrow$ is an $\mathcal{ALC}$ downward refinement operator.

PROOF We have to show that $D \in \rho_\downarrow(C)$ implies $D \sqsubseteq_\mathcal{T} C$. We do this by structural induction of $\mathcal{ALC}$ concepts in negation normal form. Obviously in all cases where $D = C \sqcap C'$, i.e. C is extended conjunctively by a concept C' we have $D \sqsubseteq C$, so these cases can be ignored.

- $C = \bot$: $D \in \rho_\downarrow(C)$ is impossible, because $\rho_\downarrow(\bot) = \emptyset$.
- $C = \top$: $D \sqsubseteq_\mathcal{T} C$ is trivially true.

- $C = A$ ($A \in N_C$): $D \in \rho_\downarrow(C)$ implies that D is also an atomic concept and $D \sqsubset_\mathcal{T} C$.

- $C = \neg A$: $D \in \rho_\downarrow(C)$ implies that D is of the form $\neg A'$ with $A \sqsubset_\mathcal{T} A'$. $A \sqsubset_\mathcal{T} A'$ implies $\neg A' \sqsubset_\mathcal{T} \neg A$ by the semantics of negation.

- $C = \exists r.C'$: $D \in \rho_\downarrow(C)$ implies that D is of the form $\exists r.D'$. We have $D' \sqsubseteq_\mathcal{T} C'$ by induction. For existential restrictions $\exists r.E \sqsubseteq_\mathcal{T} \exists r.E'$ if $E \sqsubset_\mathcal{T} E'$ holds in general (Badea and Nienhuys-Cheng, 2000). Thus we also have $\exists r.D' \sqsubseteq_\mathcal{T} \exists r.C'$.

- $C = \forall r.C'$: This case is analogous to the previous one. For universal restrictions $\forall r.E \sqsubseteq_\mathcal{T} \forall r.E'$ if $E \sqsubset_\mathcal{T} E'$ holds in general (Badea and Nienhuys-Cheng, 2000).

- $C = C_1 \sqcap \cdots \sqcap C_n$: In this case one element of the conjunction is refined, so $D \sqsubseteq_\mathcal{T} C$ follows by induction.

- $C = C_1 \sqcup \cdots \sqcup C_n$: In this case one element of the disjunction is refined, so $D \sqsubseteq_\mathcal{T} C$ follows by induction. ■

We have shown that $\rho_\downarrow$ is an $\mathcal{ALC}$ downward refinement operator. The most important property for an $\mathcal{ALC}$ refinement operator to be useful is weak completeness, which we will investigate next, but first we want to discuss other characteristics of $\rho_\downarrow$.

$\rho_\downarrow$ is infinite. There are two sources of infinity: First of all a refinement of the top concept can have an infinite number of universal quantifications. The reason is the equality $\forall r.\top \equiv \top$. This refinement (i.e. $\top \rightsquigarrow \forall r.\top$) is not useful, so in refinements of the form $\top \rightsquigarrow \forall r.C$ we require C to be a refinement of the top concept, which can again contain a universal quantification etc. The second source of infinity is that we allow to create a disjunction with an arbitrary number of elements as a refinement of the top concept. In fact refining the top concept is the only way to introduce disjunctions, whereas in all other cases we allow to add concepts conjunctively.

One of the nice properties of $\rho_\downarrow$ compared to other refinement operators for learning concepts in Description Logics (Badea and Nienhuys-Cheng, 2000; Esposito et al., 2004) is that it makes use of the subsumption hierarchy of concepts in the knowledge base. Other operators usually select an atomic concept, but do not allow to refine it. Allowing the traversal of the subsumption hierarchy by a refinement operator does not directly affect its theoretical properties, but is useful since it makes use of knowledge contained in the TBox.

When observing $\rho_\downarrow$ we see that there are four possible syntactic changes, which it can perform. These changes are the replacement of a top symbol, denoted by $\overset{\top}{\rightsquigarrow}$, adding a concept conjunctively, denoted by $\overset{\sqcap}{\rightsquigarrow}$, replacing an atomic concept by a more general one, denoted by $\overset{A}{\rightsquigarrow}$, and replacing a negated atomic concept by a negated more special atomic concept, denoted by $\overset{\neg A}{\rightsquigarrow}$.

4.4 Weak Completeness of the Operator

In the sequel we will show the weak completeness of $\rho_\downarrow$. We will do this stepwise. First we define a set $S_\downarrow$ of $\mathcal{ALC}$ concepts. For these concepts we show that every $\mathcal{ALC}$ concept, which is not equivalent to $\top$, is equivalent to an element of $S_\downarrow$. Then we show that all concepts in $S_\downarrow$ can be reached from $\top$ by $\rho_\downarrow$.

Definition 4.20 ($S_\downarrow$)
The set $S'_\downarrow$ is defined as follows:

1. If $A \in N_C$ then $A \in S'_\downarrow$ and $\neg A \in S'_\downarrow$.
2. If $r \in N_R$ then $\forall r.\bot \in S'_\downarrow$, $\exists r.\top \in S'_\downarrow$.
3. If $C, C_1, \dots, C_m$ are in $S'_\downarrow$ then the following concepts are also elements of $S'_\downarrow$:
   - $\exists r.C$
   - $\forall r.C$
   - $C_1 \sqcap \dots \sqcap C_m$
   - $C_1 \sqcup \dots \sqcup C_m$ if for all i $(1 \leq i \leq m)$ C_i is not of the form $D_1 \sqcap \dots \sqcap D_n$ where all D_j $(1 \leq j \leq n)$ are of the form $E_1 \sqcup \dots \sqcup E_p$

The set $S_\downarrow$ is defined as $S_\downarrow = S'_\downarrow \cup \{\bot\}$. □

The set $S_\downarrow$ is obviously a set of concepts in negation normal form. However, we do not use the $\top$ and $\bot$ symbols directly and we give a restriction on disjunctions, i.e. we do not allow that elements of a disjunction are conjunctions, which in turn only consist of disjunctions.

For technical reasons we will always assume that disjunctions are not nested in disjunctions and conjunctions are not nested in conjunctions, e.g. we do not write $(C_1 \sqcap C_2) \sqcap C_3$, but $C_1 \sqcap C_2 \sqcap C_3$.

Lemma 4.21 ($S_\downarrow$)
For any $\mathcal{ALC}$ concept C there exists a concept $D \in S_\downarrow$ such that $D \equiv C$.

Proof We first transform C to negation normal form (see Section 2.3). The proof consists of three steps: First we will eliminate $\top$ symbols unless they occur in existential restrictions (because in Definition 4.20 $\exists r.\top$ is used in the induction base opposed to using $\top$ directly as in Definition 2.25). After that we do something similar with the bottom symbol. In a third step we will eliminate disjunctions violating the criterion in Definition 4.20. After these three steps we obviously obtain a concept, which is in $S_\downarrow$.

We eliminate $\top$-symbols by applying the following rewrite rules:

$$\begin{aligned} C_1 \sqcap \dots \sqcap C_{i-1} \sqcap \top \sqcap C_{i+1} \sqcap \dots \sqcap C_n &\rightarrow C_1 \sqcap \dots \sqcap C_{i-1} \sqcap C_{i+1} \sqcap \dots \sqcap C_n \\ C_1 \sqcup \dots \sqcup C_{i-1} \sqcup \top \sqcup C_{i+1} \sqcup \dots \sqcup C_n &\rightarrow \top \\ \forall r.\top &\rightarrow \top \end{aligned}$$

Obviously these $\top$-elimination steps preserve equivalence. We exhaustively apply these steps (since every step reduces the length of the concept there can be only finitely many such steps) to get a concept C'. Note that $C' \neq \top$ (otherwise $C' \equiv C \equiv \top$) and in C' the top concept only appears in existential restrictions, i.e. in the form $\exists r.\top$.

$\bot$ symbols are eliminated by the following rewrite rules:

$$\begin{aligned} C_1 \sqcap \dots \sqcap C_{i-1} \sqcap \bot \sqcap C_{i+1} \sqcap \dots \sqcap C_n &\rightarrow \bot \\ C_1 \sqcup \dots \sqcup C_{i-1} \sqcup \bot \sqcup C_{i+1} \sqcup \dots \sqcup C_n &\rightarrow C_1 \sqcup \dots \sqcup C_{i-1} \sqcup C_{i+1} \sqcup \dots \sqcup C_n \\ \exists r.\bot &\rightarrow \bot \end{aligned}$$

These steps also preserve equivalence. After exhaustively applying these steps we either get the $\bot$ symbol itself (which is in $S_\downarrow$) or the $\bot$ symbol only appears in universal restrictions, i.e. in the form $\forall r.\bot$.

Next we have to eliminate disjunctions, which do not satisfy Definition 4.20. Say we have such a disjunction $C_1 \sqcup \dots \sqcup C_m$. Then there is a C_i ($1 \leq i \leq m$), which is a conjunction consisting only of disjunctions. Without loss of generality we assume $i = 1$ (the order of elements in a disjunction is not important), i.e. we can write $C_1 = D_1 \sqcap \dots \sqcap D_n$ and $D_1 = E_1 \sqcup \dots \sqcup E_p$. This means we can apply the following equivelance preserving rewriting rule:

$$\begin{aligned} &((E_1 \sqcup \dots \sqcup E_p) \sqcap D_2 \sqcap \dots \sqcap D_n) \sqcup C_2 \sqcup \dots \sqcup C_m \quad \rightarrow \\ &\qquad (E_1 \sqcap D_2 \sqcap \dots \sqcap D_n) \sqcup \dots \sqcup (E_p \sqcap D_2 \sqcap \dots \sqcap D_n) \sqcup C_2 \dots \sqcup C_m \end{aligned}$$

Note that E_i ($1 \leq i \leq p$) cannot be a disjunction. Let C_1' be the replacement of C_1 after applying the rewriting rule. Obviously C_1' is no more a disjunction where an element is a conjunction of disjunctions (because any E_i is not a disjunction). If we apply this rule to all applicable C_i ($1 \leq i \leq m$), then we obtain a concept C'' equivalent to $C_1 \sqcup \dots \sqcup C_m$, which is in $S_\downarrow$.

Hence we have shown that we can construct a concept $C'' \equiv C' \equiv C$ with $C'' \in S_\downarrow$, which completes the proof. ■

Proposition 4.22 (weak completeness of $\rho_\downarrow$)
$\rho_\downarrow$ is weakly complete.

Proof We have to show that for any concept C with $C \sqsubset_{\mathcal{T}} \top$ a concept E with $E \equiv C$ can be reached from $\top$ by $\rho_{\downarrow}$. Due to Lemma 4.21 it is sufficient to show that all concepts in $S_{\downarrow}$ can be reached from $\top$ by $\rho_{\downarrow}$.

We claim that $\rho_{\downarrow}^*(\top) = S_{\downarrow}$ and prove this by induction over the structure of concepts in $S_{\downarrow}$ (see Definition 4.20). The bottom concept itself can be reached by a one-step refinement of the $\top$ symbol, so we just have to analyze the elements in $S'_{\downarrow}$.

- Induction Base: An atomic concept A can be reached from $\top$ by a refinement chain of the following form:

  $$\top \overset{\top}{\rightsquigarrow} A_1 \overset{A}{\rightsquigarrow} \dots \overset{A}{\rightsquigarrow} A_n \overset{A}{\rightsquigarrow} A$$

  The operator descends the subsumption hierarchy and reaches A in a finite number of steps (there are only finitely many atomic concepts). Negated atomic concepts can be handled analogously.

  $\forall r.\bot$ can also be reached by descending the subsumption hierarchy:

  $$\top \overset{\top}{\rightsquigarrow} \forall r.A_1 \overset{A}{\rightsquigarrow} \dots \overset{A}{\rightsquigarrow} \forall r.A_n \overset{A}{\rightsquigarrow} \forall r.\bot$$

  $\exists r.\top$ can be reached by a one step refinement from $\top$.

- Induction Step:
  - $\exists r.C$: We have $\top \overset{\top}{\rightsquigarrow} \exists r.\top$ and by induction we can reach C from $\top$ by $\rho_{\downarrow}$.
  - $\forall r.C$: We have $\forall r.C \in \rho_{\downarrow}^*(\top)$ if $C \in \rho_{\downarrow}^*(\top)$, which is true by induction.
  - $C_1 \sqcup \dots \sqcup C_m$: We will look at the elements of the disjunction separately. We have to show that for any i we have $C_i \in \rho^*(m)$ for an $m \in M$ (where M is defined as in the definition of $\rho_{\downarrow}$). Without loss of generality we show this for C_1, i.e. $i = 1$. We do a case distinction based on the structure of C_1 (obviously C_1 is not a disjunction).
    - C_1 is an atomic concept A: In this case we pick a most general atomic concept $A_1 \in S$ and refine it:

      $$A_1 \overset{A}{\rightsquigarrow} \dots \overset{A}{\rightsquigarrow} A_n \overset{A}{\rightsquigarrow} A$$

    - C_1 is a negated atomic concept $\neg A$: We pick a most special atomic concept $A_1 \in S$ and refine it:

      $$A_1 \overset{\neg A}{\rightsquigarrow} \dots \overset{\neg A}{\rightsquigarrow} A_n \overset{\neg A}{\rightsquigarrow} A$$

    - $C_1 = \forall r.\bot$: We refine to $\forall r.A_1$, where A_1 is itself a refinement of the top concept. Then we can refine to $\forall r.\bot$:

      $$\forall r.A_1 \overset{A}{\rightsquigarrow} \dots \overset{A}{\rightsquigarrow} \forall r.A_n \overset{A}{\rightsquigarrow} \forall r.\bot$$

- $*$ $C_1 = \exists r.\top$: $\exists r.\top$ is in M.
- $*$ $C_1 = \forall r.D$: By induction, $D \in \rho_\downarrow^*(\top)$, so there is a concept $\forall r.E_1 \in S$, which we can refine to $\forall r.D$:

$$\forall r.E_1 \leadsto \ldots \leadsto \forall r.E_n \leadsto \forall r.D$$

- $*$ $C_1 = \exists r.D$: We choose $\exists r.\top \in M$ and by induction $D \in \rho_\downarrow^*(\top)$.
- $*$ $C_1 = D_1 \sqcap \cdots \sqcap D_n$: By Definition of $S_\downarrow$ (Definition 4.20), we know that there exists a j, such that D_j is not a disjunction (and obviously also not a conjunction). So there is one D_j of the form $\exists r.\top$, $\bot$, A, $\neg A$, $\exists r.C$ or $\forall r.C$. These can be produced exactly as we have shown in the previous cases. For any of these constructs $\rho_\downarrow$ allows to extend these constructs to a conjunction with elements in $\rho_\downarrow(\top)$. By induction, we know for all i ($1 \leq i \leq n$) that $D_i \in \rho_\downarrow^*(\top)$. So we can produce a concept weakly equal to C_1 by first creating D_j and then extending this to a conjunction by generating all $D_i's$ ($1 \leq i \leq n, i \neq j$).

– $C_1 \sqcap \cdots \sqcap C_m$: By induction, we know $C_1 \in \rho^*(\top)$, so we first create C_1 and then add all other elements to the conjunction:

$$\top \overset{\top}{\leadsto} D_1 \ldots \leadsto C_1 \overset{\sqcap}{\leadsto} C_1 \sqcap D_2 \leadsto \ldots \leadsto C_1 \sqcap C_2 \leadsto \ldots \leadsto C_1 \sqcap \cdots \sqcap C_m \quad \blacksquare$$

It turns out that $\rho_\downarrow$ is even complete:

Proposition 4.23 (completeness of $\rho_\downarrow$)
$\rho_\downarrow$ is complete.

Proof Let C and D be arbitrary $\mathcal{ALC}$ concepts in $S_\downarrow$ with $C \sqsubset D$. To prove completeness of $\rho_\downarrow$ we have to show that there exists a concept E with $E \equiv C$ and $E \in \rho_\downarrow^*(D)$. $E = D \sqcap C$ satisfies this property. We obviously have $E = D \sqcap C \equiv C$, because of $C \sqsubset D$. We know that $\rho_\downarrow$ allows to extend concepts conjunctively by refinements of the top concept. Hence we know that $D \sqcap C$ can be reached from D for any concept C by the weak completeness result for $\rho_\downarrow$. Thus $\rho_\downarrow$ is complete. $\blacksquare$

The completeness of $\rho_\downarrow$ is a by-product and not a design decision, i.e. we automatically can derive completeness from weak completeness if we allow to extend concepts conjunctively (which is usually a good idea). For instance we cannot reach A_1 from $A_1 \sqcup A_2$ (because we cannot drop elements of disjunctions), but instead we can only reach $(A_1 \sqcup A_2) \sqcap A_1$. So $\rho_\downarrow$ is complete, but it is not always possible to reach the shortest concepts. (This is intentional since we will later see that we need the property that applications of $\rho_\downarrow$ cannot produce shorter concepts.) The important property with respect to the top-down learning algorithm we will design later is weak completeness.

4.5 Achieving Properness

The operator $\rho_\downarrow$ is not proper, for instance it allows the refinement (A_1 is a most general atomic concept):

$$\top \overset{\top}{\rightsquigarrow} \exists r.\top \sqcup \forall r.A_1 \quad (\equiv \top) \tag{1}$$

There is no structural subsumption algorithm for $\mathcal{ALC}$, which indicates that it is hard to define a proper operator just by syntactic rewriting rules. One could try to modify $\rho_\downarrow$, such that it becomes a proper operator. Unfortunately this is likely to lead to the weak incompleteness of the operator. Say we disallow refinement step (1). Consider the following refinement chain (A_2 is a most general atomic concept):

$$\top \overset{\top}{\rightsquigarrow} \exists r.\top \sqcup \forall r.A_1 \overset{\top}{\rightsquigarrow} \exists r.A_2 \sqcup \forall r.A_1 \tag{2}$$

If we disallow the first step we have to ensure that we can reach $\exists r.A_2 \sqcup \forall r.A_1$ from $\top$, otherwise the operator is weakly incomplete. This is just one example case we have to take care of. In particular there can be infinite chains of improper refinements:

$$\top \overset{\top}{\rightsquigarrow} \exists r.\top \sqcup \forall r.A_1 \overset{\top}{\rightsquigarrow} \exists r.(\exists r.\top \sqcup \forall r.A_1) \sqcup \forall r.A_1 \overset{\top}{\rightsquigarrow} \ldots \tag{3}$$

This example illustrates that one would have to allow very complex concepts to be generated as refinements of the top concept, if one wants to achieve weak completeness and properness.

There is a way to solve the problem: Instead of modifying $\rho_\downarrow$ directly we allow it to be improper, but consider the closure $\rho_\downarrow^{cl}$ of $\rho_\downarrow$ (see also Badea and Nienhuys-Cheng, 2000).

Definition 4.24 ($\rho_\downarrow^{cl}$)
$\rho_\downarrow^{cl}$ is defined as follows: $D \in \rho_\downarrow^{cl}(C)$ iff there exists a refinement chain

$$C \rightsquigarrow_{\rho_\downarrow} C_1 \rightsquigarrow_{\rho_\downarrow} \ldots \rightsquigarrow_{\rho_\downarrow} C_n = D$$

such that $C \not\equiv D$ and $C_i \equiv C$ for $i \in \{1, \ldots, n-1\}$. □

$\rho_\downarrow^{cl}$ is proper by definition. It also inherits the weak completeness of $\rho_\downarrow$ since we do not disallow any refinement steps, but only check if they are improper.

However, it is necessary to show that $\rho_\downarrow^{cl}$ is a meaningful operator, which we will do in the sequel. We know that $\rho_\downarrow$ is an infinite operator, so it is clear that we cannot consider all refinements of a concept at a time. The refinement operator spans a search tree of $\mathcal{ALC}$ concepts, which has an infinite branching factor. Therefore in practice we will always compute all refinements of a concept up to a fixed length. A flexible algorithm will allow this length limit to be increased if necessary. In this sense an infinite operator is not a big problem, since we are more interested in smaller concepts anyway (they generalize better to unseen examples, i.e. they are less likely to overfit the given data). However, we have to make sure that all refinements up to a given length are computable in finite time. To show this we need the following lemma.

Lemma 4.25 ($\rho_\downarrow$ does not reduce length)

1. *$D \in \rho_\downarrow(C) \implies |D| \geq |C|$*
2. *There are no infinite refinement chains of the form*

$$C_1 \rightsquigarrow_{\rho_\downarrow} C_2 \rightsquigarrow_{\rho_\downarrow} \dots$$

with $|C_1| = |C_2| = \dots$. (Or equivalently: After a finite number of steps we will reach a longer concept.)

Proof To show the first statement we need to observe the steps, which are performed by $\rho_\downarrow$. We can see that $\rho_\downarrow$ does one of four things in each refinement step:

1. add an element conjunctively ($\overset{\sqcap}{\rightsquigarrow}$)
2. refine the top concept ($\overset{\top}{\rightsquigarrow}$)
3. generalize an atomic concept ($\overset{A}{\rightsquigarrow}$)
4. generalize a negated atomic concept ($\overset{\neg A}{\rightsquigarrow}$)

Steps 1 and 2 obviously result in a concept with greater length. Step 3 and 4 result in a concept with the same length. This proves claim 1.

Claim 2 follows from the fact that there is just a finite number of atomic concepts (N_C is finite) and there are only finitely many occurences of an atomic concept within any $\mathcal{ALC}$ concept. Hence there are no infinite refinement chains using only steps 3 or 4. Thus after a finite number of refinements step 1 or 2 is used, which produce a longer concept. ■

Proposition 4.26 (usefulness of $\rho_\downarrow^{cl}$)
For any concept C in negation normal form and any natural number n the set

$$\{D \mid D \in \rho_\downarrow^{cl}(C), |D| \leq n\}$$

can be computed in finite time.

Proof Because of Lemma 4.25 we know that for any concept D in the set there exists an m such that $|D| > |C|$ with $D \in \rho_\downarrow^m(C)$. Obviously a concept has only finitely many refinements up to a fixed length. If we consider all refinement chains of a concept C by $\rho_\downarrow$ up to length n as a tree, then this tree is finite (there are only finitely many concepts of length $\leq n$ and any such concept can be reached by a finite refinement chain). The set $\{D \mid D \in \rho_\downarrow^{cl}(C), |D| \leq n\}$ is a subset of the nodes of this tree. Hence it can be computed in finite time. ■

Proposition 4.26 essentially states that using the closure of $\rho_\downarrow$ as an operator is useful, i.e. there is no risk to run in an infinite loop when computing it up to a certain length of the refinements. Hence we can use $\rho_\downarrow^{cl}$ in a learning algorithm.

4.6 Removing Redundancies

So far we have created a weakly complete and proper refinement operator. The next goal is to remove redundancies. The next example shows that $\rho_\downarrow$ and $\rho_\downarrow^{cl}$ are redundant. (This already follows from Proposition 4.12, but we show an example explicitly.)

Example 4.27 (redundancy of $\rho_\downarrow$ and $\rho_\downarrow^{cl}$)
These are two refinement chains of $\rho_\downarrow$ and $\rho_\downarrow^{cl}$ (both operators can produce these chains):

$$\top \overset{\top}{\rightsquigarrow} A_1 \overset{\sqcap}{\rightsquigarrow} A_1 \sqcap A_2$$
$$\top \overset{\top}{\rightsquigarrow} A_2 \overset{\sqcap}{\rightsquigarrow} A_2 \sqcap A_1$$

$A_1 \sqcap A_2$ and $A_2 \sqcap A_1$ are weakly equal ($A_1 \sqcap A_2 \simeq A_2 \sqcap A_1$). So there is a refinement chain from $\top$ to $A_1 \sqcap A_2$, which does go through A_1 and another refinement chain from $\top$ to a concept weakly syntactically equivalent to $A_1 \sqcap A_2$, which does not go through A_1. This satisfies the definition of redundancy (Definition 4.6, page 23). □

In the proof of Proposition 4.15 (page 31) we have seen that it is possible to define a weakly complete, non-redundant, and proper refinement operator as follows:
Let S be a maximal subset of $\{C \mid C \not\equiv_\mathcal{T} \top\}$ with $C_1, C_2 \in S \implies C_1 \not\simeq C_2$.

$$\rho(C) = \begin{cases} S & \text{if } C = \top \\ \emptyset & \text{otherwise} \end{cases}$$

This operator is clearly impractical since it generates arbitrarily complex concepts from a single concept instead of using syntactic rewriting rules to modify the input concept.

This rises the question whether there exists a weakly complete, proper, and non-redundant $\mathcal{ALC}$ refinement operator, which can be considered practically useful. One idea is to modify $\rho_\downarrow^{cl}$ by disallowing refinement steps, which cause redundancies. The following example illustrates that this is problematic.

Example 4.28 (problems with non-redundant operators)
The following two refinement chains can be produced by $\rho_\downarrow^{cl}$:

$$\top \rightsquigarrow \forall r_1.A_1 \sqcup \forall r_2.A_1 \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.A_1 \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.(A_1 \sqcap A_2)$$
$$\top \rightsquigarrow \forall r_1.A_1 \sqcup \forall r_2.A_1 \rightsquigarrow \forall r_1.A_1 \sqcup \forall r_2.(A_1 \sqcap A_2) \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.(A_1 \sqcap A_2)$$

Again, we see that $\rho_{\downarrow}^{cl}$ is redundant. However, the interesting aspect of this example is that to avoid redundancies we may consider disallowing one of the refinement steps:

$$\forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.A_1 \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.(A_1 \sqcap A_2)$$

$$\forall r_1.A_1 \sqcup \forall r_2.(A_1 \sqcap A_2) \rightsquigarrow \forall r_1.(A_1 \sqcap A_2) \sqcup \forall r_2.(A_1 \sqcap A_2)$$

Both steps result in the same concept and thereby lead to the redundancy of $\rho_{\downarrow}^{cl}$. We cannot disallow both steps since this would lead to the incompleteness of $\rho_{\downarrow}^{cl}$. Looking closer at both refinement steps, we see that both refine an element of a disjunction by rewriting A_1 to $A_1 \sqcap A_2$. This means that if we want to disallow one of both refinement steps, we need to modify the way the operator handles disjunctions. Instead of delegating the operator to elements of a disjunction like $\rho_{\downarrow}^{cl}$ ($\rho_{\downarrow}^{cl}$ can refine disjunctions by refining elements of the disjunction) we need to take the structure of each element of the disjunction into account. However, this is impractical, because $\mathcal{ALC}$ concepts can have an arbitrarily deeply nested structure. □

Example 4.28 gives an intuition why a weakly complete, proper, and non-redundant $\mathcal{ALC}$ refinement operator cannot handle disjunctions by refining elements of a disjunction (which can be considered the natural way to handle disjunctions).

Another reason why weakly complete, proper, and non-redundant $\mathcal{ALC}$ refinement operators are problematic in practice can be found in the proof of Proposition 4.23 on page 39: If a downward refinement operator ρ allows to extend a concept conjunctively, i.e. for all $\mathcal{ALC}$ concepts C we have $C \sqcap \top \in \rho(C)$ or $C \sqcap \rho(\top) \in \rho(C)$, then completeness is automatically derived from weak completeness. By Propositon 4.12 a complete operator is always redundant. Hence a weakly complete, proper, and non-redundant operator cannot extend concepts conjunctively in the way described above.

As a conclusion there are two ways to handle redundancy: The first one is to modify $\rho_{\downarrow}^{cl}$ in such a way that redundancy is reduced – but it is problematic to remove redundancy completely in a practical way. The second way is to let $\rho_{\downarrow}^{cl}$ be redundant, but remove or mark all occuring redundant concepts by choosing an appropriate search strategy for the learning algorithm. A combination of both approaches is also possible.

We will describe the second approach in more detail: A learning algorithm can be constructed as a combination of a refinement operator, which defines how the search tree can be build, and a search algorithm, which controls how the tree is traversed. The search algorithm specifies which nodes have to be expanded. (Expanding a node roughly corresponds to applying the refinement operator to the $\mathcal{ALC}$ concept represented by this node.) Whenever the search algorithm encounters a node in the search tree, i.e. an $\mathcal{ALC}$ concept, it can check whether a weakly equal concept already exists in the search tree. If yes, then this node is ignored, i.e. it will not be expanded further and it will not be evaluated.

The first observation is that this approach obviously completely removes redundancies. Each concept exists at most once in the search tree. (More exactly: For each concept there is at most representative of the equivalence class of weakly syntactical equal concepts in the search tree which is evaluated.)

The second observation is that we can still reach any potential solution. $\rho_{\downarrow}^{cl}$ handles weakly equal concepts in the same way, i.e. $\rho_{\downarrow}^{cl}(C) \simeq \rho_{\downarrow}^{cl}(D)$ if $C \simeq D$. This means that ignoring a concept if a weakly equal concept already exists in the search tree does not influence the set of concepts we can reach in the search tree (up to weak equivalence).

The third observation is that this approach is computationally expensive. Hence we considered it worthwhile to investigate how it can be handled as efficient as possible.

Summary: We gave reasons why weakly complete and non-redundant $\mathcal{ALC}$ refinement operators are impractical. We have then shown that a way to avoid redundancy is to have a search strategy, which checks for every newly created node whether a weakly equal concept already exists in the search tree.

The next step is to analyse how this can be done. We first need to define an algorithm, which decides $C \simeq D$ given two concepts C and D.

Algorithm 4.29 (checking weak equality)
Function Name: *checkEquality*
Input: $\mathcal{ALC}$ concepts C and D in negation normal form
Output: yes or no
We make a case distinction on the structure of C:

- $C = \bot$: return yes iff $D = \bot$
- $C = \top$: return yes iff $D = \top$
- $C = A$ ($A \in N_C$): return yes iff $D = A$
- $C = \neg A$ ($A \in N_C$): return yes iff $D = \neg A$
- $C = \exists r.C'$: return yes iff $D = \exists r.D'$ and *checkEquality*(C', D')
- $C = \forall r.C'$: return yes iff $D = \forall r.D'$ and *checkEquality*(C', D')
- $C = C_1 \sqcap \dots \sqcap C_n$: if D is not of the form $D_1 \sqcap \dots \sqcap D_n$ then return no otherwise:
  - call *checkEquality*(C_1, D_i) starting from $i = 1$ to $i = n$ until yes is returned; if none of the tests return yes then return no as result
  - set $C' = C_2 \sqcap \dots \sqcap C_n$ and $D' = D_1 \sqcap \dots \sqcap D_{i-1} \sqcap D_{i+1} \sqcap \dots \sqcap D_n$; call *checkEquality*$(C', D')$
- $C = C_1 \sqcup \dots \sqcup C_n$: analogously to the previous case

The equality check algorithm is rather simple. However, it is not efficient since in conjunctions and disjunctions we have the problem that we have to guess which pairs of elements are equal (case $C = C_1 \sqcap \dots \sqcap C_n$ in the algorithm). One way to solve this problem is to define an ordering over concepts and require the elements of disjunctions and conjunctions to be ordered. This would eliminate the guessing step and allow to check weak equality in linear time.

Definition 4.30 (ordering concepts)
We define a relation $\preceq$ over $\mathcal{ALC}$ concepts in negation normal form as follows:

We assume that roles are ordered in a list $[r_1, \dots, r_t]$ and atomic concepts in a list $[A_1, \dots, A_u]$ such that for $i < j$ we have $A_i \sqsubseteq A_j$.

Let $A, A_1, A_2 \in N_C \cup \{\top, \bot\}$, $r \in N_R$, and $C, D, C_1, \dots, C_n, D_1, \dots, D_n$ $(m, n > 1)$ be arbitrary $\mathcal{ALC}$ concepts in negation normal form. Then we have:

- $\bot \preceq A \preceq \top \preceq \neg A \preceq C_1 \sqcap \cdots \sqcap C_n \preceq D_1 \sqcup \cdots \sqcup D_n \preceq \exists r.C \preceq \forall r.D$ and $\forall r.D \not\preceq \exists r.C \not\preceq D_1 \sqcup \cdots \sqcup D_n \not\preceq C_1 \sqcap \cdots \sqcap C_n \not\preceq \neg A \not\preceq \top \not\preceq A \not\preceq \bot$ (This defines the ordering over different syntactic structures.)
- $A_i \preceq A_j$ iff $i \leq j$
- $\neg A_i \preceq \neg A_j$ iff $A_i \preceq A_j$
- $C_1 \sqcap \cdots \sqcap C_m \preceq D_1 \sqcap \cdots \sqcap D_n$ iff $m < n$ or $m = n$ and $((C_1 \neq D_1$ and $C_1 \preceq D_1)$ or $(C_1 = D_1$ and $C_2 \sqcap \cdots \sqcap C_m \preceq D_2 \sqcap \cdots \sqcap D_n))$
- $C_1 \sqcup \cdots \sqcup C_m \preceq D_1 \sqcup \cdots \sqcup D_n$ iff $m < n$ or $m = n$ and $((C_1 \neq D_1$ and $C_1 \preceq D_1)$ or $(C_1 = D_1$ and $C_2 \sqcup \cdots \sqcup C_m \preceq D_2 \sqcup \cdots \sqcup D_n))$
- $\exists r_i.C \preceq \exists r_j.D$ iff $i < j$ or $(i = j$ and $C \preceq D)$
- $\forall r_i.C \preceq \forall r_j.D$ iff $i < j$ or $(i = j$ and $C \preceq D)$ □

Proposition 4.31 (properties of $\preceq$)

1. *$C \preceq D$ and $D \preceq C$ implies $C = D$.*
2. *$\preceq$ is a linear order.*

PROOF The proof will not be presented in full detail, but rather as a proof sketch.

1. $C \preceq D \wedge D \preceq C \implies C = D$: From the definition of $\preceq$ it is clear that the premise $C \preceq D \wedge D \preceq C$ can only be true if C and D are equal with respect to their outmost syntactic structure (e.g. they are both disjunctions). If C and D are atomic concepts, then the equality is obvious $(i \leq j \wedge j \leq i \implies i = j)$. If $C = D = \bot$ or $C = D = \top$, then the claim obviously also holds. For other syntactic constructs the equality of C and D can easily be shown by induction.
2. - $\preceq$ is total: This holds since for two aribtrary concepts C and D in any rule in Definition 4.30 either the condition for $C \preceq D$ or $D \preceq C$ is satisfied. This is obvious for atomic concepts, $\top$ and $\bot$, and can easily be shown by induction for other concepts.

- $\preceq$ is reflexive: For any concept C of any syntactic structure $C \preceq C$ is satisfied (which can be verified by looking at all cases in the definition).
- $\preceq$ is antisymmetric: $C \preceq D$ and $D \preceq C$ implies $C = D$ according to (1). Hence there are no concepts C, D with $C \neq D$, $C \preceq D$, and $D \preceq C$.
- $\preceq$ is transitive: Assume for three concepts C, D, and E we have $C \preceq D$ and $D \preceq E$. If C, D, and E do not have the same outmost syntactic structure, then it is clear that we have $C \preceq E$ (by definition of the ordering over different syntactic structures). However, if C, D, and E have the same outmost syntactic structure, then $C \preceq E$ is not hard to show, because in the rules in Definition 4.30 the relations $\leq$, $=$ (transitive), and $\preceq$ (transitive by induction) are used. ■

Lemma 4.32 (deciding $C \preceq D$)
$C \preceq D$ can be decided in polynomial time.

Proof An algorithm for deciding $C \preceq D$ can be defined by simply following the rules in Definition 4.30. The input size of the problem $C \preceq D$ is $|C| + |D| = s$. We refer to one time step as the comparision of two symbols (e.g. $\sqcap$ and $\sqcup$). If C and D have different outmost syntactic structures, then the problem is decided in exactly one time step. If $C = D = \top$, $C = D = \bot$ or C and D are atomic concepts, then the problem is also decided in a single step. If C and D are of the form $\neg A$, $\exists r.E$ or $\forall r.E$, then we need one time step plus the time for deciding the smaller problems for the subconcepts, which can be solved in polynomial time by induction. If C and D are conjunctions or negations, then we may additionally also have to decide whether $C_1 = D_1$ holds for elements C_1 of C and D_1 of D. This check can be done in linear time (in less than s time steps). The other checks (in the cases $C_1 \sqcap \cdots \sqcap C_n$, $C_1 \sqcup \cdots \sqcup C_n$, $\exists r.C$, $\forall r.C$) are polynomial by induction. Hence the overall complexity of the decision is polynomial. ■

We have introduced a linear order $\preceq$ over concepts in negation normal form and have shown that $C \preceq D$ is decidable in polynomial time. We can use this linear order to sort the elements of disjunctions and conjunctions. We say that the resulting concept is ordered.

Definition 4.33 (ordered negation normal form)
A concept C is in ordered negation normal form iff it is in negation normal form and the following conditions hold:

- If C is of the form $\exists r.D$, then D is in ordered negation normal form.
- If C is of the form $\forall r.D$, then D is in ordered negation normal form.

- If C is of the form $C_1 \sqcap \cdots \sqcap C_n$, then $C_1 \preceq \cdots \preceq C_n$ and $C_1, \ldots, C_n$ are in ordered negation normal form.
- If C is of the form $C_1 \sqcup \cdots \sqcup C_n$, then $C_1 \preceq \cdots \preceq C_n$ and $C_1, \ldots, C_n$ are in ordered negation normal form. □

Proposition 4.34 (transformation to ordered negation normal form)
A concept in negation normal form can be transformed to a concept in ordered negation normal form in polynomial time.

Proof We will first define an algorithm for the transformation to an ordered concept. The function *transform*, which takes as input a concept in negation normal form and returns a concept in ordered negation normal form is defined as follows:

Algorithm 4.35 (transformation to ordered negation normal form)
Function Name: *transform*
Input: an $\mathcal{ALC}$ concept in negation normal form
Output: a weakly equal $\mathcal{ALC}$ concept in ordered negation normal form
We do a case distinction on the structure of C:

- $C \in \{\bot, \top, A, \neg A\}$ ($A \in N_C$): return C
- $C = \exists r.C'$: return $\exists r.\mathit{transform}(C')$
- $C = \forall r.C'$: return $\forall r.\mathit{transform}(C')$
- $C = C_1 \sqcap \cdots \sqcap C_n$: transform all elements of the disjunction and then sort them (e.g. by Quicksort) according to $\preceq$
- $C = C_1 \sqcup \cdots \sqcup C_n$: analogously to the previous case

Obviously this algorithm is correct, i.e. the resulting concept is weakly equal to the input concept and ordered.

Let C with $|C| = n$ be the input of the algorithm. The *transform* function is called less than n-times. From a complexity point of view the only interesting case is the handling of disjunctions and conjunctions (all other cases are trivial).

C does not contain more than n disjunctions (concjunctions) and each disjunction (conjunction) has a length less than n. Sorting a disjunction (conjunction) by e.g. Quicksort is done with $O(n^2)$ comparision operations (comparing means to decide $C \preceq D$ for concepts C and D). In Lemma 4.32, we have shown that such a comparision can be done in polynomial time. Hence the overall complexity of the algorithm is also polynomial. ■

We have shown that ordering concepts greatly increases the efficiency of redundancy elimination. By ordering concepts, it is possible to check weak equality in linear time, because we can avoid the guessing step in Algorithm 4.29 (page 44). This is because for concepts in ordered negation normal form checking weak equality is the same like checking equality. Note that the transformation of a concept to an ordered concept has to be done only once when inserting it into the search tree. After that it will speed up all weak equality checks.

The most important question to ask is whether it is worth to check for redundancies. We have shown that, given a concept and a search tree, we can decide if a weakly equal concepts exists in the search tree in polynomial time. However, the search tree can grow exponentially with the size of concepts. It is still worth to eliminate redundancy, because for measuring whether a concept is good or not we need to determine instances of the concept. Depending on the description language used for the background knowledge this problem is at least in PSPACE (for $\mathcal{ALC}$), but can be in higher complexity classes (NEXPTIME for $\mathcal{SHOIN}(D)$ and OWL-DL). Redundancy elimination can avoid many instance checks. Also note that when we do not eliminate redundancies the search tree may contain large subtrees of weakly equal concepts, i.e. we have different subtrees in the search tree which are identical up to weak equality. In this sense redundancy elimination also avoids the creation of further redundant concepts.

In practice we would not perform the weak equality check on all concepts of the search tree, but rather only on those with the same length, depth etc. One might think that an even better method is to start from the root of the search tree and only search those paths, which can lead to a weakly equal concept. For a concept C and a node in the search tree, which represents a concept D this means we have to check whether there can be a concept E with $E \simeq C$ and $E \in \rho^*(D)$. However, such a check is not efficient since the transformation to ordered concepts can change the ordering of elements in conjunctions and disjunctions (not performing the transformation does not help since then the necessary weak equality check is not efficient). A solution for this problem would be to modify the operator such that changes in the order of elements in conjunctions and disjunctions never occur, i.e. applying the operator to a concept in ordered negation normal form results in concepts in ordered negation normal form. This is possible, but not desirable, for instance we would have to disallow the traversal of the subsumption hierarchy. We will illustrate this in a simple example. Let $N_C = \{A_1, A_2, A_3\}$, where the ordering of concept names is $[A_1, A_2, A_3]$ with $A_3 \sqsubset A_1$, $A_2 \not\sqsubseteq A_1$ and $A_1 \not\sqsubseteq A_2$. Consider the refinement chain:

$$\top \rightsquigarrow A_1 \sqcup A_2 \rightsquigarrow A_3 \sqcup A_2$$

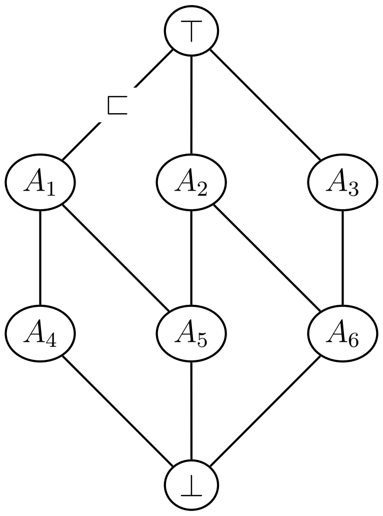
We can reach A_3 by refining A_1, but not by refining A_2. So in this case we get a concept, which is not in ordered negation normal form ($A_3 \sqcup A_2$).

4.7 Further Optimisations

In this section $\rho_\downarrow$ and thereby $\rho^{cl}_\downarrow$ will be improved while preserving their basic properties.

Improving the Traversal of the Subsumption Hierarchy

The operator $\rho^{cl}_\downarrow$ allows to traverse the subsumption hierarchy of atomic concepts. This is done by refining an atomic concept A to an atomic concept A' such that $A' \sqsubset A$ and there is no $A'' \in N_C$ with $A' \sqsubset A'' \sqsubset A$. A drawback of this approach is that there can be several ways to reach a concept. As an example let the following subsumption hierarchy be given:



We can reach A_5 by two refinement chains in $\rho^{cl}_\downarrow$:

$$\top \overset{\top}{\rightsquigarrow} A_1 \overset{A}{\rightsquigarrow} A_5$$
$$\top \overset{\top}{\rightsquigarrow} A_2 \overset{A}{\rightsquigarrow} A_5$$

However, it is not necessary that we reach A_5 by more than one chain and indeed the redundancy elimination algorithms we have presented would mark one of both results as redundant. There is an easy way to avoid this problem by transforming the subsumption graph into two trees: one tree for downward refinement of atomic concepts and one tree for upward refinement of atomic concepts in negated atomic concepts. In the tree for downward refinement for each atomic concept, which has more than one more general neighbour we just pick one neighbour and erase all others. Additionally we can remove the $\bot$ symbol, because $\rho_\downarrow$ never refines an atomic concept to the $\bot$ symbol. The resulting tree has the property that there is exactly one path from each node to the $\top$ symbol. (There way at least one path from each node to $\top$ before the transformation and we erased all paths except one.)

The tree for upward refinement can be constructed analogously. In this case we need to look at more special neighbours and we can remove the $\top$ concept.

The algorithm is not deterministic, so there can be more than one solution. These are two possible resulting trees:

$$\begin{tikzcd}
& \top \arrow[dl] \arrow[d] \arrow[dr] & \\
A_1 \arrow[d] & A_2 \arrow[d] & A_3 \arrow[d] \\
A_4 & A_5 & A_6
\end{tikzcd}
\qquad
\begin{tikzcd}
A_1 & A_2 & A_3 \\
A_4 \arrow[u] & A_5 & A_6 \arrow[ul] \arrow[u] \\
& \bot \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Now we can modify $\rho^{cl}_{\downarrow}$ such that it uses these trees instead of a direct use of the subsumption hierarchy. If an atomic concept A as a subconcept of a concept C is refined, then we use the first tree for downward refinement and if a negated atomic concept A is refined we use the second tree for upward refinement. This reduces the number of refinements and therefore the number of redundancy checks. The resulting operator is still weakly complete, since we only removed redundant refinement chains.

Using $\exists r.(C \sqcup D) \equiv \exists r.C \sqcup \exists r.D$ and $\forall r.(C \sqcap D) \equiv \forall r.C \sqcap \forall r.D$

The equivalences $\exists r.(C \sqcup D) \equiv \exists r.C \sqcup \exists r.D$ and $\forall r.(C \sqcap D) \equiv \forall r.C \sqcap \forall r.D$ can be used to modify $\rho_{\downarrow}$ without losing weak completeness.

Disjunctions in $\rho_{\downarrow}$ are only introduced in refinements of the top concept. The only existential value restrictions in these disjunctions are of the form $\exists r.\top$ for $r \in N_R$. The equivalence $\exists r.(C \sqcup D) \equiv \exists r.C \sqcup \exists r.D$ says that it is not necessary to allow several disjuncts of the form $\exists r.\top$ for a fixed role r, because we can always reach an equivalent concept by only introducing it once. Therefore we can restrict $\rho_{\downarrow}$ to produce $\exists r.\top$ only at most once per role as element of the disjunction in the refinement of the top concept, without losing weak completeness.

$\rho_{\downarrow}$ allows to refine a concept C by extending it conjunctively. If C is of the form $\forall r.D$ or of the form $C_1 \sqcap \dots \sqcap \forall r.D \sqcap \dots \sqcap C_n$, then we can restrict $\rho_{\downarrow}$ to disallow adding an element of the form $\forall r.E$ (E is an arbitrary $\mathcal{ALC}$ concept). Again, the resulting operator is still weakly complete.

By using the equalities $\exists r.(C \sqcup D) \equiv \exists r.C \sqcup \exists r.D$ and $\forall r.(C \sqcap D) \equiv \forall r.C \sqcap \forall r.D$ as described above, we have reduced the number of possible refinements, but preserved (weak) completeness.

4.8 Creating a Full Learning Algorithm

A learning algorithm can be created by combining a refinement operator with a search algorithm. In this section we will show how to combine the refinement operator $\rho^{cl}_{\downarrow}$ with a redundancy-eliminating heuristic search algorithm.

Learning concepts in Description Logics is a search process. The refinement operator $\rho^{cl}_{\downarrow}$ is used for building the search tree, while a heuristic decides which nodes to expand.

Major decision criteria are obviously the examples covered by a concept. We will make this more explicit by definining the notion of quality.

Definition 4.36 (quality)
Let $\mathcal{K}$ be a knowledge base, E^- the set of negative examples, and E^+ the set of positive examples of a learning problem. The *quality of a concept* C is a function, which maps a concept to an element of $\mathcal{Q}$ with $\mathcal{Q} = \{0, \dots, -|E^-|\} \cup \{tw\}$, defined by:

$$quality(C) = \begin{cases} tw & \text{if there is an } e \in E^+ \\ & \text{with } \mathcal{K} \cup \{C\} \not\models e \\ -|\{e \mid e \in E^- \text{ and } \mathcal{K} \cup \{C\} \models e\}| & \text{otherwise} \end{cases} \qquad \square$$

The quality of a concept is "tw" if it is too weak, i.e. it does not cover all positive examples. In all other cases we assign a number n with $n \geq 0$ to a concept, which is the number of negative examples covered.

One of the problems we have to solve in the learning algorithm is that there can be infinitely many refinements of a given concept. This means that a node can have infinitely many children. Hence it is not possible to expand all of these children. Instead we will only consider children representing concepts up to a fixed length n. We say that n is the *horizontal expansion* of a node.

We will now define a node in the search tree explicitly.

Definition 4.37 (node)
A node is a quadrupel (C, n, q, b), where C is an $\mathcal{ALC}$ concept, $n \in \mathbb{N}$ is the *horizontal expansion*, $q \in \mathcal{Q} \cup \{\text{-}\}$ is the *quality*, and $b \in \{\text{true}, \text{false}\}$ is a boolean marker for the redundancy of a node. □

The marker "-" stands for undefined. This allows us to have nodes with an undefined quality. This is useful since we do not need to evaluate the quality (which is the most expensive operation) of redundant nodes. A concept with quality 0 is correct, i.e. it covers all positives and none of the negatives.

In the learning algorithm we need to be able to find the fittest node in the search tree. We define the fitness of a node as a lexicographical order over quality, concept length and horizontal expansion.

Definition 4.38 (fitness)
Let $N_1 = (C_1, n_1, q_1, b_1)$ and $N_2 = (C_2, n_2, q_2, b_2)$ be two nodes with defined quality ($q_1, q_2 \neq -, tw$). N_1 is *fitter* than N_2, denoted by $N_2 \leq_f N_1$ iff one of the following conditions hold:

- $q_2 < q_1$
- $q_1 = q_2$ and $|C_1| < |C_2|$
- $q_1 = q_2$ and $|C_1| = |C_2|$ and $n_1 \leq n_2$ □

We have now introduced all necessary notions to specify the complete learning algorithm.

Algorithm 4.39 (learning algorithm)

- Initialize:
  - ST (search tree) is set to the tree consisting only the root node $(\top, 0, \mathit{quality}(\top), \mathit{false})$
  - *minHorizontalExpansion* is set to 0
  - set *horizontalExpansionFactor* to a number higher than 0 and at most 1
- while ST does not contain a correct concept, do:
  - choose a node $N = (C, n, q, b)$ with the highest fitness in ST
  - expand N horizontally up to length $n+1$, i.e. add all nodes $(D, n+1, -, \mathit{checkRedundancy}(ST, D))$ with $D \in \mathit{transform}(\rho_\downarrow^{cl}(C))$ and $|D| = n+1$ as children of N
  - change N to $(C, n+1, q, b)$
  - set *minHorizontalExpansion* to max(*minHorizontalExpansion*, $\lceil \mathit{horizontalExpansionFactor} * (n+1)) \rceil$)
  - expand all nodes, which are neither redundant nor too weak, to at least $minHorizontalExpansion$ and change their horizontal expansion values accordingly
  - evaluate the quality of all concepts in non-redundant nodes

We see that the usual expansion in a search algorithm is replaced by a one step horizontal expansion. If we only expand the fittest node we may not explore large parts of the search space. In order to avoid this, a minimum horizontal expansion is used, which specifies that all nodes have to be expanded at least up to this length. This is a tradeoff between expanding only the fittest node and exploring other parts of the search space. For instance if we explore nodes with concepts of length 7 we may want to be sure that we explore at least all concepts of length 3. This tradeoff is controlled by the horizontal expansion factor.

Note that in the algorithm we do not remove overly special or redundant nodes. We do not remove these nodes, because they can be used to make it more efficient to compute the closure of $\rho_\downarrow$. There are (at least) two strategies for implementing the closure efficently: The first one is to store a tree (up to a maximal length of concepts) of refinements of $\rho_\downarrow$ for each node in the search tree. Whenever we increase the horizontal expansion of this node, we expand this tree by applying $\rho_\downarrow$ to its leafs. The second strategy is not to store any additional information in a node in the search tree, but recompute the refinements every time a node is expanded horizontally (in some sense this corresponds to iterative deepening search). In this case it is useful not to delete overly special or redundant nodes, since we can stop computing further refinements of $\rho_\downarrow$ whenever such a node is reached.

Proposition 4.40 (correctness)
If a learning problem has a solution, then Algorithm 4.39 eventually terminates and computes a correct solution of the learning problem.

Proof Assume there is a solution C (which is an $\mathcal{ALC}$ concept) of a learning problem. By the weak completeness of $\rho_\downarrow^{cl}$, we know that there is a concept D with $D \equiv C$ and $D \in \rho_\downarrow^*(\top)$. Because the horizontal expansion factor in Algorithm 4.39 is higher than 0 we will eventually explore D unless another solution is found before. In both cases the proposition is satisfied. ∎

Characteristics of the Algorithm

In this section we will summarize the characteristics of the learning algorithm we have created and describe how it integrates with other research.

First we have shown that the refinement operator $\rho_\downarrow$ is weakly complete. An important aspect related to weak completeness is the difference in length between the best reachable concept, i.e. the smallest concept C with $C \in \rho_\downarrow^{cl*}(\top)$, and the shortest solution. In our approach the operator $\rho_\downarrow$ uses the negation normal form of $\mathcal{ALC}$ concepts. It can reach any concept in $S_\downarrow$, which contains most of the formulas in negation normal form. In practice this means that for many learning problems a possible solution of the learning algorithm is close (with respect to length) to the shortest solution. Of course, it cannot always find the shortest solution, e.g. if all solutions are equivalent to $\neg(A_1 \sqcap A_2)$ then the algorithm finds the solution $\neg A_1 \sqcup \neg A_2$ (which is longer). Other learning algorithms work on $\mathcal{ALC}$ normal form (Esposito et al., 2004), which is a restricted version of negation normal form (see Section 2.3). In this case the algorithm is more likely to produce a longer solution. So on the one hand it is desirable not to restrict the structure of concepts too much, but on the other hand removing such restrictions makes the refinement process more complex and increases the search space.

Note that our learning algorithm avoids difficulties, which arise in other approaches. Some bottom-up approaches compute the *most specific concept* (MSC) (Cohen and Hirsh, 1994) of the examples or consider approximations of the MSC, since in some description languages like $\mathcal{ALC}$ it need not exist (Brandt et al., 2002). Then *least common subsumers* (LCS) (Cohen et al., 1993) of these concepts are computed to generalize the most specific concepts. However, this usually leads to overly specific concept definitions, which are long and overfit the data. Some approaches use most specific concepts, but avoid to use least common subsumers (Iannone and Palmisano, 2005). However, they still tend to produce very long solutions. These problems do not occur in our learning algorithm, since it is a top down approach, which was designed not only to produce a correct solution, but also to carefully take the length of a solution into account. As explained above we also avoided to impose too many restrictions on the structure of concepts.

Another nice feature of the learning algorithm is that it makes use of the subumption hierarchy of atomic concepts in the knowlege base. Often this allows to ignore large parts of the search space, e.g. if the concept `Male` is already overly special then we do not need to check `Man`, `Father`, and `Uncle` (assuming a meaningful knowledge base with these concepts is given).

To avoid improper refinement steps we have considered the closure $\rho_\downarrow^{cl}$ of $\rho_\downarrow$. We have also shown how the problem of infinite operators can be handled by not fully expanding nodes, but just compute the refinements up to a specified length of concepts. In Proposition 4.26 we have shown that for $\rho_\downarrow^{cl}$ this can always be done in finite time.

Further we have shown that meaningful weakly complete operators are usually redundant. For this reason the problem of redundancy was handled by creating a redundancy eliminating heuristic (instead of trying to define a non-redundant operator). We described exactly how the redundancy elimination works. In particular we presented a polynomial time check for weak equality, which relies on an ordered negation normal form of $\mathcal{ALC}$ concepts.

Related Work

Some links to related work have already been given above in the discussion of the characteristics of the developed learning algorithm. However, we want to make the two main influences of our research more explicit.

An interesting paper, which is closest to our work, is (Badea and Nienhuys-Cheng, 2000). It suggests a refinement operator for the $\mathcal{ALER}$ description logic. They also investigate some theoretical properties of refinement operators. However, unlike in this thesis, this is not a full analysis. As we have done with the design of $\rho_\downarrow$, they also favour the use of a downward refinement operator to enable a top-down search. They use $\mathcal{ALER}$ *normal form* (see the paper for a detailed description), which is easier to handle than $\mathcal{ALC}$ negation normal form, because $\mathcal{ALER}$ is not closed under boolean operations. As a consequence, they obtain a simpler refinement operator. Although they took a similar strategy for solving the learning problem, our work is more comprehensive, for instance we investigate the theoretical properties of refinement operators in more depth, propose a different way to deal with infinite operators, show how the subsumption hierarchy of atomic concepts can be used efficiently, and describe how redundancy can be avoided.

The second area of ongoing related work, is the research at the University of Bari in Italy, described in (Esposito et al., 2004; Iannone and Palmisano, 2005). They take a different approach for solving the learning problem by using approximated MSCs. As described above, when we discussed the characteristics of our learning algorithm, the main problem of their algorithms is that, like other approaches using MSCs, they tend to produce correct, but often unnecessarily long solutions of learning problems.

One problem is that MSCs for the description language $\mathcal{ALC}$ and more expressive languages do not exist and hence can only be approximated. Previous work (Cohen et al., 1993; Cohen and Hirsh, 1994) in the research area of learning in Description Logics was mostly focused on approaches using MSCs and LCSs, which face this problem to an even greater extend than the algorithms developed in Bari. In our approach we also cannot guarantee that we obtain the shortest possible solution of a learning problem. However, as described above, the learning algorithm was carefully designed to produce short solutions. The solution we produce, will be close to the shortest existing solution in negation normal form. In fact, we can handle this by varying the horizontal expansion factor in the learning algorithm (see page 52). In their algorithms they also make use of refinement operators, but do not focus on theoretical properties.

An Example Run

In this section we show an example run of the algorithm for the problem of learning the concept `Father`. Let the following knowledge base be given:

$\texttt{Male} \equiv \neg\texttt{Female}$	Male(MARC)
	Male(STEPHEN)
hasChild(STEPHEN,MARC)	Male(JASON)
hasChild(MARC,ANNA)	Male(JOHN)
hasChild(JOHN,MARIA)	Female(ANNA)
hasChild(ANNA,JASON)	Female(MARIA)
	Female(MICHELLE)

The positive examples in this case are `STEPHEN`, `MARC`, and `JOHN`. The negative examples are `JASON`, `ANNA`, `MARIA`, and `MICHELLE`. The concept $\texttt{Male} \sqcap \exists\texttt{hasChild}.\top$ is a solution, so by Proposition 4.40 the algorithm will solve the learning problem.

Figure 2 shows the search tree in each step of the algorithm. In this case we have chosen 0.4 as horizontal expansion factor. The role `hasChild` is abbreviated by `h`. `Male` is abbreviated by `Ma`, and `Female` by `Fe`. The selected fittest node is gray. We do not show nodes, which were classified as overly special in previous steps. On the left of each node we see the quality of the concept represented by the node. On the right of the node is its horizontal expansion. The example is too small to highlight all features of the learning algorithm, but is sufficient to get a general overview of the workings of the algorithm.

What we have presented in this section is a deterministic learning algorithm, which uses a search tree to find a solution of a learning problem. In the next section we will look at a stochastic approach as an alternative learning algorithm. Both methods have different properties and their relative performance is likely to be problem specific.

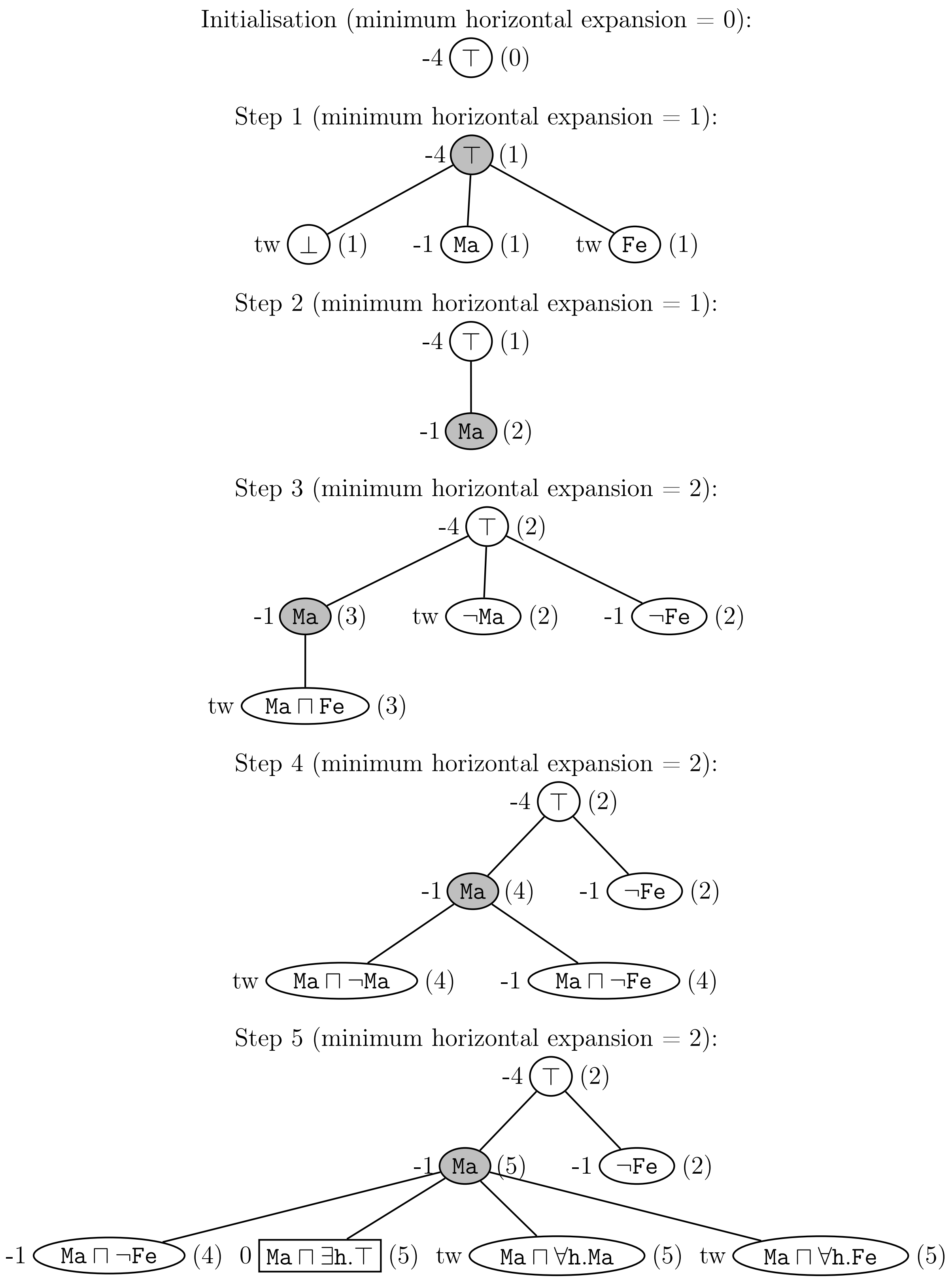

Initialisation (minimum horizontal expansion = 0):
-4 ⊤ (0)
Step 1 (minimum horizontal expansion = 1):
-4 ⊤ (1)
tw ⊥ (1)
-1 Ma (1)
tw Fe (1)
Step 2 (minimum horizontal expansion = 1):
-4 ⊤ (1)
-1 Ma (2)
Step 3 (minimum horizontal expansion = 2):
-4 ⊤ (2)
-1 Ma (3)
tw ¬Ma (2)
-1 ¬Fe (2)
tw Ma ⊓ Fe (3)
Step 4 (minimum horizontal expansion = 2):
-4 ⊤ (2)
-1 Ma (4)
-1 ¬Fe (2)
tw Ma ⊓ ¬Ma (4)
-1 Ma ⊓ ¬Fe (4)
Step 5 (minimum horizontal expansion = 2):
-4 ⊤ (2)
-1 Ma (5)
-1 ¬Fe (2)
-1 Ma ⊓ ¬Fe (4)
0 Ma ⊓ ∃h.⊤ (5)
tw Ma ⊓ ∀h.Ma (5)
tw Ma ⊓ ∀h.Fe (5)

Figure 2: learning the concept `Father` (horizontal expansion factor is 0.4)

5 Concept Learning and Genetic Programming

In this section we first briefly introduce Evolutionary Programming (EP) and then focus on Genetic Programming (GP) as a special kind of evolutionary programming. Later we show how to apply Genetic Programming to the learning problem for Description Logics. After that we will unite Genetic Programming and refinement operators to improve the performance of the learner. Finally, interesting extensions like learning from uncertain data and concept invention are presented.

5.1 Basics of Evolutionary Computing

Genetic Programming is an evolutionary algorithm and more general a Machine Learning technique. Evolutionary algorithms are inspired by the observation and computer simulation of biological evolution in the real world. One of the main sources of inspiration is Darwin's theory of evolution. In nature traits found in parents can be passed to their offspring. Usually both parents influence the new offspring and additionally mutations can produce new traits. In evolution there is a process called natural selection, which gives individuals, which are better adapted to their environment, i.e. have a high fitness, a better chance to survive and produce new offspring.

These concepts in nature are mapped to computers as shown in Table 5. The given problem, which we want to solve, is seen as an environment in which a population of individuals evolves. Each individual corresponds a solution of the problem. In every generation natural selection is used to select the fittest individuals among the population, which can then reproduce and form a new generation.

From a more technical point of view evolutionary computing is a search algorithm. It takes a set of possible solutions (the population), chooses some of them (selection), modifies these using genetic operators and forms a new set of solutions (the next generation of the population). The aim of the search algorithm is to find a good solution, i.e. an individual having a high fitness. This is how the algorithm in its general form looks like:

Algorithm 5.1 (generic evolutionary algorithm)

```
- create population
- while the termination criterion is not met:
        - select a subset of the population based on their fitness
        - produce offspring using genetic operators on selected individuals
        - create a new population from the old one and the offspring
```

Of course, this algorithm is very abstract. Many questions are still open: How do we represent individuals? How do we create a population of individuals? Which termination criteria exist? What selection methods are used? Which genetic operators can be used

nature	evolutionary computing
individual	problem solution
fitness	quality of a solution
chromosome	encoding of a solution
crossover, mutation	genetic search operators
natural selection	reuse of good solutions

Table 5: mapping concepts in nature to evolutionary computing

and how do they work? These questions will be answered in the Section 5.2 for the special case of Genetic Programming.

Before we will briefly give an overview of existing evolutionary computing algorithms and their distinctive features:

Genetic Algorithms This is a very popular evolutionary algorithm. The solutions of a problem are usually represented as a fixed length string of numbers (in most cases binary). It is the predecessor of Genetic Programming.

Genetic Programming Genetic Programming is the evolutionary algorithm which we focus on in this thesis. In contrast to Genetic Algorithms solutions are represented as variable length programs. A tree representation is very common, but linear Genetic Programming is also used.

Evolutionary Programming Evolutionary Programming makes no assumptions on the structure of the representation of an individuals. An individual could be represented as a neural network or a finite state machine. Usually operators do not change the structure of the solution itself, but its numerical parameters. Such an operator could for instance change the weights of a neural network or state transitions in a finite state machine.

Evolutionary Strategy Evolutionary Strategies are similar to evolutionary programming. It works on vectors of real numbers as representations of solutions. A difference between evolutionary strategies and evolutionary programming is that the latter does not use any kind of recombination between different individuals, but uses mutation style operators, which only change a single individual. In this way Evolutionary Strategies can be seen as an evolutionary process on the basis of individuals and Evolutionary Programming as a process on the basis of species (different species do not combine).

Learning Classifier Systems Learning Classifier Systems are a combination of Genetic Algorithms and Reinforcement Learning techniques. They are used for learning IF-THEN-rules.

All approaches have in common that they are population based and inspired by biological evolution and natural selection. They mainly differ in the way solutions are represented. After this general overview we will now deal with the specifics of Genetic Programming.

5.2 Introduction to Genetic Programming

Genetic Programming is one way to automatically solve problems. It is a systematic method to evolve programs and has been shown to deliver human-competitive machine intelligence in many applications (Koza et al., 2003).

The distinctive feature of Genetic Programming within the area of Evolutionary Computing is to represent individuals as variable length programs. It is an extension of Genetic Algorithms, which use fixed length strings (corresponding to the DNA of an individual). We will introduce Genetic Programming directly without covering the special case of Genetic Algorithms.

The primary use of Genetic Programming is to automatically find a program for solving a given task. Programs can have a tree (e.g. LISP) or linear (e.g. imperative languages) structure. Systems for the latter case fall under the category of Linear Genetic Programming. Although both cases are similar, Linear Genetic Programming faces different problems (i.e. syntactic correctness). In this thesis only tree based Genetic Programming will be introduced as it is the approach, which we will use to learn Description Logics.

Tree Representation and Alphabet

We will now look at the tree representation of programs in more detail. Let us assume we want to construct a GP algorithm for finding a function of two variables approximating given data i.e. in an interpolation task. Figure 3 shows a tree representation of the arithmetic expression $(2-y)*(3+(x*2))$, which is a potential solution of our problem. The actual individual or solution is called the *phenotype* whereas the encoding of the solution is the *genotype*.

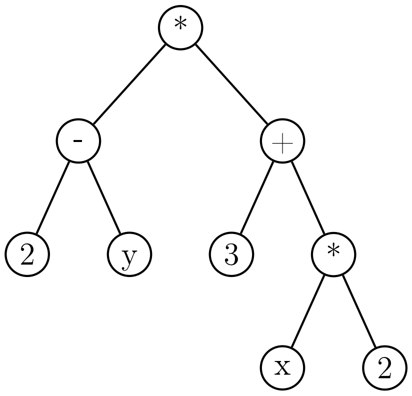


Figure 3: simple program in tree structure

The *function set* is the set of all internal nodes, which can occur in a tree, and the *terminal set* is the set of all leaf nodes, which can occur in a tree. The union of both is called the *alphabet*.

All elements in the alphabet can be assigned a *data type* (their return type). In our examples this would be the natural numbers. An alphabet is said to have the *closure* property if any function symbol can handle as an argument any data type and value returned by an alphabet symbol. In particular an alphabet has the closure property if all nodes have the same argument and data type. An alphabet having the closure property is called *closed*. We need the closure property, because later on we want to modifiy trees using genetic operators, e.g. replacing subtrees by other subtrees. Without the closure property we would need additional methods to make sure that the obtained program is meaningful. There are approaches to handle different data types in the area of *Strongly Typed Genetic Programming* (Montana, 1995). In this approach data types are specified explicitly and in all modifications of trees it is ensured that no illegal trees (with respect to data types) are produced.

Example 5.2 (closure property)
The following alphabet is closed:

$$T = \{x, y, True, False\}, F = \{AND, OR, NOT\}$$

For this example to be closed we assume that x and y are boolean variables and all other symbols have the obvious meaning.

Another closed alphabet is ("$-$" is subtraction):

$$T = \{x, y, ERC[0,1]\}, F = \{*, +, -\}$$

Note that $ERC[0,1]$ stands for "ephemeral random constant between 0 and 1". This means that when creating an individual a random number between 0 and 1 is chosen, which remains fixed after creation (so when evaluating the tree it is actually a constant). Here and in the following examples x and y are variables of type integer.

The following alphabet is not closed, because division (/) by zero is not defined:

$$T = \{x, y, 0, 1\}, F = \{*, +, -, /\}$$

It can be closed by using a protected division. This involves explicitly defining the behaviour in case of a division by 0, e.g. return a value of 1 in this case.

Another non-closed alphabet:

$$T = \{x, y\}, F = \{+, -, \sin, \log\}$$

It is not closed, because the logarithm is only defined for positive numbers. Again, we can protect it by e.g. returning 0 if the argument is a negative number. □

Note that although boolean and arithmetical examples are shown here, GP's are by no means limited to such tasks. They can also be applied to a variety of other tasks like robot control, electrical circuit design, neural network construction and many more.

Another property of an alphabet is *sufficiency*. Sufficiency means that the alphabet symbols are sufficient to express the solution of a problem.

Example 5.3 (sufficiency)
The alphabet $T = \{x, y\}, F = \{AND, OR, NOT\}$ is sufficient to learn any boolean function of two variables x and y, e.g. XOR.

The alphabet $T = \{x, 0, 1\}, F = \{*, +, -\}$ is not sufficient to learn the function e^x, because e^x cannot be expressed (only approximated) using only polynomials. □

Note that in many cases it is very complicated or impossible to determine whether or not an alphabet is sufficient. For some tasks even the choice of the alphabet is difficult. Including many symbols in the alphabet increases the search space, but not including certain symbols can greatly degrade performance. It depends on the specific problem, which alphabet is suitable.

Creating an Initial Population

Before the evolutionary progress of a GP algorithm can start, an initial population has to be created. There are traditonally three main methods to do this, which we will introduce briefly. Finally, a slightly modified creation method is presented, we considered useful for possible future experiments.

The Grow Method The grow method creates one individual at a time. As parameter it takes the maximum depth d of the tree to be created. If $d = 0$, the grow method just returns a terminal symbol. If $d > 0$, the method randomly selects a symbol from the alphabet. If this symbol is a terminal, we return it. If it is a function symbol of arity n, we (recursively) call the grow method with depth parameter $d - 1$ n-times to generate all children.

This way a tree of maximum depth d is created. However, it does not necessarily need to have depth d nor does it have to be a complete tree. (A complete tree is a tree where all leaf nodes have equal depth.)

The Full Method The full method works like the grow method, but we just generate function symbols until the maximum depth d is reached, when we will generate a terminal. This method creates complete trees.

Ramped-Half-and-Half The ramped-half-and-half method is a combination of the grow and full method. It is very popular, because it increases the variation in the structure of the generated trees. Again, a maximum depth parameter d is used. The population, which we want to generate, is divided in $d - 1$ parts. In the first part all trees have a maximum depth of 2. Half of them are generated by the grow and the other half by the full method. In the second part the maximum depth is 3, in the third part the maximum depth is 4 etc. In part $d - 1$ the maximum depth is d. This way it is ensured that there is a variety of different tree depths of complete as well as (possibly) incomplete trees within the population.

Function-Based-Half-and-Half This is a more flexible extension of the ramped-half-and-half method. (We do not know if it is already mentioned in literature.) A disadvantage of the ramped-half-and-half method is that it generates the same number of individuals for each depth between 2 and the maximum depth d. However, the number of different trees increases (exponentially) with increasing tree depth. It seems to be sensible to generate more trees with higher tree depths. This is done by the function-based-half-and-half method. It generates one tree at a time. First it decides wether to use the full or the grow tree method (both with equal probability). Then it decides which maximum tree depth to use. To do this one has to specify a function $f : D \rightarrow \mathcal{R}^+$ with $D = \{2 \ldots d\}$ (the set of all possible maximum tree depths). Then a tree with maximum depth i is generated with probability *prob*(i):

$$prob(i) = \frac{f(i)}{\sum\limits_{i=2}^{d} f(i)}$$

The sum of all possible tree depths obviously adds up to 1. The ramped-half-and-half method can be approximated by using $f(i) = 1$. A useful function could be $f(i) = i$ or even $f(i) = i^2$ to generate more trees with high maximum depth.

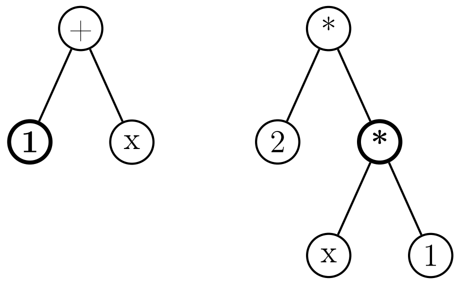
Genetic Operators

The creation of the initial population is already a search for a possible solution, but it is blind and random. For complex problems it is unlikely that any reasonable solution will be produced in the initialisation phase. However, some of the individuals may contain parts of a useful solution. It is the role of the evolutionary process (selection and genetic operators) to find and combine the pieces to a good solution. In this section we will give an overview over the operators used in GP. All operators use one or two individuals and modify them. The main operators are *reproduction* and *crossover*. *Mutation* can be used as a secondary operator and we will also briefly mention other operators, which can be useful. Last bot not least it should be mentioned that one can also define own operators for a specific problem, which is a way of adding knowledge to the search process. We will later do this to incorporate refinement operators in the Genetic Programming framework.

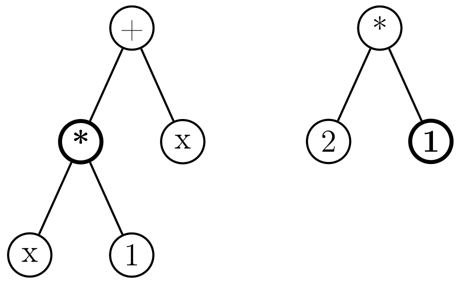
Reproduction Reproduction is a very simple operator. It selects one individual and leaves it unchanged. The effect is that the individual is copied over in the next generation. The selection function in a GP is designed to select fitter individuals with a higher probability, so reproduction is one way to ensure that fitter individuals survive.

Crossover The crossover operator selects two individuals (parents) and produces two offspring. In both parents one node is randomly selected. These are the *crossover points*. The subtrees rooted by the two points are then swapped, which produces two new individuals. The next example illustrates this process.

Example 5.4 (crossover)
Parents (crossover points highlighted):



Offspring:

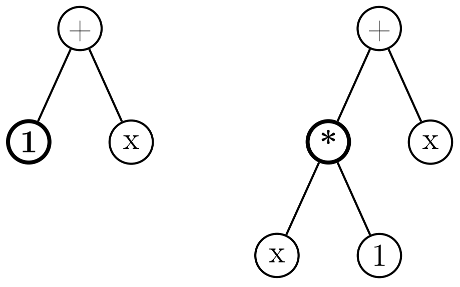


The closure property ensures that we obtain legal trees. Note that if both crossover points are roots of the parent trees then crossover is equivalent to reproduction. Another interesting observation is that identical parents are likely to produce different offspring. This means that even in a population with many identical individuals we can still produce new solutions. (Remark: This property does not hold for Genetic Algorithms, where the genotypes are strings of numbers.)

A technical issue is that crossover can produce trees of considerable depth. This is the case when one crossover point is close to the root and the other one has a high depth. For this reason a depth limit or a depth penalty can be used. Crossover is the main operator in Genetic Programming. It is typically used with a probability of around 85 to 90 percent. However, we will later see that the probability of the use of an operator is higly problem specific. In some Genetic Programming systems crossover is not used at all.

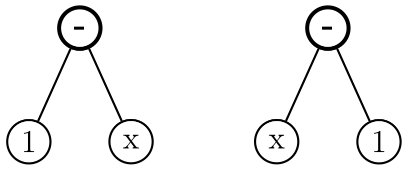
Mutation Mutation selects one individual from the population and randomly chooses a node in the corresponding tree (the *mutation point*). The subtree rooted by this mutation point is removed and replaced by a randomly generated subtree. If this subtree consists only of a single node (a terminal) this is called a *point mutation.* Usually to generate the subtree the grow method is used with the same maximum depth like in the initialisation. Mutation is used sparingly. Typically its probability is around 1 percent.

Example 5.5 (mutation)
The selected individual is shown on the left and the resulting individual after mutation on the right (mutation point highlighted):



Permutation Permutation produces a random change in the order of the arguments of a function. It selects one individual and randomly chooses a non-terminal node in the corresponding tree (the *permutation point*). After that a random permutation of the function arguments is generated and the children of the permutation point are ordered according to this permutation.

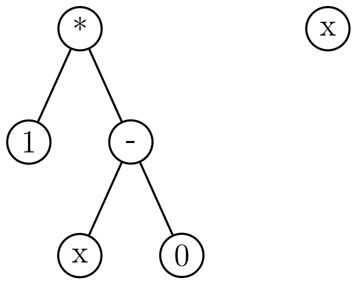
Example 5.6 (permutation)
The selected individual is shown on the left and the resulting individual after permutation on the right (permutation point highlighted):



Permutation only makes sense if the order of the arguments is relevant for the selected function. Therefore one may restrict permutation to select only such functions, e.g. "-" can be selected, but not "+".

Editing A GP tree can be highly redundant, because the GP does not have any knowledge about the structure it is working on. Naturally one idea is to remove redundancies in the tree representation. This is often done by recursively applying simplification rules to a tree, which are problem specific. As a simple example *True AND True* can be replaced by *True* and $x + 0$ can be replaced by x.

Example 5.7 (editing)
The selected individual is shown on the left and the edited individual on the right:



Although editing seems to be a useful genetic operator its effects are unclear. Extraneous parts of the tree, which when removed do not alter the result, are called *introns*.

They emerge when variable length structures are modified. In early generations of a GP run introns make up a small part of the code, whereas towards the end of a run they tend to make up almost all of the code (if one does nothing to prevent introns). This process is called *bloat*. At first glance it seems to be beneficial to remove introns completely, however, they can be useful, because they increase the *effective fitness* of an individual. Effective fitness is the likelihood that an individual's descendants survive. It is determined by the fitness of the individual (because selection prefers fitter individuals) and how fit the offspring is likely to be. Introns can shield a tree from the destructive effects of crossover, so they increase the surivivability of the offspring and therefore the effective fitness. Of course, introns can also be problematic. If introns make up almost all of the code, then crossover does not efficiently produce new solutions (it just swaps introns). Too many introns can also negatively affect runtime. A common means to avoid this is to introduce a fitness penalty for the size of the generated tree. This way individuals without introns are slightly fitter.

Selection

Selection methods in GP algorithms are used to simulate natural selection in nature, sometimes also refered to as *survival of the fittest*. Technically this means that individuals with high fitness are selected with higher probability compared to individuals with low fitness.

How do we actually measure fitness? Of course, this is highly problem dependant, but most approaches have in common that the fitness of an individual is decoded as a single number. A higher number denotes higher fitness of the individual. Although the fitness value is a single number the measuring process can be quite complex. It could involve letting a robot act on a problem for some time or to test the performance of a neural network. It can also be as easy as the evaluation of a simple formula.

In many cases one has to encode multiple objectives into a single fitness function. For instance if we want to find a function approximating a given data set, one criterion is usually how close the function is to the given data. Another criterion is the length of the function we find. A short function usually generalizes better to unseen data. In such cases the fitness function can also be used to bias the search process (correctness vs. low complexity). The choice of the fitness function can greatly influence the result of a GP algorithm.

Assuming an appropriate fitness function a single fitness value can be assigned to each individual. Usually one individual is selected at a time. A selection can be viewed as a probability distribution of the individuals in a population and a population can be viewed as a list of fitness values. There are a variety of different selection methods and only some of these will be described here.

Fitness Proportionate Selection (FPS) FPS is a popular selection method. Its foundation is that the probability of an individual being selected is proportional to its fitness. If we denote f_i as the fitness of individual i and $s = \sum_{k=1}^{n} f_k$ as the sum of all fitness values in a population of n individuals, then the probability of individual i

being selected is $\frac{f_i}{s}$, denoted by $prob(f_i)$. There are several algorithms for computing this efficiently, which will not be described here.

FPS is a very straightforward way to implement natural selection. In practical applications it has been shown that it can be used to quickly converge to a good solution. A drawback of FPS is *stagnation*. At the end of a GP run the population often consists mostly of individuals with similar fitness and some low fitness individuals. In this case all individuals have almost the same probability of being selected and there is virtually no *selective pressure*. This can be avoided by using a fitness scaling technique. One such scaling technique is *sigma truncation*. It replaces the fitness function f by f' with $f' = \max\left(0, f - \overline{f} + c\sigma\right)$ where $\overline{f} = \frac{s}{N}$ is the average fitness within the population, σ is the standard deviation of the population and c is a factor (usually $1 \le c \le 3$). Whether or not sigma truncation is necessary depends on the fitness measure.

Example 5.8 (sigma truncation)
Assume we have 5 individuals in our population and the following fitness list:

$$[123, 121, 122, 123, 108]$$

We can compute the following values:

$$N = 5, \quad s = 597, \quad \overline{f} = 119.4, \quad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(f_i - \overline{f})^2} = 5.748\ldots$$

With standard FPS we get this probability list:

$$[20.6\%, 20.3\%, 20.4\%, 20.6\%, 18, 1\%]$$

Using sigma-truncation with $c = 2$ we get the following list for f':

$$[15.1, 13.1, 14.1, 15.1, 0.1]$$

This results in this probability list:

$$[26.2\%, 22.8\%, 24.5\%, 26.2\%, 0.2\%]$$

The variance in the probability list has been increased, so the GP actually distinguishes between individuals of similar fitness (here: 121-123). This shows that sigma-truncation can increase selective pressure. □

Rank Selection Another way to solve the stagnation problem is rank selection. In this method individuals are sorted by their fitness, i.e. each individual gets a rank within the population. The probability of being selected in rank selection only depends on the rank of the individual and not directly on its fitness value. The probability is then a function of the rank (this can be any sensible function). Rank selection usually converges slower than FPS, because even less fit individuals can be selected. Note that for rank selection a fitness function is not strictly necessary, but one just needs an ordering on individuals. It is also not necessary for the fitness values to be positive.

Tournament Selection Tournament selection is similar to rank selection, but does not require the individuals to be sorted by their fitness. It takes m ($m \geq 2$) individuals from the population and selects the best one (all others are discarded). Tournament selection is more naturally inspired than rank selection.

In all the selection methods one can use a flag called *elitism*. It means that the best individual always surives, i.e. it is automatically passed on to the next generation.

Termination

An issue we have not yet discussed is the termination of the GP algorithm. In some cases it is possible to detect whether or not an optimal solution has been found. In such cases one can simply terminate after such an individual has been found. Often an optimal individual does not exist (or it is unknown whether it exists). In these cases one can terminate if the fitness of the best individual has exceeded a certain threshold, which means that it is sufficiently close to an optimal solution. Another more complex termination criterion is to verify whether the genetic diversity of the population is high enough to produce better solutions. If all individuals are equal or very similar this usually means that the GP has converged and is unlikely to produce new better solutions. In a lot of applications a much simpler criterion for termination is used: a fixed number of generations or a manual abort by the user.

We defined another simple method to test for convergence. (We did not find an explicit mention of it in the literature, but it is straightforward.) The user can specify a number of *post convergence generations*. If the GP algorithm does not find a fitter individual within this number of generations, it stops. This is a very natural targeted criterion, because the task of the GP algorithm is to find the fittest individual possible. If it does not find any better solution for a sufficiently large number of generations, it has converged with a high probability.

Overall Algorithm

We have introduced all necessary ingredients of a GP algorithm and now want to show how to combine them.

We can distinguish two types of GP algorithms: *generational* and *steady-state*. They are, of course, both specialisations of the general evolutionary computing algorithm (Algorithm 5.1). In a generational algorithm the new generation completely replaces the old generation, i.e. the new generation is build completely from individuals, which have been produced by genetic operators. In contrast in a steady-state algorithm only a subset of the old generation is replaced by the offspring. Usually the weakest individuals are replaced, while the fitter individuals survive. (The exact percentage of individuals which have to be replaced is a parameter of the GP algorithm.)

There are four steps to define a standard GP problem:

1. Define the alphabet.

2. Define the fitness measure.
3. Choose a termination criterion.
4. Fix all parameters of the GP.

Parameters of a GP algorithm include population size, algorithm type, initialisation method, selection method, and the probabilities of the genetic operators used. The population size allows the GP to be adjusted to the computational resources available. A higher population size usually means that the probability of finding a good solution is higher.

Algorithm 5.9 (Genetic Programming)

```
- create initial population (section 5.2)
- while the termination criterion (section 5.2) is not met:
      - evaluate the fitness of all individuals in the population
      - while the number of individuals to generate is not reached:
            - choose a genetic operator (section 5.2)
            - select (section 5.2) the appropriate numbers of individuals
              for this operator and generate new individuals
            - copy these individuals into the new population
```

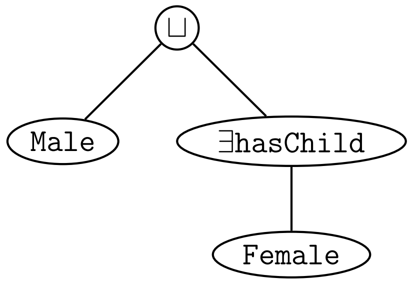
This algorithm also applies to other areas of Evolutionary Computing. This distinctive feature of Genetic Programming is the use of variable length programs to represent possible solutions.

5.3 Application to Description Logics

In this thesis we are concerned with learning $\mathcal{ALC}$ concepts. So far, we have shown how Genetic Programming works and we now want to apply it to the learning problem for Description Logics. The way we do it is natural and straightforward.

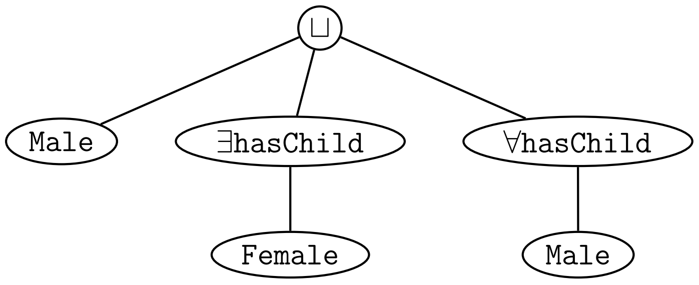
We can represent each $\mathcal{ALC}$ concept as a tree. There are the following types of nodes: concept names, $\top$, $\bot$, $\sqcup$, $\sqcap$, $\neg$, $\forall r$, and $\exists r$. The leaf nodes are, of course, the concept names, $\top$, and $\bot$. All other node types are function symbols of arity one or two.

Example 5.10 (representing ALC concepts as trees)
The ALC concept `Male` $\sqcup$ `∃hasChild.Female` can be represented as the following tree: □



More formally the alphabet we use is $T = N_C \cup \{\top, \bot\}$ and $F = \{\sqcup, \sqcap, \neg\} \cup \{\forall r \mid r \in N_R\} \cup \{\exists r \mid r \in N_R\}$. An important question we have to ask is whether this way of representing ALC concepts satisfies the closure property. Indeed it is satisfied, because the way trees are build exactly corresponds to the way ALC concepts are defined recursively. This means that all nodes are of type concept, so closure is satisfied. This ensures that crossover and mutation cannot construct illegal trees.

Please note that the tree representation we have shown is just one way to encode $\mathcal{ALC}$ concepts. Another possibility would be to consider $\sqcap$ and $\sqcup$ as nodes of arbitrary arity (≥ 2), e.g. a conjunction `Male` $\sqcup$ $\exists$`hasChild.Female` $\sqcup$ $\forall$`hasChild.Male` can be represented as:



When using Genetic Programming we could, of course, also learn in more expressive description logics than $\mathcal{ALC}$, which allow the use of role constructors in concepts. The following remark explains how we could learn in such logics.

Remark 5.11 (role constructors)
Some description languages (for instance $\mathcal{ALCI}_{reg}$) also allow role constructors to appear within concepts. If such description languages should be learned by Genetic Programming the closure property no longer holds, because we now have two different node types: concepts and roles. (Note than in $\mathcal{ALC}$ we also have roles, but they are hidden within the $\forall r$ and $\exists r$ nodes.) We can still use Genetic Programming by now extending it to strongly typed Genetic Programming (STGP). In STGP we have to explicitly assign argument- and return-types of each operator (if we allow e.g. disjunction for concepts and roles, then this would be two separate operators). Additionally we have to assign a type to our learning task: Either concept for concept learning or role for learning roles. When generating trees we have to take care that we only generate valid trees, which respect the type restrictions (this is not difficult). Additionally all used genetic operators have to be modified to be type aware. For crossover this means that we still randomly select a node in the first parent, but we have to select a node of the same type in the second parent. For mutation this means that the node at the mutation point has to be replaced by a tree of the same type. This brief description shows that Genetic Programming can be applied to various description languages. □

Fitness Measurement

Apart from being able to represent an $\mathcal{ALC}$ concept we also need to define a fitness measure. Of course, this measurement is derived from the background knowledge and the positive and negative examples. The fitness measurement may take several things into account (efficiency of measurement, correct classification, avoiding overfitting, learning from uncertain information etc.). We introduce some shortcuts to be able to define the fitness function in a more compact way.

Definition 5.12 (covered examples)
Let `Target` be the target concept, $\mathcal{K}$ a knowledge base, and C an arbitrary $\mathcal{ALC}$ concept. The *set of positive examples covered by* C, denoted by $pos_{\mathcal{K}}(C)$, is defined as:

$$pos_{\mathcal{K}}(C) = \{\texttt{Target}(a) \mid a \in N_I, K \cup \{\texttt{Target} \equiv C\} \models \texttt{Target}(a)\} \cap E^+$$

Analogously the *set of negative examples covered by* C, denoty by $neg_{\mathcal{K}}(C)$, is defined as:

$$neg_{\mathcal{K}}(C) = \{\texttt{Target}(a) \mid a \in N_I, K \cup \{\texttt{Target} \equiv C\} \not\models \texttt{Target}(a)\} \cap E^- \qquad \square$$

Of course, the fitness measurement should give credit to covered positive examples and penalize covered negative examples. In addition to these classification criteria it is also useful to bias the GP algorithm towards shorter solutions. A possible fitness functions is:

$$f_{\mathcal{K}}(C) = |pos_{\mathcal{K}}(C)| - |neg_{\mathcal{K}}(C)| - \frac{1}{a}|C| \quad (a > 0)$$

a is a parameter, which specifies the importance of simplicity compared to correct classification. (Intuitively a is the increase in length of a possible solution which justifies that one more example is not classified correctly.) The parameter a is also used to handle noise. If we know that the data is noisy, then a should have a low value. In this way GP algorithms can handle noise easily.

Often a weighted version of a fitness function is more appropriate to be more independant of the number of examples given:

$$f_{\mathcal{K}}(C) = \frac{|pos_{\mathcal{K}}(C)|}{|E^+|} - \frac{|neg_{\mathcal{K}}(C)|}{|E^-|} - \frac{1}{b}|C| \quad (b > 0)$$

For the approaches we consider in this section both fitness functions are suitable. We will later slightly extend the fitness function to learn from uncertain data.

Being able to represent solutions and measuring their fitness is already sufficient to apply Genetic Programming to a problem. However, we may also ask if it is a good idea to use it. We present some advantages and problems of the introduced approach.

Advantages of Genetic Programming

One of the reasons why GP is tried as a method to learn in Description Logics is its flexibility. As explained above, it can be used to learn in several description languages, because it is a very general learning method. Genetic Programming has shown very promising results in practice (Koza et al., 2003), so we considered it worh investigating.

GP is especially suited in situations, where it is difficult to directly obtain good solutions and approximate solutions are acceptable (Koza and Poli, 2003). An advantage of GP in general is that it can make use of computational resources, i.e. if more resources (time and memory) exist its parameters can be changed to increase the probability of finding good solutions. This may seem obvious, but in fact this does not hold for all (deterministic) solution methods. Genetic Programming also allows for a variety of extensions and is able to handle noise and can learn from uncertain and inconsistent examples. A GP algorithm is also highly configurable (initialisation method, genetic operators and their probabilities, population size, selection method etc.). This allows to encode knowledge in a GP algorithm and biasing its search process.

Problems of the Standard Approach

With the standard approach we have introduced we can solve the learning problem using Genetic Programming. However, despite the advantages of GP, there are also drawbacks. One problem is that the crossover operator is too destructive. For GP to work in a meaningful way we need to have the property that applying genetic operators to individuals having a high fitness is likely to produce offspring with a high fitness. This is the reason why the selection methods we have shown work better than merely selecting individuals at random. For crossover on $\mathcal{ALC}$ concepts it is problematic that small changes in a concept can drastically change its meaning. Similar problems arise when using GP in ILP and indeed a lot of systems use non-standard operators (Divina, 2006).

Another problem of the standard approach is that we do not use all knowledge we have. An essential insight in Machine Learning (Mitchell, 1997) is that the approaches, which use most knowledge about the learning problem they want to solve usually perform best. The standard GP algorithm does not exploit the subsumption hierarchy of concepts. In Section 4 the main goal of the learning algorithm we created was to traverse the subumption hierarchy of concepts in an efficient way by using refinement operators. Using subsumption as ordering over concepts can increase the performance of a learning algorithm significantly. Thus a natural idea is to enhance the standard GP algorithm by operators, which exploit the subsumption order.

Usage of Evolutionary Techniques in Inductive Logic Programming

To the best of our knowledge there has been no attempt to apply Genetic Programming to the learning problem in Description Logics. However, there have been several approaches to use evolutionary techniques in Inductive Logic Programming. We will give some pointers to such approaches since they may be of interest for readers, who want to have a look at the role of evolutionary approaches in logical induction in general. We will not explain them, but instead refer to a recent article (Divina, 2006), which describes and compares most of the systems, which are mentioned below. An experimental comparision of evolutionary and standard approaches for learning recursive list functions can be found in (Tang et al., 1998).

The ILP systems, which use evolutionary algorithms, usually use variants of Genetic Algorithms or Genetic Programming. The target is to learn a set of clauses for a target predicate. EVIL_1 (Reiser and Riddle, 1999) is a system based on Progol (Muggleton, 1995b, 1996), where an indivual represents a set of clauses (called the *Pittsburgh approach*) and crossover operators are used. REGAL (Neri and Saitta, 1995; Giordana and Neri, 1996) is a system, which consists of a network of genetic nodes to achieve high parallelism. Each individual encodes a partial solution (called the *Michigan approach*). It uses classic mutation and several crossover operators. GNET is a descendant of REGAL. It also uses a network of genetic nodes, but takes a co-evolutionary approach (Anglano et al., 1998), i.e. two algorithms are used to converge to a solution. DOGMA (Hekanaho, 1996, 1998) is a system, which uses a combination of the Pittsburgh and Michigan approach on different levels of abstraction. All these systems use a simple bit string representation. This is possible by requiring a fixed template, which the solution must fit in. We did not consider this approach when learning in Description Logics due to its restricted flexibility. The systems mentioned in the sequel use a high level representation of individuals. SIA01 (Augier et al., 1995) is a bottom-up approach, which starts with a positive example as seed and grows a population until it reaches a bound (so the population size is not fixed as in the standard approach). ECL (Divina and Marchiori, 2002) is a system using only mutation style operators for finding a solution. In contrast GLPS (Wong and Leung, 1995) uses only crossover style operators and a tree (more exactly forest) representation of individuals. In (Tamaddoni-Nezhad and Muggleton, 2000) a binary representation of clauses is introduced, which is shown to be processable by genetic operators in a meaningful way. (Tamaddoni-Nezhad and Muggleton, 2003) extends this framework by a fast fitness evaluation algorithm.

5.4 Refinement Operators in Genetic Programming

Transforming Refinement Operators to Genetic Refinement Operators

As argued before it is useful to modify the standard GP approach to make learning more efficient. The idea we propose is to combine refinement operators and Genetic Programming. A disadvantage of Genetic Programming is that it does not make use of the subsumption hierarchy of concepts, which is exactly what refinement operators are designed to do (see Section 4). By combining both, we hope to increase the performance of the learner (compared to the standard GP approach). We show how refinement operators and Genetic Programming can be combined in general and then present a concrete refinement operator.

Some steps need to be done in order to be able to use refinement operators as genetic operators. The first problem is that a refinement operator is a mapping from one concept to an arbitrary number of concepts. Naturally the idea is to select one of the possible refinements. In order to be able to do this efficiently we assume that the refinement operators we are looking at are finite. Then we can randomly select a refinement. Every refinement can have the same probability of being selected. Of course, it would be possible to bias the probabilities of refinements being selected towards smaller concepts.

However, the bias for smaller concepts is already incorporated in the fitness function, so a uniform distribution of refinements should perform reasonable. Whether other distributions are more suitable depends on the specific refinement operator which is used.

The second problem when using refinement operators in Genetic Programming is that a concrete refinement operator only performs either specialisation or generalisation, but not both. However, in Genetic Programming we are likely to find too strong as well as too weak concepts, so there is a need for upward and downward refinement. A simple approach is to use two genetic operators: an adapted upward and an adapted downward refinement operator.

A better way to solve the problem is to use one genetic operator, which stochastically chooses whether downward or upward refinement is used. This allows to adjust the probabilites of upward or downward refinement being selected to the classification of the concept we are looking at. For instance consider an overly general concept, i.e. it covers all positive examples, but does also cover some negative examples. In this case we always want to specialize, so the probability for using downward refinement should be 1. In the opposite case for an overly specific concept, i.e. none of the negatives is covered, but some positives, the probability of downward refinement should be 0.

For all other concepts, which are neither overly general nor overly specific, we need to assign appropriate probabilites different from 0 and 1. How do we assign appropriate probabilites? We came up with some points, which seemed reasonable to us:

1. The probability of downward refinement, denoted by $p_\downarrow$, should depend on the percentage of covered negative examples. Using α as variable factor we get:

   $$p_\downarrow(\mathcal{K}, C) = \alpha \cdot \frac{|neg_\mathcal{K}(C)|}{|E^-|}$$

   In particular for $|neg_\mathcal{K}(C)| = 0$ (consistent concept) we get $p_\downarrow(\mathcal{K}, C) = 0$.

2. The probability of upward refinement, denoted by $p_\uparrow$, should depend on 1 minus the percentage of covered positive examples. As factor we will use the same factor as in the first case, so we get:

   $$p_\uparrow(\mathcal{K}, C) = \alpha \cdot \left(1 - \frac{|pos_\mathcal{K}(C)|}{|E^+|}\right)$$

   In particular for $|pos_\mathcal{K}(C)| = |E^+|$ (complete concept) we get $p_\uparrow(\mathcal{K}, C) = 0$.

3. For any concept $p_\downarrow(\mathcal{K}, C) + p_\uparrow(\mathcal{K}, C) = 1$.

From these points we can derive the following formulae for the probablities of upward and downward refinement:

$$p_\downarrow(\mathcal{K}, C) = \frac{\frac{|neg_\mathcal{K}(C)|}{|E^-|}}{1 + \frac{|neg_\mathcal{K}(C)|}{|E^-|} - \frac{|pos_\mathcal{K}(C)|}{|E^+|}}$$

$$p_\uparrow(\mathcal{K}, C) = \frac{1 - \frac{|pos_\mathcal{K}(C)|}{|E^+|}}{1 + \frac{|neg_\mathcal{K}(C)|}{|E^-|} - \frac{|pos_\mathcal{K}(C)|}{|E^+|}}$$

Note that the return value of the formula is undefined, because of division by zero, for cases in which the given concept C is complete and consistent. However, in this case C is a solution for the learning problem and we can stop the algorithm (it is also possible to continue the algorithm by just randomly selecting whether upward or downward refinement is used).

This way we have given a possible solution to both problems: transforming the refinement operator to a mapping from a concept to exactly one concept and managing specialisation and generalisation. Overall for a given finite upward refinement operator $\varphi_\uparrow$ and a finite downward refinement operator $\varphi_\downarrow$ we can construct a genetic operator φ, which is defined as follows (rand selects an element of a given set uniformly at random):

$$\varphi(C) = \begin{cases} \text{rand}(\varphi_\downarrow(C)) & \text{with probability } \frac{\frac{|neg_\mathcal{K}(C)|}{|E^-|}}{1+\frac{|neg_\mathcal{K}(C)|}{|E^-|}-\frac{|pos_\mathcal{K}(C)|}{|E^+|}} \\ \text{rand}(\varphi_\uparrow(C)) & \text{with probability } \frac{1-\frac{|pos_\mathcal{K}(C)|}{|E^+|}}{1+\frac{|neg_\mathcal{K}(C)|}{|E^-|}-\frac{|pos_\mathcal{K}(C)|}{|E^+|}} \end{cases} \qquad (4)$$

Note, that we need two refinement operators to construct one genetic refinement operator. The most straightforward way to do this is to define one of the refinement operators (for upward or downward refinement) and design the other operator in a dual fashion.

In the sequel we will call operators like φ, which are created from upward and downward refinement operators in this fashion, *genetic refinement operators.*

A Genetic Refinement Operator

So far, we have shown how to transform a finite refinement operator to a genetic refinement operator. Now we want to define a specific refinement operator, which is suitable for learning in the GP framework.

Note that the operator $\rho_\downarrow$ presented in Section 4.3 cannot be used, because it is finite. We will design a finite (and complete) operator. However, by Proposition 4.10 such an operator has to be improper.

Another reason, why $\rho_\downarrow$ is not suitable, is that it cannot drop elements of a disjunction, e.g. it is not possible to reach A_1 from $A_1 \sqcup A_2$ (see also the discussion of Proposition 4.23 on page 39). This is no problem for the top down search $\rho_\downarrow$ was designed for, e.g. we can reach A_1 directly from $\top$. However, for the Genetic Programming approach, which does not use a search tree, it is better if we allow to drop elements of a disjunction.

We will first define a downward refinement operator for the GP framework.

$$\varphi_\downarrow(C) = \begin{cases} \{C_1 \sqcap \cdots \sqcap C_{i-1} \sqcap D \sqcap C_{i+1} \sqcap \cdots \sqcap C_n & \text{if } C = C_1 \sqcap \cdots \sqcap C_n (n \geq 2) \\ \quad \mid D \in \varphi_\downarrow(C_i), 1 \leq i \leq n\} & \\ \cup \{C_1 \sqcap \cdots \sqcap C_n \sqcap \top\} & \\ \{C_1 \sqcup \cdots \sqcup C_{i-1} \sqcup D \sqcup C_{i+1} \sqcup \cdots \sqcup C_n & \text{if } C = C_1 \sqcup \cdots \sqcup C_n (n \geq 2) \\ \quad \mid D \in \varphi_\downarrow(C_i), 1 \leq i \leq n\} & \\ \cup \{C_1 \sqcup \cdots \sqcup C_{i-1} \sqcup C_{i+1} \sqcup \cdots \sqcup C_n & \\ \quad \mid 1 \leq i \leq n\} & \\ \cup \{C \sqcap \top\} & \\ \{A' \mid A' \sqsubset_\mathcal{T} A, A' \in N_C \cup \{\bot\}, & \text{if } C = A\ (A \in N_C) \\ \text{ there is no } A'' \in N_C \text{ with } A' \sqsubset_\mathcal{T} A'' \sqsubset_\mathcal{T} A\} & \\ \cup \{C \sqcap \top\} & \\ \{\neg A' \mid A \sqsubset_\mathcal{T} A', A' \in N_C, & \text{if } C = \neg A\ (A \in N_C) \\ \text{ there is no } A'' \in N_C \text{ with } A \sqsubset_\mathcal{T} A'' \sqsubset_\mathcal{T} A'\} & \\ \cup \{C \sqcap \top\} & \\ \{\exists r.E \mid E \in \varphi_\downarrow(D)\} \cup \{C \sqcap \top\} & \text{if } C = \exists r.D \\ \{\forall r.E \mid E \in \varphi_\downarrow(D)\} \cup \{C \sqcap \top\} & \text{if } C = \forall r.D \\ \emptyset & \text{if } C = \bot \\ \{\forall r.\top \mid r \in N_R\} \cup \{\exists r.\top \mid r \in N_R\} & \text{if } C = \top \\ \cup \{\top \sqcup \top\} & \\ \cup \{A \mid A \in N_C, & \\ \quad \text{there is no } A' \in N_C \text{ with } A \sqsubset_\mathcal{T} A' \sqsubset_\mathcal{T} \top\} & \\ \cup \{\neg A \mid A \in N_C, & \\ \quad \text{there is no } A' \in N_C \text{ with } \bot \sqsubset_\mathcal{T} A' \sqsubset_\mathcal{T} A\} & \end{cases}$$

Proposition 5.13 ($\mathcal{ALC}$ refinement operator $\varphi_\downarrow$)
$\varphi_\downarrow$ is an $\mathcal{ALC}$ downward refinement operator.

Proof The proof is similar to the proof of Proposition 4.19 (page 34). We have to show that $D \in \varphi_\downarrow(C)$ implies $D \sqsubseteq_\mathcal{T} C$. Again, we can do this by structural induction over $\mathcal{ALC}$ concepts in negation normal form. We can ignore refinements of the form $C \rightsquigarrow C \sqcap \top$, because obviously $C \sqsubseteq_\mathcal{T} C \sqcap \top$ ($C \equiv_\mathcal{T} C \sqcap \top$).

- $C = \bot$: $D \in \varphi_\downarrow(C)$ is impossible, because $\varphi_\downarrow(\bot) = \emptyset$.
- $C = \top$: $D \sqsubseteq_\mathcal{T} C$ is trivially true for each concept D (and hence also for all refinements).

- $C = A$ ($A \in N_C$): $D \in \varphi_\downarrow(C)$ implies that D is also an atomic concept or the bottom concept and $D \sqsubset C$.
- $C = \neg A$: $D \in \varphi_\downarrow(C)$ implies that D is of the form $\neg A'$ with $A \sqsubset_\mathcal{T} A'$. $A \sqsubset_\mathcal{T} A'$ implies $\neg A' \sqsubset_\mathcal{T} \neg A$ by the semantics of negation.
- $C = \exists r.C'$: $D \in \varphi_\downarrow(C)$ implies that D is of the form $\exists r.D'$. We have $D' \sqsubseteq_\mathcal{T} C'$ by induction. For existential restrictions $\exists r.E \sqsubseteq_\mathcal{T} \exists r.E'$ if $E \sqsubset_\mathcal{T} E'$ holds in general (Badea and Nienhuys-Cheng, 2000). Thus we also have $\exists r.D' \sqsubseteq \exists r.C'$.
- $C = \forall r.C'$: This case is analogous to the previous one. For universal restrictions $\forall r.E \sqsubseteq_\mathcal{T} \forall r.E'$ if $E \sqsubset_\mathcal{T} E'$ holds in general (Badea and Nienhuys-Cheng, 2000).
- $C = C_1 \sqcap \cdots \sqcap C_n$: In this case one element of the conjunction is refined, so $D \sqsubseteq_\mathcal{T} C$ follows by induction.
- $C = C_1 \sqcup \cdots \sqcup C_n$: One possible refinement is to apply $\varphi_\downarrow$ to one element of the disjunction, so $D \sqsubseteq_\mathcal{T} C$ follows by induction. Another possible refinement is to drop an element of the disjunction, when $D \sqsubseteq_\mathcal{T} C$ obviously also holds (dropping an element of the disjunction cannot generalise a concept in negation normal form). ■

Proposition 5.14 (completeness of $\varphi_\downarrow$)
$\varphi_\downarrow$ is complete.

Proof We will first show weak completeness of $\varphi_\downarrow$. We do this by structural induction over $\mathcal{ALC}$ concepts in negation normal form, i.e. we show that every concept in negation normal form can be reached by $\varphi_\downarrow$ from $\top$.

- Induction Base:
  - $\top$: $\top$ can trivially be reached from $\top$.
  - $\bot$: $\top \rightsquigarrow A_1 \rightsquigarrow \ldots \rightsquigarrow A_n \rightsquigarrow \bot$ (descending the subsumption hierarchy)
  - $A \in N_C$: $\top \rightsquigarrow A_1 \rightsquigarrow \ldots \rightsquigarrow A_n \rightsquigarrow A$ (descending the subsumption hierarchy until A is reached)
  - $\neg A (A \in N_C)$: $\top \rightsquigarrow \neg A_1 \rightsquigarrow \ldots \rightsquigarrow \neg A_n \rightsquigarrow \neg A$ (ascending the subsumption hierarchy of atomic concepts within the scope of a negation symbol)
- Induction Step:
  - $\exists r.C$: $\top \rightsquigarrow \exists r.\top \rightsquigarrow^* \exists r.C$ (last step is possible by induction)
  - $\forall r.C$: $\top \rightsquigarrow \forall r.\top \rightsquigarrow^* \forall r.C$ (last step is possible by induction)

- $C_1 \sqcap \cdots \sqcap C_n$: $\top \rightsquigarrow^* C_1$ (by induction) $\rightsquigarrow C_1 \sqcap \top \rightsquigarrow^* C_1 \sqcap C_2 \rightsquigarrow^* C_1 \sqcap \cdots \sqcap C_n$
- $C_1 \sqcup \cdots \sqcup C_n$: $\top \rightsquigarrow \top \sqcup \top \rightsquigarrow^* C_1 \sqcup \top$ (by induction) $\rightsquigarrow C_1 \sqcup \top \sqcup \top \rightsquigarrow^* C_1 \sqcup C_2 \sqcup \top \rightsquigarrow^* C_1 \sqcup \cdots \sqcup C_n$

We have shown that $\varphi_\downarrow$ is weakly complete. The completeness can be derived from weak completeness by similar arguments as in the proof of Proposition 4.23 (page 39): If we have two $\mathcal{ALC}$ concepts C and D in negation normal form with $C \sqsubseteq_\mathcal{T} D$, then we can construct a concept $E = D \sqcap C$ for which we have $E \equiv_\mathcal{T} C$. E can be reached by the following refinement chain from D:

$$D \rightsquigarrow D \sqcap \top \rightsquigarrow^* D \sqcap C \text{ (by weak completeness of } \varphi_\downarrow)$$

Thus we have shown that we can reach a concept equivalent to C, which proves the completeness of $\varphi_\downarrow$. ■

Proposition 5.15 (finiteness of $\varphi_\downarrow$)
$\varphi_\downarrow$ is finite.

Proof Some rules in the definition of $\varphi_\downarrow$ apply $\varphi_\downarrow$ recursively to inner structures, e.g. specialising an element of a conjunction. Overall there are just five transformations which $\varphi_\downarrow$ applies to a concept (in brackets we write why this results in only finitely many possible refinements):

- dropping an element of a disjunction (there are only finitely many such elements)
- transforming a subconcept C to $C \sqcap \top$ (there are only finitely many subconcepts)
- specialising an atomic concept by an atomic concept (or bottom) further down in the subsumption hierarchy (there are only finitely many occurences of atomic concepts in a concept)
- specialising a negated atomic concept (see previous case)
- transforming a $\top$ symbol (there are only finitely many occurences of the $\top$ symbol in a concept and the set of possible refinements of $\top$ is finite)

Hence $\varphi_\downarrow$ is finite. ■

We have shown that $\varphi_\downarrow$ is complete and finite, which makes it suitable to be used in a genetic refinement operator (we will define the dual upward operator below). Avoiding redundancy is not very important within the GP framework since we only follow one specific refinement chain anyway. (Also note that due to Proposition 4.12 redundancy cannot be avoided in a complete $\mathcal{ALC}$ refinement operator even if we would consider this a desired goal.) Properness is more interesting than non-redundancy. However, by

Proposition 4.10 it is impossible to obtain properness without loosing finiteness (which we stated as a prerequisite for transforming the refinement into a genetic refinement operator). In particular note that the closure construct introduced for $\rho_\downarrow$ in Section 4 would lead to an infinite operator.

This is the dual upward refinement operator, which we need to get a genetic refinement operator:

$$\varphi_\uparrow(C) = \begin{cases} \{C_1 \sqcap \dots \sqcap C_{i-1} \sqcap D \sqcap C_{i+1} \sqcap \dots \sqcap C_n & \text{if } C = C_1 \sqcap \dots \sqcap C_n (n \geq 2) \\ \quad \mid D \in \varphi_\uparrow(C_i), 1 \leq i \leq n\} & \\ \cup \{C_1 \sqcap \dots \sqcap C_{i-1} \sqcap C_{i+1} \sqcap \dots \sqcap C_n & \\ \quad \mid 1 \leq i \leq n\} & \\ \cup \{C \sqcup \bot\} & \\ \{C_1 \sqcup \dots \sqcup C_{i-1} \sqcup D \sqcup C_{i+1} \sqcup \dots \sqcup C_n & \text{if } C = C_1 \sqcup \dots \sqcup C_n (n \geq 2) \\ \quad \mid D \in \varphi_\uparrow(C_i), 1 \leq i \leq n\} & \\ \cup \{C \sqcup \bot\} & \\ \{A' \mid A \sqsubset_\mathcal{T} A', A' \in N_C, & \text{if } C = A \ (A \in N_C) \\ \ \text{there is no } A'' \in N_C \text{ with } A \sqsubset_\mathcal{T} A'' \sqsubset_\mathcal{T} A'\} & \\ \cup \{C \sqcup \bot\} & \\ \{\neg A' \mid A' \sqsubset_\mathcal{T} A, A' \in N_C \cup \{\top\}, & \text{if } C = \neg A \ (A \in N_C) \\ \ \text{there is no } A'' \in N_C \text{ with } A' \sqsubset_\mathcal{T} A'' \sqsubset_\mathcal{T} A\} & \\ \cup \{C \sqcup \bot\} & \\ \{\exists r.E \mid E \in \varphi_\uparrow(D)\} \cup \ \{C \sqcup \bot\} & \text{if } C = \exists r.D \\ \{\forall r.E \mid E \in \varphi_\uparrow(D)\} \cup \ \{C \sqcup \bot\} & \text{if } C = \forall r.D \\ \emptyset & \text{if } C = \top \\ \{\forall r.\bot \mid r \in N_R\} \cup \{\exists r.\bot \mid r \in N_R\} & \text{if } C = \bot \\ \cup \{\bot \sqcap \bot\} & \\ \cup \{A \mid A \in N_C, & \\ \quad \text{there is no } A' \in N_C \text{ with } \bot \sqsubset_\mathcal{T} A' \sqsubset_\mathcal{T} A\} & \\ \cup \{\neg A \mid A \in N_C, & \\ \quad \text{there is no } A' \in N_C \text{ with } A \sqsubset_\mathcal{T} A' \sqsubset_\mathcal{T} \top\} & \end{cases}$$

Proposition 5.16 (properties of $\varphi_\uparrow$)
$\varphi_\uparrow$ is a complete and finite $\mathcal{ALC}$ upward refinement operator.

We omit the proof of Proposition 5.16 since it is very similar to what we have already shown in Propositions 5.13, 5.14, 5.15. $\varphi_\uparrow$ is constructed in the same way like $\varphi_\downarrow$ and has the same the properties.

From $\varphi_\downarrow$ and $\varphi_\uparrow$ we can construct a genetic refinement operator as described in Equation 4 (page 74). This new operator is ready to be used within the GP framework and combines classical induction with evolutionary approaches. Below we will look at the features of this approach and compare it with classical induction and standard Genetic Programming.

Characteristics of the Introduced Approach

In Section 4 we analysed $\mathcal{ALC}$ refinement operators and showed how they can be usefully combined with a heuristic to form a full learning algorithm. In this section we also used refinement operators, but combined them with Genetic Programming. What are the differences between both methods? In the heuristic search we have a search tree, whereas in the GP case there is a fixed number of individuals. In this way we can think of the GP search as a set of individuals moving in the search space. Their movement is stochastic, but not completely random (by the design of the genetic refinement operator). We can see that the two approaches are very different with respect to space complexity. The GP approach needs constant space if we use the number of stored concepts as a measure. In contrast the search tree keeps growing (exponentially with the size of concepts). This is, of course, necessary to traverse the search space in a well structured deterministic redundancy-free fashion. Another difference is that in the GP approach we do both upward and downward refinements. In Section 4 we traversed the search space in only one direction (top-down or bottom-up). Using both directions of refinement allows to start the search from random points in the search space, which may be beneficial compared to starting from either $\top$ or $\bot$.

Compared to the standard GP algorithm we use a mutation style genetic refinement operator as main operator whereas this is usually crossover in the standard case. We can, of course, still use crossover (and the other operators presented in the introduction) as secondary operators. This can be useful to overcome local maxima. The changes of the operator were necessary, because learning concepts in Description Logics is not a task GP algorithms can handle naturally in an efficient way. This is mainly due to the fact that the fitness landscape of concepts is very ragged.

The bias of the learning algorithm is mostly controlled by the fitness function. We prefer smaller concepts since they generalise better to unseen examples. The representation language was only restricted in that we limit the search to concepts in negation normal form. Of course, this is no limitation with respect to the semantics of concepts, i.e. we can always find a solution if one exists.

Related Work

To the best of our knowledge, there has been no attempt to use evolutionary approaches for learning concepts in Description Logics. Hence, there is no closely related work we are aware of. For an overview of research on non-evolutionary approaches see the description of related work in Section 4 on page 54. Although evolutionary methods have not been considered for learning in Description Logics before, they have been used for inducing logic programs. On page 71 we summarized these approaches and gave some pointers to interesting papers. Some of these approaches require a fixed template the solution must fit in. Such approaches are usually based on Genetic Algorithms, i.e. they do not allow a variable length of the solution. We did not consider this approach due to its restricted flexibilty. The systems based on Genetic Programming, i.e. SIA01 (Augier et al., 1995), ECL (Divina and Marchiori, 2002), and GLPS (Wong and Leung, 1995) are closer to our approach. Similar to our research, they also concluded that standard Genetic Programming is not sufficient to solve their learning problem. As a consequence, they invented new operators. As far as we know, they did not try to connect refinement operators and Genetic Programming explicitly, which distinguishes them from our approach. This strategy enables us to make use of the powerful theory of refinement operators, in particular the full property analyses we performed in Section 4. We cannot directly compare the operators, which are used in their systems, with the operators we have developed, since the target language (logic programs) is different.

5.5 Other Improvements

In this section we discuss other ways of improving or modifying the standard Genetic Programing approach besides the use of genetic refinement operators.

Learning From Uncertain Data

When we discussed the fitness measurement in our GP algorithm we have noted that it is easy to handle noise by including a penalty for the length of a concept in the fitness function. We will now show that is also possible (albeit a bit more difficult) to learn from uncertain data. We first have to introduce what we mean by learning from uncertain data.

An example in this scenario is a tuple $(z, \texttt{Target}(a))$, where `Target` is the concept we want to learn, $a \in N_I$ is an object, and $0 \leq z \leq 1$ $(z \in \mathbb{R})$ a number expressing the certainty that the fact is true. $z = 0$ means that we are certain that the fact is not true. This is what we classically considered to be a negative example. $z = 1$ is a classical positive example. Often this kind of data arises naturally from preprocessing steps like Bayesian or Neural Network classification, where one can compute the certainty that a classification is correct.

To incorporate uncertain data into the GP framework we have to modify the fitness measurement. The idea is, of course, that a positively classified example should have a high z-value and an example, which is not classified as positive, should have a low z-value. For $z = 0.5$ it should not matter whether the corresponding example is classified positively or not by a concept. ($z = 0.5$ means that the probability that the fact is true equals the probability that the fact is false.) The following fitness function can be used (we designed it such that fitness values are always positive):

$$f_{\mathcal{K}}(C) = \sum_{e \in E} s_{\mathcal{K}}(C, e)$$

$$s_{\mathcal{K}}(C, (z, \texttt{Target}(a))) = \begin{cases} z & \text{if } \mathcal{K} \cup \{\texttt{Target} \equiv C\} \models \texttt{Target}(a) \\ 1 - z & \text{if } \mathcal{K} \cup \{\texttt{Target} \equiv C\} \not\models \texttt{Target}(a) \end{cases}$$

The modification of the fitness function is sufficient to be able to apply the standard GP approach. For the genetic refinement operators we have introduced we need to do additional work. Remember that we calculated a value which determined whether upward or downward refinement is used. This value depends on the percentage of covered examples (see Equation 4 on page 74). When learning from uncertain data we no longer have this strict separation between positive and negative examples, so we have to modify this approach.

The idea is to replace the expressions $\frac{|neg_{\mathcal{K}}(C)|}{|E^-|}$ and $\frac{|pos_{\mathcal{K}}(C)|}{|E^+|}$ in Equation 4 by corresponding expressions, which take the uncertainty into account. We look at examples with $z < 0.5$ as negative examples and $z \geq 0.5$ corresponds to classical positive examples. The sum of the fitness values for all examples we consider as negative is:

$$\sum_{\substack{(z,\texttt{Target}(a)) \in E \\ \text{with } z<0.5}} s_{\mathcal{K}}(C, (z, \texttt{Target}(a)))$$

To turn this into an expression, which corresponds to $\frac{|neg_{\mathcal{K}}(C)|}{|E^-|}$ we have to normalize this value such that a value of 1 is obtained if every negative example is classified correctly and we get a value of 0 if none of the negative examples is classified correctly. The resulting expression we get is listed as v below (for better readability the denominator ist not simplified). Analogously w corresponds to $\frac{|pos_{\mathcal{K}}(C)|}{|E^+|}$. To get a better overview we show the complete definition of a genetic refinement operator with support for reasoning from uncertain data:

$$\varphi(C) = \begin{cases} \text{rand}(\varphi_{\downarrow}(C)) & \text{with probability } \frac{v}{1+v-w} \\ \text{rand}(\varphi_{\uparrow}(C)) & \text{with probability } \frac{1-w}{1+v-w} \end{cases} \tag{5}$$

$$v = \frac{\sum\limits_{\substack{(z,\texttt{Target}(a)) \in E \\ \text{with } z<0.5}} s_{\mathcal{K}}(C, (z, \texttt{Target}(a))) - z}{\sum\limits_{\substack{(z,\texttt{Target}(a)) \in E \\ \text{with } z<0.5}} (1-z) - z}$$

$$w = \frac{\sum\limits_{\substack{(z,\texttt{Target}(a)) \in E \\ \text{with } z\geq0.5}} s_{\mathcal{K}}(C, (z, \texttt{Target}(a))) - (1-z)}{\sum\limits_{\substack{(z,\texttt{Target}(a)) \in E \\ \text{with } z\geq0.5}} z - (1-z)}$$

This extension of the presented framework may be very useful since, as we already mentioned, a lot of data which is preprocessed by other Machine Learning algorithms (Bayesian Networks, Neural Networks, etc.) produces classification probabilites instead of only binary classifications (true or false). What we have described above can be used

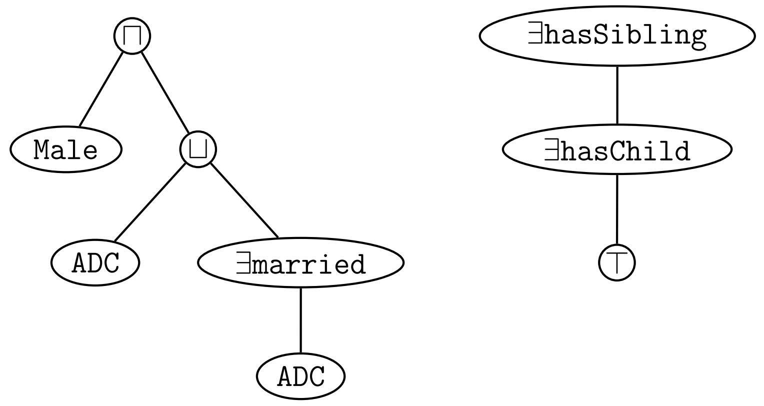


Figure 4: learning with ADCs - main tree on the left and ADC tree on the right

to model the fact that we do not know exactly whether a fact is true or not. Please note that the underlying background knowledge base has not changed, so we can use the same reasoning algorithms as before. The only changes we made were: a) an adaptation of the fitness function and b) a change in the calculation of the probabilities for upward and downward refinement in the genetic refinement operator.

Automatically Defined Concepts

One key idea present in Inductive Logic Programming (Nienhuys-Cheng and de Wolf, 1997) is the automatic invention of new predicates, e.g. by the inverse resolution method (Muggleton, 1995a). For learning in Description Logics this corresponds to the invention of new concepts, which are then used in the definition of the target concept we want to learn. This idea is very interesting since people tend to view a machine, which automatically invents something to improve its performance on a given task, as intelligent.

Genetic Programming offers the possibility of inventing new concepts. To do this we slightly modified an approach already known as *Automatically Defined Functions (ADFs)* (Koza, 1993) and called it *Automatically Defined Concepts (ADCs)*. Roughly the idea in ADFs is to give the tree, which represents an individual, a fixed structure, where one subtree is the main tree and other parts of the tree are function trees. Function trees can take arguments and return values similar to functions in imperative programming languages.

We adapted this approach to the learning problem in Description Logics and simplified this in order to reach a reasonable performance. Instead of evolving just one tree we now evolve two trees. One tree is the main tree and the other the ADC tree. The alphabet of the main tree is extended by a symbol `ADC`, which represents the concept defined in the ADC tree. An example for two such trees is shown in Figure 4.

It is not hard to change the standard GP framework to allow ADCs. The genetic operators have to be changed, such that they only work on compatible trees. For crossover we first select either the main or ADC tree randomly. If the main tree is selected then we also have to select the main tree of the second parent. Analogously if the ADC tree of the first parent is selected then we also have to select the ADC tree of the second parent. After that standard crossover can be performed. Other genetic operators have to be treated in a similar way.

The ADC approach is not completely compatible with the genetic refinement operators we have introduced. In principle we can first select whether we want to perform upward or downward refinement as usual. In a second step we select whether the main or the ADC tree is refined and finally we apply the refinement operator to the corresponding tree. However, the problem is that if the automically defined concept occurs negated and non-negated in the definition of the target concept, then the effect of e.g. upward refinement on the ADC tree may not be an upward refinement of the target concept definition. This means that changes on the ADC tree would not necessarily have the desired effect.

Hill Climbing Operators

An interesting idea is to combine hill climbing methods with Genetic Programming. The idea of hill climbing as a search algorithm is to look at the neighbourhood of a point in the search space and move to that point in the neighbourhood, which is most promising according to some measurement. In our case this measurement is fitness and the neighbourhood of a point in the search space, which is an $\mathcal{ALC}$ concept, is a set of syntactically similar concepts. More exactly for a given concept C we define the neighbourhood of C as the set:

$$N(C) = \{\neg C\} \cup \{C \sqcup A \mid A \in N_C\} \cup \{C \sqcap A \mid A \in n_C\} \cup \{\exists r.C \mid r \in N_R\} \cup \{\forall r.C \mid r \in N_R\}$$

The genetic hill climbing operator works as follows: It takes one individual, which represents a concept C, as input. Given C it evaluates the fitness of all concepts in $N(C)$. Then it computes a subset of individuals with the highest fitness of all individuals in $N(C)$ and C itself (it is a set and not a single individual, because individuals may have the same fitness). From this set one element is randomly selected and returned.

This operator makes sense if we are able to evaluate the fitness of concepts in the neighbourhood very fast. In Section 6 we will introduce an algorithm, which is able to do this. In particular it can use the fact that we already evaluated C (obviously C is in the population so we have evaluated its fitness). The algorithm in Section 6 can easily be extended to compute the (approximate) fitness of all individuals in the neighbourhood with only a few basic set operations. Summed up the hill climbing operator is a way to efficiently search a possibly large set of related concepts.

6 Concept Quality Measurement

6.1 Problems in Concept Learning

When learning $\mathcal{ALC}$ concepts within the introduced learning framework problems can arise, which we will describe in the sequel. As we have introduced the goal of learning is to find a concept C such that when adding the definition $\texttt{Target} \equiv C$ to our background knowledge base the positve examples follow and the negatives do not follow. Example 6.1 shows, which problems can arise and we will later see how they can be solved.

Example 6.1 (problems in concept learning)
Consider the following knowledge base $\mathcal{K}$ with an empty TBox and the following ABox $\mathcal{A}$:

```
Male(A).
Female(B).
Male(C).
Male(D).
Female(E).
hasChild(A,C).
hasChild(A,D).
hasChild(B,C).
hasChild(B,D).
hasChild(C,E).
```

Let $E^+ = \{\texttt{Target(A)}, \texttt{Target(B)}\}$ and $E^- = \{\texttt{Target(C)}\}$.

Clearly the hypothesis $\forall\texttt{hasChild.Male}$ seems to be an intuitive solution. However, it does not solve the learning problem, because the positive examples `A` and `B` are not covered. The reason is, of course, that we have the Open World Assumption in Description Logics. This means that `A` and `B` could have more children than the ones described in the knowledge base and these children do not necessarily need to be male. $\square$

In general the problem is that for concepts of the form $\forall r.C$ we have that even if all known r-fillers in $\mathcal{A}$ for a positive example a satisfy C we cannot deduce $\forall r.C(a)$. Even if the used description language allows negative role assertions of the form $\neg r(a, b)$ this problem cannot be avoided, because it could still be that an unknown object, i.e. it does not appear in $\mathcal{A}$, is an r-filler of a that does not satisfy C. This is due to the Open World Assumption (OWA). Although the OWA is usually prefered when reasoning in Description Logics it is a restriction for concept learning as we have seen above. One approach to overcome this problem is to fix the domain of the interpretations we allow (e.g. we only allow objects appearing in the ABox). Together with allowing negated role assertions this is one way to solve the problem[2]. To support this kind of reasoning in learning algorithms in Description Logics we create an efficient retrieval algorithm. The purpose is not only to solve the mentioned problem, but also to provide a very efficient method for testing the quality of concepts.

[2]Another way to solve this problem by epistemic operators is described in (Badea and Nienhuys-Cheng, 2000). See also the report (Motik and Rosati, 2004) about closing Semantic Web ontologies.

6.2 A Fast Retrieval Algorithm

As we have seen retrieval is a reasoning service, which can be used for learning concepts. In this section we will introduce a fast and simple retrieval algorithm, which makes it possible to reason under a fixed domain. Before we do this, it is necessary to introduce some notions.

Definition 6.2 (Δ-interpretation)
A Δ-interpretation is an interpretation with domain Δ and $a^{\mathcal{I}} = a$ for all objects. A Δ-model is a model, which is a Δ-interpretation. □

Definition 6.3 (instance and retrieval with respect to a fixed domain)
Let $\mathcal{A}$ be an Abox, $\mathcal{T}$ a TBox, $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ a knowledge base, Δ an interpretation domain, C a concept, and $a \in N_I$.

a is an instance of C with respect to $\mathcal{A}$ and Δ, denoted by $\mathcal{A} \models_\Delta C(a)$, iff in any Δ-model $\mathcal{I}_\Delta$ with $N_I \subseteq \Delta$ we have $a^{\mathcal{I}_\Delta} \in C^{\mathcal{I}_\Delta}$.

a is an instance of C with respect to $\mathcal{K}$ and Δ, denoted by $\mathcal{K} \models_\Delta C(a)$, iff in any Δ-model $\mathcal{I}_\Delta$ with $N_I \subseteq \Delta$ we have $a^{\mathcal{I}_\Delta} \in C^{\mathcal{I}_\Delta}$.

The *retrieval $R_{\mathcal{A},\Delta}(C)$ of a concept C with respect to $\mathcal{A}$ and Δ* is defined as $R_{\mathcal{A},\Delta}(C) = \{a \mid a \in N_I \text{ and } \mathcal{A} \models_\Delta C(a)\}$.

The *retrieval $R_{\mathcal{K},\Delta}(C)$ of a concept C with respect to $\mathcal{K}$ and Δ* is defined as $R_{\mathcal{K},\Delta}(C) = \{a \mid a \in N_I \text{ and } \mathcal{K} \models_\Delta C(a)\}$. □

The algorithms works on a flat ABox, which only consists of positive and negative assertions about atomic concepts and roles.

Definition 6.4 (flat Abox)
An Abox $\mathcal{A}$ is called *flat* if it contains only assertions of the form $A(a)$, $\neg A(a)$, $r(a,b)$, and $\neg r(a,b)$ $(A \in N_C, r \in N_R, a, b \in N_I)$. □

It is possible to transform a knowledge base into a flat ABox by executing retrievals of the form $\neg A$ and A for any atomic concept A and by finding all tuples which fulfill $r(a,b)$ and $\neg r(a,b)$ for any role r and objects a, b. The latter type of reasoning is supported by some DL-reasoners e.g. KAON2. Currently many description languages and in particular the OWL ontology language do not support negated role assertions. However, they can be emulated by augmenting Description Logics with other Logics (such as F-Logic (Balaban, 1995)) and the upcoming OWL 1.1 is likely to support negated role assertions, because it will be based on $\mathcal{SROIQ}$ (Horrocks et al., 2006).

It should be noted that eliminating the TBox in this way can cause a loss of information. This means that a reduction of a knowledge base to a flat ABox does not preserve equivalence. If $\mathcal{K}$ is a knowledge base and $\mathcal{A}$ the flat ABox obtained by transforming $\mathcal{K}$ to $\mathcal{A}$, then it can be the case that for some concept C and object a we have $\mathcal{K} \models C(a)$, but $\mathcal{A} \not\models C(a)$. As an example, consider an empty TBox and an ABox consisting only of the assertion $A_1 \sqcup A_2(a)$, where A_1, A_2 are atomic concepts, and a an object. The transformation to a flat ABox $\mathcal{A}$ results in an empty ABox. Hence, we have $\mathcal{K} \models A_1 \sqcup A_2(a)$, but $\mathcal{A} \not\models A_1 \sqcup A_2(a)$.

In the other direction, however, the situation is different. If for a flat ABox $\mathcal{A}$ we have $\mathcal{A} \models C(a)$, then for the corresponding knowledge base $\mathcal{K} = (\mathcal{T}', \mathcal{A}')$ we also have $\mathcal{K} \models C(a)$. The reason is that Description Logics, as we have presented it in this thesis, are a monotonic knowledge representation formalism. This means that adding knowledge never invalidates a deduction. The knowledge base $\mathcal{K} = (\mathcal{T}', \mathcal{A}')$ is equivalent to the knowledge base $\mathcal{K}' = (\mathcal{T}', \mathcal{A}' \cup \mathcal{A})$, because the transformed flat ABox does not contain any assertions, which we cannot already deduce from $\mathcal{K}$. Hence, $\mathcal{A} \models C(a)$ implies $\mathcal{K} \models C(a)$.

Definition 6.5 ($C^+_{\mathcal{A},\Delta}$, $C^-_{\mathcal{A},\Delta}$, $r^+_{\mathcal{A}}$, $r^-_{\mathcal{A}}$)
Let $\mathcal{A}$ be a flat Abox, N_C the set of concept names, N_R the set of role names, N_I the set of individual names appearing in $\mathcal{A}$ and Δ a superset of N_I.

- If $r \in N_R$, then $r^+_{\mathcal{A}} = \{(a,b) \mid r(a,b) \in \mathcal{A}\}$ and $r^-_{\mathcal{A}} = \{(a,b) \mid \neg r(a,b) \in \mathcal{A}\}$.
- If $C \in N_C$ is an atomic concept, then $C^+_{\mathcal{A},\Delta} = \{a \mid C(a) \in \mathcal{A}\}$ and $C^-_{\mathcal{A},\Delta} = \{a \mid \neg C(a) \in \mathcal{A}\}$. Additionally we have $\top^+_{\mathcal{A},\Delta} = \Delta$, $\top^-_{\mathcal{A},\Delta} = \emptyset$, $\bot^+_{\mathcal{A},\Delta} = \emptyset$, and $\bot^-_{\mathcal{A},\Delta} = \Delta$.
- $C^+_{\mathcal{A},\Delta}$ and $C^-_{\mathcal{A},\Delta}$ are defined inductively over the structure of $\mathcal{ALC}$ concepts:
  - $(\neg D)^+_{\mathcal{A},\Delta} = D^-_{\mathcal{A},\Delta}$
  - $(\neg D)^-_{\mathcal{A},\Delta} = D^+_{\mathcal{A},\Delta}$
  - $(D \sqcap E)^+_{\mathcal{A},\Delta} = D^+_{\mathcal{A},\Delta} \cap E^+_{\mathcal{A},\Delta}$
  - $(D \sqcap E)^-_{\mathcal{A},\Delta} = D^-_{\mathcal{A},\Delta} \cup E^-_{\mathcal{A},\Delta}$
  - $(D \sqcup E)^+_{\mathcal{A},\Delta} = D^+_{\mathcal{A},\Delta} \cup E^+_{\mathcal{A},\Delta}$
  - $(D \sqcup E)^-_{\mathcal{A},\Delta} = D^-_{\mathcal{A},\Delta} \cap E^-_{\mathcal{A},\Delta}$
  - $(\exists r.D)^+_{\mathcal{A},\Delta} = \{a \mid \exists b.(a,b) \in r^+_{\mathcal{A}} \wedge b \in D^+_{\mathcal{A},\Delta}\}$
  - $(\exists r.D)^-_{\mathcal{A},\Delta} = \{a \mid \forall b.((a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}}) \implies b \in D^-_{\mathcal{A},\Delta}\}$
  - $(\forall r.D)^+_{\mathcal{A},\Delta} = \{a \mid \forall b.((a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}}) \implies b \in D^+_{\mathcal{A},\Delta}\}$
  - $(\forall r.D)^-_{\mathcal{A},\Delta} = \{a \mid \exists b.(a,b) \in r^+_{\mathcal{A}} \wedge b \in D^-_{\mathcal{A}}\}$ □

Algorithm 6.6 (retrieval)
We can define a retrieval algorithm for a concept C with respect to a flat ABox $\mathcal{A}$ and a domain Δ by computing $C^+_{\mathcal{A},\Delta}$. ($C^-_{\mathcal{A},\Delta}$ is then the answer of the retrieval for $\neg C$.) For the execution of the algorithm we assume that the C is stored as a tree (see Section 5) and the sets $D^+_{\mathcal{A},\Delta}$ and $D^-_{\mathcal{A},\Delta}$ for each node, which represents a concept D, are computed from leaf to root.

We write $A \vdash_\Delta C(a)$ iff $C(a)$ is a consequence of the algorithm with respect to the flat ABox $\mathcal{A}$ and domain Δ. This relation is defined in the following way:

$$A \vdash_\Delta C(a) \text{ iff } a \in C^+_{\mathcal{A},\Delta}$$

We will now analyse the properties of this algorithm.

Proposition 6.7 (soundness)
$\mathcal{A} \vdash_\Delta C(a)$ *implies* $\mathcal{A} \models_\Delta C(a)$.

PROOF To prove the proposition we show that $a \in C^+_{\mathcal{A},\Delta}$ implies $\mathcal{A} \models_\Delta C(a)$ and $a \in C^-_{\mathcal{A},\Delta}$ implies $\mathcal{A} \models_\Delta \neg C(a)$. We do this by an induction over the structure of $\mathcal{ALC}$ concepts.

1. *Induction Base:*
   a) C is $\top$.
      $a \in C^+_{\mathcal{A},\Delta}$: In this case $\mathcal{A} \models_\Delta \top(a)$ is trivially true.
      $a \in C^-_{\mathcal{A},\Delta}$: This is never true by Definition 6.5, so the implication is true.
   b) C is $\bot$.
      $a \in C^+_{\mathcal{A},\Delta}$: This is never true by Definition 6.5, so the implication is true.
      $a \in C^-_{\mathcal{A},\Delta}$: In this case $\mathcal{A} \models_\Delta \neg\bot(a)$ is trivially true.
   c) C is an atomic concept.
      $a \in C^+_{\mathcal{A},\Delta}$ means that we have the assertion $C(a)$ in $\mathcal{A}$ by definition of $C^+_{\mathcal{A},\Delta}$. So by definition of a model of an ABox (Definition 2.11, page 10) we have $\mathcal{A} \models_\Delta C(a)$.
      $a \in C^-_{\mathcal{A},\Delta}$ means that the assertion $\neg C(a)$ is in $\mathcal{A}$, so by the same argument we get $\mathcal{A} \models_\Delta \neg C(a)$.
2. *Induction Step:*
   a) C is of the form $\neg D$:
      i. $C^+_{\mathcal{A},\Delta}$:

$$\begin{aligned} & a \in (\neg D)^+_{\mathcal{A},\Delta} \\ \iff\ & a \in D^-_{\mathcal{A},\Delta} \quad \text{(by Definition 6.5)} \\ \implies\ & \mathcal{A} \models_\Delta \neg D(a) \quad \text{(by induction)} \end{aligned}$$

ii. $C^-_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (\neg D)^-_{\mathcal{A},\Delta} \\
\iff & a \in D^+_{\mathcal{A},\Delta} \quad \text{(by Definition 6.5)} \\
\implies & \mathcal{A} \models_\Delta D(a) \quad \text{(by induction)} \\
\iff & \mathcal{A} \models_\Delta \neg(\neg D(a))
\end{aligned}$$

b) C is of the form $D \sqcap E$:

i. $C^+_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (D \sqcap E)^+_{\mathcal{A},\Delta} \\
\iff & a \in D^+_{\mathcal{A},\Delta} \wedge a \in E^+_{\mathcal{A},\Delta} \quad \text{(by Definition 6.5)} \\
\implies & \mathcal{A} \models_\Delta D(a) \wedge \mathcal{A} \models_\Delta E(a) \quad \text{(by induction)} \\
\implies & \forall I_\Delta . \mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies (a^{\mathcal{I}_\Delta} \in D^{\mathcal{I}_\Delta} \wedge a^{\mathcal{I}_\Delta} \in E^{\mathcal{I}_\Delta}) \quad \text{(by Definition 6.3)} \\
\iff & \mathcal{A} \models_\Delta (D \sqcap E)(a) \quad \text{(see Table 1 on page 9)}
\end{aligned}$$

ii. $C^-_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (D \sqcap E)^-_{\mathcal{A},\Delta} \\
\iff & a \in D^-_{\mathcal{A},\Delta} \vee a \in E^-_{\mathcal{A},\Delta} \quad \text{(by Definition 6.5)} \\
\implies & \mathcal{A} \models_\Delta \neg D(a) \vee \mathcal{A} \models_\Delta \neg E(a) \quad \text{(by induction)} \\
\implies & \forall I_\Delta . \mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies (a^{\mathcal{I}_\Delta} \notin D^{\mathcal{I}_\Delta} \vee a^{\mathcal{I}_\Delta} \notin E^{\mathcal{I}_\Delta}) \quad \text{(by Definition 6.3)} \\
\iff & \forall I_\Delta . \mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \neg(a^{\mathcal{I}_\Delta} \in D^{\mathcal{I}_\Delta} \wedge a^{\mathcal{I}_\Delta} \in E^{\mathcal{I}_\Delta}) \\
\iff & \forall I_\Delta . \mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \neg(a^{\mathcal{I}_\Delta} \in (D \sqcap E)^{\mathcal{I}_\Delta}) \\
\iff & \mathcal{A} \models_\Delta \neg(D \sqcap E)(a)
\end{aligned}$$

c) C is of the form $D \sqcup E$ (similar to the previous case):

i. $C^+_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (D \sqcup E)^+_{\mathcal{A},\Delta} \\
\iff & a \in D^+_{\mathcal{A},\Delta} \vee a \in E^+_{\mathcal{A},\Delta} \quad \text{(by Definition 6.5)} \\
\implies & \mathcal{A} \models_\Delta D(a) \vee \mathcal{A} \models_\Delta E(a) \quad \text{(by induction)} \\
\implies & \mathcal{A} \models_\Delta (D \sqcup E)(a) \quad \text{(analogous to 2b)}
\end{aligned}$$

ii. $C^-_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (D \sqcup E)^-_{\mathcal{A},\Delta} \\
\iff & a \in D^-_{\mathcal{A},\Delta} \wedge a \in E^-_{\mathcal{A},\Delta} \quad \text{(by Definition 6.5)} \\
\implies & \mathcal{A} \models_\Delta \neg D(a) \wedge \mathcal{A} \models_\Delta \neg E(a) \quad \text{(by induction)} \\
\implies & \forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies (a^{\mathcal{I}_\Delta} \notin D^{\mathcal{I}_\Delta} \wedge a^{\mathcal{I}_\Delta} \notin E^{\mathcal{I}_\Delta}) \quad \text{(by Definition 6.3)} \\
\iff & \forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \neg(a^{\mathcal{I}_\Delta} \in D^{\mathcal{I}_\Delta} \vee a^{\mathcal{I}_\Delta} \in E^{\mathcal{I}_\Delta}) \\
\iff & \forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \neg(a^{\mathcal{I}_\Delta} \in (D \sqcup E)^{\mathcal{I}_\Delta}) \\
\iff & \mathcal{A} \models_\Delta \neg(D \sqcup E)(a)
\end{aligned}$$

d) C is of the form $\exists r.D$:

i. $C^+_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (\exists r.D)^+_{\mathcal{A},\Delta} \\
\iff & \exists b.(a,b) \in r^+_{\mathcal{A}} \wedge b \in D^+_{\mathcal{A},\Delta} \quad \text{(by Definition 6.5)} \\
\implies & \exists b.(a,b) \in r^+_{\mathcal{A}} \wedge \mathcal{A} \models_\Delta D(b) \quad \text{(by induction)} \\
\implies & \exists b.\forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies (a,b) \in r^{\mathcal{I}_\Delta} \wedge b \in D^{\mathcal{I}_\Delta} \\
\implies & \forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \exists b.(a,b) \in r^{\mathcal{I}_\Delta} \wedge b \in D^{\mathcal{I}_\Delta} \\
\iff & \mathcal{A} \models_\Delta (\exists r.D)(a)
\end{aligned}$$

ii. $C^-_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (\exists r.D)^-_{\mathcal{A},\Delta} && (6) \\
\implies & \forall b.((a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}} \implies b \in D^-_{\mathcal{A},\Delta}) \quad \text{(by Definition 6.5)} && (7)
\end{aligned}$$

Let $\mathcal{I}_\Delta$ be an arbitrary Δ-model of $\mathcal{A}$. To prove $\mathcal{A} \models_\Delta \neg(\exists r.D)(a)$ we have to show $\neg(\exists b.(a,b) \in r^{\mathcal{I}_\Delta} \wedge b \in D^{\mathcal{I}_\Delta})$, which is equivalent to $\forall b.(a,b) \notin r^{\mathcal{I}_\Delta} \vee b \notin D^{\mathcal{I}_\Delta}$.
If we have $(a,b) \notin \Delta \times \Delta \setminus r^-_{\mathcal{A}}$, then $(a,b) \in r^-_{\mathcal{A}}$ and an assertion $\neg r(a,b)$ exists in $\mathcal{A}$. In this case the disjunction is true, because the first disjunct is true.
If we have $(a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}}$ we get $b \in D^-_{\mathcal{A},\Delta}$ by (7). So by induction we have $A \models_\Delta \neg D(b)$ and $b \notin D^{\mathcal{I}_\Delta}$. Thus the disjunction is true, because the second disjunct is true.

e) C is of the form $\forall r.D$:

i. $C^+_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (\forall r.D)^+_{\mathcal{A},\Delta} && (8) \\
\implies & \forall b.((a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}} \implies b \in D^+_{\mathcal{A},\Delta}) \quad \text{(by Definition 6.5)} && (9)
\end{aligned}$$

Let $\mathcal{I}_\Delta$ be an arbitrary Δ-model of $\mathcal{A}$. To prove $\mathcal{A} \models_\Delta \forall r.D(a)$ we have to show $\forall b.(a,b) \in r^{\mathcal{I}_\Delta} \rightarrow b \in D^{\mathcal{I}_\Delta}$.
If we have $(a,b) \notin \Delta \times \Delta \setminus r^-_\mathcal{A}$, then $(a,b) \in r^-_\mathcal{A}$ and an assertion $\neg r(a,b)$ exists in $\mathcal{A}$. In this case the implication is true, because the premise is false.
If we have $(a,b) \in \Delta \times \Delta \setminus r^-_\mathcal{A}$ we get $b \in D^+_{\mathcal{A},\Delta}$ by (9). So by induction we have $A \models_\Delta D(b)$ and $b \in D^{\mathcal{I}_\Delta}$. Thus the implication is true.

ii. $C^-_{\mathcal{A},\Delta}$:

$$\begin{aligned}
& a \in (\forall r.D)^-_{\mathcal{A},\Delta} \\
\iff & \exists b.(a,b) \in r^+_\mathcal{A} \wedge b \in D^-_{\mathcal{A},\Delta} \quad \text{(by Definition 6.5)} \\
\implies & \exists b.(a,b) \in r^+_\mathcal{A} \wedge \mathcal{A} \models_\Delta \neg D(b) \quad \text{(by induction)} \\
\implies & \exists b.\forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies (a,b) \in r^{\mathcal{I}_\Delta} \wedge b \notin D^{\mathcal{I}_\Delta} \\
\implies & \forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \exists b.(a,b) \in r^{\mathcal{I}_\Delta} \wedge b \notin D^{\mathcal{I}_\Delta} \\
\implies & \forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \neg\neg(\exists b.(a,b) \in r^{\mathcal{I}_\Delta} \wedge b \notin D^{\mathcal{I}_\Delta}) \\
\implies & \forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \neg(\forall b.\neg((a,b) \in r^{\mathcal{I}_\Delta}) \vee b \in D^{\mathcal{I}_\Delta}) \\
\implies & \forall I_\Delta.\mathcal{I}_\Delta \models_\Delta \mathcal{A} \implies \neg(\forall b.(a,b) \in r^{\mathcal{I}_\Delta} \implies b \in D^{\mathcal{I}_\Delta}) \\
\iff & \mathcal{A} \models_\Delta \neg(\forall r.D)(a)
\end{aligned}$$ ■

The Open World Assumption in Description Logics means that we view the information provided in a knowledge base as incomplete. What the algorithm essentially does is to check only two interpretations: One interpretation, which interprets everything as false unless explicitly stated otherwise (by assertions of the form $r(a,b)$ and $A(a)$), and one interpretation, which interprets everything as true unless explicitly stated otherwise (by assertions of the form $\neg r(a,b)$ and $\neg A(a)$). This approach is sound, but incomplete.

Proposition 6.8 (incompleteness)
$\mathcal{A} \models_\Delta C(a)$ *does not imply* $\mathcal{A} \vdash_\Delta C(a)$.

Proof See Example 6.9 below. ■

Example 6.9 (Oedipus example)
The following is the Oedipus example which stipulated a lot of research in Description Logics (Baader et al., 2003, chap. 2). The following flat ABox is given:

```
hasChild(IOKASTE,OEDIPUS).
hasChild(OEDIPUS,POLYNEIKES).
hasChild(IOKASTE,POLYNEIKES).
hasChild(POLYNEIKES,THERSANDROS).
Patricide(OEDIPUS).
¬ Patricide(THERSANDROS).
```

Assume we want to make the following instance test (where the domain Δ is assumed to be the set of all objects in $\mathcal{A}$):

$$\exists\texttt{hasChild}.(\texttt{Patricide} \sqcap \exists\texttt{HasChild}.\neg\texttt{Patricide})(\texttt{IOKASTE})$$

We will compute the set $(\exists\texttt{hasChild}.(\texttt{Patricide} \sqcap \exists\texttt{hasChild}.\neg\texttt{Patricide}))^+_{\mathcal{A},\Delta}$:

$$\begin{aligned}
\texttt{Patricide}^+_{\mathcal{A},\Delta} &= \{\texttt{OEDIPUS}\} \\
\texttt{Patricide}^-_{\mathcal{A},\Delta} &= \{\texttt{THERSANDROS}\} \\
(\neg\texttt{Patricide})^+_{\mathcal{A},\Delta} &= \{\texttt{THERSANDROS}\} \\
(\neg\texttt{Patricide})^-_{\mathcal{A},\Delta} &= \{\texttt{OEDIPUS}\} \\
(\exists\texttt{hasChild}.\neg\texttt{Patricide})^+_{\mathcal{A},\Delta} &= \{\texttt{POLYNEIKES}\} \\
(\exists\texttt{hasChild}.\neg\texttt{Patricide})^-_{\mathcal{A},\Delta} &= \{\emptyset\} \\
(\texttt{Patricide} \sqcap \exists\texttt{hasChild}.\neg\texttt{Patricide})^+_{\mathcal{A},\Delta} &= \{\emptyset\} \\
(\texttt{Patricide} \sqcap \exists\texttt{hasChild}.\neg\texttt{Patricide})^-_{\mathcal{A},\Delta} &= \{\emptyset\} \\
(\exists\texttt{hasChild}.(\texttt{Patricide} \sqcap \exists\texttt{hasChild}.\neg\texttt{Patricide}))^+_{\mathcal{A},\Delta} &= \{\emptyset\} \\
(\exists\texttt{hasChild}.(\texttt{Patricide} \sqcap \exists\texttt{hasChild}.\neg\texttt{Patricide}))^-_{\mathcal{A},\Delta} &= \{\emptyset\}
\end{aligned}$$

This means that the algorithm can not deduce that `IOKASTE` is an instance of the aforementioned concept. However, it is indeed an instance, by the following arguments:

Let the models of $\mathcal{A}$ be divided in two classes. In one class `POLYNEIKES` is a `Patricide` and in the other he is not a `Patricide`. In the first class `IOKASTE` has a child `POLYNEIKES`, which fulfills $\texttt{Patricide}\sqcap\exists\texttt{hasChild}.\neg\texttt{Patricide}$ (*) as one can easily check. In a model of the second class `OEDIPUS` is the child of `IOKASTE`, which fulfills (*). Thus in all models `IOKASTE` is an instance of the concept we wanted to test. □

We have shown that the algorithm is incomplete. However, we will show that it is complete if there is no uncertain knowledge. The reason is that in this case there can be at most one model of the ABox.

Definition 6.10 (closed flat ABox)
We say that a flat ABox is *closed with respect to* Δ iff the following conditions hold:

1. for all atomic concepts A: $a \in \Delta$ implies $(A(a) \in \mathcal{A}$ or $\neg A(a) \in \mathcal{A})$
2. for all roles r: $(a, b) \in \Delta \times \Delta$ implies $(r(a, b) \in \mathcal{A}$ or $\neg r(a, b) \in \mathcal{A})$ □

Proposition 6.11 (completeness under closure)
Let $\mathcal{A}$ be a closed consistent flat ABox with respect to a domain Δ. Then we have $\mathcal{A} \models_\Delta C(a)$ implies $\mathcal{A} \vdash_\Delta C(a)$.

Proof For a consistent closed flat ABox the following holds:

1. for all atomic concepts A: $a \in \Delta$ implies that either $A(a) \in \mathcal{A}$ or $\neg A(a) \in \mathcal{A}$ (but not both, because of consistency)
2. for all roles r: $(a, b) \in \Delta \times \Delta$ implies that either $r(a, b) \in \mathcal{A}$ or $\neg r(a, b) \in \mathcal{A}$ (but not both)

By the definition of a Δ-model (Definition 6.2, page 85), the interpretation domain and the mapping of individuals is fixed. Additionally a consistent closed ABox also fixes the mapping of atomic concepts to subsets of Δ and the mapping of roles to subsets of $\Delta \times \Delta$. This interpretation $\mathcal{I}_\Delta$ is defined by: $A^{\mathcal{I}_\Delta} = \{a \mid C(a) \in \mathcal{A}\}$ and $r^{\mathcal{I}_\Delta} = \{(a, b) \mid r(a, b) \in \mathcal{A}\}$ for atomic concepts A and roles r. It is easy to see that this is the only model of $\mathcal{A}$.

What we will show is that in this case the retrieval algorithm corresponds exactly to the semantics of $\mathcal{ALC}$ concepts (see Table 1, page 9).

As a prerequisite we need two statements for closed consistent ABoxes:

1. $r^+_{\mathcal{A}} = \Delta \times \Delta \setminus r^-_{\mathcal{A}}$
   This is easy to see, because $r(a, b)$ is in $\mathcal{A}$ iff $\neg r(a, b)$ is not in $\mathcal{A}$.
2. $C^-_{\mathcal{A},\Delta} = \Delta \setminus C^+_{\mathcal{A},\Delta}$
   We can show this by induction over the structure of C:
   a) Induction Base:
      i. $C = \top$: $\bot^-_{\mathcal{A},\Delta} = \emptyset = \Delta \setminus \Delta = \Delta \setminus \top^+_{\mathcal{A},\Delta}$
      ii. $C = \bot$: $\bot^-_{\mathcal{A},\Delta} = \Delta = \Delta \setminus \emptyset = \Delta \setminus \bot^+_{\mathcal{A},\Delta}$
      iii. $C = A$: $A^-_{\mathcal{A},\Delta} = \Delta \setminus A^+_{\mathcal{A},\Delta}$, because for any object a we have $A(a)$ in $\mathcal{A}$ iff $\neg A(a)$ is not in $\mathcal{A}$.
   b) Induction Step:
      i. $C = \neg D$: $(\neg D)^-_{\mathcal{A},\Delta} = D^+_{\mathcal{A},\Delta} = \Delta \setminus D^-_{\mathcal{A},\Delta}$ (by induction) $= \Delta \setminus (\neg D)^+_{\mathcal{A},\Delta}$
      ii. $C = D \sqcap E$:

$$\begin{aligned}(D \sqcap E)^-_{\mathcal{A},\Delta} =& D^-_{\mathcal{A},\Delta} \cup E^-_{\mathcal{A},\Delta}\\ =&(\Delta \setminus D^+_{\mathcal{A},\Delta}) \cup (\Delta \setminus E^+_{\mathcal{A},\Delta})\\ =&\Delta \setminus (D^+_{\mathcal{A},\Delta} \cap E^+_{\mathcal{A},\Delta})\\ =&\Delta \setminus (D \sqcap E)^+_{\mathcal{A},\Delta}\end{aligned}$$

      iii. $C = D \sqcup E$:

$$\begin{aligned}(D \sqcup E)^-_{\mathcal{A},\Delta} =& D^-_{\mathcal{A},\Delta} \cap E^-_{\mathcal{A},\Delta}\\ =&(\Delta \setminus D^+_{\mathcal{A},\Delta}) \cap (\Delta \setminus E^+_{\mathcal{A},\Delta})\\ =&\Delta \setminus (D^+_{\mathcal{A},\Delta} \cup E^+_{\mathcal{A},\Delta})\\ =&\Delta \setminus (D \sqcup E)^+_{\mathcal{A},\Delta}\end{aligned}$$

iv. $C = \exists r.D$:

$$\begin{aligned}(\exists r.D)^-_{\mathcal{A},\Delta} &= \{a \mid \forall b.((a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}}) \implies b \in D^-_{\mathcal{A},\Delta}\}\\ &= \{a \mid \forall b.\neg((a,b) \in r^+_{\mathcal{A}}) \vee b \in D^-_{\mathcal{A},\Delta}\}\\ &= \{a \mid \neg(\exists b.(a,b) \in r^+_{\mathcal{A}} \wedge \neg(b \in D^-_{\mathcal{A},\Delta}))\}\\ &= \{a \mid \neg(\exists b.(a,b) \in r^+_{\mathcal{A}} \wedge b \in D^+_{\mathcal{A},\Delta})\}\\ &= \Delta \setminus (\exists r.D)^+_{\mathcal{A},\Delta}\end{aligned}$$

v. $C = \forall r.D$:

$$\begin{aligned}(\forall r.D)^-_{\mathcal{A},\Delta} &= \{a \mid \exists b.((a,b) \in r^+_{\mathcal{A}}) \wedge b \in D^-_{\mathcal{A},\Delta}\}\\ &= \{a \mid \exists b.((a,b) \in r^+_{\mathcal{A}}) \wedge \neg b \in D^+_{\mathcal{A},\Delta}\}\\ &= \{a \mid \neg(\forall b.\neg((a,b) \in r^+_{\mathcal{A}}) \vee b \in D^+_{\mathcal{A},\Delta})\}\\ &= \{a \mid \neg(\forall b.((a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}}) \implies b \in D^+_{\mathcal{A},\Delta})\}\\ &= \Delta \setminus (\forall r.D)^+_{\mathcal{A},\Delta}\end{aligned}$$

Using these statements we have:

$$\begin{aligned}(\neg C)^+_{\mathcal{A},\Delta} &= C^-_{\mathcal{A},\Delta} = \Delta \setminus C^+_{\mathcal{A},\Delta}\\ (\forall r.C)^+_{\mathcal{A},\Delta} &= \{a \mid \forall b.((a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}}) \implies b \in D^+_{\mathcal{A},\Delta}\}\\ &= \{a \mid \forall b.(a,b) \in r^+_{\mathcal{A}} \implies b \in D^+_{\mathcal{A},\Delta})\}\end{aligned}$$

If we use this together with the rules in Definition 6.5 we see that computing $C^+_{\mathcal{A},\Delta}$ corresponds exactly to the semantics of $\mathcal{ALC}$ (see Table 1, page 9).

Summary: We have shown that there is exactly one model $\mathcal{I}_\Delta$ for $\mathcal{A}$. $\mathcal{A} \models_\Delta C(a)$ means that we have $a \in C^{\mathcal{I}_\Delta}$. We have $a \in C^{\mathcal{I}_\Delta} \iff a \in C^+_{\mathcal{A},\Delta}$, because applying the interpretation function is the same like computing $C^+_{\mathcal{A},\Delta}$. $a \in C^+_{\mathcal{A},\Delta}$ means that we have $\mathcal{A} \vdash_\Delta C(a)$ by Definition of the retrieval algorithm. ■

Below we will show that the algorithm runs in polynomial time.

Proposition 6.12 (complexity of the algorithm)
Let C be a concept with $|C| = n$ and $|\Delta| = m$.
The space complexity of Algorithm 6.6 is $O(m \cdot n)$.
The time complexity of Algorithm 6.6 is at most $O(n \cdot m^2 \cdot \log m)$.

PROOF We assume that we have given $A^+_{\mathcal{A},\Delta}$ and $A^-_{\mathcal{A},\Delta}$ for all atomic conceps. Additionally we need $r^+_{\mathcal{A}}$ and $\Delta \times \Delta \setminus r^-_{\mathcal{A}}$ for all roles.

We will first have a look at the space complexity. For each node in the tree (remember that the query concept is stored as tree in the algorithm) we have to store two lists of individuals. There are no more than n nodes. The list of individuals has length at most m. Hence the space complexity is $O(m \cdot n)$.

For time complexity we look at the number of operations we have to perform for each of at most n nodes. The most expensive operations have to be done for quantifications. As an example we consider a node of the form $\exists r.D$, i.e. we have to calculate:

$$(\exists r.D)^+_{\mathcal{A},\Delta} = \{a \mid \exists b.(a,b) \in r^+_{\mathcal{A}} \wedge b \in D^+_{\mathcal{A},\Delta}\}$$
$$(\exists r.D)^-_{\mathcal{A},\Delta} = \{a \mid \forall b.((a,b) \in \Delta \times \Delta \setminus r^-_{\mathcal{A}}) \implies b \in D^-_{\mathcal{A},\Delta}\}$$

To traverse $r^+_{\mathcal{A}}$ (or $\Delta \times \Delta \setminus r^-_{\mathcal{A}}$) we have to look at $m \cdot m$ elements. For each such element we may have to do a lookup whether an individual belongs to $D^+_{\mathcal{A},\Delta}$ (or $D^-_{\mathcal{A},\Delta}$). Assuming that the individuals are ordered this takes $O(\log m)$ time steps (binary search). Overall we get a time complexity of $O(n \cdot m^2 \cdot \log m)$. ■

Why is the Algorithm Useful for Solving the Learning Problem?

There are reasons why the algorithm we have presented is useful despite its simplicity. First of all it is very efficient. It has a low polynomial time and space complexity. This low complexity comes at a price. We have shown that the algorithm is incomplete. For a more specific case (closed flat ABox) completeness holds. We have shown that the algorithm is sound under the fixed domain assumption we made. We have also justified that the fixed domain assumption can be useful when learning in Description Logics (see Example 6.1 on page 84).

In the learning problem we need to test a large number of concepts against a fixed knowledge base. For this reason it makes sense to do a large pre-processing step, i.e. the transformation of a knowledge base to a flat ABox, if the time per query can be significantly reduced. Especially in algorithms, which do not explicitly avoid redundancy, e.g. the Genetic Programming approach, it may be a useful idea to use the algorithm for an approximate fitness measure. This way the use of sound and complete reasoning algorithms for $\mathcal{ALC}$, which are PSPACE-complete (Baader et al., 2003), can be reduced. (The complexity of sound and complete reasoning can be even higher than PSPACE if the background knowledge description language is more expressive than $\mathcal{ALC}$.)

An advantage of the algorithm is that it performs a complete retrieval at once instead of performing single instance tests (like most tableau algorithms). This is useful, because we have to do an instance test for each example anyway. Moreover when doing a retrieval for a concept C it automatically also does a retrieval for $\neg C$, which can be very useful if we define the learning problem in such a way that negative examples of the form $\neg$`Target`(a) have to be covered (see Section 3 for details).

A drawback of the algorithm is, of course, its incompleteness. While sound and complete reasoning in Description Logics is usually very hard, it is less difficult to design incomplete algorithms. However, there is a way to combine the sound and incomplete retrieval algorithm with sound and complete instance tests: We first try to classify all examples in the learning problem using the fast incomplete retrieval algorithm. By the soundness of the algorithm, we know that if it can classify an example then this classification is correct. For the examples, which could not be classified, we can use sound and complete instance tests.

The goal of this section was to look at ways to measure the quality of a concept efficiently and see which problems can arise in concept learning in general. In particular we showed that the Open World Assumption can be a problem in learning in Description Logics and gave a concrete algorithm as a way to overcome this problem. While the major focus of this thesis is on creating intelligent learning algorithms we also considered it worthwhile to devote a section to look deeper at the reasoning algorithms, which form the basis of these learning algorithms.

7 Summary, Outlook, Further Work

In this section we will briefly summarise the advances we made in learning Description Logics and look at future work and perspectives.

Summary In the thesis we analysed the learning problem in Description Logics. First we introduced a general framework for different learning methods. Then we analysed refinement operators as the main method to travers the space of concepts ordered by subsumption. We made a full analysis of the properties of refinement operators and concluded that learning $\mathcal{ALC}$ concepts can be considered a hard learning task. Nevertheless we were able to construct a complete and proper refinement operator, which may be useful in practice. Then we showed how a redundancy-eliminating heuristic can be implemented efficiently by introducing an ordered normal form of concepts. Finally we analysed the properties of the resulting algorithm, compared it to other approaches and outlined its advantages. A new approach was introduced in Section 5. We analysed the use of Genetic Programming for learning in Description Logics. The standard approach does not make use of the subsumption order of concepts and the used genetic operators are too destructive. To overcome these problems we proposed to combine Genetic Programming with the usage of refinement operators. We showed how this can be done in general and then defined concrete refinement operators, which can be used in the Genetic Programming framework. Some other features like noise handling, learning from uncertain data, and concept invention were briefly described. In Section 6 we looked at the problems in quality measurement of concepts and developed a simple approximate retrieval algorithm as a first step to overcome these problems.

Further Work From a practical point of view, a lot of work still needs to be done. The approaches developed in this thesis need to be implemented in order to evaluate them. We did not perform an implementation and evaluation of the proposed approaches yet, since this would have gone beyond the intended scope of this thesis. However, we plan to do this in the future.

Although learning in Description Logics is currently an interesting research topic, there are not enough standard benchmark data sets available yet. These need to be created to make it possible to compare the performance in practice between different learning approaches. The Semantic Web is a driving force behind the research in concept learning. So naturally an idea is to embed a concept learning module in an ontology editor. In this way we would be able to see whether knowledge engineers make use of such a module and to what extend it helps to build up ontologies. The standard approach to implement such a module is likely to be close to what we presented in Section 4, i.e. a refinement operator combined with a heuristic. We consider the usage of Genetic Programming as a viable alternative for other applications of concept learning, especially in classical Inductive Logic Programming domains.

The emphasis in this thesis was intentionally on theoretical aspects to build up a solid foundation for later practical work. However, there is is still a lot of room for future theoretical advances. The refinement operator analysis made for $\mathcal{ALC}$ concepts should be extended to other description languages. This would allow us to gain a better overview of the hardness of the learning problem for different languages (or more exactly the concept constructors in these languages). We could also use it to compare them with other target languages, e.g. logic programs. The major piece of theoretical work will be to extend the approaches in this thesis to more powerful description languages than $\mathcal{ALC}$. Ontology languages like OWL DL offer much more constructs than $\mathcal{ALC}$. A desirable goal is to be able to handle all class constructors in OWL DL, which is a prerequisite in order to use the learning methods in OWL ontology editors. Another extension is to develop suitable approaches for learning only from positive data, which has turned out to be crucial in Inductive Logic Programming (Muggleton, 1996).

Outlook There has been research in the area of learning in Description Logics for more than ten years. However, we can still consider it to be in an early stage compared to the induction of logic programs. As of now it is open whether it will gain enough momentum to develop into a field of research on its own. The success of logic programming languages like Prolog was crucial for the success of Inductive Logic Programming. In the same manner the potential future success of Description Logics and ontologies in the Semantic Web may be an important factor for the further development and implementation of learning methods in Description Logics. We consider it to be an interesting research field with high potential.

Due to the close relation to Inductive Logic Programming (in the broadest sense it can be considered to be part of it) there will be a lot of knowledge transfer from ILP to learning in Description Logics. However, both target languages face different problems, so it is by no means trivial to use advances in ILP for learning in DLs.

From our point of view, this thesis constitutes a useful scientific contribution to the field and many ideas may serve as a basis of future research or influence it.

List of Figures

List of Tables

List of Algorithms

List of Definitions

List of Theorems, Propositions, Corollaries, and Lemmata

List of Examples and Remarks

References

Anglano, C., Giordana, A., Bello, G. L., and Saitta, L. (1998). An experimental evaluation of coevolutive concept learning. In *Proc. 15th International Conf. on Machine Learning*, pages 19–27. Morgan Kaufmann, San Francisco, CA.

Augier, S., Venturini, G., and Kodratoff, Y. (1995). Learning first order logic rules with a genetic algorithm. In Fayyad, U. M. and Uthurusamy, R., editors, *The First International Conference on Knowledge Discovery and Data Mining*, pages 21–26, Montreal, Canada. AAAI Press.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Badea, L. and Nienhuys-Cheng, S.-H. (2000). A refinement operator for description logics. In Cussens, J. and Frisch, A., editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 40–59. Springer-Verlag.

Balaban, M. (1995). The F-logic approach for description languages. *Ann. Math. Artif. Intell*, 15(1):19–60.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1990). Occam's razor. In Shavlik, J. W. and Dietterich, T. G., editors, *Readings in Machine Learning*, pages 201–204. Morgan Kaufmann.

Brachman, R. J. (1978). A structural paradigm for representing knowledge. Technical Report BBN Report 3605, Bolt, Beraneck and Newman, Inc., Cambridge, MA.

Brandt, S., Küsters, R., and Turhan, A.-Y. (2002). Approximation and difference in description logics. In Fensel, D., Giunchiglia, F., McGuinness, D. L., and Williams, M.-A., editors, *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02), Toulouse, France, April 22-25, 2002*, pages 203–214. Morgan Kaufmann.

Bratko, I. and Muggleton, S. (1995). Applications of Inductive Logic Programming. *Communications of the ACM*, 38(11):65–70.

Cohen, W. W., Borgida, A., and Hirsh, H. (1993). Computing least common subsumers in description logics. In Rosenbloom, P. and Szolovits, P., editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 754–760, Menlo Park, California. American Association for Artificial Intelligence, AAAI Press.

Cohen, W. W. and Hirsh, H. (1994). Learning the CLASSIC description logic: Theoretical and experimental results. In Doyle, J., Sandewall, E., and Torasso, P., editors, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR94)*, pages 121–133. Morgan Kaufmann.

Divina, F. (2006). Evolutionary concept learning in first order logic: An overview. *AI Commun.*, 19(1):13–33.

Divina, F. and Marchiori, E. (2002). Evolutionary concept learning. In Langdon, W. B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., and Jonoska, N., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 343–350, New York. Morgan Kaufmann Publishers.

Esposito, F., Fanizzi, N., Iannone, L., Palmisano, I., and Semeraro, G. (2004). nowledge-intensive induction of terminologies from metadata. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, pages 441–455. Springer.

Gennari, H., J., Musen, A., M., Fergerson, W., R., Grosso, E., W., Crubezy, M., Eriksson, H., Noy, F., N., Tu, and W., S. (2003). The evolution of protege: an environment for knowledge-based systems development. *International Journal of Human-Computer Studies*, 58(1):89–123.

Giordana, A. and Neri, F. (1996). Search-intensive concept induction. *Evolutionary Computation Journal*, 3(4):375–416.

Haarslev, V. and Möller, R. (2003). Racer: A core inference engine for the semantic web. In Sure, Y. and Corcho, Ó., editors, *EON2003, Evaluation of Ontology-based Tools, Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools held at the 2nd International Semantic Web Conference ISWC 2003, 20th October 2003 (Workshop day), Sundial Resort, Sanibel Island, Florida, USA*, volume 87 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Hekanaho, J. (1996). Background knowledge in GA-based concept learning. In *Proc. 13th International Conference on Machine Learning*, pages 234–242. Morgan Kaufmann.

Hekanaho, J. (1998). DOGMA: A GA-based relational learner. In Page, D., editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 205–214. Springer-Verlag.

Horrocks, I., Kutz, O., and Sattler, U. (2006). The even more irresistible SROIQ. In Doherty, P., Mylopoulos, J., and Welty, C. A., editors, *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 57–67. AAAI Press.

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

Iannone, L. and Palmisano, I. (2005). An algorithm based on counterfactuals for concept learning in the semantic web. In Ali, M. and Esposito, F., editors, *Innovations in Applied Artificial Intelligence*, pages 370–379, Bari, Italy. Proceedings of the 18th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems.

Koza, J., Keane, M., Streeter, M., Mydlowec, W., Yu, J., and Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers.

Koza, J. R. (1993). Hierarchical automatic function definition in genetic programming. In Whitley, L. D., editor, *Foundations of Genetic Algorithms 2*, pages 297–318, San Mateo. Morgan Kaufmann.

Koza, J. R. and Poli, R. (2003). A genetic programming tutorial. In Burke, E., editor, *Introductory Tutorials in Optimization, Search and Decision Support.*

Mitchell, T. (1997). *Machine Learning.* McGraw Hill, New York.

Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230.

Motik, B. (2006). *Reasoning in Description Logics using Resolution and Deductive Databases.* PhD thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany.

Motik, B. and Rosati, R. (2004). Closing Semantic Web Ontologies. Technical report, University of Manchester, UK.

Muggleton, S. (1995a). Inductive logic programming: Inverse resolution and beyond. In Mellish, C. S., editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 997–997, San Mateo. Morgan Kaufmann.

Muggleton, S. (1995b). Inverse entailment and progol. *New Generation Computing*, 13(3&4):245–286.

Muggleton, S. (1996). Learning from positive data. In Muggleton, S., editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, volume 1314 of *Lecture Notes in Artificial Intelligence*, pages 358–376. Springer-Verlag.

Neri, F. and Saitta, L. (1995). Analysis of genetic algorithms evolution under pure selection. In Eshelman, L., editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 32–39, San Francisco, CA. Morgan Kaufmann.

Nienhuys-Cheng, S.-H. and de Wolf, R., editors (1997). *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science.* Springer.

Reiser, P. G. K. and Riddle, P. J. (1999). Evolution of logic programs: Part-of-speech tagging. In *1999 Congress on Evolutionary Computation*, pages 1338–1345, Piscataway, NJ. IEEE Service Center.

Russel, S. and Norvig, P. (2003). *Artificial Intelligence - A Modern Approach*. Prentice Hall, 2nd edition.

Sirin, E. and Parsia, B. (2004). Pellet: An OWL DL reasoner. In Haarslev, V. and Möller, R., editors, *Proceedings of the 2004 International Workshop on Description Logics (DL2004), Whistler, British Columbia, Canada, June 6-8, 2004*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Tamaddoni-Nezhad, A. and Muggleton, S. (2000). Searching the subsumption lattice by a genetic algorithm. In Cussens, J. and Frisch, A., editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 243–252. Springer-Verlag.

Tamaddoni-Nezhad, A. and Muggleton, S. (2003). A genetic algorithms approach to ILP. In Matwin, S. and Sammut, C., editors, *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *Lecture Notes in Artificial Intelligence*, pages 285–300. Springer-Verlag.

Tang, L. R., Califf, M. E., and Mooney, R. J. (1998). An experimental comparison of genetic programming and inductive logic programming on learning recursive list functions. Technical Report AI 98-271, Artificial Intelligence Lab, University of Texas at Austin.

Wong, M. L. and Leung, K. S. (1995). Inducing logic programs with genetic algorithms: The genetic logic programming system. *IEEE Expert*, 10 5:68–76.

Statement of Academic Honesty

Hereby, I declare that this is my work and that I did not use any other sources than the ones cited.