

The Design and Implementation of a
Game-Theoretic Decision Procedure for the
Constructive Description Logic *cALC*

Diplomarbeit

im Studiengang Wirtschaftsinformatik
der Fakultät Wirtschaftsinformatik
und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Verfasser: Martin Sticht

Gutachter: Prof. Michael Mendler, PhD

CONTENTS

List of Figures	iv
List of Tables	v
0. Introduction	1
1. Introduction to Description Logic	3
1.1. Description Logic	3
1.1.1. Concepts and Roles	4
1.1.2. Families of Description Languages and their Semantics	7
1.1.3. Formalizing and Representing Knowledge	10
1.1.4. Inference	14
1.1.5. Tableau-Based Reasoning	16
1.1.6. Description Logic and Modal Logic	23
1.1.7. Summary	24
1.2. The Constructive Description Logic $c\mathcal{ALC}$	25
1.2.1. Why Constructive?	25
1.2.2. Syntax and Semantics	30
1.2.3. Intuitionistic and Constructive Description Logic	33
1.2.4. Inference	36
1.2.5. Tableau-Based Proofs for $c\mathcal{ALC}$	38
1.2.6. Summary	45
2. A Game-Theoretic Decision Procedure	47
2.1. Introduction to Dialogical Games for First-Order Logics	47
2.1.1. Why and How?	48
2.1.2. Rules	49
2.1.3. Some Examples	55

2.1.4. Summary	59
2.2. Dialogues for the Description Logic \mathcal{ALC}	59
2.2.1. Aims and Problems	60
2.2.2. Notation	60
2.2.3. Rules	62
2.2.4. Strategies	64
2.2.5. Some Examples	65
2.3. Dialogues for $c\mathcal{ALC}$	67
2.3.1. The Language	67
2.3.2. Rules	68
2.3.3. Strategies	74
2.3.4. Inference	74
2.3.5. More Examples	76
2.4. Conclusion and Miscellaneous	81
3. Design and Implementation of a Dialogue-Based Prover for $c\mathcal{ALC}$	83
3.1. Tools for the Implementation	84
3.1.1. A Functional Programming Language	84
3.1.2. Development Environment	84
3.1.3. Creating Graphs	85
3.2. Architecture	85
3.2.1. Expressions	86
3.2.2. Assertions	88
3.2.3. Entities	92
3.2.4. Games	92
3.2.5. Dialogues	93
3.2.6. Particle Rules	95
3.2.7. Annotations	97
3.2.8. Moves	100
3.2.9. Steps	101
3.3. Particle Rules and their Parameters	102
3.3.1. Parameters	102
3.3.2. Implementing a Particle Rule	104
3.4. Structural Rules	106
3.4.1. Rules Affecting the Player	107

3.4.2. Rules Affecting Moves	108
3.4.3. Updating Annotations	115
3.4.4. Backtrack	120
3.4.5. Strategies	121
3.5. A Simple User-Interface	122
4. Evaluation and Further Steps	125
4.1. Testing	125
4.1.1. Some Basic Intuitionist Features	126
4.1.2. IK1 to IK5	128
4.1.3. Other Formulæ	129
4.1.4. Conclusion	130
4.2. Dialogues in Scope	130
4.2.1. Advantages Compared to the Tableau-Based Algorithm	131
4.2.2. Disadvantages Compared to the Tableau-Based Algorithm	131
4.2.3. Solving Problems	132
4.2.4. How to Show Soundness and Completeness	133
4.3. The Implementation in Scope	133
4.3.1. Experiences with Haskell	133
4.3.2. Possible Improvements	134
4.4. Final Conclusion	135
A. Rules	139
A.1. Particle Rules for $c\mathcal{ALC}$	139
A.2. Structural Rules for $c\mathcal{ALC}$	140
B. Notation and Representation of Concepts	143
B.1. Infix Notation for Concept Descriptions	143
B.2. Concept-Representation in ASCII-Terminals	144
C. Content of the CD	145
C.1. Content	145
C.2. Module Descriptions	146
Bibliography	147

LIST OF FIGURES

1.1. A Simplified Semantic Network	5
1.2. Architecture of a Knowledge Based System	11
1.3. Illustration of a Tableau	23
1.4. The Queen's Knowledge about her Apples	28
1.5. Snow White's Knowledge about Apples	28
1.6. Completion Rules of the Constraint Calculus for $c\mathcal{ALC}$	40
1.7. The First Step of Proving IK2 in $c\mathcal{ALC}$	44
1.8. A $c\mathcal{ALC}$ -Tableau after Applying two Rules	44
1.9. A $c\mathcal{ALC}$ -Tableau with two Inactive Entities	44
1.10. The same Entity with different Constraints	44
2.1. Illustration of a Dialogue	58
2.2. Relationships of Entities in IK4 (Alternative 1)	76
2.3. Relationships of Entities in IK4 (Alternative 2)	77
2.4. Dialogue Tree for IK4	78
2.5. Dialogue Tree for IK5	78
2.6. Relationships of Entities for Wine and Meat	80
3.1. Overview of Data Types	85
3.2. Showing Expressions in a Terminal Window	87
3.3. Showing Assertions in a Terminal Window	91
3.4. A Redundant Dialogue Structure	94
3.5. The three Types of Annotations	98
3.6. Adding two Possible Moves to a Dialogue at the same Time	121
3.7. Selecting the Level of Interaction	122
3.8. The Option Menu	124
3.9. The User is Asked to Select a Move For Player P	124

LIST OF TABLES

1.1. A Simple ABox	13
1.2. Transformation Rules of the Satisfiability Algorithm for \mathcal{ALC}	17
1.3. Semantic Interpretation Structure of $c\mathcal{ALC}$	32
1.4. Axioms for the Intuitionistic Modal Logics IK and FS	34
1.5. Axioms for the Constructive Modal Logic CK	35
1.6. Axioms which are valid in Classical but not in Intuitionistic Logics . .	35
1.7. Constraint System for $c\mathcal{ALC}$	36
2.1. Particle Rules for FOL-Semantics	50
2.2. The Excluded Third in a Classical Dialogue	56
2.3. The Excluded Third in an Intuitionist Dialogue	57
2.4. The Opponent Tracks Back	57
2.5. Closed but Unfinished Dialogue	58
2.6. Particle Rules for \mathcal{ALC} -Semantics	62
2.7. An \mathcal{ALC} -Dialogue	65
2.8. Dualities in a Dialogue	66
2.9. Particle Rules for $c\mathcal{ALC}$ -Semantics	69
3.1. Particle Rule Parameters	103
3.2. Domains Affected by Structural Rules	107
4.1. Sometimes it is Better not to Choose the most Refining Entity	126
4.2. Peirce's Law Causes Cycles	128
B.1. Infix Notation for Concept Descriptions	143
B.2. Alternative Concept-Representation in ASCII-Terminals	144
C.1. Content of the CD	145
C.2. Module Descriptions	146

0

INTRODUCTION

In the last years, several languages of Description Logic have been introduced to model knowledge and perform inference on it. There have been several propositions for different application scenarios. The constructive Description Logic $c\mathcal{ALC}$ deals with uncertain or dynamic knowledge. We make use of a game-theoretic dialogue-based proving technique that comes from the fields of philosophy and introduce rules so that we can perform reasoning in $c\mathcal{ALC}$. It can be considered as an alternative technique to tableau-based proofs, emphasizing the semantics. As we will see, showing validity efforts a high complexity, but for this, we have a philosophical approach that might make it possible to find out more about the logics and that provides possibilities to extend or alter the underlying semantics.

We will see a Haskell-implementation of a simple dialogue-based reasoner for $c\mathcal{ALC}$ that is able to show the validity of descriptive formulæ or to refute them. For this implementation, the fuzzy structural rules will be formalized. Unfortunately, because of high branching factors, it is not able to cope with some formulæ.

In Chapter 1, we first look at general Description Logic, its terminology and at a tableau algorithm for the language \mathcal{ALC} . We then consider the constructive logic $c\mathcal{ALC}$, its purpose and its relationship to intuitionistic logics. We will finally see a tableau-based algorithm for $c\mathcal{ALC}$.

In Chapter 2, we introduce a dialogue-based proving technique for the constructive Description Logic. For this, we begin with dialogues for first-order logic and then extend the underlying rules step by step until we have a system for $c\mathcal{ALC}$.

The implementation of a dialogue-based reasoner is explained in Chapter 3. We first introduce the main data structures and then have a closer look at the realisation of the rules.

Eventually, we test the implementation for some selected formulæ in Chapter 4. Then we compare the dialogical approach to the tableau-based. Finally, we provide some suggestions on how to improve both the rules for the dialogues and the implementation.

1

INTRODUCTION TO DESCRIPTION LOGIC

In this first chapter, we have a general introduction to Description Logic followed by an explanation of a constructive language called *cALC*, as this is the logic we are going to cope with in the Chapters 2 and 3. But in order to be able to understand *cALC*, it is reasonable to learn what Description Logic is and how to perform *reasoning* with it.

1.1. Description Logic

Description Logic provides a way to describe knowledge about a specific domain of interest. It can be used to formalize definitions and assertions by applying given operators which can also be read and understood by humans quite easily. These facts, stored in a *knowledge base*, can be used to obtain new knowledge automatically. Description Logics are traced back to *semantic networks* but in addition to these, they have defined *logical semantics* which can be expressed in first-order logic (see [BCM⁺05], p. 6; [BS01], p.5).

In this section we have an introduction to Description Logic. First, the terminologies and the syntax of Description Logic are explained. Afterwards, an overview of descrip-

tion languages is given, followed by a short introduction to inferences and proofs. In the end, we compare Description Logic to modal logic.

1.1.1. Concepts and Roles

Using Description Logic (DL) is a way to describe *concepts* and their *relationships* to each other. In this way, knowledge can be represented.

Concepts can be defined as “sets or classes of individual objects” ([BCM⁺05], p. 5). These concepts describe for example any kind of object such as persons, vegetables, etc., also abstract objects like geometric forms.

Now, we can also describe several types of relationships between concepts. Here is a simple example:

Example 1.1

Let us consider seven concepts: **person**, **captain**, **inferior**, **navigator**, **janitor**, **vacuum cleaner** and **space ship**. A *captain commands inferiors*. All inferiors and all captains **are persons**. *Janitors* and *navigators* **are** both *inferiors*. Further, janitors **navigate vacuum cleaners**, while navigators can navigate both *ships* and *vacuum cleaners* (they will probably not do both at the same time). Let us suppose that not every navigator *navigates* vacuum cleaners (because he/she has someone else at home who does that for him/her). □

This scene can be illustrated by a semantic network (see Figure 1.1). It is just a simplified illustration, for example, the cardinalities are missing: the captain usually commands more than one inferior, but at least one, whereas it is not obvious that not all navigators are able to navigate vacuum cleaners.

In this example, the *is_a*-relations are displayed by dashed arrows, while all other relations are represented by solid, labelled ones. *Is_a*-relations indicate an increase of generality, e.g. a captain *is a* person, where *person* is a more general concept than *captain*. In Description Logic, an *is_a*-relation can be expressed by the subsumption

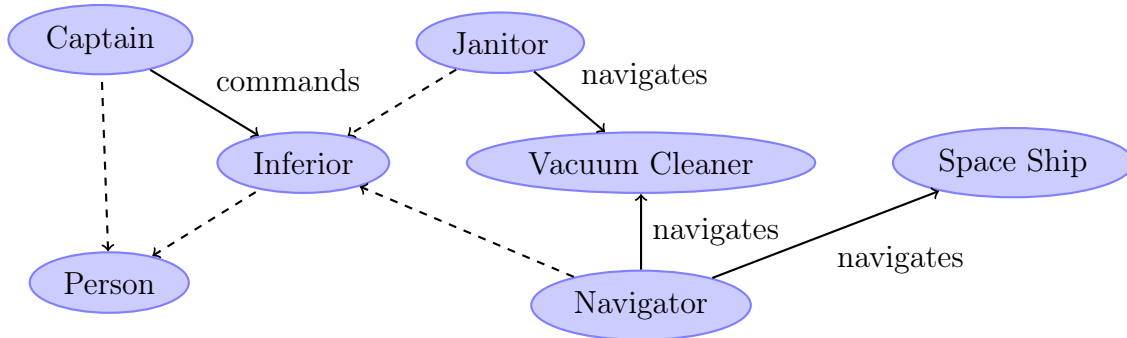


Figure 1.1.: A Simplified Semantic Network

symbol \sqsubseteq . It can also be read as an implication. So, for showing the relation between a captain and a person the expression

$$Captain \sqsubseteq Person$$

might be used which indicates that every captain is also a person.

The relations illustrated by solid arrows indicate properties of concepts, called *roles*. A role can also be seen as a set of pairs of individuals (see [BCM⁺05], p. 5, 7, 46). For instance, in the example shown above, the captain has the property to command inferiors.

Description Logic has the ability to describe these relationships between concepts. Until now, we have only considered *atomic* concepts and roles. If we want to talk about more complex structures, we have to use composition operators. For example, a *crew member* can be defined to be either the captain *or* an inferior (or maybe both). To express this statement, the following logical definition might be used:

$$Crew_Member \equiv Captain \sqcup Inferior$$

The symbol \sqcup can be read as *or* or as *union*, when considering the concepts as sets: as described at the very beginning, concepts can be seen as sets of individual objects. To obtain the set of all crew members, the set of captains (which consists of only one element in this case) has to be joined with the set of all inferiors. We get a set of individuals which are either captains or inferiors (or theoretically even both).

The sentence from above can be converted to the first-order logic expression

$$\forall x \text{ Crew_Member}(x) \longleftrightarrow \text{Captain}(x) \vee \text{Inferior}(x)$$

if x acquires the values of all possible individuals (see [BCM⁺05], p. 7).

There are also other operators which can be applied to concepts. In order to obtain the *intersection* of two concepts, the operator \sqcap is used, which is similar to the logical connective \wedge (for *and*) known from first-order logic. Further, negations (i.e. set complements) can be expressed using the usual symbol \neg .

So, for example, to derive the concept of inferiors from the concept of crew members, one can write

$$\text{Crew_Member} \sqcap \neg \text{Captain}$$

The operators described so far are only used on concepts. In order to express relations, other constructs are needed. Let us begin with the *existential quantification* which has the form $\exists R.C$ where R is a role and C represents an arbitrary concept (see [BCM⁺05], p. 8, 47). For example, the sentence $\exists \text{ navigates.Space_Ship}$ characterizes the concept *navigator*, or to be more specific, it specifies all individuals, that have **at least one** *navigates*-relation to the concept *space ship*.

The so-called *value restriction* $\forall R.C$ is another possibility to describe roles. The sentence $\forall \text{ navigates.Space_Ship}$ expresses something different than the existential version. Here, we get the individuals who navigate a space ship, but no vacuum cleaners, because the universal quantifier indicates only those individuals from which **all** *navigates*-relations lead to *space ship* (see [DLNN97], p. 3). So, for the obtained individuals it is necessary that there is no *navigates*-relation leading to another concept than *space ship*.

The target concept of a possible relation and its individuals are called *role-fillers* (see [BCM⁺05], p. 8). In the last two examples above, the individuals of the concept *space ship* are the role-fillers, but we also say that the *concept* “Space Ship” is the role-filler.

There are languages which offer more constructs to make restrictions on roles. The *number restrictions* indicate those individuals that have a minimal or a maximal number of role-fillers (see [BCM⁺05], p. 8). Let us suppose that a *mighty captain* is a captain

who commands at least 500 individuals. This can be defined in this way:

$$\text{Mighty_Captain} \equiv \text{Captain} \sqcap \geq 500 \text{ commands}$$

Instead of \geq one can also use \leq in order to indicate individuals that do not have more than a given number of role-fillers.

1.1.2. Families of Description Languages and their Semantics

The Language \mathcal{AL}

There are different *description languages* which are more expressive than others. For example in the basic language called \mathcal{AL} (for *attributive language*), the expression $C \sqcup D$ is not defined, while $C \sqcap D$ is possible. Further, negations are only allowed on *atomic* concepts, that is why the statement $\neg(C \sqcap D)$ is also impossible (see [BCM⁺05], p. 47).

From now on, the letter A is used for atomic concepts, while R stands for atomic roles. C and D represent concept descriptions, i.e. they might be atomic concepts or more complex descriptions which are built by various *concept constructors* (such as \sqcup , \sqcap , \forall , \exists , ...) and therefore are more complex than atomic concepts.

According to [BCM⁺05], the basic language \mathcal{AL} accepts expressions formed by the following rules:

C, D	\longrightarrow	A		(atomic concept)
		\top		(universal concept)
		\perp		(bottom concept)
		$\neg A$		(atomic negation)
		$C \sqcap D$		(intersection)
		$\forall R.C$		(value restriction)
		$\exists R.\top$		(limited existential quantification)

It is also remarkable, that the existential quantification can only be applied on the universal concept \top as role-filler. That means that the expression $\exists \text{ navigates.Space_Ship}$ is not possible in \mathcal{AL} . By contrast, the statement $\exists \text{ navigates}.\top$, indicating all individ-

uals that navigate *something* (i.e. navigate an arbitrary individual) is allowed. Number restrictions are not supported anyway.

Semantics and Interpretation

Semantics are defined by a set $\Delta^{\mathcal{I}}$ which includes all *interpreted* individuals and by an interpretation function $\cdot^{\mathcal{I}}$. Together they make the *Interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$.

“In order to define a formal semantics of \mathcal{AL} -concepts, we consider interpretations \mathcal{I} that consist of a non-empty set $\Delta^{\mathcal{I}}$ (the domain of the interpretation) and an interpretation function, which assigns to every atomic concept A a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and to every atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$.”

[BCM⁺05], p. 48

The following definitions declares how concept descriptions are interpreted semantically:

$$\begin{aligned}
 \top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
 \perp^{\mathcal{I}} &= \emptyset \\
 (\neg A)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus A^{\mathcal{I}} \\
 (C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
 (\forall R.C)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \forall b.(a, b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}} \} \\
 (\exists R.\top)^{\mathcal{I}} &= \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a, b) \in R^{\mathcal{I}} \}
 \end{aligned}$$

Here, we see what is behind the operators, e.g. that the conjunction (\sqcap) is interpreted as an *intersection* of two sets. The negation can be understood as a subtraction of a set (representing an atomic concept) from our interpretation domain and \top represents a set of all individuals, while \perp is interpreted as the empty set.

Eventually, equivalence of two concepts ($C \equiv D$) can be defined as consequence of $C^{\mathcal{I}} = D^{\mathcal{I}}$ for all interpretations \mathcal{I} (see [BCM⁺05], p. 48).

Language Extensions

By adding constructs to our basic language \mathcal{AL} , we can make it more and more expressive and powerful. To achieve that *unions* of concepts are also possible, the constructor \mathcal{U} has to be added. Such an extension is described by a syntax $(C \sqcup D)$ and a semantics $(C^{\mathcal{I}} \cup D^{\mathcal{I}})$ (see [BCM⁺05], p. 48, 488).

The new language \mathcal{ALU} is now able to deal with disjunctions. For this reason, our expression $Captain \sqcup Inferior$ is no problem anymore.

Until now, existential quantification is only allowed on the universal concept \top as role-filler (see above). In order to make other concepts accessible as role-fillers, a further extension named \mathcal{E} is needed. It is able to deal with *full existential quantification*. Its interpretation is defined thus:

$$(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \exists b.(a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}$$

Still we face the problem with expressions like $\neg(Captain \sqcup Inferior)$, because in \mathcal{AL} , the negation can only be applied to atomic concepts. The third important extension \mathcal{C} (for *complement*) solves this problem by adding the semantics interpretation

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$$

There are more constructors, which can be added to \mathcal{AL} , such that \mathcal{N} , which are able to cope with *number restrictions* (\leq, \geq, \dots), but we will not cover these here. Instead, we will restrict our view (at least for now) to the constructors explained above.

All constructors can be combined to extend the basic language \mathcal{AL} . We merely add the constructors which we need. In this way, we can compose the languages

$$\mathcal{ALU}, \mathcal{ALE}, \mathcal{ALC}, \mathcal{ALUE}, \mathcal{ALUC}, \mathcal{ALEC} \text{ and } \mathcal{ALUEC}$$

from \mathcal{AL} and the constructors \mathcal{U} , \mathcal{E} and \mathcal{C} .

Anyway, it is remarkable that from the semantic point of view, if we use just \mathcal{C} with \mathcal{AL} , then \mathcal{U} and \mathcal{E} are not really needed, because the disjunction (or union) can be expressed

as a negated conjunction (or intersection), as well as a the full existence quantification can be rewritten as a negated value restriction in this way (see [BCM⁺05], p. 49):

$$C \sqcup D \equiv \neg(\neg C \sqcap \neg D)$$

$$\exists R.C \equiv \neg \forall R. \neg C$$

The other way round, the negation from \mathcal{C} is also available implicitly when using $\mathcal{ALU}\mathcal{E}$, e.g

$$\begin{aligned} & \neg(\text{Captain} \sqcap \exists \text{ navigates} . (\text{Vacuum_Cleaner} \sqcup \text{Space_Ship})) \\ \equiv & \neg \text{Captain} \sqcup \forall \text{ navigates} . (\neg \text{Vacuum_Cleaner} \sqcap \neg \text{Space_Ship}) \end{aligned}$$

(note that *captain*, *vacuum cleaner* and *space ship* are all atomic concepts and that in \mathcal{AL} negation is possible on atomic concepts). That is why, it is possible to abbreviate $\mathcal{ALU}\mathcal{E}\mathcal{C}$ and even just $\mathcal{ALU}\mathcal{E}$ to \mathcal{ALC} .

So, from now on, the language $\mathcal{ALU}\mathcal{E}\mathcal{C}$ will just be referred to as \mathcal{ALC} .

1.1.3. Formalizing and Representing Knowledge

Description Logic is usually applied to represent knowledge. It is possible to formalize a query in form of a logical sentence and check if a concept exists that satisfies this query. Such a sentence might be a *subsumption* $C \sqsubseteq D$ expressing that the concept D is *more general* than the concept C . When regarding concepts as sets, you could read this as *concept C is always a subset of concept D* (see [BCM⁺05], p. 9). For example the expression $\text{Inferior} \sqsubseteq \text{Crew_Member}$ can be used as a query, where an underlying DL *knowledge base* checks if that statement is true. In this knowledge base, all relevant domain-specific knowledge is going to be stored. A typical DL knowledge base can be separated into two different components, the *TBox* and the *ABox* (see [BCM⁺05], p. 12, 46).

The architecture of a *knowledge representation* system, which is based on Description Logic as it is illustrated by [BCM⁺05] is shown in Figure 1.2.

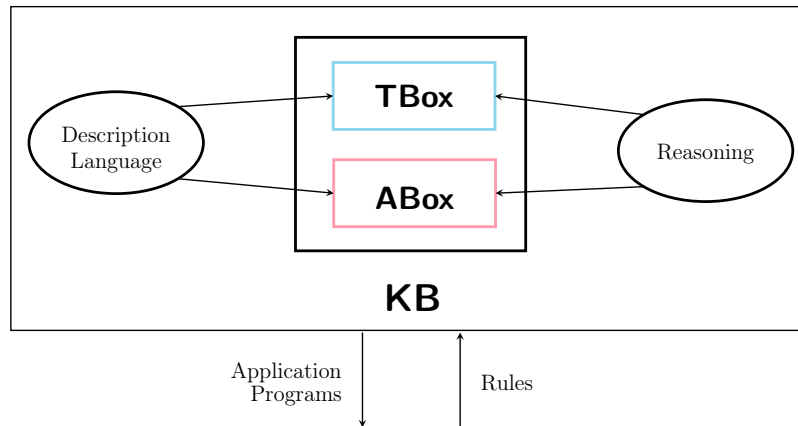


Figure 1.2.: Architecture of a Knowledge Based System as shown by [BCM⁺05]

TBox

The TBox contains the *intensional* knowledge, i.e. general knowledge about the domain. It specifies the *terminology* of the domain (that is why it is named *TBox*) by *definitions* and also by *inclusion axioms*. We have seen such a definition before. They are usually expressed by *logical equivalences*, where the left side of the equation is an atomic concept (see [BCM⁺05], p. 12 ff., 46, 51).

$$\text{Crew_Member} \equiv \text{Captain} \sqcup \text{Inferior}$$

Here again, a crew member is defined to be either a captain or an inferior (or even both). This knowledge is independent from the individuals which might be members of the concepts (or not).

An inclusion axiom has the form $C \sqsubseteq D$ with C and D being arbitrary concepts (see [BCM⁺05], p. 14).

[BS01] define the TBox thus (p. 19)¹:

“A TBox is a finite set of terminological axioms of the form $C \doteq D$, where C, D are concept descriptions. The terminological axiom $C \doteq D$ is called *concept definition* [if and only if] C is a concept name.

¹[BS01] use the symbol \doteq instead of \equiv for definitions.

An interpretation \mathcal{I} is a model of the TBox \mathcal{T} [if and only if] $C^{\mathcal{I}} = D^{\mathcal{I}}$ holds for all terminological axioms $C \doteq D$ in \mathcal{T} . [...]

A TBox is an *acyclic terminology*, if

- it does not contain *multiple definitions*
 $A \equiv C$ and $A \equiv D$...
- and it does not contain *cyclic definitions*
 $A_1 \equiv C_1, \dots, A_n \equiv C_n$ where A_i occurs in C_{i-1} ($1 < i \leq n$) and A_1 occurs in C_n

If a TBox is acyclic, then the definitions can be *unfolded*. That means, we can transform the defined atomic concepts (e.g. *Crew_Member*) back to their concept descriptions (e.g. *Captain* \sqcap *Inferior*) (see [BS01], p. 20). Doing this with TBoxes which are not acyclic would obviously cause a never-ending unfolding process.

ABox

In an ABox (or *world description*), *extensional* knowledge is stored, i.e. knowledge which addresses a certain problem. It specifies *assertions* about individuals (that is why it is named ABox) (see [BCM⁺05], p. 12, 15, 59).

The statement

$$\textit{Captain}(\textit{KATHRYN})$$

indicates that a specific individual called “KATHRYN” is a captain, while the assertions

$$\textit{navigates}(\textit{HARRY}, \textit{VOYAGER}) \quad \text{and} \quad \textit{Space_Ship}(\textit{VOYAGER})$$

describe together that a particular individual named “HARRY” navigates a specific space ship called “VOYAGER”.

With the help of the ABox, it is possible to check if a specific individual can be assigned to a particular concept (see [BCM⁺05], p. 15). So we might for example be able to obtain new knowledge by checking if HARRY is a navigator or a janitor.

From now on, the lower-case letters a , b and c are used for individual names. Generally spoken, an ABox’s assertion is

- a *concept assertion* $C(a)$, where C is a concept and a is part of it **or**
- a *role assertion* $R(b, c)$, where R is a role and c is a role-filler of R for b

An ABox is a finite set of such assertions (see [BCM⁺05], p. 59, 60). So, we can construct a simple ABox:

$Captain(KATHRYN)$	$commands(KATHRYN, HARRY)$
$Space_Ship(VOYAGER)$	$navigates(HARRY, VOYAGER)$

Table 1.1.: A Simple ABox

In order to be able to work with these individuals, it is necessary to extend the interpretation \mathcal{I} . We then have the possibility to map an *individual name* a to a certain individual $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$.

Further, we assume that every individual has its unique name, so $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ with a and b being distinct names. This feature is called the *unique name assumption* (UNA) (see [BCM⁺05], p. 60).

As mentioned before, $\Delta^{\mathcal{I}}$ is the set of all individuals, so for each individual name a , $a^{\mathcal{I}}$ is an element of $\Delta^{\mathcal{I}}$. \mathcal{I} is called to be a *model* of a particular ABox if \mathcal{I}

- *satisfies* the concept assertion $C(a)$ which is the case if $a^{\mathcal{I}} \in C^{\mathcal{I}}$;
- *satisfies* the role assertion $R(a, b)$, which is the case if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$

(see [BCM⁺05], p. 60; [BS01], p. 10).

But what does that mean for the TBox? Let us now look at the second part of the definition of the TBox by [BS01] (p. 19, 20)

“[...] The concept description D *subsumes* the concept description C w.r.t. the TBox \mathcal{T} (written $C \sqsubseteq_{\mathcal{T}} D$) [if and only if] $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all models \mathcal{I} of \mathcal{T} ; C is *satisfiable* w.r.t. \mathcal{T} [if and only if] there exists a model \mathcal{I} of \mathcal{T} such that $C^{\mathcal{I}} \neq \emptyset$. The Abox \mathcal{A} is *consistent w.r.t. \mathcal{T}* [if and only if] it has a model that is also a model of \mathcal{T} . The individual a is an instance of C w.r.t. \mathcal{A} and \mathcal{T} [if and only if] $a^{\mathcal{I}} \in C^{\mathcal{I}}$ holds for each model \mathcal{I} of \mathcal{A} and \mathcal{T} .”

To put it in a nutshell: \mathcal{I} satisfies an ABox \mathcal{A} *with respect to* a TBox \mathcal{T} if it satisfies (in addition to \mathcal{A}) also \mathcal{T} .

1.1.4. Inference

Now, after we know how to formalize knowledge, it is time to look at how to draw conclusions in order to obtain new facts or verify statements.

Satisfiability

For this, we first have to check if a given or new concept C is *satisfiable* with respect to a TBox \mathcal{T} . So, if there exists a model \mathcal{I} of \mathcal{T} such that $C^{\mathcal{I}} \neq \emptyset$, then C is satisfiable. Then \mathcal{I} is called to be a *model* of C with respect to \mathcal{T} (see [BCM⁺05], p. 62). One can also say that there has to be at least one individual in the concept C so that it is satisfiable with respect to \mathcal{T} .

In further definitions, the constraint “*with respect to \mathcal{T}* ” is dropped as it is clear that we always talk about a particular TBox.

Subsumption

An inference problem which often occurs, is to check if a concept C is *subsumed* by another concept D . That is the case if (and only if) $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all interpretations \mathcal{I} (see [BS01], p. 10).

Now, in \mathcal{ALC} we have the possibility to use *negations*. So, C is subsumed by D , if (and only if) $C \sqcap \neg D$ is *unsatisfiable* (see [BS01], p. 10). In this way, it is easy to check if one concept is subsumed by another: we just have to check unsatisfiability.

It is also possible to express unsatisfiability by subsumption:

C is unsatisfiable, if (and only if) $C \sqsubseteq \perp$ (see [BCM⁺05], p. 63), i.e. C is an empty concept.

Equivalence

Two concepts are *equivalent* if (and only if) $C^{\mathcal{I}} = D^{\mathcal{I}}$ for all interpretations \mathcal{I} (see [BCM⁺05], p. 62). This is written $C \equiv D$ and means that all individuals in C are

individuals in D and vice versa. Formally, this means that

$$C \equiv D \iff C \sqsubseteq D \ \& \ D \sqsubseteq C$$

So, as already shown with the subsumption, equivalence of C and D is present if (and only if) $C \sqcap \neg D$ and $D \sqcap \neg C$ are both unsatisfiable.

Disjointness

Finally, two concepts C and D are disjoint if (and only if) $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ for every model \mathcal{I} (see [BCM⁺05], p. 62). Disjointness of C and D means that no individual of C can be found in D and vice versa, in other words, $C \sqcap D \sqsubseteq \perp$, because $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$.

As shown before, this can be reduced to unsatisfiability of $C \sqcap D$. So again, we just have to test unsatisfiability in order to show that two concepts are disjoint.

To conclude: all four kinds of inference tests can be traced back to *(un)satisfiability* and also to *subsumption* if full negation is allowed.

Consistency

Having the assertions $Captain(KATHRYN)$ and $Janitor(KATHRYN)$ in our ABox, we might have a problem, at least if the TBox tells us that a janitor is an inferior who is commanded by the captain and nothing can be inferior to itself. So, the captain does not command himself and that is why we have an inconsistency in our ABox which should be detected.

[BCM⁺05] defines *Consistency* formally (p. 66):

“An ABox \mathcal{A} is *consistent with respect to a TBox* \mathcal{T} , if there is an interpretation that is a model of both \mathcal{A} and \mathcal{T} . We simply say that \mathcal{A} is *consistent* if it is consistent with respect to the empty TBox.”

Let us consider a set of all individuals N_I . *Satisfiability* can now be redefined using the term of *consistency* of ABoxes with the set N_I of all individuals:

1. “ C is satisfiable [if and only if] the ABox $\{C(a)\}$ for some $a \in N_I$ is consistent; and”

2. “ a is an instance of C w.r.t. \mathcal{A} [if and only if] $\mathcal{A} \cup \{\neg C(a)\}$ is inconsistent.”

[BS01], p. 10

After reducing satisfiability to the consistency problem, it is not hard to do the same with subsumption, equivalence and disjointness, because, as shown before, equivalence and disjointness can be reduced to subsumption which again can be reduced to (un)satisfiability and we have just seen that this can be reduced to (in)consistency of ABoxes.

1.1.5. Tableau-Based Reasoning

We will now look at a tableau-based algorithm for \mathcal{ALC} which has originally been introduced by [SSS91]. It uses the possibility to reduce the properties *satisfiability* and *subsumption* to the the problem of *consistency* for ABoxes. Unfortunately, this is only possible for languages which support negation (like \mathcal{ALC}). For other description logics, there are also other ways to perform reasoning (e.g. automaton based reasoning introduced by [CDL99]). We do not cover these here.

For the beginning, we omit TBoxes and perform reasoning on ABoxes only. Reasoning with TBoxes will be considered afterwards.

Rules

The tableau algorithm provides a set of *rules* which are applied to a given ABox \mathcal{A} . Such an application then leads to one or several new ABoxes \mathcal{A}' , \mathcal{A}'' ... including additional assertions. Of course, not every rule can be applied on any statement in \mathcal{A} . As in STRIPS-notation² (without negative effects), rules have a name, *conditions* and *actions* (effects). The rules for the various operators are shown in Table 1.2 (see [BS01], p. 11).

²STRIPS is an algorithm used to solve planning problems in artificial intelligence (see [GNT04], p. 76 ff.).

The \rightarrow_{\sqcap} -rule

Condition: \mathcal{A} contains $(C_1 \sqcap C_2)(x)$, but not both $C_1(x)$ and $C_2(x)$.

Action: $\mathcal{A}' := \mathcal{A} \cup \{C_1(x), C_2(x)\}$.

The \rightarrow_{\sqcup} -rule

Condition: \mathcal{A} contains $(C_1 \sqcup C_2)(x)$, but neither $C_1(x)$ nor $C_2(x)$.

Action: $\mathcal{A}' := \mathcal{A} \cup \{C_1(x)\}$, $\mathcal{A}'' := \mathcal{A} \cup \{C_2(x)\}$.

The \rightarrow_{\exists} -rule

Condition: \mathcal{A} contains $(\exists r.C)(x)$, but there is no individual name z such that $C(z)$ and $r(x, z)$ are in \mathcal{A} .

Action: $\mathcal{A}' := \mathcal{A} \cup \{C(y), r(x, y)\}$, where y is an individual name not occurring in \mathcal{A} .

The \rightarrow_{\forall} -rule

Condition: \mathcal{A} contains $(\forall r.C)(x)$ and $r(x, y)$, but it does not contain $C(y)$.

Action: $\mathcal{A}' := \mathcal{A} \cup \{C(y)\}$.

Table 1.2.: Transformation Rules of the Satisfiability Algorithm for \mathcal{ALC}

Every rule creates either one or two new ABoxes \mathcal{A}' (and sometimes \mathcal{A}'') containing all elements which have already been in \mathcal{A} , but in addition also new assertions depending on the rule which has been used. The conditions create some restrictions so that a rule can only be applied if it generates new results. Otherwise, it would be possible to apply one rule again and again without any new results and the algorithm would never terminate.

It is assumed that all concept descriptions are in *negation normal form* (NNF), so we do not need a \rightarrow_{\neg} -rule and find *clashes* more easily (*clashes* will be described below). This means, that we have the negation \neg only directly in front of concept names. With the rule of *de Morgan* and the rules for quantifiers, it is possible to transform any expression to NNF in linear time (see [BS01], p. 11; [BCM⁺05], p. 78).

Here is an example:

$$\begin{aligned}
& \neg(\exists R.(A \sqcap B)) && \text{quantification rule} \\
\Leftrightarrow & \forall R.\neg(A \sqcap B) && \text{de Morgan} \\
\Leftrightarrow & \forall R.(\neg A \sqcup \neg B) && \Longrightarrow \text{NNF}
\end{aligned}$$

An ABox \mathcal{A} is *consistent*, if (and only if) the new succeeding ABox \mathcal{A}' is consistent, too. After applying a sequence of rules to an initial ABox, we finally get an ABox where we cannot use any more rules, because the conditions do not give us an opportunity to do so. If this last ABox is consistent, the original one must also be consistent (at least if the algorithm is correct) (see [BS01], p. 12).

The existential quantification rule \rightarrow_{\exists} in x creates a role-filler y . Besides, a new *relation* $r(x, y)$ is generated. By contrast, the rule \rightarrow_{\forall} can only be applied if there is already such a generated role-filler, so the target concept C can be assigned to all the role-filling individuals y .

The \rightarrow_{\sqcup} -rule creates two different new ABoxes \mathcal{A}' and \mathcal{A}'' . The original ABox \mathcal{A} is consistent if and only if (at least) *one* of the new \mathcal{A}' or \mathcal{A}'' is so (see [BS01], p. 12). The reason is simple: $(C \sqcup D)(x)$ is true if (and only if) $C(x)$ or $D(x)$ or even both are true. So there is no need that both conform to that condition. So if for example $C(x)$ in \mathcal{A}' leads to an inconsistency, \mathcal{A}'' with $D(x)$ has to be tested on consistency.

Completeness and Clashes

“An ABox \mathcal{A} is called *complete* [if and only if] none of the transformation rules of [Table 1.2] applies to it. The ABox \mathcal{A} contains a *clash* [if and only if] $\{P(x), \neg P(x)\} \subseteq \mathcal{A}$ for some individual name x and some concept name P . An ABox is called *closed* if it contains a clash, and *open* otherwise.”

[BS01], p. 12

So, for example an ABox with both assertions

$$\textit{Captain}(x) \quad \text{and} \quad \neg\textit{Captain}(x)$$

contains a *clash*, and therefore the ABox is inconsistent and unsatisfiable. If this is a case for an ABox A' , we have to backtrack. Maybe a \sqcup has been disassembled earlier (let us say in \mathcal{A}), so there is a chance that another ABox \mathcal{A}'' is satisfiable. If such a branch does not exist, we are finished and our original ABox is inconsistent. Otherwise we have to look for clashes in \mathcal{A}'' again until the tableau is either *closed* (then the original ABox is not satisfiable) or it is *complete* and there is a branch which does not contain a clash.

Running the Algorithm

At the beginning we always start with a *prototypical* ABox with the abstract individual x . Our aim is to show that this prototype is satisfiable or it is not. We just have to apply all possible rules until the tableau is either closed or there are no more applicable rules left.

Let us look at the subsumption again. As shown before, $C \sqsubseteq D$ is true if (and only if) $C \sqcap \neg D$ is *unsatisfiable*. For a prototypical (fresh) individual x this means that we can check the subsumption by applying the algorithm on the ABox $\{C(x) \sqcap \neg D(x)\}$. If it leads to a *closed* tableau, then $\{C(x) \sqcap \neg D(x)\}$ is inconsistent and therefore $C(x) \sqsubseteq D(x)$ eventually is satisfiable.

Correctness

There are several ways to obtain new knowledge. In any case, it is important that the results are correct. The inference procedure is of no value if it tells us wrong conclusions, while it is also not very helpful if it does not find facts which are obviously there. Further, it is always helpful if the termination of an algorithm is guaranteed.

Let us consider a set of ABoxes $\mathcal{S} = \{\mathcal{A}_1, \dots, \mathcal{A}_k\}$, where \mathcal{A}_1 represents the initial ABox of our tableau followed by all ABoxes which result by applying rules on the ABox before. \mathcal{S} is consistent if and only if there is some i , $1 \leq i \leq k$, such that \mathcal{A}_i is consistent ([BCM⁺05], p. 80).

Termination *Let C_0 be an \mathcal{ALC} -concept in negation normal form. There cannot be an infinite sequence of rule applications*

$$\{\{C_0(x_0)\}\} \rightarrow \mathcal{S}_1 \rightarrow \mathcal{S}_2 \rightarrow \dots$$

([BS01], p. 12).

The proof for the extended language \mathcal{ALCN} is given by [BCM⁺05] (p. 81).

Soundness *Assume that \mathcal{S}' is obtained from the finite set of ABoxes \mathcal{S} by application of a transformation rule. Then \mathcal{S} is consistent [if and only if] \mathcal{S}' is consistent.*

([BCM⁺05], p. 81)

[BCM⁺05] explains this as an “easy consequence of the definition of the transformation rules” (p. 80) which are shown in Table 1.2. This leads us to the fact that any closed ABox \mathcal{A} is *inconsistent* (see [BS01], p. 12).

Completeness *Any complete and clash-free ABox \mathcal{A} has a model.* ([BCM⁺05], p. 82)

In other words: *Any complete and open ABox \mathcal{A} is consistent.* ([BS01], p. 12)

[BCM⁺05] provide a proof for this lemma (p. 82, 83).

With all this, we can formulate the following theorem:

It is decidable whether or not an \mathcal{ALC} -concept is satisfiable.

[BS01], p. 13

Example

It is now time to see the algorithm in action. The following example is adapted from [BCM⁺05] (p. 78):

Let us suppose that we want to prove that the following subsumption is always correct:

$$(\exists R.A \sqcap \exists R.B) \sqsubseteq (\exists R.(A \sqcap B))$$

So everything we have to do is to show that the expression

$$\exists R.A \sqcap \exists R.B \sqcap \neg \exists R.(A \sqcap B)$$

is unsatisfiable. As the algorithm does not provide rules for negation, we have to transform that statement to NNF.

$$\exists R.A \sqcap \exists R.B \sqcap \forall R.(\neg A \sqcup \neg B)$$

Let us construct an initial ABox \mathcal{A} with the abstract individual x which shall be member of each concept.

$$\mathcal{A} = \{(\exists R.A \sqcap \exists R.B \sqcap \forall R.(\neg A \sqcup \neg B))(x)\}$$

This is where the algorithm starts. First, the only applicable rule is \rightarrow_{\sqcap} . We use it twice.

$$\mathcal{A}'_1 = \mathcal{A} \cup \{(\exists R.A)(x), (\exists R.B)(x), (\forall R.(\neg A \sqcup \neg B))(x)\}$$

Now we have four single elements in our ABox \mathcal{A}'_1 . The \rightarrow_{\sqcap} -rule cannot be applied once more, because this would not lead to any new results. So we can apply the \rightarrow_{\exists} -rule on $\exists R.A$ or on $\exists R.B$. Rule \rightarrow_{\forall} cannot be applied on the last element of \mathcal{A}'_1 , because it requires that there is already a R -relation from x to at least one role-filler, but there is no such role-filler right now.

Let us apply the \rightarrow_{\exists} -rule twice (i.e. on both elements). It generates new individuals each time. We call them y and z .

$$\mathcal{A}'_2 = \mathcal{A}'_1 \cup \{A(y), R(x, y), B(z), R(x, z)\}$$

Because we now have our role-fillers with the corresponding concepts, the \rightarrow_{\exists} -rule cannot be applied again. But because we now have our relations leading from x , we can finally apply \rightarrow_{\forall} on $(\forall R.(\neg A \sqcup \neg B))(x)$. We can apply this rule twice, because we have two different relations which lead from x to other individuals (y and z).

$$\mathcal{A}'_3 = \mathcal{A}'_2 \cup \{(\neg A(y) \sqcup \neg B(y)), (\neg A(z) \sqcup \neg B(z))\}$$

We can decide if we want to apply \rightarrow_{\sqcup} on y or on z so let us start with y . But here we

have to branch, because \rightarrow_{\sqcup} constructs two different ABoxes.

$$\mathcal{A}'_4 = \mathcal{A}'_3 \cup \{\neg A(y)\} \qquad \mathcal{A}''_4 = \mathcal{A}'_3 \cup \{\neg B(y)\}$$

In \mathcal{A}'_2 we already had $A(y)$. This results in a *clash* in \mathcal{A}'_4 . Anyway, the tableau is still not *closed*, so we still do not know if \mathcal{A} is unsatisfiable. For this, we have to prove that there is also a clash in \mathcal{A}''_4 . To do so, let us apply \rightarrow_{\sqcup} on $\neg A(z) \sqcup \neg B(z)$. Again, two different ABoxes are obtained:

$$\mathcal{A}''_5 = \mathcal{A}''_4 \cup \{\neg A(z)\} \qquad \mathcal{A}'''_5 = \mathcal{A}''_4 \cup \{\neg B(z)\}$$

A *clash* occurs in \mathcal{A}'''_5 due to the element $B(z)$ which we already have in \mathcal{A}'_2 . Still, the left branch \mathcal{A}''_5 is open. Looking at the rule-conditions makes us realize that no rule is applicable here. So, we are done. There is no closed tableau and by completeness of the algorithm we conclude that \mathcal{A} is satisfiable and therefore the initial subsumption is *not* valid.

The complete tableau is shown in Figure 1.3. The crosses x mean that we have a clash at that position.

Reasoning with TBoxes

In order to involve definitions of an *acyclic TBox*, it is enough to *unfold* these. We have already seen how this can be done. If a TBox contains such a definition, it should then be unfolded *on demand* (see [BS01], p. 20, 21). So if a defined concept $A \equiv C$ occurs negated and non-negated in an ABox ($A(x), \neg A(x)$), it is not necessary to unfold them, because we then already have our contradiction and therefore a *clash*.

But if considering *general* TBoxes (which might contain cycles), the algorithm will probably not terminate when unfolding a concept again and again, without ever detecting a clash. So if (general) TBoxes are allowed, we also need a mechanism which detects such *cycles*. [BCM⁺05] describes a strategy to detect loops and how to *block* the corresponding ABoxes, so the algorithm will finally terminate (see p. 86). We will not talk about blocking here in detail.

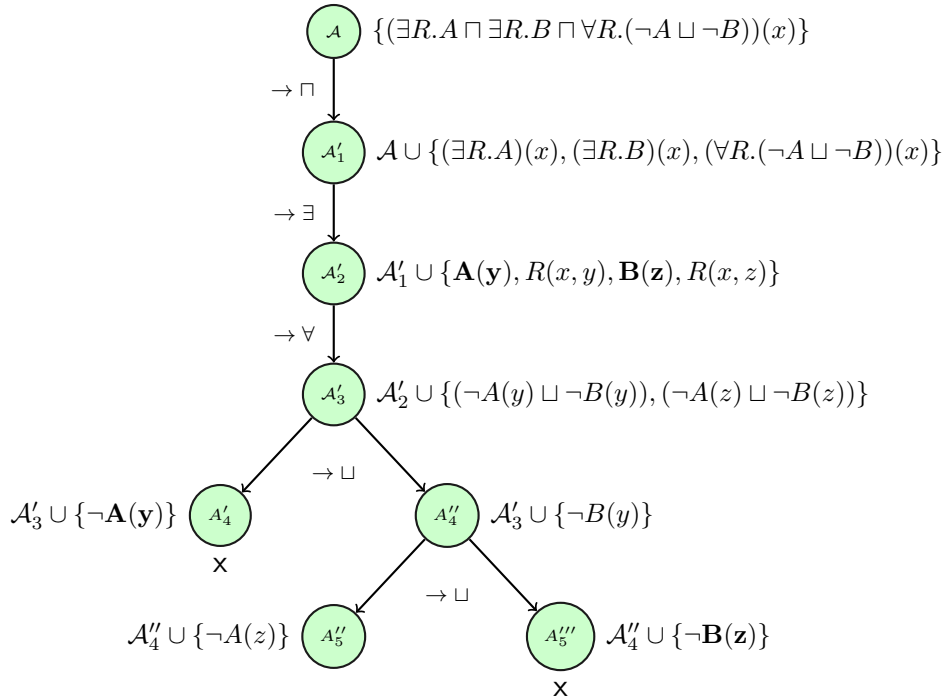


Figure 1.3.: Illustration of a Tableau

Miscellaneous

Extending \mathcal{ALC} to other, more expressive languages (like for example \mathcal{ALCN} or languages with transitive or reflexive roles) makes it compulsory to alter the set of possible rules when using a tableau algorithm. Of course, then correctness and completeness has to be checked again (see [BS01], p. 15 ff).

1.1.6. Description Logic and Modal Logic

According to [Baa09], “ \mathcal{ALC} is just a syntactic variant of the basic multimodal logic \mathbf{K}'' (p. 7). Concepts are then expressed as *propositional variables* while roles are relations between *worlds* in modal logic. The pendant to the value restriction is then the *Box operator* \square . Existential quantification is expressed by the *Diamond* \diamond .

If the used modal logic was not *multimodal*, we would not be able to distinguish between the different role names. So, the role names might be written as indices to the modal operators (\square and \diamond).

[Baa09] defines the function θ , which translates an \mathcal{ALC} concept description C to a modal formula thus (p.7):

$$\begin{aligned}
 \theta(A) &\longleftarrow a \text{ (name of atomic concept } A) \\
 \theta(C \sqcap D) &\longleftarrow \theta(C) \wedge \theta(D) \\
 \theta(C \sqcup D) &\longleftarrow \theta(C) \vee \theta(D) \\
 \theta(\neg C) &\longleftarrow \neg\theta(C) \\
 \theta(\forall r.C) &\longleftarrow \Box_r \theta(C) \\
 \theta(\exists r.C) &\longleftarrow \Diamond_r \theta(C)
 \end{aligned}$$

In this way, our example $\exists \textit{navigates.Space_Ship}$ could be translated to such a modal expression:

$$\Diamond_{\textit{navigates}} \textit{space_ship}$$

Later, we will talk about modal logics again and use them to construct bridges to Description Logic in various contexts.

1.1.7. Summary

Here are again the most important issues about Description Logic:

- Description Logic is used to describe knowledge and derive new facts. *Concepts* and *roles* are used for this.
- There are several *languages* which are more or less expressive than others. All are extensions to the basic language \mathcal{AL} .
- Concept *definitions* and terminologies with *inclusion axioms* are stored in the *TBox*, while knowledge about individuals (world descriptions) is saved in the *ABox*. Both components are elements of the *Knowledge Base*.
- The properties *equivalence* and *disjointness* of concepts can be reduced to the properties of *subsumption* and *unsatisfiability* in \mathcal{ALC} . For ABoxes, subsumption and unsatisfiability can both be reduced to *inconsistency*.
- The tableau algorithm for \mathcal{ALC} introduced by [SSS91] performs inferences on ABoxes using these properties.

- \mathcal{ALC} is similar to the multimodal modal logic K. An arbitrary \mathcal{ALC} -expression can be transformed to a modal logic formula using a simple function.

Next, we will look at the constructive language $c\mathcal{ALC}$.

1.2. The Constructive Description Logic $c\mathcal{ALC}$

In the last years, new description languages have been introduced, that have *intuitionistic* or *constructive* semantics, i.e. they have a special perspective on Description Logic.

For example, [FFF10] have introduced the *Basic Constructive Description Logic* \mathcal{BCDL} that focuses *information term semantics*. [HRdP10] have introduced an *intuitionistic* \mathcal{ALC} , called iALC, that they suggest to apply in fields of law.

The constructive language $c\mathcal{ALC}$ has been introduced by [MS09] with the aim to cope with domains which contain *dynamic* or *incomplete* knowledge.

In this section, we have an introduction to $c\mathcal{ALC}$. After showing the purpose and after explaining syntax and semantics, we will see how to perform tableau-based proofs in $c\mathcal{ALC}$.

1.2.1. Why Constructive?

Intuitionist and Classical Logic

Before we consider reasons for working with *constructive* Description Logic, it might be helpful to have a short introduction to *intuitionist logic*³ which has its roots in some mathematical fields of philosophy (see [Pri01], p. 99) and which is sometimes also referred to as *constructive logic*.

³In some sources it is called *intuitionistic logic*, in others *intuitionist logic*. We use both terminologies, but do not make any difference.

Let us consider the assertion

$$A \vee \neg A$$

in *propositional logic*. It says, that A is either *true* or *false*. It is an axiom which is always true in *classical logic*. For example, the expression

$$\textit{Sun_shining} \vee \neg \textit{Sun_shining}$$

always makes a correct assertion. Either the sun is shining, or it is not shining. This rule is called the *Law of the excluded middle* or *excluded third* (see [Min00], p. 1).

But now, let us consider the assertion “*There are infinitely many twin primes*”, where a twin prime is a pair of prime numbers (a, b) with $b - a = 2$, for example $(3, 5)$ or $(11, 13)$ are twin primes. Although we know that there are infinitely many prime numbers, there is still no known way to prove or refute that there are infinitely many twin primes (see [Pri01], p. 101).

Now, some might think, that the assertion about the twin primes is true, some might disagree, but it is obvious that (at least till now) there is no *proof* which shows that it is and also no proof which shows that it is *not* true. It might be, but we cannot be sure about it.

Intuitionist Logic deals with such proof-expressions. The assertion $A \wedge B$ says that there is a *proof* which shows that A is true *and* there is also a *proof* that shows that B is true, so it indicates a proof for A *and* B . Analogous, $A \vee B$ is an expression for a proof which again is a proof of A *or* a proof of B (see [Pri01], p. 100).

Now, $\neg A$ does not say that it is just wrong that there is a proof for A . Instead, a “proof of $\neg A$ ” is a proof that there is no proof for A ” ([Pri01], p.100). It is obvious, that then $A \vee \neg A$ is not necessarily true for all A , because there might be a way to proof A , or to refute A , but it is also possible that neither one, nor the other can be proved and this is one issue that makes intuitionist logic different to classical logic.

Constructiveness in Description Logic

Let us move away from general intuitionist logic and have a look on *constructiveness* in Description Logic.

As [MS09] describe, information which are stored in a knowledge base, are often dynamic or incomplete. The individuals which are represented in the knowledge are *abstract*. Of course, they are not real at all (p. 208), because it is not possible to store every single information of an individual in a knowledge base.

Sometimes, it is inevitable to cope with abstractions and abstracted concepts. This is especially necessary if we deal with individuals, we do not have the complete knowledge about or individuals whose features are not static.

In order to exclude wrong conclusions, it is necessary to alter our description language, so that it is able to cope with these abstract entities. The language *cALC* is such an alternation.

To illustrate that problem, here is an example which is based on the fairy tale *Snow White and the Seven Dwarfs*⁴.

Example 1.2

Once, there was a Queen who was not able to get along with the fact that she was not the most beautiful person in the land. That is why she wanted to kill her stepdaughter Snow White who has been (according to the Queen's speaking mirror) thousandfold more beautiful. Snow White fled to the Seven Dwarfs, but the evil stepmother followed her, masked as an old lady who sold apples. One of these apples (the red one) had been poisoned. Snow White did not realise that the old woman was her stepmother, but still, she did not trust her first. So, the jealous Queen ate the pale part of the manipulated apple (that was the part which was not poisoned). Snow White, now convinced that it would not be dangerous, ate the red part and died (fortunately only temporarily). □

Of course we could define a concept *Apple* by joining both concepts *Red_Apple* and *Pale_Apple*, where the red apples are poisoned, so they cause *death*, while the pale ones are still sour and cause *stomach ache* (but at least, they do not kill us).

⁴The original version can be found in *Kinder- und Hausmärchen* by the Grimm Brothers as *Schneewittchen*.

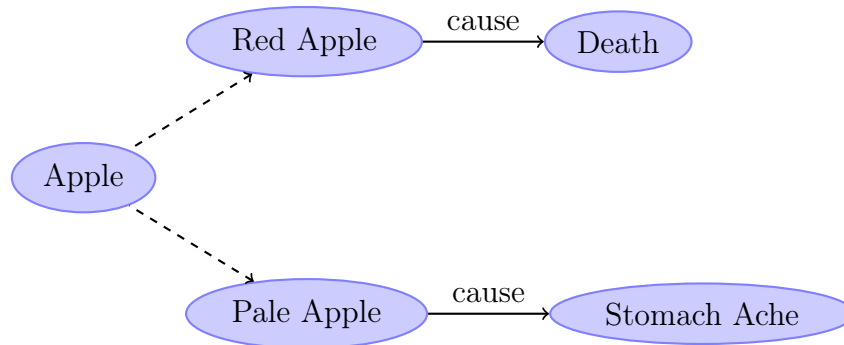


Figure 1.4.: The Queen's Knowledge about her Apples

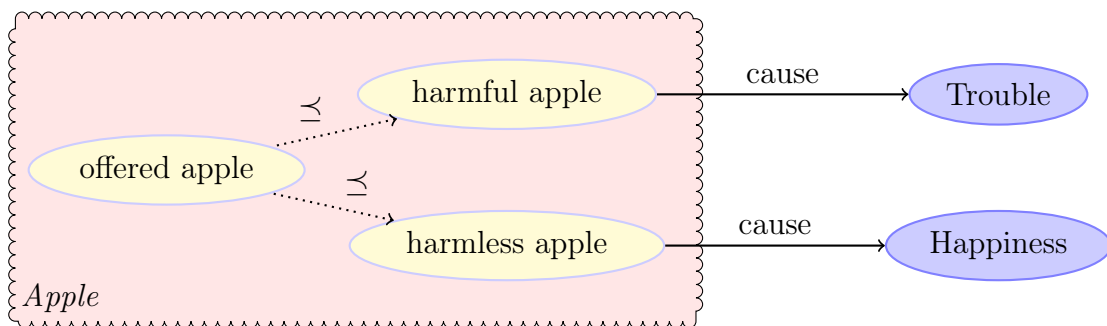


Figure 1.5.: Snow White's Knowledge about Apples

$$Apple \equiv Red_Apple \sqcup Pale_Apple$$

$$Red_Apple \sqsubseteq \forall cause.Death$$

$$Pale_Apple \sqsubseteq \forall cause.Stomach_Ache$$

So far, we do not need constructiveness, because the stepmother is able to distinguish which apples are dangerous and which can be eaten without risk (apart from the stomach ache). The Queen's knowledge about her apples is illustrated in Figure 1.4.

But now, let us consider the scene from Snow White's point of view. Knowing that her stepmother tries to kill her, she acts carefully, because the Queen has tried to murder her several times before. Her knowledge about apples is shown in Figure 1.5.

Snow White only knows that the old lady offers her an (abstract) apple and that it might be either *harmful* or *harmless*. Her problem is, that she is not able to say which

apples are the bad ones, because she does not know that her stepmother has marked the poisoned ones red. So, she cannot be sure about the outcome of eating the offered piece of red apple.

Now the Queen uses an artful trick by eating the pale part of the apple which is not harmful. So her naive stepdaughter eats the rest and dies. Anyway, she could not have known.

Regarding Figure 1.5, the harmless and the harmful apple contain more information than our *abstract* offered apple (which does not really exist in reality), i.e. they are more *concrete* than the abstract one. This increase of information is represented by *refinement*-relations, illustrated by the dotted edges and the symbol \preceq (not to mix this up with the *number restriction* \leq). Anyway, neither *offered apple*, nor *harmful apple*, nor *harmless apple* are concepts any more, but *entities*. These are all members of an enveloping concept which is indicated by the red cloud.

Application in Reality

Let us move away from fairy tales and go back to reality again. [MS09] and [MS08] outline that *auditing* in economy gives a good example for an application. Automated auditing is getting more and more important and besides, it is a delicate issue due to new strict laws like the Sarbanes-Oxley Act⁵.

Many processes which have to be monitored are dynamic, so their states are changing for a period of time. Further, people holding a certain position in a company may change, i.e. they might be replaced by others while the auditing process is taking place. Eventually, the abstract concept ‘CEO of company X’ does not say anything about personal qualities of a particular person doing the job. Finally, an abstraction from irrelevant available data can simplify the complexity of the process. Quick checks can be performed when abstracting. (see [MS09], p. 209).

⁵The Sarbanes-Oxley Act is a US law of 2002 which has been initiated to “protect investors by improving the accuracy and reliability of corporate disclosures made pursuant to the securities laws, and for other purposes” <http://www.sec.gov/about/laws/soa2002.pdf>

$c\mathcal{ALC}$ and Intuitionistic Logic

The tautologies of classical logic which do not hold for intuitionistic logic are also not valid in $c\mathcal{ALC}$ (see [MS09], p. 213). This can be explained with the lack of information about an entity. So, for example, we might not be sure if an entity x is an element of a concept C or if it is not. That is why the expression $C(x) \sqcup \neg C(x) \equiv \top$ is not a tautology anymore which means that it is probably not true for all entities x .

We will compare $c\mathcal{ALC}$ to intuitionistic logic in Section 1.2.3 in detail.

1.2.2. Syntax and Semantics

Syntax

Next, we will have a look at the syntax of $c\mathcal{ALC}$. As before, the letters C and D represent *concept descriptions*, while A is used for *atomic concepts* and R for *roles*.

C, D	\longrightarrow	A		(atomic concept)
		\top		(universal concept)
		\perp		(bottom concept)
		$\neg C$		(negation)
		$C \sqcap D$		(intersection)
		$C \sqcup D$		(union)
		$C \sqsubseteq D$		(subsumption)
		$\forall R.C$		(value restriction)
		$\exists R.C$		(full existential quantification)

(see [MS09], p. 211)

It is noteworthy that the *subsumption* has been added (it has not been part of \mathcal{ALC} 's syntax definition). The reason is that we do not need it in \mathcal{ALC} , because we can substitute it there:

$$C \sqsubseteq D \iff \neg C \sqcup D$$

This is not possible in $c\mathcal{ALC}$, because now, these operators are independent from each other. The subsumption is now denoted to be a "*concept-forming operator*". Anyway,

\top can still be written as $\neg\perp$ and the negation \neg and the bottom concept \perp can still be represented by each other thus ([MS09], p. 211):

$$\perp = A \sqcap \neg A \quad \text{and} \quad \neg C = C \sqsubseteq \perp$$

The Refinement Relation

Further, we have the new *refinement* relation expressed by the symbol \preceq . It is used on *entities*. The term *entity* is now used instead of *individual*, because in contrast to individuals, entities are abstract or “partially defined”, i.e. they do not contain the amount of information, an individual (which represents real objects) usually contains (see [MS09], p. 211).

The expression $a \preceq b$ points out that the entity b is more concrete or at least as concrete as the entity a . In other words, b carries at least the same amount of information as a . We say, that b *refines* a . This relation is reflexive⁶. However, if $a \preceq b$ and $b \preceq a$, then both entities need not to be equal. Further, if an entity a is member of a concept C and $a \preceq b$, then b must also be a member of the concept C (see [MS09], p. 211).

In our apple-example, the entity *apple* is more abstract than the *harmful apple* which contains more information (the information that it is harmful). Analogous the *harmless apple* refines the more abstract one. This can be expressed by the statements

$$apple \preceq harmful_apple \quad \text{and} \quad apple \preceq harmless_apple$$

Now, the harmless and the harmful apple both contain the same amount of information. So these statements are true, too⁷:

$$harmful_apple \preceq harmless_apple \quad \text{and} \quad harmless_apple \preceq harmful_apple$$

Therefore, both entities have the same amount of information, but obviously, they are not equal.

⁶For any reflexive relation $R \subseteq A \times A$ we have $\forall a \in A . (a, a) \in R$ (see [EMGR⁺01], p. 80).

⁷These refinement relations are not illustrated in Figure 1.5, because they might have caused confusion at that point.

Due to the property of *reflexivity*, every entity refines itself. Further, the refinement relation is also *transitive*⁸ (see [MS09], p. 211).

Semantics

The original interpretation structure \mathcal{I} used by \mathcal{ALC} has to be extended for $c\mathcal{ALC}$:

$$\mathcal{I} = (\Delta^{\mathcal{I}}, \preceq^{\mathcal{I}}, \perp^{\mathcal{I}}, \cdot^{\mathcal{I}})$$

Every part of it has its own function explained in Table 1.3 (see [MS09], p. 211, 212). The set N_C contains all *concept names*, while the *role names* are all elements of N_R .

$\Delta^{\mathcal{I}}$	A set containing all <i>entities</i> , i.e. <i>abstract</i> or <i>partially defines</i> individuals
$\preceq^{\mathcal{I}}$	A transitive and reflexive relation over $\Delta^{\mathcal{I}}$
$\perp^{\mathcal{I}}$	A set containing the <i>fallible</i> entities; it is a subset of $\Delta^{\mathcal{I}}$
$\cdot^{\mathcal{I}}$	An interpretation function which maps each role name $R \in N_R$ to a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and each atomic concept $A \in N_C$ to a set $\perp^{\mathcal{I}} \subseteq A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$

Table 1.3.: Semantic Interpretation Structure of $c\mathcal{ALC}$

The refinement relation has already been explained before.

The set of *fallible* entities can be considered thus:

“Fallible elements $b \in \perp^{\mathcal{I}}$ may be thought of as over-constrained tokens of information, self-contradictory objects of evidence or undefined computations. [...] Fallible entities are information-wise maximal elements and therefore included in every concept, i.e., $\perp^{\mathcal{I}} \subseteq C^{\mathcal{I}}$ for all C .”

[MS09], p. 212

Let us now look at how the concept descriptions are interpreted semantically. For this, we use the set $\Delta_c^{\mathcal{I}}$ for *non-fallible* entities. It can be defined as $\Delta^{\mathcal{I}} \setminus \perp^{\mathcal{I}}$.

⁸Transitivity is defined as $\forall a, b, c \in A . (a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$ for $R \subseteq A \times A$ (see [EMGR⁺01], p. 80).

$$\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} &= \{x \mid \forall y \in \Delta_c^{\mathcal{I}}. x \preceq^{\mathcal{I}} y \Rightarrow y \notin C^{\mathcal{I}}\} \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(C \sqsubseteq D)^{\mathcal{I}} &= \{x \mid \forall y \in \Delta^{\mathcal{I}}. (x \preceq^{\mathcal{I}} y \ \& \ y \in C^{\mathcal{I}}) \Rightarrow y \in D^{\mathcal{I}}\} \\
(\forall R.C)^{\mathcal{I}} &= \{x \mid \forall y \in \Delta^{\mathcal{I}}. x \preceq^{\mathcal{I}} y \Rightarrow \forall z \in \Delta^{\mathcal{I}}. (y, z) \in R^{\mathcal{I}} \Rightarrow z \in C^{\mathcal{I}}\} \\
(\exists R.C)^{\mathcal{I}} &= \{x \mid \forall y \in \Delta^{\mathcal{I}}. x \preceq^{\mathcal{I}} y \Rightarrow \exists z \in \Delta^{\mathcal{I}}. (y, z) \in R^{\mathcal{I}} \ \& \ z \in C^{\mathcal{I}}\}
\end{aligned}$$

(see [MS09], p. 212)

In \mathcal{ALC} , $(\neg C)^{\mathcal{I}}$ is defined as $\Delta^{\mathcal{I}} \setminus C$. But now, in $c\mathcal{ALC}$, this expression is stronger as it indicates that all of C 's refinements are fallible, too. \top , \sqcap , \sqcup are similar in \mathcal{ALC} , but the subsumption $C \sqsubseteq D$ now also holds for all refinements of C . Also, the quantifications to the role-filler C hold for C 's refinements.

An important feature of $\perp^{\mathcal{I}}$ is that for all concepts C , $\perp^{\mathcal{I}} \subseteq C^{\mathcal{I}}$. This can be proved by induction (see [MS09], p. 230, 231)

1.2.3. Intuitionistic and Constructive Description Logic

$c\mathcal{ALC}$ and Modal Logic

As we have seen in Section 1.1.6, \mathcal{ALC} is similar to the multimodal logic K . For $c\mathcal{ALC}$, there is also such a counterpart called CK (constructive K , see [Mdp05]). There is a CK which has been introduced before by [Wij90] but which is not completely the same as the CK we will consider here in detail, although it is quite close to it.

Differences Between Constructive and Intuitionistic Logics

Not every axiom that holds in an intuitionistic modal logic, also holds in CK (see [MS10] p. 2 ff). [PS86] have introduced five axioms which hold in the intuitionistic modal logic IK . The axioms provided by [FS80] (FS) are similar but with a focus on the “duality between \Box and \Diamond ” ([Sim94], p. 51). Both include all theorems of *intuitionistic propositional logic* (IPL) (see [MS10], p. 2). The axioms of IK and FS are listed in Table 1.4.

Axioms (IK)	Axioms (FS)
<i>All theorems of IPL</i>	<i>All theorems of IPL</i>
IK1 : $\Box(A \supset B) \supset (\Box A \supset \Box B)$	FS1 : $\Box \top$
IK2 : $\Box(A \supset B) \supset (\Diamond A \supset \Diamond B)$	FS2 : $\Box(A \wedge B) \equiv (\Box A \wedge \Box B)$
IK3 : $\neg \Diamond \perp$	FS3 : $\neg \Diamond \perp$
IK4 : $\Diamond(A \vee B) \supset (\Diamond A \vee \Diamond B)$	FS4 : $\Diamond(A \vee B) \equiv (\Diamond A \vee \Diamond B)$
IK5 : $(\Diamond A \supset \Box B) \supset \Box(A \supset B)$	FS5 : $(\Diamond A \supset \Box B) \supset \Box(A \supset B)$
	FS6 : $\Diamond(A \supset B) \supset (\Box A \supset \Diamond B)$

Table 1.4.: Axioms for the Intuitionistic Modal Logics IK and FS (see [MS10], p. 2)

Now, [MS10] present several reasons, why not all of these axioms hold in *constructive* modal logic. For example the axiom IK4/FS4 is not valid anymore (see p. 2 ff). It is not so easy to understand why, that is why we will explain it with an illustrative example.

Example 1.3

Let us suppose that I have a door and two keys which look the very same, so I cannot distinguish them. A key might be able to open a door, but then it cannot close it (then it is an *o-key*). It is also possible that a key closes a door but then it cannot open it (then it is a *c-key*). Anyway, I do not know if I have one *o-key* and one *c-key*, or if both of my keys are *c-keys* or both are *o-keys*. However, as both of my keys look equal, I assume that they are both the same type of key. \square

Our modal operator \diamond_{put} denotes the process of turning one of my keys in the key-hole of the door. So, no matter which key I take, the formula $\diamond_{put}(open \vee closed)$ is always true, because after I have put the key into the key-hole and turned it around, the door is open or closed.

Now, after putting one of my keys into the key-hole and turning it, I can check if the door is closed. Let us suppose that it is, then $\diamond_{put}closed$ is true and $\diamond_{put}open$ is false. Still, if I take the other key by mistake (which is an *o-key* as I cannot distinguish them) next time and put it into the keyhole, then suddenly $\diamond_{put}closed$ is not true anymore.

Although $\diamond_{put}open \vee \diamond_{put}closed$ has been true before my first try, I have made my decision after putting the c -key into the key-hole for the first time.

The modal operator \diamond_{put} has been dissolved and $\diamond_{put}closed$ is assumed to be true, but in reality, it is not because I have not tested both keys. So, the formula

$$\diamond_{put}(open \vee closed) \supset (\diamond_{put}open \vee \diamond_{put}closed)$$

is not true in CK.

The axioms IK3/FS3 and IK5/FS5 do not hold in CK either (see [MS10], p. 2 ff). The set of axioms is reduced, so that only the following remain and these are valid for constructive issues:

Axioms (CK-1)	Axioms (CK-2)
<i>All theorems of IPL</i>	<i>All theorems of IPL</i>
IK1 : $\Box(A \supset B) \supset (\Box A \supset \Box B)$	FS1 : $\Box \top$
IK2 : $\Box(A \supset B) \supset (\Diamond A \supset \Diamond B)$	FS2 : $\Box(A \wedge B) \equiv (\Box A \wedge \Box B)$
	FS6 : $\Diamond(A \supset B) \supset (\Box A \supset \Diamond B)$

Table 1.5.: Axioms for the Constructive Modal Logic CK (see [MS10], p. 4)

Back to Description Logic

So because CK is related to $c\mathcal{ALC}$, the axioms might simply be converted syntactically. IK3, IK4, IK5 and also their counterparts FS3, FS4 and FS5 can be refuted by the tableau proofing systems introduced by [MS09], while the remaining axioms hold.

So, there are axioms from intuitionistic logic which are not valid in constructive logic, whereas there are also classical axioms which do not hold in intuitionistic and therefore also not in constructive logic. Table 1.6 gives an overview of some of these *tautologies* (see [MS09], p. 217).

<i>Excluded Middle</i>	$C \sqcup \neg C = \top$
<i>Double Negation</i>	$\neg\neg C = C$
<i>Dualities</i>	$\exists R.C = \neg\forall R.\neg C$
	$\forall R.C = \neg\exists R.\neg C$

Table 1.6.: Axioms which are valid in Classical but not in Intuitionistic Logics

1.2.4. Inference

Let us now look at the properties of *satisfiability* and *subsumption* in $c\mathcal{ALC}$. They are slightly different to the associated notions of \mathcal{ALC} . In addition, we have to consider the *fallible entities*, too.

To be able to cope with these in a simple tableau calculus, the ABox is not enough anymore. So, a *constraint system* is introduced.

Constraints

Constraints are used for a tableau algorithm which is explained in the next section. A constraint can be written as one of these expressions:

$$x : +C, \quad x : -C, \quad xRy, \quad x \preceq x', \quad x : -_R D$$

The letters x , x' and y represent *entities*, while C and D are used for *concept descriptions* and $R \in N_R$ for *role names* again (see [Sch]). The meanings of the different constraint types are explained in Table 1.7 (see [Sch]).

$x : +C$	For the entity x the concept C is <i>true</i> , i.e. $x \in C^{\mathcal{I}}$
$x : -C$	For the entity x the concept C is <i>false</i> , i.e. $x \notin C^{\mathcal{I}}$
xRy	There is a role relation R between the entities x and y
$x \preceq x'$	The entity x' refines the entity x
$x : -_R D$	For all constructible R - <i>successors</i> of the entity x , D is <i>false</i>

Table 1.7.: Constraint System for $c\mathcal{ALC}$

Right now, only the first two rows of the table are interesting. We will talk about the others later.

Now the question occurs, why we need a constraint like $x : -C$, because it might be enough to deal with negated concepts $\neg C$ instead.

Well, it is important to know, that $x : +(\neg C)$ does not express the same as $x : -C$. As we have seen in the definition of the semantics in Section 1.2.2, $(\neg C)^{\mathcal{I}}$ represents all entities which are not element of $C^{\mathcal{I}}$ including their refining entities. So $x : +(\neg C)$ means that the entity x is not element of $C^{\mathcal{I}}$, but all of x 's refinements are not element

of $C^{\mathcal{I}}$ either. On the other hand, $x : -C$ does not say anything about the refinements of x . We only know that $x \notin C^{\mathcal{I}}$ and that $x : +(-C)$ implies $x : -C$, but not vice versa (see [Sch]).

Constraint System

For the next steps, we first need to know what a *constraint system* is. [Sch] provides the following definition:

“A constraint system is a pair $S = (\mathcal{C}, \mathcal{A})$ where \mathcal{C} is a finite, non-empty set of constraints and the second component $\mathcal{A} \subseteq V$ [with V being an alphabet of variable symbols,] is a set of variables, called the *active set* of S , such that every element of \mathcal{A} occurs in at least one of the constraints from \mathcal{C} . The set of variables occurring in \mathcal{C} is called the *support* of S , written $Supp(S)$. Note that $\mathcal{A} \subseteq Supp(S)$ and $Supp(S)$ is not empty.”

Constraint Satisfiability and Subsumption

Let us now consider *satisfiability*⁹.

An entity x is said to *satisfy* a concept C in an interpretation \mathcal{I} , if and only if $x \in C^{\mathcal{I}}$. In [MS09], the validity relation \models is used to express the same thing: $\mathcal{I}; x \models C$ (see [MS09], p. 212).

Further, \mathcal{I} is said to be a model of C , written $\mathcal{I} \models C$, if and only if all possible entities $x \in \Delta^{\mathcal{I}}$ satisfy the concept C (see [MS09], p. 212).

[Sch] defines the constraint satisfiability thus:

“Let \mathcal{I} be an interpretation. An \mathcal{I} -assignment is a valuation function α mapping each variable symbol $x \in V$ to an element of $\Delta^{\mathcal{I}}$. We say that α

⁹General satisfiability, subsumption, disjointness and equivalence with respect to a given *TBox* for $c\mathcal{ALC}$ are not explained here because we will concentrate on proofs based on constraint systems. They are covered by [MS09].

satisfies a constraint w in \mathcal{I} , written $\mathcal{I}; \alpha \models w$, according to the following rules:

$$\begin{aligned} \mathcal{I}; \alpha \models x : +C & \text{ if } \alpha(x) \in C^{\mathcal{I}}, \\ \mathcal{I}; \alpha \models x : -C & \text{ if } \alpha(x) \notin C^{\mathcal{I}}, \\ \mathcal{I}; \alpha \models xRy & \text{ if } (\alpha(x), \alpha(y)) \in R^{\mathcal{I}}, \\ \mathcal{I}; \alpha \models x \preceq x' & \text{ if } (\alpha(x), \alpha(x')) \in \preceq^{\mathcal{I}} \\ \mathcal{I}; \alpha \models x : -_RD & \text{ if } \forall z \in \Delta^{\mathcal{I}}. (\alpha(x), z) \in R^{\mathcal{I}} \Rightarrow z \notin D^{\mathcal{I}}. \end{aligned}$$

A constraint system $S = (\mathcal{C}, \mathcal{A})$ is satisfied by an interpretation \mathcal{I} and an \mathcal{I} -assignment α if for all $w \in \mathcal{C}$ it holds that $\mathcal{I}; \alpha \models w$ and for all variables $x \in \mathcal{A}$ the assignment $\alpha(x)$ is infallible, i.e., $\alpha(x) \notin \perp^{\mathcal{I}}$. We call the pair (\mathcal{I}, α) a model of S . A constraint system S is satisfiable if it has a model.”

For a concrete concept C , this means that

“ C is satisfiable (w.r.t. the empty TBox) [if and only if] the constraint system $S = (\{x : +C, x : -\perp\}, \{x\})$ is satisfiable”

The subsumption is defined analogously:

“A concept C is subsumed by a concept D (w.r.t. the empty TBox) [if and only if] the constraint system $S = (\{x : +C, x : -D\}, \{x\})$ is not satisfiable”

This reminds us of the subsumption for \mathcal{ALC} , but now, we do not have $\neg D(x)$, instead, D is enveloped by a *negative* constraint. The corresponding proofs are provided by [Sch] and are not discussed here.

1.2.5. Tableau-Based Proofs for $c\mathcal{ALC}$

Now we consider a proving technique introduced by [Sch], that is based on constraint systems.

Rules

Similar to the tableau algorithm by [SSS91], this calculus comes with a set of *rules*. However, now we have two different sets: one handles *positive* constraints ($x : +C$)

while the other copes with *negative* ones ($x : -C$). In addition, we have rules for refinements, for $-_R$ -constraints, one special rule for \perp and rules for implication, because, as mentioned before, in $c\mathcal{ALC}$, the implication is a *concept-forming operator* ([MS09], p. 211). All rules are listed in Figure 1.6. Note that [Sch] uses the symbol \supset for the implication instead of \sqsubseteq .

To understand the actions and preconditions, we have to explain some terms (see [Sch]):

- As mentioned before, a *R-successor* y of an entity x is a role-filler y in our constraint system S . Such a relation between two entities x and y is expressed by the constraint xRy . So, y is a *R-successor* of x if $xRy \in \mathcal{C}$.
- x' is a \preceq^* -*successor* of x if $x \preceq x' \in \mathcal{C}$. Since the refinement-relation is reflexive, all entities refine themselves, too.
- If we want to exclude those self-refinements, we talk about \preceq^+ -*successors*. In other words: x' is a \preceq^+ -successor of x if and only if x' is a \preceq^* -successor of x and $x \neq x'$ ([Sch]).

Saturation and Clashes

“A constraint system $S = (\mathcal{C}, \mathcal{A})$ is called *saturated* if no completion rule is applicable to it. A saturated constraint system S^* with $S^* \supseteq S$ is called a *saturation* of S .”

[Sch]

This is similar to the *completeness* of the satisfiability algorithm for \mathcal{ALC} . By contrast, a *clash* detection is achieved differently.

“ S contains a *clash* if for some $a \in \text{Supp}(\mathcal{C})$ [with $\text{Supp}(\mathcal{C})$ being the set of all variables occurring in \mathcal{C}] and arbitrary concept description C one of the following conditions holds:

1. $a \in \mathcal{A}$ and $a : +C$ and $a : -C$ is in \mathcal{C} ;
2. $a \in \mathcal{A}$ and $a : +\perp$ is in \mathcal{C} ;
3. $a \in \mathcal{A}$ and $a : +\perp$ and $a : -C$ is in \mathcal{C} ;

- ($\rightarrow_{\neg-}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\neg-} S' = (\{x \preceq x', x' : +C\} \cup \mathcal{C}, \mathcal{A} \cup \{x'\})$
if for some $x \in \mathcal{A}$, $x : \neg C$ is in \mathcal{C} , x' is a new variable and there exists no \preceq^+ -successor x'' of x in S , with $x'' : +C \in \mathcal{C}$.
- ($\rightarrow_{\neg+}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\neg+} S' = (\{x : -C\} \cup \mathcal{C}, \mathcal{A})$
if for some $x \in \mathcal{A}$, $x : +\neg C$ is in \mathcal{C} and $x : -C$ is not in \mathcal{C} .
- ($\rightarrow_{\cap+}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\cap+} S' = (\{x : +C, x : +D\} \cup \mathcal{C}, \mathcal{A})$
if for some $x \in \mathcal{A}$, $x : +C \cap D$ is in \mathcal{C} and \mathcal{C} does not contain both $x : +C$ and $x : +D$.
- ($\rightarrow_{\cap-}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\cap-} S' = (\{x : E\} \cup \mathcal{C}, \mathcal{A})$; for $E = -C$ or $E = -D$
if for some $x \in \mathcal{A}$, $x : -C \cap D$ is in \mathcal{C} and neither $x : -C$ nor $x : -D$ is in \mathcal{C} .
- ($\rightarrow_{\sqcup+}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\sqcup+} S' = (\{x : E\} \cup \mathcal{C}, \mathcal{A})$; for $E = +C$ or $E = +D$
if for some $x \in \mathcal{A}$, $x : +C \sqcup D$ is in \mathcal{C} and neither $x : +C$ nor $x : +D$ is in \mathcal{C} .
- ($\rightarrow_{\sqcup-}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\sqcup-} S' = (\{x : -C, x : -D\} \cup \mathcal{C}, \mathcal{A})$
if for some $x \in \mathcal{A}$, $x : -C \sqcup D$ is in \mathcal{C} and $x : -C, x : -D$ are not both in \mathcal{C} .
- ($\rightarrow_{\supset+}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\supset+} S' = (\{x : E\} \cup \mathcal{C}, \mathcal{A})$; for $E = -C$ or $E = +D$
if for some $x \in \mathcal{A}$, $x : +C \supset D$ is in \mathcal{C} , and neither $x : -C$ nor $x : +D$ is in \mathcal{C} .
- ($\rightarrow_{\supset-}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\supset-} S' = (\{x \preceq x', x' : +C, x' : -D\} \cup \mathcal{C}, \mathcal{A} \cup \{x'\})$
if for some $x \in \mathcal{A}$, $x : -C \supset D$ is in \mathcal{C} , x' is a new variable and there exists no \preceq^* -successor x'' of x in S , with $x'' : +C, x'' : -D$ in \mathcal{C} .
- ($\rightarrow_{\forall+}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\forall+} S' = (\{y : +C\} \cup \mathcal{C}, \mathcal{A})$
if for some $x \in \mathcal{A}$, $x : +\forall R.C$ is in \mathcal{C} and there exists a R -successor y of x in S with $y \in \mathcal{A}$ and $y : +C$ is not in \mathcal{C} .
- ($\rightarrow_{\forall-}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\forall-} S' = (\{x \preceq x', x' R y, y : -D\} \cup \mathcal{C}, \mathcal{A} \cup \{x', y\})$
if $x : -\forall R.D$ is in \mathcal{C} , x', y are new variables and there exists no \preceq^* -successor x'' of x and R -successor y'' of x'' in S , with $y'' : -D$ in \mathcal{C} .
- ($\rightarrow_{\exists+}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\exists+} S' = (\{x R y, y : +C\} \cup \mathcal{C}, \mathcal{A})$
if for some $x \in \mathcal{A}$, $x : +\exists R.C$ is in \mathcal{C} , y is a new variable and there is no R -successor z of x in S such that $z : +C$ is in \mathcal{C} .
- ($\rightarrow_{\exists-}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\exists-} S' = (\{x \preceq x', x' : -R D\} \cup \mathcal{C}, \mathcal{A} \cup \{x'\})$
if for some $x \in \mathcal{A}$, $x : -\exists R.D$ is in \mathcal{C} , x' is a new variable and there exists no \preceq^* -successor x'' of x in S such that $x'' : -R D$ is in \mathcal{C} .
- (\rightarrow_{R-}) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{R-} S' = (\{y : -D\} \cup \mathcal{C}, \mathcal{A} \cup \{y\})$
if for some $x \in \mathcal{A}$, $x : -R D$ is in \mathcal{C} , there exists a R -successor y of x in S such that $y : -D$ is not in \mathcal{C} .
- ($\rightarrow_{\preceq+}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\preceq+} S' = (\{x' : +C\} \cup \mathcal{C}, \mathcal{A})$
if for some x , $x : +C$ is in \mathcal{C} and x' is a \preceq^+ -successor of x in S and $x' : +C$ is not in \mathcal{C} .
- ($\rightarrow_{R\perp+}$) $S = (\mathcal{C}, \mathcal{A}) \rightarrow_{R\perp+} S' = (\{y : +\perp\} \cup \mathcal{C}, \mathcal{A})$
if for some x , $x : +\perp$ is in \mathcal{C} and y is a R -successor of x in S and $y : +\perp$ is not in \mathcal{C} .

Figure 1.6.: Completion Rules of the Constraint Calculus for $c\mathcal{ALC}$ ([Sch])

If S contains no clash it is called *clash-free*. [...] If a constraint system contains a clash, then it is not satisfiable.”

[Sch]

The other way round, this means that a constraint system is *satisfiable* if it is saturated and clash-free.

So, we now have to check if there are any *positive* \perp s or if we have the same concept description C in a positive and a negative constraint regarding the same *active* entity.

[Sch] provides proofs for satisfiability and subsumption and also for the correctness of the algorithm. We will not consider these here because this would go beyond the scope of this work.

Running the Algorithm

We start the algorithm with a new constraint system $S_0 = (\mathcal{C}_0, \mathcal{A}_0)$. We first introduce an initial abstract entity x that we put into the (previously empty) *active set*, so $\mathcal{A}_0 = \{x\}$.

\mathcal{C}_0 contains our initial constraints. If we want to check satisfiability for a concept description C , we add the constraints $x : +C$ and $x : -\perp$ to \mathcal{C} and begin to apply rules. As soon as S is satisfied and clash-free, C is proved to be satisfiable (see [Sch]).

For proving a subsumption $C \sqsubseteq D$, we add the two constraints $x : +C$ and $x : -D$ to an empty \mathcal{C} and try to lead S to a *clashing system*, i.e. a system containing clashes for each alternative (see [Sch]).

Example

It is time to see the algorithm in action: Let us consider IK2 in Description Logic.

$$\forall R.(A \sqsubseteq B) \sqsubseteq (\exists R.A \sqsubseteq \exists R.B)$$

We want to show that this axiom is always true in $c\mathcal{ALC}$. For this, we begin with generating an initial constraint system $S_0 = (\mathcal{C}_0, \mathcal{A}_0)$ with

$$\mathcal{C}_0 = \{x : +\forall R.(A \sqsubseteq B), x : -\exists R.A \sqsubseteq \exists R.B\}$$

and $\mathcal{A}_0 = \{x\}$. This first state is illustrated in Figure 1.7 (on page 44). Entities that are elements of the active set are represented by yellow rectangles. On the left side, we have the *positive* constraints ($x : +C$) and on the right side the *negative* ones ($x : -C$) and the $-_R$ -constraints ($x : -_R D$).

Now, on the left side, we have a value restriction. However, we cannot apply $(\rightarrow_{\forall+})$ because we do not have an active R -successor. On the right side we have a subsumption. The conditions for $(\rightarrow_{\supset-})$ hold, so let us apply it. We get $S_1 = (\mathcal{C}_1, \mathcal{A})$ with

$$\mathcal{C}_1 = \mathcal{C}_0 \cup \{x \preceq x', x' : +\exists R.A, x' : -\exists R.B\} \quad \mathcal{A}_1 = \mathcal{A}_0 \cup \{x'\}$$

So, we now have a \preceq^+ -successor x' of x . As we know, if an expression is true for an arbitrary entity x , then it is also true for all of its refinements, i.e. all of x 's \preceq^* -successors. For our example, this means that $\forall R.(A \sqsubseteq B)$ must be true for x' either. The rule $(\rightarrow_{\preceq+})$ moves a *positive* constraint of an entity to a direct \preceq^+ -successor. So let us apply it to $x : +\forall R.(A \sqsubseteq B)$. We get $S_2 = (\mathcal{C}_2, \mathcal{A}_2)$ with

$$\mathcal{C}_2 = \mathcal{C}_1 \cup \{x' : \forall R.(A \sqsubseteq B)\} \quad \mathcal{A}_2 = \mathcal{A}_1$$

The new state is illustrated in Figure 1.8. Note that refinement relations between two entities are displayed by dotted arrows.

Now we have two possibilities. Either we apply $(\rightarrow_{\exists+})$ to $x' : +\exists R.A$ or $(\rightarrow_{\exists-})$ to $x : -\exists R.B$. Let us try the first one. We generate a R -successor y of x' and obtain $S_3 = (\mathcal{C}_3, \mathcal{A}_3)$ with

$$\mathcal{C}_3 = \mathcal{C}_2 \cup \{x' R y, y : +A\} \quad \mathcal{A}_3 = \mathcal{A}_2$$

This might become a problem: y is not element of the active set. To be more concrete, this means that we now have a R -successor of x' , so at first sight it might be possible to apply $(\rightarrow_{\forall+})$ to $x' : +\forall(A \sqsubseteq B)$, but another condition is that the R -successor must be an element of the active set. So this is a dead-end.

Now let us apply $(\rightarrow_{\exists-})$ to $x' : -\exists R.B$. We then have $S_4 = (\mathcal{C}_4, \mathcal{A}_4)$ with

$$\mathcal{C}_4 = \mathcal{C}_3 \cup \{x' \preceq x'', \quad x'' : -_R B\} \quad \mathcal{A}_4 = \mathcal{A}_3 \cup \{x''\}$$

So, as we have the new \preceq^+ -successor x'' of x' , we can move the positive constraints from x' to x'' by applying rule $(\rightarrow_{\preceq+})$ twice. We then have $S_5 = (\mathcal{C}_5, \mathcal{A}_5)$ with

$$\mathcal{C}_5 = \mathcal{C}_4 \cup \{x'' : +\exists R.A, \quad x'' : +\forall R.(A \sqsubseteq B)\} \quad \mathcal{A}_5 = \mathcal{A}_4$$

Now, the only possibility we have is to apply rule $(\rightarrow_{\exists+})$ to $x'' : +\exists R.A$. This will generate an inactive R -successor again. Let us call it z . Our new constraint system $S_6 = (\mathcal{C}_6, \mathcal{A}_6)$ is illustrated in Figure 1.9. Role-relations are displayed by solid arrows, while entities that are not listed in \mathcal{A} are denoted by grey rectangles. We have

$$\mathcal{C}_6 = \mathcal{C}_5 \cup \{x'' R z, \quad z : +A\} \quad \mathcal{A}_6 = \mathcal{A}_5$$

But now, as we have a new R -successor of x'' , we can apply (\rightarrow_{R-}) . And this rule has the side-effect that it *activates* the R -successor. We obtain $S_7 = (\mathcal{C}_7, \mathcal{A}_7)$ with

$$\mathcal{C}_7 = \mathcal{C}_6 \cup \{z : -B\} \quad \mathcal{A}_7 = \mathcal{A}_6 \cup \{z\}$$

As z is now a member of the active set, we can finally apply $(\rightarrow_{\forall+})$ in x'' . We get $S_8 = (\mathcal{C}_8, \mathcal{A}_8)$ with

$$\mathcal{C}_8 = \mathcal{C}_7 \cup \{z : +A \sqsubseteq B\} \quad \mathcal{A}_8 = \mathcal{A}_7$$

The only possible way to continue is to apply $(\rightarrow_{\supset+})$ to $z : +A \sqsubseteq B$. The result contains two different constraint systems. Let us call them S_9 and S'_9 with $S_9 = (\mathcal{C}_9, \mathcal{A}_9)$ and $S'_9 = (\mathcal{C}'_9, \mathcal{A}'_9)$ where

$$\mathcal{C}_9 = \mathcal{C}_8 \cup \{z : -A\} \quad \mathcal{C}'_9 = \mathcal{C}_8 \cup \{z : +B\} \quad \mathcal{A}_9 = \mathcal{A}'_9 = \mathcal{A}_8$$

Both constraint systems, S_9 and S'_9 , are *saturated*, i.e. we are not able to apply any more rules. Entity z is illustrated in Figure 1.10 in both alternatives and we see that they both contain clashes (underlined constraints). Therefore, shown by refutation, we know that $\forall R.(A \sqsubseteq B) \sqsubseteq (\exists R.A \sqsubseteq \exists R.B)$ is valid in $c\mathcal{ALC}$.

$$x : +\forall R.(A \sqsubseteq B) \quad \boxed{x} \quad x : -\exists R.A \sqsubseteq \exists R.B$$

Figure 1.7.: The First Step of Proving IK2 in $c\mathcal{ALC}$

$$\begin{array}{ccc} x : +\forall R.(A \sqsubseteq B) & \boxed{x} & x : -\exists R.A \sqsubseteq \exists R.B \\ \vdots & \downarrow & \\ x' : +\exists R.A & x : +\forall R.(A \sqsubseteq B) & \boxed{x'} & x' : -\exists R.B \end{array}$$

Figure 1.8.: A $c\mathcal{ALC}$ -Tableau after Applying two Rules

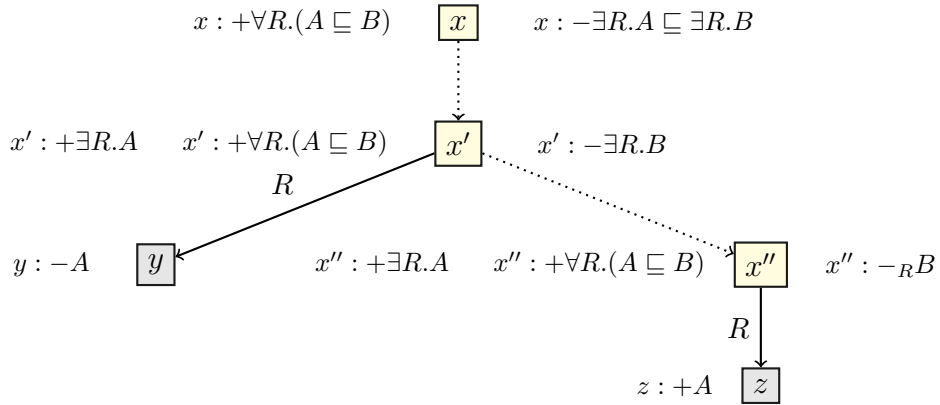


Figure 1.9.: A $c\mathcal{ALC}$ -Tableau with two Inactive Entities

$$\begin{array}{ccccccc} \underline{z : +A} & z : +A \sqsubseteq B & \boxed{z} & z : -B & \underline{z : -A} & & \\ & & & & & & \\ \underline{z : +B} & z : +A & z : +A \sqsubseteq B & \boxed{z} & \underline{z : -B} & & \end{array}$$

Figure 1.10.: The same Entity with different Constraints

1.2.6. Summary

Here are again the most important issues about the constructive Description Logic $c\mathcal{ALC}$:

- Constructiveness is needed when dealing with incomplete or dynamic knowledge.
- Constructive logics are similar to *intuitionistic logics* but even more restrictive.
- The constructive Description Logic $c\mathcal{ALC}$ has a modal counterpart, CK.
- We use *refinement*-relations that are transitive and reflexive to express an increase of information from one entity to another one in $c\mathcal{ALC}$.
- The tableau-algorithm for $c\mathcal{ALC}$ introduced by [Sch] performs inferences on ABoxes making use of *constraint systems*.

2

A GAME-THEORETIC DECISION PROCEDURE

Let us now deal with *dialogical games*.

Dialogical Logic as we consider it here has been introduced by Paul Lorenzen and Kuno Lorenz as a method to decide about the validity of logical formulæ, especially in intuitionistic logic. The idea behind this procedure is that two *players* battle against each other in a logical way (see [RK05], p. 360; [Kei09]).

In this chapter, we first look at how dialogical logic works for first-order logics. We will then extend the underlying rules step by step until we have a decision procedure for $cALC$ that can be used as an alternative to tableau-based reasoning.

2.1. Introduction to Dialogical Games for First-Order Logics

This section gives a short introduction to dialogical logic. Let us move away from Description Logic for a while: after looking at the general idea behind dialogical logic, we will consider intuitionistic and classical first-order logic in that dialogical way.

2.1.1. Why and How?

Why Dialogues?

Dialogical Logic is related to argumentations between two parties. Several kinds of logics can be transformed to dialogical rules in order to specify their semantics. These rules vary from logic to logic. New logics have also been invented by altering rules. For example, classical logic uses slightly different rules as intuitionistic logic. Eventually, a way is given to check validity of logical formulæ, in other words: we can use dialogues to prove the truth (or the falsity) of an expression (see [RK05], p. 362; [Kei09]).

There are several fields of application where Dialogical Logic can be used. Regarding the set of rules, one could reveal possible relations to other logics. Another application can be found in the area of artificial intelligence, for example when trying to design multi-agent systems where both logic and games theories are applied (see [RK05], p. 362).

Principle

When we consider an ordered dialogue, we probably think about a sequence of arguments. First, an agent (maybe a person) P (let us call him Peter) makes an *assertion*. Talking to his friend O (Olga), who is not in line with him, leads to an *attack* against Peter's assertion. O gives a new argument, so P now has the choice: either he *counterattacks* Olga's argument or he *defends* his own assertion (see [RK05], p. 364). In this way, both can go on until they realize that one of them is not right. Finally, they might come to the result that Peter's assertion was either right or wrong.

In Dialogical Logic, the two parties can *attack* or *defend* a given formula, depending on the underlying argumentation rules. In any case, the parties are not allowed to argue in a jumbled way. The discussion flow is also bound to special rules. We will consider these later.

Language and Expressions

For now, we talk about classical and intuitionistic *first-order logic*. We use a language L_{FOL} based on this. An *expression* of this language can be (see definition given by [Kei09]):

- any first-order formula
(with the connectives \wedge , \vee , \rightarrow , \neg and the quantifiers \forall and \exists)
- the symbols L , R , \vee , $\forall x/c$, $\exists x$
where x represents a variable and c a constant of our language

A *dialogically signed expression* has the form $\langle X, f, e \rangle$ with

- X being a *label* O or P representing one of the two players
- f being a *force symbol* $?$ or $!$
- e being an *expression*, e.g. $\forall x(A(x) \vee \neg A(x))$

Regarding the players, the label P is used for *Proponent*, while O means *Opponent*. The force symbol $?$ depicts an *attack* of an disputant's argument and $!$ is used to indicate a *defence*.

The variables X and Y , representing the players, are elements of the set $\{O, P\}$; in addition, because we talk about two different *agents* battling each other, we assume that $X \neq Y$ (see [RK05], p. 363; [Kei09]).

So eventually, the signed expression $\langle X, f, e \rangle$ indicates that the player X performs the action f (which is $?$ or $!$) with a given expression e . From now on, we will omit the brackets and write X - f - e instead (see [Kei09]).

2.1.2. Rules

There are two different sets of rules with different functions. On the one hand we have *particle rules* which define how a given formula can be *attacked* and how to *defend* a formula against a specific attack. On the other, there are *structural rules* which handle the “*organisation*” of a game (see [RK05], p. 364, 367). We will now look at these two kinds of rules in detail.

Particle Rules

Particle rules define which attacking and defending actions can be performed depending on the primary connective or quantification of the corresponding formula. They are said to “state the local semantics”, because they “show how the game runs locally” ([RK05], p. 364), i.e. particle rules are applied independently from the rest of the dialogue (see [Kei09]).

Table 2.1 gives an overview of the different rules. The row *Assert* indicates the structure of a given signed expression, where the letters A and B represent arbitrary first-order formulæ. The second row then shows the possible attack(s) and the last how to defend against such an attack.

	\wedge	\vee	\neg
Assert	$X\text{-!-}A \wedge B$	$X\text{-!-}A \vee B$	$X\text{-!-}\neg A$
Attack	$Y\text{-?-}L$ or $Y\text{-?-}R$	$Y\text{-?-}\vee$	$Y\text{-!-}A$
Defend	$X\text{-!-}A$ resp. $X\text{-!-}B$	$X\text{-!-}A$ or $X\text{-!-}B$	—

	\rightarrow	\forall	\exists
Assert	$X\text{-!-}A \rightarrow B$	$X\text{-!-}\forall x A$	$X\text{-!-}\exists x A$
Attack	$Y\text{-!-}A$	$Y\text{-?-}\forall x/c^\dagger$	$Y\text{-?-}\exists$
Defend	$X\text{-!-}B$	$X\text{-!-}A[x/c]$	$X\text{-!-}A[x/c]^\ddagger$

Table 2.1.: Particle Rules for FOL-Semantics (see [Kei09])

When attacking the formula $A \wedge B$, Y can choose if he challenges A (i.e. the left side L of the conjunction) or B (the right side R), because before, player X asserted that both A and B are true. By contrast, with the expression $A \vee B$, X asserts that either A or B (or both) is true. So if Y attacks this formula, then X may choose if he wants to show that A is true or if he prefers to show that B is true.

Handling the quantifiers is quite similar. If player X asserts a formula $\forall x A$, then Y can choose an arbitrary constant c and demand X to show that his assertion is true for x being replaced by Y 's c . The other way round, the expression $\exists x A$ indicates that there must be *at least one* x , for which A is true, so if Y attacks this, then X may

[†]for any c that Y chooses

[‡]for any c that X chooses

choose the constant c for which he wants to show that A is true (see [RK05], p. 365; [Kei09]).

Here are two examples:

Assertion	$P-!\forall x \text{ sunShines}(x)$	<i>The sun always shines.</i>
Attack	$O-?\forall x/\text{today}$	<i>Show me for today!</i>
Defence	$P-!\text{-sunShines}(\text{today})$	<i>The sun shines today.</i>
Assertion	$P-!\exists x \text{ sunShines}(x)$	<i>There is at least one day, when the sun shines.</i>
Attack	$O-?\exists$	<i>Show me!</i>
Defence	$P-!\text{-sunShines}(\text{today})$	<i>The sun shines today.</i>

In both examples, our constant is the term *today*. In the first, O chooses the day, later P does.

Y attacks negated formulæ by asserting the contrary (i.e. omitting \neg). This is the only case where no defence is possible. However, with this attack, Y provides a new assertion which might be *counterattacked* by X (see [Kei09]):

Assertion	$P-!\neg(\text{rain}(\text{today}) \wedge \text{rain}(\text{yesterday}))$	<i>It is not true that it rains today and it rained yesterday</i>
Attack	$O-!\text{-rain}(\text{today}) \wedge \text{rain}(\text{yesterday})$	<i>You are wrong! It is true!</i>
Attack	$P-?-L$	<i>Show me that it rains today!</i>

Now the implication: the assertion $A \rightarrow B$ expresses that if A is true, then B is also true. This expression is correct anyway if A is not true, so Y attacks by asserting A in order to claim that X is not able to construct a proof showing that B is also true (see [Kei09]).

An attack and the corresponding defence together form a pair that we call a *round*. An attack *opens* a round whereas a defence *closes* it (see [Kei09]).

Note that an atomic formula which is not decomposable (e.g. *rain(today)*) cannot be attacked. We call such expressions *prime formulæ*.

Structural Rules

Structural rules provide a set of guidelines about the “general organisation of the game” ([RK05], p. 367) or the “global semantics” ([Kei09]). With other rules we follow other aims, for example (as mentioned before), intuitionistic logic uses slightly different rules than classical logic. Here, we will concentrate on rules which make it possible to check *validity* of logical formulæ.

In this case, we assume that the first player is P whose initial assertion is called *thesis*, while all following actions are named *moves* (‘P moves’ are performed by the proponent, while ‘O moves’ are done by the opponent). P now tries to prove this first statement, whereas O tries to refute it. O starts by attacking P’s thesis (first move). If P finds a way to *win* the game, no matter how O performs his moves (e.g. which constants he chooses in attacks), the thesis is correct ([Kei09]).

A *dialogical game* is defined to be a sequence of dialogical signed expressions *X-f-e*. A *dialogue* is a set of such dialogical games. It has the structure of a tree with the root containing several premises and P’s thesis. Every decision about the next move which is made by O in the game, generates a new branch of the tree. For now, P’s moves will not create any new branches (see [Kei09]).

A new branch is generated in these cases (see [Kei09]):

- O attacks a conjunction (selecting *L* or *R*)

Assertion	P-!-A ∧ B	
Attack	O-?-L	or
		O-?-R

- O defends a disjunction

Assertion	O-!-A ∨ B	
Attack	P-?-∨	
Defence	O-!-A	or
		O-!-B

- O has the choice of either defending an assertion against an attack performed by P or counterattacking P’s attack:

Assertion	O-!-(A ∨ B) → C
Attack	P-!-A ∨ B

Defence or Attack | O-!-C or O-?-V

A *strategy* can be understood as a function of commands telling a player what to do in the next step depending on all moves which have been done in the game. If P has a strategy which makes him win all games of a dialogue, then his strategy is a *winning strategy* (see [Kei09]).

So let us now look at the different structural rules. We concentrate on the rules which make the dialogues test *validity*. There are also other possibilities, for example regarding *persuasion* instead of validity (see [RK05], p.371). To keep the rules correct and simple, only the following “*asymmetric rules*” are presented where we assume that “O makes the best possible move” ([RK05], p. 372).

The first rule only makes clear, inter alia, that P introduces the thesis and that a P move is followed by an O move and vice versa.

(SR-ST0) (starting rule): “Expressions are numbered and alternately uttered by P and O. The thesis is uttered by P. All even numbered expressions including the thesis are P-labelled, all odd numbered expressions are O moves. Every move below the thesis is a reaction to an earlier move with another player label and performed according to the particle and the other structural rules.”

[RK05], p. 372

The next rule gives definitions for the terms *closed*, *open* and *finished* and specifies under which conditions a player wins the dialogue. These definitions evoke the presented tableaux for the description logics \mathcal{ALC} and $c\mathcal{ALC}$.

(SR-ST1)¹ (winning rule): “A [dialogical game] is closed [if and only if] it contains two copies of the same prime formula, one stated by X and the other one by Y , and neither of these copies occurs within the brackets ‘<’ and ‘>’ (where any expression which has been bracketed between these signs in a [dialogical game] either cannot be counterattacked in this [game], or it has been chosen in this [game] not to be counterattacked). Otherwise it is open. The player who stated the thesis wins the [game] [if and only if] the [dialogical game] is closed. A [game] is finished if it is closed or if no other move is allowed by the (other) structural and particle rules of the game. The player who started the dialogue as a challenger wins if the [dialogical game] is finished and open.”

[RK05], p. 372

¹In [RK05] the term “dialogical game” is not used, whereas it is in [Kei09], so some rules have been slightly adjusted. Note that we sometimes write “game” instead of “dialogical game” what means the same.

Well, the formulation for this rule is quite confusing because, according to the particle rules, a prime formula can never be *attacked*. So, it is not clear in which cases the brackets ‘<’ and ‘>’ appear. As prime formulæ cannot be counterattacked anyway, it seems that they always has to be bracketed, what does not make much sense.

Omitting the part with those brackets does not improve it. It would be possible to prove formulæ like $(A \rightarrow B) \rightarrow (A \rightarrow C)$, but of course, this is nonsense (we will consider the dialogue for this example at the end of Section 2.1.3).

So let us reformulate that rule and replace the old one by this:

(SR-ST1) (*altered winning rule*): A dialogical game is closed if and only if it contains two copies of the same prime formula, one stated by X and the other one by Y . Otherwise it is open. A dialogical game is finished if no other move is allowed by the (other) structural and particle rules of the game. The player who stated the thesis wins the game if and only if it is closed and finished and the last move of the game has been performed by this player. The player who started the dialogue as a challenger wins if the game is finished and if he/she has performed the last move of the game.

So far, the rules are applicable for classical and intuitionistic logic. Next, we have to introduce two different rules depending on the logic we use. Here we define when we are allowed to attack and defend (independently of the particle rules). Regarding defence, the intuitionistic version is much stricter than the classical one. In later examples, we will see why.

(SR-ST2I) (*intuitionist ROUND closing rule*): “In any move, each player may attack a (complex) formula asserted by his partner or he may defend himself against the last not already defended attack. Defences may be postponed as long as attacks can be performed. Only the latest open attack may be answered: if it is X ’s turn at position n and there are two open attacks m, l such that $m < l < n$, then X may not at position n defend himself against m .”

(SR-ST2C) (*classical ROUND closing rule*): “In any move, each player may attack a (complex) formula asserted by his partner or he may defend himself against *any* attack (including those which have already been defended).”

[RK05], p. 372

The next rule states that only O may create new branches in the dialogue and under which conditions this happens.

(SR-ST3) (*strategy branching rule*): “At every propositional choice (i.e., when O defends a disjunction, reacts to the attack against a conditional or attacks a conjunction),

O will motivate the generation of two [dialogical games] differentiated only by the expressions produced by the choice. O will move into a second [dialogical game] [if and only if] he loses the first chosen one. No other move will generate new [games].”

[RK05], p. 372

Restrictions on prime formulæ: P is not allowed to introduce them, because O would not be able to attack them, so P would win too easily.

(SR-ST4) (*formal use of prime formulæ*): “P cannot introduce prime formulæ: any prime formula must be stated by O first. Prime formulæ can not be attacked.”

[RK05], p. 373

Last but not least, there is a rule for repetitions. If repeating a particle rule on a specific expression was allowed unrestrictedly, the dialogue would probably never end (see [RK05], p. 369). Again, we have an intuitionist and a classical version:

(SR-ST5C) (*classical no delaying tactics rule*): “P may perform once a new defence (attack) of an existential (universal) quantifier using a different constant (but not new) [if and only if] the first defence (attack) compelled P to introduce a new constant. No other repetitions are allowed.”

(SR-ST5I) (*intuitionist no delaying tactics rule*): “P may perform a repetition of an attack if and only if O has introduced a new prime formula which can now be used by P.”

(rule split; original rule (SR-ST5) from [RK05], p. 373)

Eventually, the complete dialogue has the form of a tableau. With the definitions provided by the rules, we can use dialogues to check the validity of a logical formula given by the proponent. So, if the dialogue is started by him with his thesis to be proved, this thesis is *valid* if the resulting dialogue/tableau is closed (see [RK05], p. 373).

2.1.3. Some Examples

To make it clearer how dialogical logics work, some examples are provided here.

In these examples the dialogues are presented in tables with the opponent and the proponent having their own columns. The right sub-column, where each belongs to a player, shows an assertion, i.e. a thesis, an attack or a defence. The left sub-column

indicates if the corresponding move is an attack or a defence and which assertion it refers to. For example ‘?1’ denotes that the expression in the right sub-column is an attack referring to the rival’s assertion in row 1, whereas ‘!2’ means that in this move, the player defends his own assertion against the rival’s attack in row 2.

The Excluded Third

As already explained in Section 1.2.1, the formula $A \vee \neg A$ is valid in classical but not in intuitionistic logic. We will see both variants here² but let us begin with the classical one:

	O		P	
1				P - ! - $A \vee \neg A$
2	?1	O - ? - \vee	!2	P - ! - $\neg A$
3	?2	O - ! - A	!3	P - ! - A

Table 2.2.: The Excluded Third in a Classical Dialogue

1. P starts with his assertion which shall be proved to be true. Then it is O’s turn.
2. Everything she can do is to attack the thesis, i.e. attack the \vee . One might think that P may choose if he defends the thesis with A or $\neg A$. But that is not really true, because according to rule **SR-ST4** it is not allowed for the proponent to introduce prime formulæ unless they have been stated by the opponent before. So P may only defend with $\neg A$.
3. O attacks the negation. This is her ‘mistake’, because now O has stated A which is a prime formula. P may defend the thesis again because the rule **SR-ST2C** allows this.

The same prime formula A has now been stated by both the proponent and the opponent. According to the winning rule (**SR-ST1**), the dialogical game is *closed* and P is the winner. As O is not able to backtrack (see next example), this makes the assertion to be valid in classical logic.

As we know, the statement $A \vee \neg A$ is not valid in intuitionistic logic. This is the game:

²This example with its explanations is adapted from [RK05], p. 370, 371.

	O		P	
1				P - ! - $A \vee \neg A$
2	?1	O - ? - \vee	!2	P - ! - $\neg A$
3	?2	O - ! - A	—	

Table 2.3.: The Excluded Third in an Intuitionist Dialogue

Instead of rule **SR-ST2C** we now have to obey **SR-ST2I**. It is not allowed to defend the same formula twice. Generally, it is only possible to defend “*the last not already defended* attack”. No further moves are possible, so the dialogue is *finished* and *open* and therefore O wins the game (**SR-ST1**). The formula is not valid in intuitionistic logic.

Backtracking

This example³ shows what happens if O loses a game but is able to retry.

	O		P	
1				P - ! - $(A \vee B) \rightarrow A$
2	?1	O - ! - $A \vee B$?2	P - ? - \vee
3	!2	O - ! - A	!2	P - ! - A
3'	!2	O - ! - B	—	

Table 2.4.: The Opponent Tracks Back

This is the explanation:

1. The proponent announces his thesis $(A \vee B) \rightarrow A$.
2. O attacks the thesis by claiming that $A \vee B$ is true. P cannot defend the thesis because A is a *prime formula* which has not been stated by O yet (**SR-ST4**), so P counterattacks by doubting O’s assertion.
3. O defends her assertion of row 2. She selects the left side A . P is now allowed to defend the thesis and claims A . So the game is closed and therefore O seems to have lost, but ...

³adapted from [Kei09]

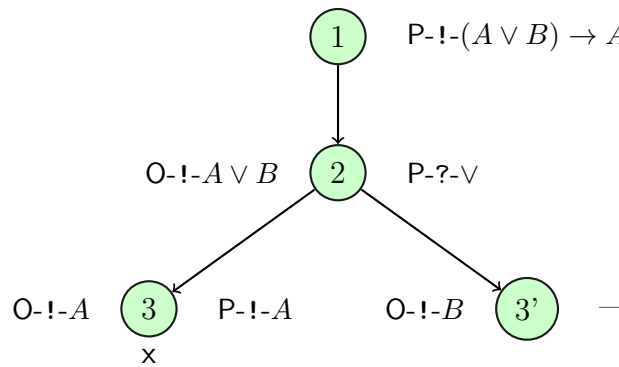


Figure 2.1.: Illustration of a Dialogue

3'. ... because O has defended a disjunction, a second *dialogical game* has been generated (indicated by the prime (') after the row number) and O is now allowed to move to this (**SR-ST3**). That is why she may defend her assertion of row 2 once more. P is not able to counterattack this and loses the game.

So because the complete dialogue could not be closed but only one of the branches, the formula $(A \vee B) \rightarrow A$ is not valid. The structure of the dialogue tree is illustrated in Figure 2.1. Each line in the dialogue-table is represented by a node. O moves are written to the left of these nodes and P moves to the right. The x marks the branched game which is closed.

Closed and Not Finished

This last examples shows that it is not enough for P to win a game if it is just closed. As stated before, the formula $(A \rightarrow B) \rightarrow (A \rightarrow C)$ is not valid (if you do not believe this, you can check this for example with a calculus for logical proofs like the *Fitch Calculus*⁴).

	O		P	
1				P - ! - $(A \rightarrow B) \rightarrow (A \rightarrow C)$
2	?1	O - ! - $A \rightarrow B$!2	P - ! - $A \rightarrow C$
3	?2	O - ! - A	?2	P - ! - A

Table 2.5.: Closed but Unfinished Dialogue

Here, P closes the game with a counterattack against $A \rightarrow B$, so the prime formula A is stated by both P and O . However, the game is not won by P because O may still answer with a defence and after that, P is not able to move further:

⁴The Fitch Calculus is described and explained in [BE99].

	O		P	
1				$P - ! - (A \rightarrow B) \rightarrow (A \rightarrow C)$
2	?1	$O - ! - A \rightarrow B$!2	$P - ! - A \rightarrow C$
3	?2	$O - ! - A$?2	$P - ! - A$
4	!3	$O - ! - B$	—	

2.1.4. Summary

This was a short introduction to Dialogical Logic. Here are the most important issues again:

- Dialogical Logic carries semantics in its rules.
- It can be used for different logics. The set of rules merely has to be modified.
- The proponent tries to verify a formula. The opponent tries to refute it.
- Validity of formulæ can be checked with the proponent introducing them as thesis and winning the games.
- Dialogue-based proofs have similarities to tableau-based proofs.

2.2. Dialogues for the Description Logic \mathcal{ALC}

It is now time to extend our game semantics so that we are able to perform game-theoretic proofs for Description Logics. For now, we restrict our view to the language \mathcal{ALC} . Other extensions might also be possible but are not important here, because they might not be helpful for our aim which is to find a procedure for $c\mathcal{ALC}$.

All explanations are referred to dialogues for *modal logic*. Because of the close relationship to \mathcal{ALC} and because this work is about Description Logic, these extended dialogical proofs are presented directly for \mathcal{ALC} by using the corresponding syntax and terminology.

2.2.1. Aims and Problems

We are looking for a method to create dialogues for \mathcal{ALC} . In Description Logics, truth depends on individuals we are considering at that moment. This is similar to truth in modal logics, where it depends on a given *context* or *world*.

Let us again consider the assertion based on Example 1.1 from page 4:

$$Crew_Member \sqcap \neg Captain$$

It is a concept description for inferiors and therefore describes a set of individuals who are crew members but no captains. For these individuals, the expression $\exists comands.Inferior$ is obviously wrong, while it should be true for all members of the concept *Captain*. That is why truth might be considered to be dependent from the individual we are looking at at the moment we make an assertion.

The tableau-based algorithm by [SSS91] we discussed in Section 1.1.5 refers in each step to one or several abstract individuals (e.g. x, y, \dots) of a prototypical ABox \mathcal{A} . It is suggested to do something similar in the dialogues. We start our proofs for the abstract individual x . After some steps, new (also abstract) individuals are introduced, as it has been done in the tableau-based proof.

Our aim is to extend the structural rules for first-order logics of Section 2.1 by adding new rules. The original rules shall remain as they are as far as possible, because the system should also work with *propositional logics*, considering them as special cases of Description Logic, i.e. we then work only with respect to one single individual x and do not generate new ones (see [RR98], p. 6).

[RR98] have introduced such a dialogical system for different modal logics. As we use the term *individual*, they talk about ‘Dialogkontexte’ (*dialogue contexts*).

2.2.2. Notation

In order to distinguish different abstract individuals, we could use variables as before, but instead, we now prefer numbers, because it is easier to calculate with them. Our starting individual receives the number 1 (see [RR98], p. 6). All individuals generated in 1 then have a higher number, of course.

Let us define our new language $\mathbf{L}_{\mathcal{ALC}}$. There are similarities to $\mathbf{L}_{\mathcal{FOL}}$. An *expression* of this new language can be:

- any concept description
 $A, C \sqcap D, C \sqcup D, C \sqsubseteq D, \neg C, \forall r.C$ or $\exists r.C$
with A being an atomic concept, C and D being arbitrary concept descriptions (including atomic concepts) and r being an arbitrary role (from now on, we use r for roles instead of R which is still the counterpart of L)
- the symbols $L, R, \sqcup, \forall r, \exists r$
where r represents a role.

For now, we write $A \sqcap \neg A$ for \perp and $\neg(A \sqcap \neg A)$ for \top , so we do not need these symbols in our language. Of course, when dealing with classical description logic, we can also write $A \sqcup \neg A$ for \top .

A *dialogically signed expression* has the form $\langle X, f, e, i \rangle$ with

- X being a *label* O or P representing one of the two players
- f being a *force symbol* $?$ or $!$
- e being an *expression*, e.g. $\forall r.(C \sqcup \neg C)$
- i being an *individual*

Of course $\forall r$ and $\exists r$ are different from $\forall x$ and $\exists x$ of first-order logic. However, we do not need the first-order versions in Description Logic, so we can replace them. But as we will see later, there are also similarities between the descriptive and the first-order version of \forall and \exists .

When using abstract individuals, we let i be a number such that $i \in \mathbb{Z}^{+5}$. If we are applying a certain ABox, it might be reasonable to let i be a string representing the individual as it is defined in the ABox.

Note that [RR98] use their context numbers in a different way. They do not attach the number to a move but mark the change of a context by a separating line in their dialogue tables. Adding the number to a move (i.e. attaching it to an expression) is similar to the notations of *hybrid logic* which is a variant of modal logic. There, each

⁵ \mathbb{Z}^+ is the set of all positive integers.

expression refers to a certain *world* in which the formula is true (see [Bla01]). For example the statement

$$@_1 \Box(A \vee B)$$

expresses that in world number 1, the assertion $\Box(A \vee B)$ is true and therefore, $A \vee B$ is true for all of 1's successors.⁶

Regarding Description Logic, it seems to make more sense to use such a hybrid notation representing our individuals than context switches as [RR98] use them. Due to the ABox in Description Logic, a reference to a certain individual is always given. That is why we add our individual number or name i to the move information.

As before, X and Y are variables which represent the players, where $X \neq Y$. The symbol $?$ is used for attacks and $!$ for defences. We write $X-f-e-i$ instead of $\langle X, f, e, i \rangle$.

2.2.3. Rules

Particle Rules

The particle rules for \mathcal{ALC} do not vary much from those of first-order logic, except for the quantification. [Kei09] uses a similar notation for modal dialogic.

	\Box		\sqcup		\neg
Assert	$X - ! - C \Box D - i$		$X - ! - C \sqcup D - i$		$X - ! - \neg C - i$
Attack	$Y - ? - L - i$	$Y - ? - R - i$	$Y - ? - \sqcup - i$		$Y - ! - C - i$
Defend	$X - ! - C - i$	$X - ! - D - i$	$X - ! - C - i$	$X - ! - D - i$	—

	\sqsubseteq	$\forall r$	$\exists r$
Assert	$X - ! - C \sqsubseteq D - i$	$X - ! - \forall r.C - i$	$X - ! - \exists r.C - i$
Attack	$Y - ! - C - i$	$Y - ? - \forall r/i^* - i^\dagger$	$Y - ? - \exists r - i$
Defend	$X - ! - D - i$	$X - ! - C - i^*$	$X - ! - C - i^* \ddagger$

Table 2.6.: Particle Rules for \mathcal{ALC} -Semantics

Prime formulæ, i.e. *atomic concepts*, cannot be attacked, of course.

⁶For a detailed introduction to hybrid logic, see for example [BvBW07], p. 821–868.

[†]for any r -filling individual i^* of i that Y chooses

[‡]for any r -filling individual i^* of i that X chooses

When attacking the formula $\forall r.C$ for an individual i , Y may choose a role-filler i^* for which X has to defend his assertion, because before, X has claimed that *all* of i 's role-fillers are in the concept C . By contrast, if X asserts $\exists r.C$ for an individual i and Y attacks that assertion, then X may choose the role-filler i^* , because he has asserted that there is just (at least) *one* role-filler that is an element of concept C .

It is obvious that the only way to change the *individual in focus*, i.e. the individual a player is making an assertion about, is either the attack of $\forall r$ or the defence of $\exists r$. We call these two moves *choice of individual*. All other moves do not change the scope (see [RR98], p. 7). Of course, it is always possible to attack or defend an earlier assertion of an individual which has already been in focus before, as long as the structural rules do not forbid this.

Here is one example:

Assertion	P - ! - $\forall \textit{navigates.Space_Ship} - \textit{HARRY}$ <i>Everything that Harry navigates is a space ship.</i>
Attack	O - ? - $\forall \textit{navigates}/\textit{VOYAGER} - \textit{HARRY}$ <i>Show me that this is true for the Voyager, which is navigated by Harry!</i>
Defence	P - ! - $\textit{Space_Ship} - \textit{VOYAGER}$ <i>The Voyager is a space ship.</i>

With her attack, O introduces the new individual *Voyager* as role-filler and changes the focus from Harry to it. Because P has claimed that everything which is navigated by Harry is a space ship, he now has to show that Voyager is also a space ship.

Structural Rules

Now, the structural rules have to be altered so that our game semantics is valid for \mathcal{ALC} .

As before, P may state prime formulæ if O has stated them before. It is reasonable to modify that rule, so that a prime formula can only be asserted by P for a given individual, if O has stated it for *the same individual* before, because for other individuals we have other truths. We alter rule **SR-ST4** thus:

(**SR-ST4ALC**) (*formal use of prime formulæ for Description Logics*): “only O may introduce prime formulæ. P cannot use a prime formula O did not utter first [*for the same individual*]. O can introduce a new prime formula anytime he wants, according to the other rules.”

see [RK05], p. 388 “for modal formal use of prime formulæ”

Now, what about *introducing* new individuals, i.e. changing the individual in focus to an individual which has not occurred in the game yet? In fact, it would be a problem to allow P to introduce new individuals, so only O is allowed to do this.

(**SR-ST6ALC**) (*formal rule for individuals*): O may introduce a new individual anytime the other rules let him do so. P cannot introduce a new individual, and his choices when changing the focus of an individual are restricted to individuals which are direct role-fillers of the individual in focus.

based on **SR-ST9.1** and **SR-ST9.2K** by [RK05], p. 388, 389

‘*Direct role-filler*’ means that only roles to another individual (that have been introduced by O before) which are reachable from the individual of focus, can be accessed. Reflexivity, symmetry or transitivity are not supported as we also do not have these features in *ALC*.

The other rules may remain as they are. We use **SR-ST2I** and **SR-ST5I** for intuitionistic *ALC* and **SR-ST2C** and **SR-ST5C** for classical.

2.2.4. Strategies

Let us suppose that both players, P and O, are agents (maybe even persons) which both want to win. For O, it is always wise to introduce a new individual whenever it is possible. By contrast, it is better for P to keep the focus on individuals which have already been introduced (according to **SR-ST6ALC**, he is not allowed to introduce new individuals anyway). The reason is simple: P may state only prime formulæ for a certain individual for which O has stated the same formula before, so O will try to ‘run away’ to new individuals and state her expressions there (see [RR98], p. 8).

2.2.5. Some Examples

Now let us try to prove some tautologies. In Section 1.2.3, we have seen some for CK. With a translated syntax, they should also be valid for intuitionistic \mathcal{ALC} .

IK2 for \mathcal{ALC}

Let us suppose that A and B are atomic concepts and therefore prime formulæ.

		O		P	
1				P - ! -	$\forall r.(A \sqsubseteq B) \sqsubseteq (\exists r.A \sqsubseteq \exists r.B)$ - 1
2	?1	O - ! -	$\forall r.(A \sqsubseteq B)$ - 1	!2	P - ! - $\exists r.A \sqsubseteq \exists r.B$ - 1
3	?2	O - ! -	$\exists r.A$ - 1	!3	P - ! - $\exists r.B$ - 1
4	?3	O - ? -	$\exists r$ - 1	?3	P - ? - $\exists r$ - 1
5	!4	O - ! -	A - 2	?2	P - ? - $\forall r/2$ - 1
6	!5	O - ! -	$A \sqsubseteq B$ - 2	?6	P - ! - A - 2
7	!6	O - ! -	B - 2	!4	P - ! - B - 2

Table 2.7.: An \mathcal{ALC} -Dialogue

Here is the explanation:

1. P states his thesis. As an abstract dummy individual, the number 1 is used. In the tableau algorithm, we might have used x instead.
2. O attacks the thesis by asserting the left side of the subsumption. So, she claims that this left part is true for the dummy individual. P reacts with the corresponding defence.
3. O does the same as before with the new subsumption. P defends his defence.
4. Now it is getting interesting: O attacks the existence quantification. This can be read as “Show me that there is a role-filler for individual 1 which is an element of the concept A ”. The problem is that P is not allowed to introduce new individuals, so he cannot change the focus (**SR-ST6ALC**). A defence is not possible, so it is time to attack one of O’s earlier assertions. P chooses the existence quantification of row 3.

5. O is allowed to introduce new individuals. So she does by changing the focus to the individual number 2 and states the prime formula A for this individual. One might think that P is now allowed to change the focus to defend himself against O's last attack. In fact, rule **SR-ST6ALC** would allow this, because individual 2 is accessible from individual 1, but then he would state the prime formula B which has not been stated by O yet and this is not allowed (**SR-ST4ALC**)! So, everything he can do is to attack O's value restriction of row 2. Note that he could not have done this before, because there has been no individual which he could have referred to. O has introduced the only available role-filler just in her last step.
6. O defends herself. She has to state the assertion for individual number 2, because P told her to do so. The resulting subsumption is attacked by P.
7. O defends again by stating the prime formula B for individual 2. Now finally, P is allowed to defend himself against O's attack of row 4. He changes the focus from 1 to 2 and states B . No other moves are possible and P wins the game because there is no way for O to backtrack.

Dualities

This is an example adapted from [RR98]. It shows that the distribution of $\forall r$ and $\exists r$ is not given in intuitionistic logic. Again, A represents an atomic concept.

	O			P		
1					P - ! - $\neg\forall r.\neg A \sqsubseteq \exists r.A$	- 1
2	?1	O - ! - $\neg\forall r.\neg A$	- 1	!2	P - ! - $\exists r.A$	- 1
3	?2	O - ? - $\exists r$	- 1	?2	P - ! - $\forall r.\neg A$	- 1
4	?3	O - ? - $\forall r/2$	- 1	!4	P - ! - $\neg A$	- 2
5	?4	O - ! - A	- 2	!3	P - ! - A	- 2

Table 2.8.: Dualities in a Dialogue

P's defence in row 5 is only possible in classical logic, because for intuitionistic logic, according to rule **SR-ST2I**, P would only be allowed to defend himself against the last attack of the rival which has been performed by O in row 5 and which cannot be defended anyway (see [RR98], p. 12).

2.3. Dialogues for $c\mathcal{ALC}$

In order to create $c\mathcal{ALC}$ -semantics for dialogical games, we have to embed the refinement relations somehow. For this, we will follow the tableau calculus presented in Section 1.2.5. Considering the tableau rules, it seems that there are already some similarities to the dialogical approach.

For example, if we want to prove an expression C in $c\mathcal{ALC}$, we usually turn it into a negative constraint $x : \neg C$ and try to close the tableau. There are different rules for positive and negative constraints, as the proponent and the opponent have slightly different rules to obey in dialogues. As we will see in this section, rules for positive constraints are applied by the proponent, while rules for negative constraints are used by the opponent.

[Bla01] provides an introduction to dialogues for *hybrid logic*. He shows that rules for tableau-based algorithms correspond to the players' moves in hybrid dialogues. We use this idea to generate our structural rules for $c\mathcal{ALC}$ -dialogues.

2.3.1. The Language

Our new language $\mathbf{L}_{c\mathcal{ALC}}$ does not differ much from $\mathbf{L}_{\mathcal{ALC}}$. The only difference is that we now do not have a focus on individuals, but on *entities* which might be more abstract than others. To emphasize this, we use another letter ϵ for entity instead of i for individual. The rest stays as before.

So, our *dialogically signed expression* has the form $\langle X, f, e, \epsilon \rangle$ for which we will usually write $X-f-e-\epsilon$. Again, X represents the *player*, f the *force symbol* and e the expression, i.e. a concept description.

Note that \perp is still written as $A \sqcap \neg A$, while \top has to be stated by the expression $\neg(A \sqcap \neg A)$. The reason is that in $c\mathcal{ALC}$ we have to obey intuitionist rules and therefore \top must not be written as $A \sqcup \neg A$ (see [MS09], p. 211).

For better readability, we will not only use positive integers for representing abstract entities. Instead, we use an enumeration of the form $n.m$ where $n \in \mathbb{Z}^+$ and $m \in \mathbb{N}^7$.

⁷ \mathbb{N} is the set of all natural numbers (including 0), i.e. $\mathbb{N} =_{df} \mathbb{Z}^+ \cup \{0\}$.

The representation ‘1.0’ denotes a general entity. Every refinement of it makes m increase, so refinements of 1.0 are 1.1, 1.2, 1.3 The refinements of $n.0$ (including $n.0$ itself) are all members of a set we call the ‘*entity group of n* ’.

When changing the focus to a role-filler, we increase n while m is reset to 0, e.g. a role-filler of 1.3 could be 2.0. A new entity group is then created.

2.3.2. Rules

Particle Rules

We now have to make some changes in the definitions of our particle rules so that our players can handle refinements.

Let us again have a look at the tableau rules of Section 1.2.5, especially at those rules which are applied to negative constraints. In most cases, a new *refining entity* will be created when using such a rule. The only cases where no refining entities are generated are the $(\rightarrow_{\sqcap-})$ - and the $(\rightarrow_{\sqcup-})$ rule. This fact is ignored for now.

In \mathcal{ALC} -dialogues, introducing new individuals or changing the focus of an individual was only possible by attacking $\forall r$ or defending $\exists r$. In $c\mathcal{ALC}$, among refining entities, it seems to be possible to change the focus for (almost) all operations.

Here is a very small example which might help to understand this:

Let us suppose that we want to prove that $C \sqsubseteq D$ is valid for two concept descriptions C and D . So we introduce an abstract entity x and assign the expression to a negative constraint:

$$x : -(C \sqsubseteq D)$$

The only possible rule to be applied is $(\rightarrow_{\sqsubseteq-})$. According to the tableau rules, a refined entity x' is generated:

$$x' : +C \qquad x' : -D$$

So, if an asserted expression of the form $C \sqsubseteq D$ is attacked, then a new entity has to be created in certain circumstances. This will be defined in the structural rules.

These are our new particle rules:

	\sqcap		\sqcup		\neg
Assert	$X - ! - C \sqcap D - \epsilon$		$X - ! - C \sqcup D - \epsilon$		$X - ! - \neg C - \epsilon$
Attack	$Y - ? - L - \epsilon'$	$Y - ? - R - \epsilon'$	$Y - ? - \sqcup - \epsilon'$		$Y - ! - C - \epsilon'$
Defend	$X - ! - C - \epsilon'$	$X - ! - D - \epsilon'$	$X - ! - C - \epsilon'$	$X - ! - D - \epsilon'$	—

	\sqsubseteq	$\forall r$	$\exists r$
Assert	$X - ! - C \sqsubseteq D - \epsilon$	$X - ! - \forall r.C - \epsilon$	$X - ! - \exists r.C - \epsilon$
Attack	$Y - ! - C - \epsilon'$	$Y - ? - \forall r/\epsilon'^* - \epsilon'$	$Y - ? - \exists r - \epsilon'$
Defend	$X - ! - D - \epsilon'$	$X - ! - C - \epsilon'^*$	$X - ! - C - \epsilon'^*$

Table 2.9.: Particle Rules for $cALC$ -Semantics

For the definition, it is important to add that

- ϵ' is a *refinement* of ϵ , i.e. $\epsilon \preceq \epsilon'$ and
- ϵ'^* is a *role-filler* of ϵ' .

So, a player who attacks a rival's assertion may always select a refining entity for which he or she states his or her assertion and for which the rival has to defend him- or herself. That is why the focus is also changed by the attack. As mentioned before, all entities refine themselves, so it is possible to attack an assertion with $\epsilon' = \epsilon$. It is important to keep this in mind. We call ϵ' the “*claimed refining entity*”.

In order to highlight the refinement relations for attacks, we will use a slightly different notation: we write $\epsilon \preceq \epsilon'$ instead of just ϵ' , e.g.

$$Y - ? - \forall r/\epsilon'^* - \epsilon \preceq \epsilon'$$

For the defence, the defender has to obey the attacker's wish concerning the refinement and therefore, ‘ $\epsilon \preceq$ ’ is omitted.

Let us look at a dialogical example to make the rules clear. It is based on Example 1.2 of page 27.

Assertion	$P - ! - \forall \text{causes.Trouble} - APPLE$ <i>Anybody eating any apple will always have trouble.</i>
Attack	$O - ? - \forall \text{cause}/SNOW_WHITE - APPLE \preceq RED_APPLE$ <i>Show me that Snow White gets into trouble when eating a red apple!</i>

Defence	$P - ! - Trouble - SNOW_WHITE$ <i>After eating the red apple, Snow White is in trouble.</i>
---------	---

With this small dialogue, a structure of three different entities is created. We first have the abstract entity *Apple* which is said to cause trouble. Then there is the entity *Red Apple* introduced by O , which refines the abstract one. O also introduces *Snow White* as entity (in fact she is an individual) and asks P to show that she is a member of the concept *Trouble*.

Structural Rules

[Bla01] shows that in hybrid logic, a tableau proof is very close to the corresponding dialogue where the proponent wants to prove the same thing as in the tableau. So, a tableau might be translated to a dialogue. It is interesting that rules which are applied on negated hybrid expressions (e.g. $\neg @_i \neg(A \wedge B)$) are played by the opponent, while rules applied on positive expressions (e.g. $@_i \neg(A \wedge B)$) correspond to moves performed by the proponent. So, it is obvious that there are some similarities to the tableau rules for $c\mathcal{ALC}$, where we have rules for negative constraints and others for positive ones. To describe it more generally, this means that negated hybrid expressions of the form $\neg @_i \phi$ always correspond to assertions stated by the proponent as $@_i \phi$, whereas positive expressions such as $@_i \psi$ correspond to the opponent's arguments.

So let us try to convert this idea into dialogues for $c\mathcal{ALC}$. Here, we have negative and positive constraints in our tableau, so let us assume that expressions of negative constraints correspond to the assertions of the proponent whose aim is to close the dialogue/tableau.

As the tableau rules show, only the negative rules (such as $(\rightarrow_{\square-})$ or $(\rightarrow_{\forall-})$) generate new refining entities, positive rules do not. We can formulate this fact in a structural rule by adjusting **SR-ST6ALC**:

(SR-ST6cALC) (formal rule for entities): O may introduce a new entity anytime the other rules let him do so. P cannot introduce a new entity, and his choices when changing the focus of an entity are restricted to entities which are refinements, direct role-fillers or role-fillers of refinements of the entity in focus.

This sounds more complex than it is. In fact, we have already seen in an example what this means for *O*. Whenever she attacks an assertion of *P*, she may introduce a new *refining* entity for which she states her expression. For *P*, this rule means that he is not allowed to create a new refinement but he may force *O* to defend herself for an existing refining entity chosen by *P*.

An example should make this clear:

Assertion	P	-	!	-	$(A \sqsubseteq B) \sqsubseteq (\neg A \sqcup B)$	-	1.0
Attack	O	-	?	-	$A \sqsubseteq B$	-	$1.0 \preceq 1.1$
Defence	P	-	!	-	$\neg A \sqcup B$	-	1.1

In this first part, *O* generates the refinement 1.1 and forces *P* to answer for this refined entity. Generally, if *P* attacks one of *O*'s assertions, he may also select the refining entity. The only restriction is that it must have been introduced by *O* before.

With this rule, we also imply the $(\rightarrow_{\preceq+})$ -rule of the tableau system: let us suppose that *O* had to assert the prime formula *A* for the entity 1.2 which is a refinement of 1.1. *P* may now attack one of *O*'s earlier assertions (e.g. *O*'s attack $A \sqsubseteq B$) for entity 1.1 and move the focus from 1.1 to 1.2. Due to the fact that the refinement relation is transitive and reflexive, he could also stay in 1.1 or move to 1.3 if it had been introduced for example as a refinement of 1.2.

...	O	-	!	-	A	-	1.2
Attack	P	-	!	-	A	-	$1.1 \preceq 1.2$
Defence	O	-	!	-	B	-	1.2

But we are not finished yet. The rule about prime formulæ must also be altered due to the fact that if *O* introduces a prime formula for an entity ϵ , she automatically states it for all refinements ϵ' .

(SR-ST4cALC) (*formal use of prime formulæ for cALC*): only *O* may introduce prime formulæ. *P* cannot use a prime formula *O* did not utter first for the same entity or an entity which is refined by the entity *P* wants to make an assertion about. *O* can introduce a new prime formula anytime he wants, according to the other rules.

Here is an example:

	O			P		
1				P - ! -	$A \sqsubseteq (B \sqsubseteq A)$	- 1.0
2	?1	O - ! -	$A - 1.0 \preceq 1.1$!2	P - ! -	$B \sqsubseteq A - 1.1$
3	?2	O - ! -	$B - 1.1 \preceq 1.2$!3	P - ! -	$A - 1.2$

Unfortunately we are still not finished, because if we keep the rules as they are now, then the formula $\neg\exists r.\perp$ would be valid (later, we will consider the corresponding dialogue in detail). But as we have seen in Section 1.2.3, this must not be possible in $c\mathcal{ALC}$. So, what is the reason for this failure?

For the tableau algorithm, the rule $(\rightarrow_{\exists+})$ is defined thus:

$$S = (\mathcal{C}, \mathcal{A}) \rightarrow_{\exists+} S' = (\{xRy, y : +C\} \cup \mathcal{C}, \mathcal{A})$$

if for some $x \in \mathcal{A}$, $x : +\exists R.C$ is in \mathcal{C} , y is a new variable and there is no R -successor z of x in S such that $z : +C$ is in \mathcal{C} .

[Sch]

It is remarkable that after applying this rule, y is not added to the active set \mathcal{A} . That is why formulæ regarding this new role-filling entity may not be touched unless this entity becomes an element of \mathcal{A} . The only way to do so is provided by rule (\rightarrow_{R-}) which can only be applied after making use of $(\rightarrow_{\exists-})$.

The fact that $(\rightarrow_{\forall+})$ also does not update the active set \mathcal{A} is no problem, because its application requires that the role-filler already exists and therefore it has to be active anyway. So, we only need a rule which solves the problem with $(\rightarrow_{\exists+})$.

(SR-ST7cALC) (coupling rule for existential quantifications): If P forces O to introduce a new role-filler by an attack, then the entity introduced by O's answer is protected against further attacks, i.e. no formulæ stated for that entity may be attacked, unless it *would* be possible for P to access that entity with a defence.

Let us explain it a little bit more formally. This is the initiated situation:

1. O has stated the following assertion with an arbitrary **non-atomic** concept description C , a role r and for an entity ϵ .

$$O - ! - \exists r.C - \epsilon$$

2. P attacks that assertion for an already introduced entity ϵ' such that $\epsilon \preceq \epsilon'$.

$$P - ? - \exists r - \epsilon \preceq \epsilon'$$

3. With her defence, O introduces a role-filling entity ϵ'^* .

$$O - ! - C - \epsilon'^*$$

Now, P is not allowed to attack C if he is not able to access the entity ϵ'^* with a defence. For a defence, an attack must have been stated before. Let D be an arbitrary concept description and $\epsilon^\#$ an entity such that $\epsilon^\# \preceq \epsilon'$. Then this dialogue might have taken place:

$$\begin{array}{l|l} \text{Assertion} & P - ! - \exists r.D - \epsilon^\# \\ \text{Attack} & O - ? - \exists r - \epsilon^\# \preceq \epsilon' \end{array}$$

With a defence, P is now able to access a role-filler of ϵ' , i.e. also ϵ'^* . But this is important: it is not relevant if he is actually allowed to do so, e.g. he might be prevented because of prime-formula-restrictions to perform the defence (rule **SR-ST4cALC**). So, he now may attack O's defending C . Here is a concrete example:

	O		P	
1				P - ! - $\exists r.(A \sqcap B) \sqsubseteq \exists r.A$ - 1.0
2	?1	O - ! - $\exists r.(A \sqcap B)$ - 1.0 \preceq 1.1	?2	P - ? - $\exists r$ - 1.1 \preceq 1.1
3	!2	O - ! - $A \sqcap B$ - <u>2.0</u>	?3	P - ? - L - 2.0 \preceq 2.0

P's last move is illegal. In row 2, he forces O to introduce entity 2.0. P has no possibility to access it with a defence and therefore it is *protected* (indicated by the underline). O's defence in row 3 cannot be attacked. By contrast, this way is possible:

	O		P	
1				P - ! - $\exists r.(A \sqcap B) \sqsubseteq \exists r.A$ - 1.0
2	?1	O - ! - $\exists r.(A \sqcap B)$ - 1.0 \preceq 1.1	!2	P - ! - $\exists r.A$ - 1.1
3	?2	O - ? - $\exists r$ - 1.1 \preceq 1.2	?2	P - ? - $\exists r$ - 1.1 \preceq 1.2
4	!3	O - ! - $A \sqcap B$ - 2.0	?4	P - ? - L - 2.0 \preceq 2.0
5	!4	O - ! - A - 2.0	!3	P - ! - A - 2.0

In row 3, O makes it possible for P to follow a role leading from entity 1.2 as soon as the time comes. P then forces O to introduce the role-filler 2.0 which is accessible by P due to O's previous attack. So, he may attack O's defence in row 4 as it is not protected. Now, with P's defence, he finally sets the focus to 2.0, states A and closes the game.

Still, it is not always necessary for P to access ϵ'^* by defending against O's previous attack. This is different with regard to the $c\mathcal{ALC}$ tableau-rules. One might think that P has to defend himself finally to win the game, because, in the tableau, the role-filling entity is *activated* after applying rule (\rightarrow_{R-}) . Yet, if we made this a compulsory restriction in our dialogues, then we would get into trouble with some theses. We will see a corresponding example at the end of Section 2.3.5.

2.3.3. Strategies

As for \mathcal{ALC} , whenever it is possible, O will generate new role-filling entities, so that P has to wait for O's assertion of a prime formula for these before P himself may state them.

In addition, O may now also 'run away' with each attack by generating new refinements of the entity in focus. P is allowed to 'move' expressions from more general entities to refined ones by attacking them, as well as he may state prime formulæ if they have been stated by O for the same or a more general entity. But still, he has to defend himself for the new entity which is claimed by O.

As we know, P is not allowed to introduce new entities. Yet, when he attacks one of O's assertions, it is wise to move to the refining entity which has just been created by O by attacking him. O is then forced to answer for that entity she wanted to *flee* to. After that defence, P might finally defend himself against O's last attack.

In the later examples, we will see these strategies in the players' moves.

2.3.4. Inference

The four properties *validity*, *subsumption*, *equivalence* and *disjointness* with respect to an empty TBox can be checked with dialogues quite easily.

Validity

To show that a concept description C is valid, we just let P state it as his thesis for a new entity 1.0.

$$P \text{ -- } ! \text{ -- } C \text{ -- } 1.0$$

If he has a winning-strategy, then C is valid w.r.t. the empty TBox. Otherwise, it is not.

Subsumption

This is quite the same. To check that D is subsumed by C , i.e. $C \sqsubseteq D$ is valid, we have to check the validity of this concept description, so P just states $C \sqsubseteq D$ in his thesis and if he has a winning-strategy, then C is subsumed by D w.r.t. the empty TBox.

$$P \text{ -- } ! \text{ -- } C \sqsubseteq D \text{ -- } 1.0$$

Equivalence

To check equivalence $C \equiv D$, we have to test two subsumptions, $C \sqsubseteq D$ and $D \sqsubseteq C$. So, if P has a winning-strategy for both subsumptions, then C and D are equivalent w.r.t. the empty TBox.

Disjointness

As we have seen before, disjointness of two concepts C and D can be reduced to subsumption:

$$C \sqcap D \sqsubseteq \perp$$

But as we have no particle rules for \perp , we just reformulate P 's thesis to $\neg(C \sqcap D)$, because $\neg C = C \sqsubseteq \perp$ (see Section 1.2.2).

$$P \text{ -- } ! \text{ -- } \neg(C \sqcap D) \text{ -- } 1.0$$

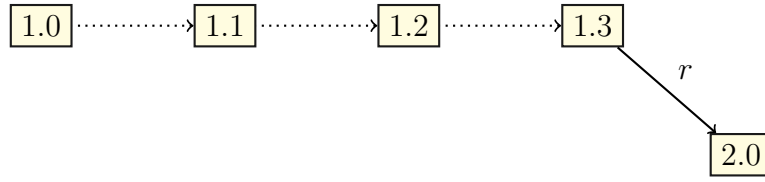


Figure 2.2.: Relationships of Entities in IK4 (Alternative 1)

2.3.5. More Examples

In Section 1.2.3, we have seen some tautologies for CK. With a translated syntax, some should also be valid for $c\mathcal{ALC}$, others should not. Let us first have a look at IK4.

IK4 for $c\mathcal{ALC}$

As before, A and B are atomic concepts.

		O		P	
1				P - ! - $\exists r.(A \sqcup B) \sqsubseteq (\exists r.A \sqcup \exists r.B)$	- 1.0
2	?1	O - ! - $\exists r.(A \sqcup B)$	- 1.0 \preceq 1.1	!2 P - ! - $\exists r.A \sqcup \exists r.B$	- 1.1
3	?2	O - ? - \sqcup	- 1.1 \preceq 1.2	!3 P - ! - $\exists r.A$	- 1.2
4	?3	O - ? - $\exists r$	- 1.2 \preceq 1.3	?2 P - ? - $\exists r$	- 1.1 \preceq 1.3
5	!4	O - ! - $A \sqcup B$	- 2.0	?5 P - ? - \sqcup	- 2.0 \preceq 2.0
6	!5	O - ! - B	- 2.0	—	

P loses. The relationships between the entities are shown in Figure 2.2. Refinement relations are illustrated by dotted arrows.

If P would not have defended against O's attack in row 3, but instead attacked O's quantification, he would have lost anyway:

		O		P	
1				P - ! - $\exists r.(A \sqcup B) \sqsubseteq (\exists r.A \sqcup \exists r.B)$	- 1.0
2	?1	O - ! - $\exists r.(A \sqcup B)$	- 1.0 \preceq 1.1	!2 P - ! - $\exists r.A \sqcup \exists r.B$	- 1.1
3	?2	O - ? - \sqcup	- 1.1 \preceq 1.2	?2 P - ? - $\exists r$	- 1.1 \preceq 1.2
4	!3	O - ! - $A \sqcup B$	- 2.0	!3 P - ! - $\exists r.A$	- 1.2
5	?4	O - ? - $\exists r$	- 1.2 \preceq 1.3	?5 P - ? - \sqcup	- 2.0 \preceq 2.0
6	!6	O - ! - A	- 2.0	—	

In row 5, P may not follow O to the role-filling entity 2.0, because he is in another refinement where no role exists (see Figure 2.3 which shows the entities' relationships of this game). Note that P is not allowed to attack O in row 4 due to rule

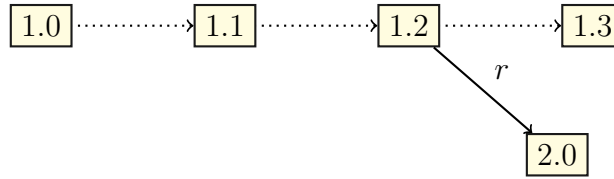


Figure 2.3.: Relationships of Entities in IK4 (Alternative 2)

SR-ST7cALC. It is also remarkable that it might have been a better strategy for O to assert B instead of A in row 6. But as P is not able to defend against O 's attack (row 5), it does not matter for her if she defends her disjunction with A or B . She wins anyway.

However, these are not all possible games. In order to show that the formula is not valid, we have to test every possible move. Only if P has no strategy (no matter how O reacts), we definitely know that the thesis is not valid. That is why we will generate a *dialogue tree* which illustrates all possible games. Figure 2.4 shows a *pruned* version. Only those of P 's attacks, with which he claims the most refining entities he is able to choose, are displayed. Note that also only those of O 's decisions are displayed which let her win, because she is allowed to backtrack if she loses. The others are reduced to dots. Repetitions allowed by **SR-ST5I** are omitted, too.

The complete tree can be found in a PDF file on the CD attached to this work. See Section 4.1 for more information.

IK5 for $cALC$

The expression $(\exists r.A \sqsubseteq \forall r.B) \sqsubseteq \forall r.(A \sqsubseteq B)$ is also not valid in $cALC$. The (pruned) dialogue tree, with O 's decisions that let her win, is shown in Figure 2.5.

IK3 in $cALC$

As announced before, we will now show that $\neg\exists r.\perp$ is not valid in $cALC$, too. For this, we replace \perp by $A \sqcap \neg A$.

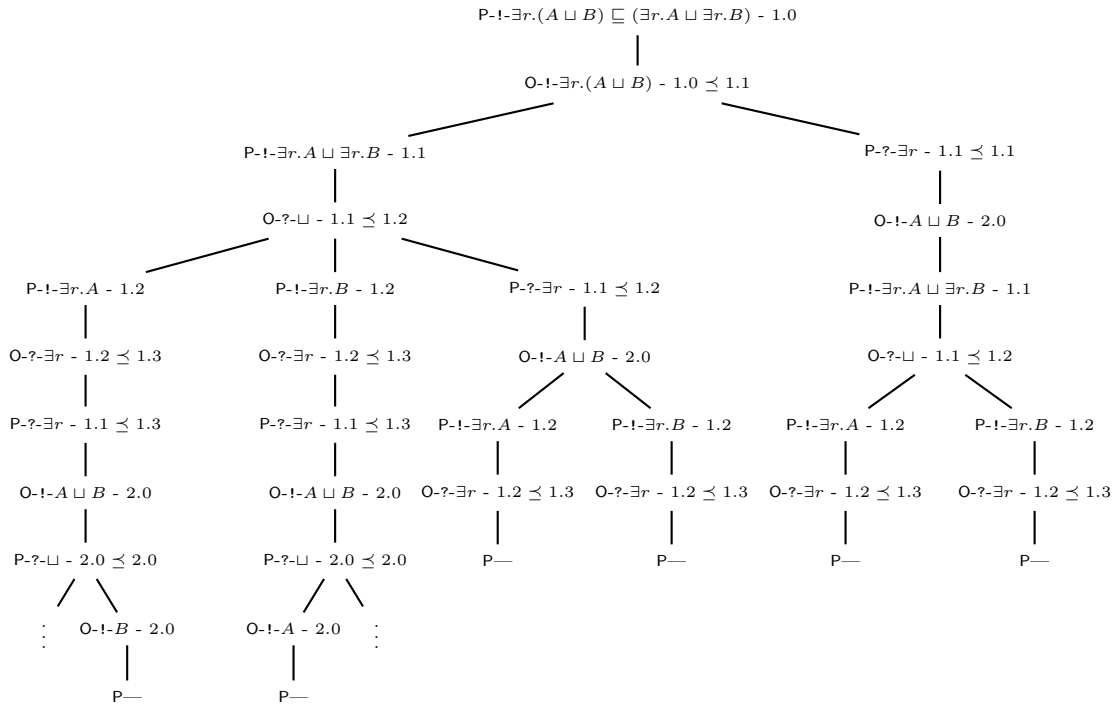


Figure 2.4.: Dialogue Tree for IK4

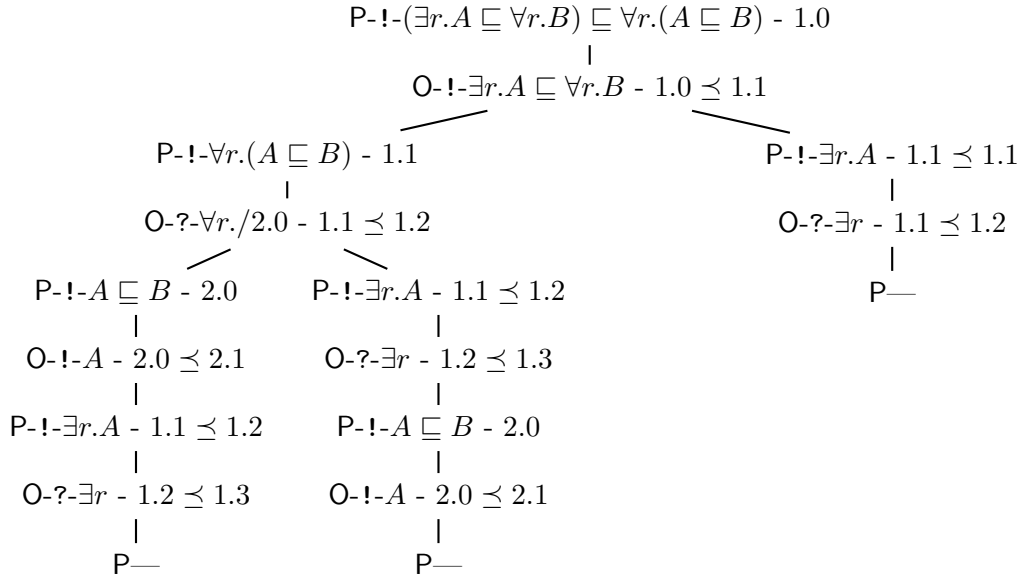


Figure 2.5.: Dialogue Tree for IK5

		O		P	
1					P - ! - $\neg\exists r.(A \sqcap \neg A)$ - 1.0
2	?1	O - ! - $\exists r.(A \sqcap \neg A)$	- 1.0 \preceq 1.1	?2	P - ? - $\exists r$ - 1.1 \preceq 1.1
3	!2	O - ! - $A \sqcap \neg A$	- 1.1	—	

P has attacked an existential quantification in row 2 but O does not attack an equivalent existential quantification for the same entity. That is why P is not allowed to attack O in row 3 (**SR-ST7cALC**).

Wine and Meat

This example is more practical. It is adapted from [BMPS⁺91] and [BFFF07]. [MS09] provide a Hilbert-style proof for $c\mathcal{ALC}$ (p. 219 ff.). Now we also have two axioms. Axioms are stated by the opponent at the very beginning of a dialogue, as it is assumed that they are true anyway (see [Kei09]). Of course, axioms are part of the TBox (see [MS09], p. 219). We do not assign them to a particular entity as they are true for every entity.

- $Ax_1 =_{df} \text{FOOD} \sqsubseteq \exists \text{goesWith}.\text{COLOR}$
- $Ax_2 =_{df} \text{COLOR} \sqsubseteq \exists \text{isColorOf}.\text{WINE}$

We want to prove: $\text{FOOD} \sqsubseteq \exists \text{goesWith}.(\text{COLOR} \sqcap \exists \text{isColorOf}.\text{WINE})$.

Because of a lack of space, we make use of some abbreviations: let us write F for FOOD, W for WINE, C for COLOR, *ico* for *isColorOf* and *gw* for *goesWith*.

		O		P	
A_1		O - ! - F $\sqsubseteq \exists gw.C$			
A_2		O - ! - C $\sqsubseteq \exists ico.W$			
1					P - ! - F $\sqsubseteq \exists gw.(C \sqcap \exists ico.W)$ - 1.0
2	?1	O - ! - F	- 1.0 \preceq 1.1	!2	P - ! - $\exists gw.(C \sqcap \exists ico.W)$ - 1.1
3	?2	O - ? - $\exists gw$	- 1.1 \preceq 1.2	? A_1	P - ! - F - 1.0 \preceq 1.2
4	!3	O - ! - $\exists gw.C$	- 1.2	?4	P - ? - $\exists gw$ - 1.2 \preceq 1.2
5	!4	O - ! - C	- 2.0	!3	P - ! - C $\sqcap \exists ico.W$ - 2.0
6	?5	O - ? - L	- 2.0 \preceq 2.1	!6	P - ! - C - 2.1

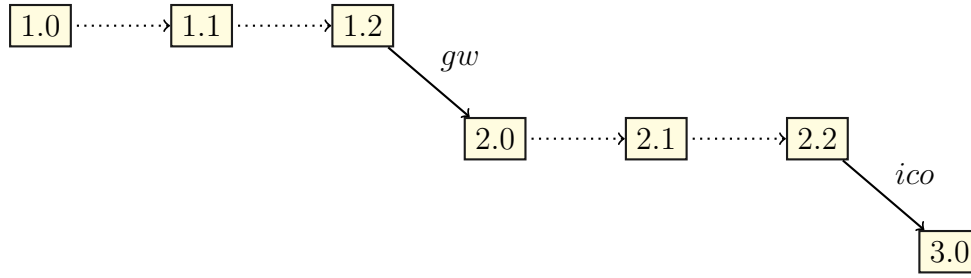


Figure 2.6.: Relationships of Entities for Wine and Meat

		O		P	
6'	?5	O - ? - R	- 2.0 \preceq 2.1	!6'	P - ! - \exists ico.W - 2.1
7'	?6'	O - ? - \exists ico	- 2.1 \preceq 2.2	?A ₂	P - ! - C - 2.0 \preceq 2.2
8'	!7'	O - ! - \exists ico.W	- 2.2	?8'	P - ? - \exists ico - 2.2 \preceq 2.2
9'	!8'	O - ! - W	- 3.0	!7'	P - ! - W - 3.0

The entities' relationships are illustrated in Figure 2.6. Note that after row 6 O tracks back because that game is won by P. He may state the prime formula COLOR because O has stated it for a more general entity of the same entity group. So, for the second game this row may be ignored.

Accessible Without Accessing

This last example shows that it could be impossible for P to defend himself against one of O's attack providing access to a role-filling entity.

		O		P	
A ₁		O - ! - $\forall r.A$			
A ₂		O - ! - $\forall r.B$			
1				P - ! - $\exists r.(\neg(A \sqcap B)) \sqsubseteq \exists r.foo$ - 1.0	
2	?1	O - ! - $\exists r.(\neg(A \sqcap B))$	- 1.0 \preceq 1.1	!2	P - ! - $\exists r.foo$ - 1.1
3	?2	O - ? - $\exists r$	- 1.1 \preceq 1.2	?2	P - ? - $\exists r$ - 1.1 \preceq 1.2
4	!3	O - ! - $\neg(A \sqcap B)$	- 2.0	?4	P - ? - $A \sqcap B$ - 2.0 \preceq 2.0
5	?4	O - ? - L	- 2.0 \preceq 2.1	?A ₁	P - ? - $\forall r/2.0$ - 1.0 \preceq 1.2
6	!5	O - ! - A	- 2.0	!5	P - ! - A - 2.1
5'	?4	O - ? - R	- 2.0 \preceq 2.1	?A ₂	P - ? - $\forall r/2.0$ - 1.0 \preceq 1.2
6'	!5'	O - ! - B	- 2.0	!5'	P - ! - B - 2.1

As we see, P has never defended against O's attack that she performed in row 3. In fact, he is not able, as foo is considered to be a prime formula that has not been stated by O. However, P wins. The reason is quite simple. In the tableau, we would apply (\rightarrow_{R-}) to *activate* entity 2.0, because there, we do not have the restriction about prime formulæ. By contrast, for the games we have it and therefore the feature of P's *accessibility* w.r.t rule **SR-ST7cALC** is enough for us.

2.4. Conclusion and Miscellaneous

We have seen an alternative reasoning method for $c\mathcal{ALC}$. It is constructed by extending the structural rules and altering the particle rules of \mathcal{ALC} -dialogues. In fact, if O was not allowed to create refining entities with her attacks and if we removed rule **SR-ST7cALC**, we would have \mathcal{ALC} -dialogues again.

It is also interesting that [RR98] can alter their modal logic system by exchanging one single rule to deal with other modal *frames*. It is then possible to deal with *reflexive*, *symmetric* or *transitive* transition systems. For our Description Logic this means that it is also possible to construct dialogue systems supporting transitive, reflexive or symmetric roles by altering or replacing rule **SR-ST6(c)ALC**.

In fact, dialogue-based proofs for $c\mathcal{ALC}$ have some advantages but also disadvantages to tableau-based proofs. Those are discussed in Chapter 4.

3

DESIGN AND IMPLEMENTATION OF A DIALOGUE-BASED PROVER FOR *cALC*

It is now time to build a dialogue-based prover for *cALC*. It should be able to perform user-guided and automated reasoning the way it has been shown in the previous chapter. With it, automated validation of concept descriptions can be performed. It is also important that we formalize the fuzzy structural rules with the implementation. Eventually, we have a tool that illustrates dialogues to achieve a better understanding of the proof method.

First, we take a look at the tools we use to implement the prover and choose a programming language which shall satisfy our needs. Afterwards, we will design the required data structures and then we will have a closer look at the implementation of the particle and structural rules that are the most complex part of the program. At the end, we will see the completed program in action.

3.1. Tools for the Implementation

3.1.1. A Functional Programming Language

Functional programming languages provide a high level of abstraction what makes it often possible to write powerful functions in a short way (see [RL99], p. ix). Usually, no global variables are permitted and therefore the results of a function depend only on the given input parameters and whenever the function is called with the same parameters, the function's result will be the same, too. This fact leads to a higher level of security and makes it easier to test our functions (see [OGS09], p. xxiv).

We use *Haskell* as our programming language. It is a *pure* functional language applying *lazy evaluation* by default, so terms are evaluated when it is really necessary, i.e. after nothing else is possible. It is a statically typed language and supports *pattern matching* which contributes to the simplicity of creating functions quickly and easily (see [OGS09], p. xxiv-xxvii, 50–55).

We use the *Glasgow Haskell Compiler* (GHC for short)¹ in version 6.12.1 that also comes with a Haskell interpreter. There are other interpreters, for example *Hugs*, but according to [OGS09], “GHC is much more suited to ‘real work’: it compiles to native code, supports parallel execution, and provides useful performance analysis and debugging tools” (p. 1). Further, GHC is supported by our development environment *Leksah*.

3.1.2. Development Environment

*Leksah*² is an *integrated development environment* (IDE) written in Haskell and especially for Haskell projects. It helps to maintain an overview of the different modules and functions and provides debug mechanisms. We will use it for the development of our project.

¹<http://www.haskell.org/ghc/>

²<http://www.leksah.org/>

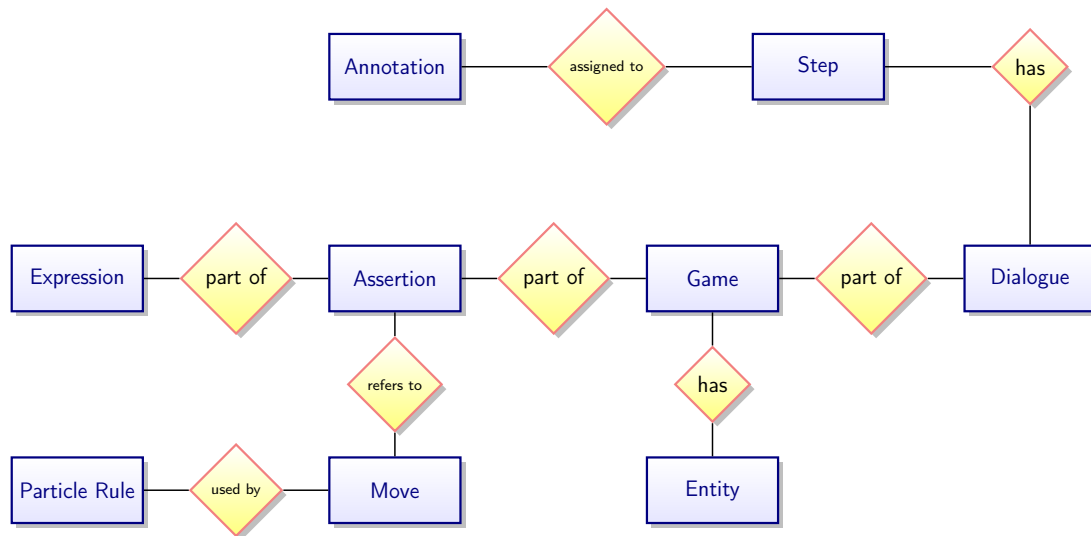


Figure 3.1.: Overview of Data Types

3.1.3. Creating Graphs

For the visualization of dialogue trees, we will make use of the free software *GraphViz*³. Our program will generate DOT source code which is then translated by a program (that is also called *DOT* and that is part of the GraphViz package) to a graph-structure which is then written to a PDF file.⁴

3.2. Architecture

Let us now look at the different data structures we are going to use and how they are interacting. First, we consider the interaction of the most important structures and then discuss them in detail.

Let us begin with a rough overview of the most important data types and their relationships to each other. These are illustrated in an Entity-Relationship-Diagram⁵ (Figure 3.1). Note that this illustration is not complete but should give an overview. Nevertheless, it might appear confusing for now. But at the end of this section, the meanings should be clear.

³<http://www.graphviz.org/>

⁴see <http://www.graphviz.org/Documentation.php> for more details

⁵The Entity-Relationship Model has originally been introduced by [Che76].

As we see, an *expression* is part of an *assertion* that is part of a (dialogical) *game* which again is part of a *dialogue*. A *step* holds one dialogue while there are several *annotations* which are assigned to it. A game usually manages some *entities*, whereas *moves* use *particle rules* and refer to assertions.

We will now regard every single data type in detail. Every type is assigned to a *module* i.e. a Haskell source file containing functions and type definitions for a certain subject (see [OGS09], p. 113 ff.). The module names and dependencies are mentioned at the beginning of each type description. We begin with the simple type *Expression* and move to more complex types afterwards.

3.2.1. Expressions

<i>Type</i>	Exp
<i>Module</i>	Expression

Expressions represent *concept descriptions*, for example

$$\exists \text{commands.}(\text{Janitor} \sqcup \text{Navigator})$$

could be such a statement. Information about individuals/entities are not part of it. We use a recursive definition to define what an expression is:

```

data Exp = Atom AtomType | -- atomic concept
          Not Exp         | -- not/negation/complement
          And Exp Exp     | -- and/intersection
          Or Exp Exp      | -- or/union
          Impl Exp Exp    | -- implication/subset
          All RType Exp   | -- for all / value restriction
          Some RType Exp  | -- there is / some
deriving (Eq, Read, Show)

```

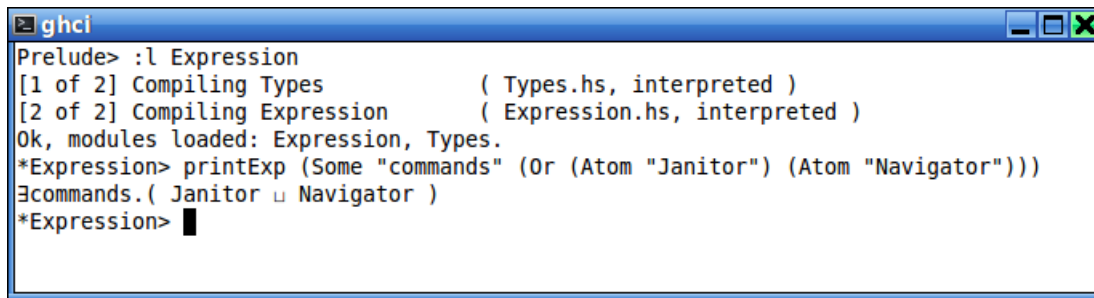
In this definition, some types are used which have not been defined yet.

```

type AtomType = String
type RType = String

```

Both types `AtomType` and `RType` are represented as *strings* of characters. So, an atomic concept *Janitor* is written `Atom "Janitor"` in our Haskell program. The complement



```

ghci
Prelude> :l Expression
[1 of 2] Compiling Types          ( Types.hs, interpreted )
[2 of 2] Compiling Expression      ( Expression.hs, interpreted )
Ok, modules loaded: Expression, Types.
*Expression> printExp (Some "commands" (Or (Atom "Janitor") (Atom "Navigator")))
∃commands.( Janitor ⊔ Navigator )
*Expression> █

```

Figure 3.2.: Showing Expressions in a Terminal Window

of *Janitor* can just be written as `Not (Atom "Janitor")`. The word `Exp` represents any concept description, so we can negate every concept description that can be built with the given constructors.

`RType` is used to represent *roles*. With the `Some`-constructor we can now express the concept description we have had before:

```
Some "commands" (Or (Atom "Janitor") (Atom "Navigator"))
```

The value restriction works in the same way but with the key-word `All` instead of `Some`.

It is notable that prefix-notation is applied. While the constructor `Or` is used for the infix-symbol \sqcup , we can write `And` for \sqcap and `Impl` for \sqsubseteq .

The concepts \top and \perp are missing here because we do not need them for our dialogues (see Section 2.3.1). Whenever they are needed, we can add them easily.

The last line of our description `deriving (Eq, Read, Show)` states that the *equality* of two expressions can be tested (`==`), an expression can be *read* from a string and eventually transformed into a string. Anyway, for better readability, we define our own function which prints an expression to the terminal. You can test this in the Haskell-interpreter *ghci*⁶ (see Figure 3.2). Note that it looks only this good in shells which are able to represent UTF-8 symbols. Shells like those from Microsoft[®] Windows[®] are not able to display special characters like \sqcap , \sqcup , \sqsubseteq , \forall , \exists or \preceq (see Appendix B.2 for more details).

⁶Type `ghci` in the command shell window to start it. Type `:q` to quit.

3.2.2. Assertions

<i>Type</i>	Assertion
<i>Module</i>	Assertion

An assertion can be either a player's move (*attack* or *defence*) or a *thesis* which is stated by the proponent. But before we look at the type-definition of assertions we have to understand some simpler types.

Player

This is a very simple type. In Chapter 2 we have used the letter P for proponent and O for opponent. Now, we do the same:

```
data Player = P | O
          deriving (Eq, Show)
```

As before, players can be displayed (as the letters P and O) and tested for equality.

Rows and References

In Chapter 2, we have used the term *row* for a row of the dialogue table. Now we use row as an identifier for an arbitrary assertion. The *thesis* usually has the row number 0, the first move has number 1 and so on. So, rows are represented by integers (**Int** in Haskell).

We use the type *reference* for referring earlier rows (usually identifying the rival's assertions). This data type is not really necessary because we could also use **Row** for this. Nevertheless, in order to distinguish between the identifier of the assertion we are talking about and an assertion we refer to, we use both types.

```
type Row = Int
type Reference = Row
```


Entity IDs

Every assertion refers to a certain (abstract) entity for which it is stated. We use a pair of integers to identify such an entity. The first number represents the *entity group*, the second one the refinement with 0 being the most general entity of an entity group.

```
type EntityID = (Int, Int)
```

Attacks, Defences and the Thesis

We define assertions thus:

```
data Assertion = Attack Row Player Reference AttackType |
                — ?/! (all kinds of attacks)
              Defence Row EntityID Player Reference Exp |
                — ! (defences)
              Thesis Row EntityID Player Exp
                — ! (thesis)
  deriving (Eq, Show)
```

The *thesis* is explained easily. It has just a *row* (usually 0), it refers to an *entity*, usually 1.0 and a player (in our case always P) and it always holds an *expression* stated by P. That is the initial assertion of a game or dialogue. For example, the thesis

$$0. \quad P \text{ — ! — } \exists r.(A \sqcup B) \sqsubseteq (\exists r.A \sqcup \exists r.B) \text{ — } 1.0$$

in row 0 can be written thus:

```
Thesis 0 (1,0) P (Impl (Some "r" (Or (Atom "A") (Atom "B")))
                    (Or (Some "r" (Atom "A")) (Some "r" (Atom "B")))).
```

Although it is not really necessary to include the player information in the thesis, we still do it to make the system more flexible for further development. In fact, it is claimed by one of the structural rules that P is the first player so this is defined somewhere else.

For the attack and the defence, a *player* is required. Additionally, a *reference* is needed that indicates which of the rival's assertion is attacked or defended against.

Further, regarding the attack, we have to care about an *attack type*. We have to introduce this because an attack can have two forms.

- If we attack \sqcap , \sqcup , $\exists r$ or $\forall r$, then an attacking *label* is stated as assertion (L , R , \sqcup , $\exists r$ or $\forall r$).
- If we attack \neg or \sqsubseteq then we attack with a formula/expression.

To distinguish between these two types of attacks, we introduce the *AttackType*:

```
data AttackType = Label AttackLabel           |  -- (label)
                  Assert Exp EntityID EntityID  -- (expression)
deriving (Eq, Show)
```

Let us first look at the second case indicated by the constructor *Assert*. The expression with which the player wants to attack has to be stated followed by two entities. The first is the entity of the assertion which is going to be attacked, while the second one is the refining entity for which the attack is taking place and for which the rival (i.e. the other player) is forced to defend him- or herself (if he/she chooses to defend).

For a label-attack, we need one more data type. It defines the labels which are available:

```
data AttackLabel = LaL EntityID EntityID      |
                  LaR EntityID EntityID      |
                  LaOr EntityID EntityID     |
                  LaAll RType EntityID EntityID EntityID |
                  LaSome RType EntityID EntityID
deriving (Eq, Show)
```

The constructors *LaL*, *LaR*, and *LaOr* are used for the labels L , R , and \sqcup . The required entity IDs have the same meanings as for attacks with expressions: the first is the entity of the attacked assertion and the second is the refining one.

The constructors *LaAll* and *LaSome* which are used for $\forall r$ and $\exists r$ require a role type r in order to define which role is attacked. *LaAll* has a third entity ID. It represents the role-filler the attacking player forces his rival to access.

```

ghci
Prelude> :l Assertion.hs
[1 of 3] Compiling Types      ( Types.hs, interpreted )
[2 of 3] Compiling Expression  ( Expression.hs, interpreted )
[3 of 3] Compiling Assertion  ( Assertion.hs, interpreted )
Ok, modules loaded: Assertion, Expression, Types.
*Assertion> printAssertion (Defence 2 (1,1) P 1 (Or (Some "r" (Atom "A")) (Some "r" (Atom "B"))))
2. ! : 1
   P - ! - ( ∃r.A ) ∪ ( ∃r.B ) - 1.1
*Assertion>

```

Figure 3.3.: Showing Assertions in a Terminal Window

Now, as the explanation of the attack is complete, let us look at an example. The thesis stated at the beginning can be attacked by O with the assertion

$$1. \quad O - ! - \exists r.(A \sqcup B) - 1.0 \preceq 1.1$$

in row 1. This is an attack with an expression. For our program, we write it thus:

```
Attack 1 0 0 (Assert (Some "r" (Impl (Atom "A") (Atom "B")))) (1,0) (1,1)) .
```

Now the defence. The entity ID represents the entity for which the defence takes place. The expression at the end is the formula with which the player defends himself against the attack.

This is P 's defence against O 's attack in row 2:

$$2. \quad P - ! - \exists r.A \sqcup \exists r.B - 1.1$$

To make our program understand this, we type:

```
Defence 2 (1,1) P 1 (Or (Some "r" (Atom "A")) (Some "r" (Atom "B")))) .
```

In order to print an assertion to the terminal, we can make use of the function `printAssertion` which works the same way as `printExp` works for expressions. In Figure 3.3, we see the outcome of calling that function. The first line shows the row number followed by the force-symbol (! for defences and ? for attacks) and the row of the referred assertion (attack in this case). The second line shows the *dialogically signed expression* as we know it from Section 2.3.1.

3.2.3. Entities

<i>Type</i>	Entity
<i>Module</i>	Entity

We now use an entity structure in order to record the atomic formulæ which have been stated by the players. This helps us to check quickly if a game or dialogue is closed or if O has stated an atomic formula for a certain entity so that P may state it, too (see rule **ST-SR4cALC**). Note that this structure is based on the data type *World* which has originally been used in a tableau-based reasoner for $c\mathcal{ALC}$ for a similar purpose. We reuse it here in order to have access to the functions of the world-module without needing to implement them once more. For a better understanding, we rewrite the type definition here for entities:

type Entity = (EntityID , [Exp] , [Exp] , [RExp])
— <i>ID</i> , <i>O-atoms</i> , <i>P-atoms</i> , <i>r-minus (unused)</i>

The first element is an entity ID which we already know from the assertions. It makes it possible to retrieve entity information by searching for that ID.

The second element of the tuple is a list of atomic expressions stated by O while the third one contains atoms states by P .

We do not need the r-minus list for our purposes. It is just a relict of the world-structure, so we do not care about it. Whenever we use the entity-structure we let the last element be an empty list ($[\]$).

Note that the entity-type is used for recording atomic expressions only but could also be used to record any expressions.

3.2.4. Games

<i>Type</i>	Game
<i>Module</i>	Game

Now, let us turn to a game branch. As we know from Chapter 2, a game is a linear sequence of assertions. It begins with the thesis which is followed by the players' moves (see rule **SR-ST0**). So far, we do not consider branches which might occur if a player makes a decision about what move to perform next.

type Game = ([Assertion] , [Entity] , [Rel] , [Ref] , [EntityID])
— <i>assertions</i> , <i>entities</i> , <i>roles</i> , <i>refinements</i> , <i>all entities</i>

As we see, a *game* is defined as a tuple of five different lists.

The first list contains all assertions of a game. The second one lists entities which contain atomic formulæ. Only those entities are listed for which atomic expressions have been stated.

Now we have two new types. The first describes role relationships between two entities (`Rel` stands for *relation*), while the second one does the same for refinement-relationships. As roles are described by a *role-type* and two entities, we need a triple to define them. For refinement relations, a pair is enough.

```

type Rel = (RType, EntityID, EntityID) — role-name, source, destination
type Ref = (EntityID, EntityID)      — source, destination

```

As we know, refinement-relations are transitive. However, not all combinations are elements of this list (that would be too much). Only immediate links are recorded here. The others can be derived from them.

The last list contains the IDs of all entities which appear in the dialogical game.

3.2.5. Dialogues

<i>Type</i>	Dialogue
<i>Module</i>	Dialogue

As explained before, a dialogue consists of several games. We have seen such dialogue trees in Section 2.3.5. Whenever a player makes a decision, new branches are generated. Rule **SR-ST3** gives **O** the right to backtrack if she loses a game. If she has a choice, two branches appear. By contrast, **P** might have more possibilities, so it might be necessary to generate more than two branches for him (but, of course, branches for **P** are only created if he is also allowed to backtrack). Anyway, we use just *one* multi-branching tree for both players and their decisions.

```

data Dialogue = Straight Game |
                Branch Row Player [Dialogue]
deriving Show

```

Let us suppose that we have a dialogue without any decisions. Then we just have a *straight* game. For this, we use the constructor **Straight**.

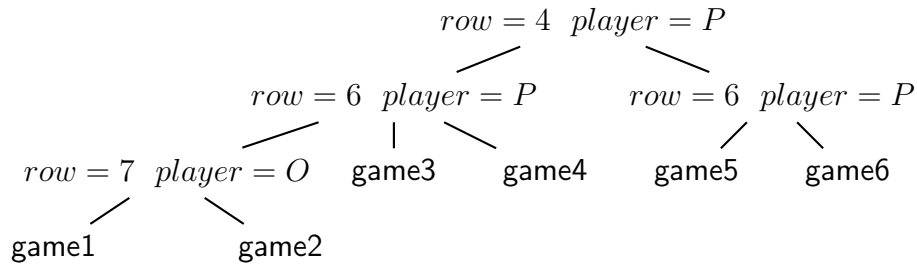


Figure 3.4.: A Redundant Dialogue Structure

Now, if we examine different runs through the dialogue, we make use of the **Branch**-constructor followed by the *row*-number indicating where the branching takes place, the player whose decision causes the split of the dialogue and a list of dialogues containing the different possibilities of the player's moves (the node's children). An instantiated dialogue tree as we have just defined it here, might look like the one illustrated in Figure 3.4.

The leafs **game1** to **game6** represent games of the structure we have defined in Section 3.2.4. All nodes which are no leafs only contain information about a player and a row number. Games are only recorded in the leafs of the tree and all these games begin with the same row 0 (the thesis), i.e. there is a redundancy in each leaf. That is why, in our example tree, the first four assertions (rows 0 to 3) are the same for all game-leafs, while even the first seven assertions (rows 0 to 6) of the leafs **game1** and **game2** are identical.

Of course, this redundancy means that more space is needed when coping with a dialogue structure. But the advantage is that we need less time when performing a back-track because the next possible game is available in the next branch where we can access it directly without merging games. It is also reasonable because the branched games might have different role- or refinement-relationships or even different *annotations* (see Section 3.2.7).

Like [PR05], we call those nodes indicating a decision made by P, P-nodes, while nodes indicating a decision made by O are named O-nodes.

Translating the dialogue of Figure 3.4 to our Haskell-structure looks thus:

```
Branch 4 P
  [ (Branch 6 P
    [ (Branch 7 0 [(Straight game1), (Straight game2)]),
      (Straight game3),
      (Straight game4)
    ] )
    (Branch 6 P [(Straight game5), (Straight game6)]) ]
```

Note that the constants `game1` to `game6` must have been defined before defining this dialogue.

Navigation

In order to navigate to a certain game of a dialogue, we construct a data type that we call `GameNav` (for *game navigation*). It is represented by an integer greater or equal to zero indicating a branch to select (0 refers to the first ‘leftmost’ branch).

```
type GameNav = Int
```

To access a certain game, we need a list of *GameNav*-integers representing the search-path. For example, the path `[0,0,1]` applied on the dialogue of Figure 3.4 leads to `game2`. In order to retrieve a game from a *straight dialogue*, we use an empty list `[]` as navigation path.

3.2.6. Particle Rules

<i>Type</i>	<code>PRule</code>
<i>Module</i>	<code>ParticleRule</code>

Particle rules work syntactically as they specify the *local semantics*. They depend on the assertions but not on a complete game or dialogue. However, the definition of particle rules is more complex than of the structures we have discussed so far, but it makes it possible to add or remove rules very easily.

```
type PRule
    = (PRuleName, PRuleAttack, PRuleDefence, PRuleParRequire)
```

So, a particle rule (`PRule`) is a tuple of four different ‘values’ or functions. The first is a name. This can always be extracted so that we can find out which rule we are using (or which is suggested by the program). The rule-name is defined as a string.

```
type PRuleName = String
```

We then have something called `PRuleAttack` and `PRuleDefence`. These are functions that can be extracted from the `PRule` tuple. One generates an attack and the other generates a defence. Let us first look at the attack function.

```
type PRuleAttack = Assertion -> [PRulePar] -> Maybe Assertion
```

An attack function receives an assertion which shall be attacked and some *particle rule parameters* (we talk about them in Section 3.3.1). The result of the function is an attacking assertion, but only if the particle rule is applicable syntactically. This is indicated by the data type `Maybe`. If the rule is not applicable then the constructor `Nothing` is returned. Otherwise, we receive the constructor `Just` followed by the attacking assertion. The *Maybe*-type is defined in Haskell’s prelude. This is a definition taken from [OGS09] (p.57):

```
data Maybe a = Just a
              | Nothing
```

Which assertion is returned by our function depends on the input assertion that shall be attacked and the input parameters. Every rule has its own parameters for attacks and its own parameters for defences. For example, the attack of $C \sqcap D$ has less parameters than the attack of $\forall r.C$, which in turn requires more parameters than the defence of $\forall r.C$.

In order to obtain the defence against an attack, we need more than the attacking assertion and some parameters. Let us suppose that player `O` attacks `P` with $O-?-L-xy$ (we are not interested in the entity). This is obviously an attack, but if we only have

this attacking assertion, then we do not know how to defend, because we need the attacked assertion, too, or at least an expression.

```
type PRuleDefence
    = Exp -> Assertion -> [PRulePar] -> Maybe Assertion
```

The first argument is that expression which has been attacked by the rival's assertion (second argument) and that we are going to defend with rule parameters (third argument). As before, if no defence is possible then `Nothing` is returned.

The last element of our `PRule`-tuple is a function, too. We need it so that we can find out which parameters are required by the particle rule.

```
type PRuleParRequire = Action -> [PRulePar]
```

As mentioned before, an attack requires other parameters than a defence. That is why we need an extra type defining what parameters we wish to get (for an attack or for a defence). Unfortunately, the constructors `Attack` and `Defence` are already taken by the assertion-type. That is why we use the progressive verb-form instead:

```
data Action = Attacking | Defending
    deriving (Eq, Show)
```

The function then returns a list of parameters with *dummy elements*. We are just interested in the constructors of the parameter types but it is not possible to return them without values.

The particle rule parameters are not interesting for now. We will talk about them in Section 3.3.

3.2.7. Annotations

<i>Type</i>	Annotation
<i>Module</i>	Annotation, DialogAnn

In order to obey the structural rules, we have to remember what has already happened to which assertions and for which entities. For example, we have to remember which

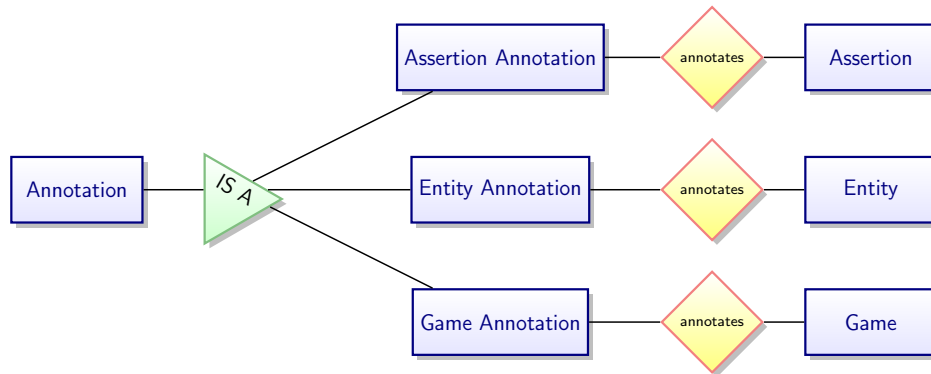


Figure 3.5.: The three Types of Annotations

assertions have been attacked so that they are *locked* for further attacks. To do so, we make use of so-called *annotations*.

Annotations are stored in a separate data structure so that we can change the annotation structures, add other annotation values or remove them without touching the annotated structures.

In order to grant quick access to the annotation of an arbitrary data object, we base our new data-type on a predefined one: the *map*. Maps have a better performance than the usual list. Data can be accessed, stored and deleted using a unique *key* (e.g. an integer) that refers to the *value* that is interesting for us (see [OGS09], p. 301–303).

We have three different data types we want to annotate, so we need three different kinds of annotations (see Figure 3.5). The most important is the *assertion annotation*, but annotations are also important for *entities* and *games*.

Assertion Annotations

In order to refer to a certain assertion of a dialogue, we need a *navigation information* that identifies the game and a *row number*. We aggregate both values to a pair and this is how we obtain a *key* for assertion annotations.

```

type AssertionAColl = AnnoColl AssertionKey [AssertionAnn]
type AssertionKey   = ([GameNav], Row)
  
```

The data-type `AnnoColl` is used as a synonym for the `Map` (this has been defined in the module `Annotation`) using keys of the type `AssertionKey` and values of the type `[AssertionAnn]`. In other words: an assertion key, consisting of a navigation information and a row, refers to a list of *assertion annotations*. This list contains all annotations that have been assigned to that specific assertion. Let us assume that this list is unordered and contains every assigned annotation only *once*, i.e. we consider the list as a *set* of annotations.

Let us have a look at the different *annotation values*.

```

data AssertionAnn =
  IsDefence      |
    — assertion is a defence (otherwise attack or thesis)
  LeftAttacked  |
    — an AND has been attacked claiming the left side
  RightAttacked |
    — an AND has been attacked claiming the right side
  Locked        |
    — assertion cannot be attacked any more (for the moment)
  AnsweredAttack |
    — assertion is an attack which has already been answered
  AppliedMoves [AbstractMove]
    — moves that have been applied to the assertion
deriving (Eq, Read, Show)

```

The meaning of the annotations are explained in the comments. The last one is the only annotation type that carries further information: a list of *abstract moves* that have been applied to the annotated assertion. We will discuss abstract moves in Section 3.2.8.

Entity Annotations

Constructing a key type for entities is similar to the construction of a type for assertions. Instead of the row number, we now use the *entity ID*. In order to get along with the coupling rule (**SR-ST7cALC**), we need two different annotation values for entities.

```

type EntityAColl = AnnoColl EntityKey [EntityAnn]
type EntityKey   = ([GameNav], EntityID)

```

```

data EntityAnn =
  Protected |
    — no attacks can be performed due to SR-ST7cALC
  PAccessible
    — P would be able to access the entity with a defence.
deriving (Eq, Show, Read)

```

Game Annotations

To identify a game in a dialogue, the navigation information provides enough input. Therefore, our new key's structure is quite simple. We have only two different types of game annotations.

```

type GameAColl = AnnoColl GameKey [GameAnn]
type GameKey = [GameNav]
data GameAnn =
  GameFinished |
    — Game finished (no more moves possible)
  GameClosed
    — Game closed (atomic formula stated by both players)
deriving (Eq, Show, Read)

```

We will talk about the annotations and their meanings again later when considering the implementation of the structural rules.

3.2.8. Moves

<i>Type</i>	Move
<i>Module</i>	ParticleRule

A *move* contains all information needed to create a new assertion for a given game. This includes

- an *action*, i.e. the constructor **Attacking** or **Defending** (see 3.2.6)
- a *particle rule* including all its functions
- a list of *particle rule parameters*
- a *reference* to the assertion we want to attack or defend against with the move

We aggregate these data to a tuple:

```
type Move = ( Action , PRule , [PRulePar] , Reference )
```

Now we have a small problem: `PRule` contains functions that cannot be compared to each other or printed to the console. So, a particle rule cannot be compared to other particle rules or displayed on the screen either. But we need these features for *assertion annotations* (see 3.2.7). So, let us introduce a further type:

```
type AbstractMove = ( Action , PRuleName , [PRulePar] , Reference )
```

The *abstract move* does not contain the complete particle rule with its functions, but instead it just holds its name which we assume to be a unique name identifying a certain particle rule. A move can easily be transformed to an abstract one and also vice versa with some extra information.

There is another difference between abstract moves and moves, but we will talk about that later, because it concerns one of the structural rules.

3.2.9. Steps

<i>Type</i>	DialogStep
<i>Module</i>	StructuralRule

`DialogStep` is the data type that combines a dialogue with all its annotations. As most of the types explained before, it can be understood as a tuple of values:

```
type DialogStep
  = ( Dialogue , GameAColl , EntityAColl , AssertionAColl , DialogTrace )
```

The first element represents the dialogue's complete tree-structure that is followed by three *annotation collections* (for games, entities and assertions). The last value is something we call *trace*.

The *trace* is not needed to make our program work. It simply contains information about what move has been performed last. If we keep all these trace elements in a list with the initial dialogue (i.e. with the thesis only), we are able to reconstruct the final one by applying all moves from the traces. Because a move is game-specific rather than dialogue-specific, the trace information contains also a navigation information.

```

data DialogTrace = None |
                    Trace [GameNav] AbstractMove
                    deriving (Read, Show)

```

As we might want to store steps to files or read them from files, it is necessary to use the *abstract move* instead of the *usual* one. The constructor **None** is used if no move has been performed yet (i.e. in the initial state of the dialogue).

3.3. Particle Rules and their Parameters

The data structure of particle rules has already been explained in Section 3.2.6. Let us now have a closer look at the implementation and especially at the rule parameters.

3.3.1. Parameters

There are six different parameter constructors needed to generate new assertions with a particle rule. One of them is used as dummy parameter only. We will see soon what that means.

```

data PRulePar =
  NewRow Row          | — the row of the new assertion
  TargetRefEntity EntityID | — a refining entity as target
  RoleFiller EntityID | — a role-filling entity as target
  AssertLeft          | — Only for attacking And and defending Or
  AssertRight         | — Only for attacking And and defending Or
  AssertLeftOrRight   | — Dummy parameter for And and Or
  deriving (Read, Show, Eq)

```

- **NewRow** specifies the row number of the new assertion that is created by applying the particle rule.
- **TargetRefEntity** is used for attacks only. It specifies the entity for which the attack will be stated and the attacked rival will be forced to defend himself.

- `RoleFiller` is used when attacking *value restrictions* or defending *existential quantifications*. It specifies the role-filler which is going to be accessed.
- `AssertLeft` and `AssertRight` are used when attacking a conjunction or defending a disjunction in order to specify which part of it is attacked or defended (the left or the right).
- `AssertLeftOrRight` is a dummy parameter. It cannot be used to generate an attacking or a defending assertion. It is a *wildcard* for either `AssertLeft` or `AssertRight` and has to be replaced by one of those so that an assertion can be derived.

For example, the resulting list of the `PRuleParRequire`-function of the \sqcap -rule (action is attack) contains the dummy parameter `AssertLeftOrRight` because the function does not know what side is going to be attacked. It must be replaced by either `AssertLeft` or `AssertRight`.

Table 3.1 gives an overview of the particle rules and the parameters that are returned by the `PRuleParRequire` function for the corresponding action. The variable n represents a row-number, x' a refining entity and y a role-filling entity. Note that the function returns 0 for n and $(0,0)$ for x' and y .

Particle Rule	Attack	Defence
<code>andParticleRule</code> (\sqcap)	<code>NewRow n</code> <code>TargetRefEntity x'</code> <code>AssertLeftOrRight</code>	<code>NewRow n</code>
<code>orParticleRule</code> (\sqcup)	<code>NewRow n</code> <code>TargetRefEntity x'</code>	<code>NewRow n</code> <code>AssertLeftOrRight</code>
<code>impParticleRule</code> (\sqsubseteq)	<code>NewRow n</code> <code>TargetRefEntity x'</code>	<code>NewRow n</code>
<code>notParticleRule</code> (\neg)	<code>NewRow n</code> <code>TargetRefEntity x'</code>	--
<code>allParticleRule</code> ($\forall r$)	<code>NewRow n</code> <code>TargetRefEntity x'</code> <code>RoleFiller y</code>	<code>NewRow n</code>
<code>someParticleRule</code> ($\exists r$)	<code>NewRow n</code> <code>TargetRefEntity x'</code>	<code>NewRow n</code> <code>RoleFiller y</code>

Table 3.1.: Particle Rule Parameters

Together with the correct parameters, the particle rule functions are able to create an assertion (attack or defence) on a completely syntactical level.

3.3.2. Implementing a Particle Rule

We will now see how to construct a particle rule. We use the $\exists r$ -rule as an example. The other rules are implemented in a similar way, requiring other input-expressions and parameters, of course.

First, let us define the rule generally.

```
someParticleRule :: PRule
someParticleRule = ("SomeParticle", attackSome, defendSome, reqSome)
```

The constant `someParticleRule` is of the type `PRule`, so it consists of a name, an attacking function, a defending function and a parameter requirement function. The name has now been defined, the rest has to be constructed now. Let us begin with the attacking function.

```
attackSome :: Assertion -> [PRulePar] -> Maybe Assertion

attackSome (Thesis row src player (Some rtype e))
  [(NewRow nRow), (TargetRefEntity ref)]
= Just (Attack nRow (changePlayer player) row
  (Label (LaSome rtype src ref)))

attackSome (Attack row player reference
  (Assert (Some rtype e) _ src))
  [(NewRow nRow), (TargetRefEntity ref)]
= Just (Attack nRow (changePlayer player) row
  (Label (LaSome rtype src ref)))

attackSome (Defence row src player reference (Some rtype e))
  [(NewRow nRow), (TargetRefEntity ref)]
= Just (Attack nRow (changePlayer player) row
  (Label (LaSome rtype src ref)))

attackSome _ _ = Nothing
```


We distinguish four cases by using *pattern matching*⁷. The first three cases react to the different types of assertions to be attacked. The first defines an attack against a thesis, the second one defines a counterattack and the third an attack against a defence. In each case, the first parameter type has to be a `NewRow` parameter followed by a `TargetRefEntity` information. The new assertion contains an attacking label `LaSome` for existing quantifications, wrapped inside `Maybe`'s `Just` constructor.

If none of the three patterns fits the given input, e.g. if the assertion that is about to be attacked does not contain an expression that is an *existential quantification* or if the parameter types are not the required ones, then the fourth case is applied (the one with the wildcards “_ _”). Then `Nothing` is returned, indicating that the rule cannot be applied.

The function `changePlayer` transforms `P` to `O` and `O` to `P`. This is needed so that the player does not attack himself with the new assertion, but his rival.

Now let us look at the defending function:

```
defendSome :: Exp -> Assertion -> [PRulePar] -> Maybe Assertion
defendSome
  (Some rt e)
  (Attack rowA player reference (Label (LaSome rtype _ ref)))
  [(NewRow nRow), (RoleFiller dst)]
| rt == rtype
= Just (Defence nRow dst (changePlayer player) rowA e)
| otherwise = error ("Wrong intention in defendSome. " ++
                    "Relation type of attack does not match!")
defendSome _ _ _ = Nothing
```

As we can only defend against an attack with a corresponding label for this particle rule, we only have to consider two different cases. The second one returns `Nothing` and is used if the function receives invalid input values, i.e. if the first pattern does not fit.

The attacked expression (that is going to be defended) (first input value) must be an existential quantification, too. Together with the attacking assertion and the rule parameters, a defence can be created, but of course only if the role of the attack and

⁷A detailed introduction to pattern matching can be found in [OGS09], p. 50–54.

the role of the attacked assertion are the same. This is checked by the *guards* (indicated by the symbol $|$)⁸.

Last but not least, here is the *parameter-require function*:

```
reqSome :: Action -> [PRulePar]
reqSome Attacking = [(NewRow 0), (TargetRefEntity (0,0))]
reqSome Defending = [(NewRow 0), (RoleFiller (0,0))]
```

This function receives an *action* (i.e. **Attacking** or **Defending**) as input and answers with a list of parameters carrying *dummy values*. For the row number, 0 is returned, while (0,0) represents a dummy entity. Using this function makes it possible to find out which rule parameters are required by the other two rule functions.

3.4. Structural Rules

The structural rules are the most complex part of our program. They are implemented in the module **StructuralRule**. A central function is

```
getPossibleMoves
  :: DialogStep -> [GameNav] -> Player -> Action -> [Move]
```

As inputs it receives

1. a *dialogue* with its *annotations*
2. a *navigation information* indicating the current *game* of the dialogue
3. the *player* whose turn it is next
4. an *action* (**Attacking** or **Defending**)

and generates all possible moves for the player in the game depending on the structural rules and the *action*.

However, this function is not able to cover all structural rules, because some are independent from moves. Those which are not, might make restrictions on possible *particle*

⁸Guards are explained by [OGS09], p. 68, 69.

rule parameters for the next move or even disallow expressions (independently from the parameters, e.g. atomic formulæ).

Table 3.2 shows which rule affects which *domain* and which function covers which rule.

Rule	Function	Affects
SR-ST0 (starting rule)	<code>getBeginningPlayer</code>	<i>Player</i>
SR-ST1 (winning rule)	<code>getWinnerOfGame</code>	<i>Player</i>
SR-ST2 (round-closing rule)	<code>getPossibleMoves</code>	<i>Moves, Parameters</i>
SR-ST3 (strategy-branching)	<i>No function in module</i>	<i>Dialogue</i>
SR-ST4cALC (prime formulæ)	<code>getPossibleMoves</code>	<i>Moves, Parameters</i>
SR-ST5 (no-delaying tactics)	<code>getPossibleMoves</code>	<i>Moves</i>
SR-ST6cALC (entities)	<code>getPossibleMoves</code>	<i>Moves, Parameters</i>
SR-ST7cALC (coupling-rule)	<code>getPossibleMoves</code>	<i>Moves</i>

Table 3.2.: Domains Affected by Structural Rules

Because the strategy branching rule depends on the *interaction mode*, we cover it at the end. The dialogues may run user-controlled or automatically and we do not need any branching if the dialogue is completely controlled by the user.

3.4.1. Rules Affecting the Player

The first function is very simple, as it just returns the first player, i.e. *P*.

```
getBeginningPlayer :: Player
getBeginningPlayer = P
```

The winner of a game is obtained thus:

```
getWinnerOfGame :: DialogStep -> [GameNav] -> Maybe Player
getWinnerOfGame (dialogue, gColl, eColl, aColl, trace) nav
| not (isEmpty (getAllPossibleMoves step nav nextPlayer)) = Nothing
  — more moves possible
| nextPlayer == P = Just O      — P can't move, O wins
```

```

| gameClosed gColl nav = Just P — O can't move and game is closed
| otherwise = Nothing — O can't move but game is still open
where step = (dialogue, gColl, eColl, aColl, trace)
        game = getGame dialogue nav
        lastAssert = getLastAssertion game
        nextPlayer = changePlayer (getPlayerFromAssertion lastAssert)
        winner = getPlayerFromAssertion lastAssert

```

Note that the function `getAllPossibleMoves` calls `getPossibleMoves` twice: once to obtain attacking moves and once for defending moves. Both results are combined to one, sorted depending on the player's strategy (see Section 3.4.5), and returned.

If the game is not finished (i.e. the next player is able to move) then we have no winner and `Nothing` is returned. Otherwise, if the last player who has performed a move is the opponent, she wins. If the last player is the proponent, the game must also be *closed* (checked with function `gameClosed`) in order for him to win. Last possibility: if `O` cannot move and the game is not closed, we have a state that is not defined by **SR-ST1**. We believe that this never happens (it is not proved and therefore still an open problem). Nevertheless, if it does, then `Nothing` is returned.

3.4.2. Rules Affecting Moves

Let us now have a look at the move-creating function `getPossibleMoves`.

```

getPossibleMoves
  :: DialogStep -> [GameNav] -> Player -> Action -> [Move]
getPossibleMoves (dialogue, gameA, entityA, assA, _)
  nav player action
  | action == Attacking = attackResults
  | otherwise          = defenceResults
  ...

```

This is just an abstract declaration. If the action is `Attacking` then attacking moves are returned, otherwise defending moves are obtained which we examine first:

Defending Moves Generally

```

...
where
  — general constants
  rival          = changePlayer player
  game           = getGame dialogue nav
  allRules       = getAllParticleRules  — all particle rules
  — Defending against an unanswered attack
  — all unanswered attacks of rival (rows)
  defendableRows = getDefendableAttacks game rival nav assA
  defendableAsserts = retrieveSortedAssertionsFromGame game
                    defendableRows
  defenceRules = map fromJust
                (map (getSuitableRule allRules game Defending)
                    defendableAsserts)
...

```

So far, we get the *defendable assertions* by obtaining the row numbers of *unlocked* attacks stated by the *rival*, followed by retrieving the corresponding assertions which might be attackable independently from the parameters. These assertions are our basis. Let us consider the function `getDefendableAttacks` later. In the next steps we will filter out those that are disallowed by certain structural rules.

The value `defenceRules` contains a list of suitable particle rules. For each attack, we get exactly one particle rule that is able to create a defence.

```

— entity for which the defence shall take place
defenceEntityIDs
  = map fromJust (map getRefDstFromAttack defendableAsserts)
— rtype of defence (if necessary)
defenceRTypes
  = map getRTypeFromAttack defendableAsserts
— particle rule parameters (if possible)
defenceParams
  = mappedParticleParameters game nav assA player Defending
    defenceEntityIDs defendableRows defenceRTypes defenceRules
— the defence rule (if there is any) with its params and rows
defenceWithAction
  = map (\(x,y,z) -> (Defending, x, y, z)) defenceParams

```

After obtaining the required entities and role-types (needed when defending against a $\forall r$ or $\exists r$) for the defences, we can generate possible rule parameters. For this, we use the function `mappedParticleParameters`. Depending on the player, existing role- and refinement-relations are used or new entities are created. This is quite a complex procedure and we discuss it at the end of this section. However, this function returns particle rules with different possible parameter bindings. The value `DefenceWithAction` contains the resulting *moves*.

```

...
— remove all moves which do not generate new possibilities.
defenceNoRepeat = removeRepetitions defenceWithAction nav assA
— remove all moves which cause atomic problems (only for proponent)
defenceWithoutAtom
  = if (player == P) then
      (filter (\x -> (not (causeAtomProblem x game))) defenceNoRepeat)
    else defenceNoRepeat
— remove all triples where no parameters are given
defenceResults
  = filter (\(a,x,y,z) -> (not (isEmpty y))) defenceWithoutAtom
...

```

Now the last steps: we remove the moves that have already been applied to the attacks. In other words: The same particle rule with the same parameters (with the exception of the `NewRow`-parameter) is not allowed to be applied twice on the same assertion. This reduces the amount of possibilities after repetitions are allowed due to rule **SR-ST5I**.

According to rule **SR-ST4cALC**, P must not state an atomic formula if it has not been stated by O for the same or a more general entity before. So, if it is P's turn to move, we have to check the situation. The function `causeAtomProblem` does this for us. Only those moves will remain which do not cause such an *atom problem*.

Finally, we remove those moves that do not contain any rule parameters. This is just a security measure. Because all particle rules need at least one parameter so that they can do something, a move without parameters does not make any sense.

Attacking Moves Generally

Let us now look at the second part of our move generator, the attacking part.

```

...
— filtered unattacked assertions of rival (row)
— (only those, the player can attack)
attackableRows = getAttackableAssertions game rival nav
                    (gameA, entityA, assA)
— filtered unattacked assertions of rival (assertions)
attackableAsserts = retrieveSortedAssertionsFromGame game attackableRows
— entities of the assertions to be attacked
attackSrcEntityIDs = map getEntityFromAssertion attackableAsserts
— expressions which are going to be attacked
expsToBeAttacked = map fromJust (map assertionToExp attackableAsserts)
— rtype of attacks (if necessary) as maybes
attackRTypes = map getRTypeFromExp expsToBeAttacked
— attacking particle rules
attackRules = map fromJust
              (map (getSuitableRule allRules game Attacking)
                 attackableAsserts )
— rules with parameters and references
attackParameters
  = mappedParticleParameters game nav assA player Attacking
    attackSrcEntityIDs attackableRows attackRTypes attackRules
— the attack rules (if there are any) with their parameters and rows
attacksWithAction
  = map (\(x,y,z) -> (Attacking, x, y, z)) attackParameters
— remove all moves which do not generate new possibilities.
attacksNoRepeat = removeRepetitions attacksWithAction nav assA
— remove all moves which cause atomic problems (only for proponent)
attacksWithoutAtom
  = if (player == P) then
    (filter (\x -> (not (causeAtomProblem x game)))) attacksNoRepeat
  else attacksNoRepeat
— remove all tuples where no parameters are given
attackResults
  = filter (\(a, x,y,z) -> (not (isEmpty y))) attacksWithoutAtom

```

As we see, it is a similar approach with respect to the defending part of the function. The main differences are that we call `getAttackableAssertions` instead of `getDefendableAttacks` and that the parameter creation which is achieved by the already used function `mappedParticleParameters` now involves the action `Attacking` instead of `Defending`. The rest should now be self-explanatory.

Defences in Focus

There are still two functions we have to consider in detail to understand how defending moves are constructed.

```

getDefendableAttacks
  :: Game -> Player -> [GameNav] -> AssertionAColl -> [Reference]
—   Game,   rival,   navigation,   Collection,   defendable attacks
getDefendableAttacks game rival nav aColl
  | isEmpty unanswAttacksRival = []   — no attack left to be defended!
  | otherwise = defendableRowsParticle — Everything that may be defended.
where
— defending player
defender           = changePlayer rival
— all unanswered attacks of rival (rows)
unanswAttacksRival = getUnansweredAttacks game rival nav aColl
— last unanswered attack of rival (row)
lastUnanswRivalRow = last unanswAttacksRival
— last unanswered attack of rival (assertion)
lastUnanswRivalAss = retrieveAssertionFromGame game lastUnanswRivalRow
...

```

We first retrieve the rival's attacks which have not been *answered*. This is done by checking the corresponding assertion annotations. If we follow the intuitionistic version of rule **SR-ST2**, the player may only defend against the *last* attack stated by the rival. We obey and obtain that attacking assertion.

```

— defending rule (as Maybe)
defendRulesMaybe
  = map (getSuitableRule getAllParticleRules game Defending)
        [lastUnanswRivalAss]

```



```

— result(s)
defendableRowsParticle
  = filterListWithMaybe [lastUnanswRivalRow] defendRulesMaybe

```

As the rest should also be applicable if we consider more than one attack (e.g. when using **SR-ST2C**), we use the map function⁹ on the list of attacking assertions. In our case, we have just one (the last attack). Eventually, only those row numbers are returned for which a particle rule exists that might be applicable (syntactically). This is achieved by filtering out those assertions for which `getSuitableRule` returns `Nothing`.

We are now finished with the defending moves except for the creation of rule parameters; still, let us delay this and look at the `getAttackableAssertions` first.

Attacks in Focus

This function is close to `getDefendableAttacks`, but of course other annotations are relevant.

```

getAttackableAssertions
  :: Game -> Player -> [GameNav] -> AnnoDiaGroup -> [Reference]
—   Game, rival, navigation, collections, attackable assertions
getAttackableAssertions game rival nav (gColl, eColl, aColl)
  = attackableRowsParticle — Everything which is not locked
where
  attacker = changePlayer rival — attacking player
—   rival's assertions which are not locked (rows)
  unlockedAndUnprotectedRows
    = getUnlockedAndUnprotectedAssertions game rival nav
      (gColl, eColl, aColl)
—   rival's assertions which are not locked (assertions)
  unlockedAndUnproAssertions
    = retrieveSortedAssertionsFromGame game unlockedAndUnprotectedRows
—   Find out which of the remaining rows can be attacked depending on
—   the particle rules; maybes are returned
  attackRules = map (getSuitableRule getAllParticleRules game Attacking)
    unlockedAndUnproAssertions

```

⁹explained in [OGS09], p. 88–90

```

— filter unattacked and unprotected assertions of rival (row)
attackableRowsParticle
= filterListWithMaybe unlockedAndUnprotectedRows attackRules

```

Note that the type `AnnoDiaGroup` is a tuple holding the three annotation collections: annotations for games, for entities and for assertions.

The function `unlockedAndUnprotectedRows` returns those rows of assertions which are not locked for attacks (i.e. having no *Locked*-annotation) and which are not stated for a *protected* entity. The rest is similar to the procedure applied for defences.

Creating Particle Rule Parameters

The parameter creation functions obey the rule **SR-ST6cALC**. If it is `O`'s turn and she wants to attack, there is no need to do much. We just create a new *refining entity* and if we attack a $\forall r$ then we additionally generate a new *role-filler*. If `O` defends against `P`'s attack of an $\exists r$, she will generate a new role-filler as well.

However, for `P` we have bigger problems. As he is not allowed to generate new entities, he has to use already defined ones. With an attack, he may change to an arbitrary refining entity including the entity for which `O` has stated her assertion that is now attacked. The only restriction: if `P` attacks a $\forall r$, he has to tell `O` which role-filler he wishes her to access. Then of course, this role-filler must exist, i.e. it must have been introduced by `O` before, but this is not the only problem. The role-filler must be accessible from the refining entity, he claims with his attack. In this case, his possibilities are restricted, of course.

There are some functions that cover these cases. It would go beyond the scope of this work to explain them in detail. The function `getPRuleParameters`, which is used by the already mentioned `mappedParticleParameters`, does exactly this. It returns a list of parameter lists that are applicable for a particle rule in a given game by a certain player. We then have to combine all parameter possibilities with all particle rule possibilities and then we have our moves.

3.4.3. Updating Annotations

So far, we have made use of assertion and entity annotations. Let us now see when it is time to add or remove annotations. Whenever an assertion is appended to a game, the annotations have to be updated.

Assertion Annotations

Whenever an assertion stated by player Y is attacked by player X then Y 's assertion will be *locked*, i.e. it receives a `Locked` annotation. The only exception arises, if P attacks a *conjunctive* formula stated by O , because he may then attack once the left and once the right side and only if both sides have been attacked then the formula is locked. However, if O is the attacker, it is enough to attack one side, as he may perform a *back-track* according to rule **SR-ST3**. This asymmetry is not mentioned explicitly in the structural rules, but it appears reasonable if we consider the tableau-rules and the translation performed by [Bla01].

So, let us use the annotation values `LeftAttacked` and `RightAttacked`. If P is attacking the left side of a conjunction, then the attacked assertion receives the `LeftAttacked`-annotation and `RightAttacked` when attacking the right side. As soon as both sides have been attacked, the assertion will eventually be *locked*.

```

getNewAttackAnnotations
  :: Assertion -> [GameNav] -> Row -> AssertionAColl -> [AssertionAnn]
— attacking assertion, nav, row of attacked assert, collection, new ann
getNewAttackAnnotations
  (Attack _ player _ (Label (LaL _ _))) nav row aColl — Attack AND (left)
  | player == O = LeftAttacked : [Locked] — locked immediately
  | otherwise = LeftAttacked : isLockedP — locked after both attacks
  where isLockedP = if (assertionAttackedRight aColl nav row)
                    then [Locked] else []
getNewAttackAnnotations
  (Attack _ player _ (Label (LaR _ _))) nav row aColl — Attack AND (right)
  | player == O = RightAttacked : [Locked]
  | otherwise = RightAttacked : isLockedP
  where isLockedP = if (assertionAttackedRight aColl nav row)
                    then [Locked] else []

```

```
getNewAttackAnnotations (Attack _ _ _ _) nav row aColl = [Locked] — Other
```

The first pattern handles attacks with *L*-labels, the second one attacks with the *R*-label and the final one handles all other attacks which will always cause locks.

Note that attacking assertions which are labels are always locked because they are not attackable.

Now, in order to realise the no-delaying tactic rule (**SR-ST5**), the *Locks* have to vanish again if *O* states an atomic formula, but of course, the locked attacking labels must stay locked.

```
unlockAttackedAssertions
  :: Game -> AssertionAColl -> [GameNav] -> Player -> AssertionAColl
—   game, assertion annotations, navigation, player, new annotations
unlockAttackedAssertions game aColl nav player
= foldl (\coll row -> removeAnnotations (nav,row)
        [Locked, LeftAttacked, RightAttacked] coll)
  aColl noLabelRows
where
  assertions      = getAllAssertions game player
  noLabelAsserts = filter (\assertion -> not (hasLabel assertion ))
                        assertions      — no label
  noLabelRows     = map getRowFromAssertion noLabelAsserts
```

With `foldl`¹⁰, the function `removeAnnotation` is applied to all *relevant rows* using the result as the next input each time, i.e. all assertions that do not have any attacking labels. For each remaining assertion, the annotations `Locked`, `LeftAttacked` and `RightAttacked` are removed (if they are available).

As this function is triggered as soon as *O* states a prime formula, rule **SR-ST5** is covered, so let us move to the next annotation types.

Every defending assertion that is added to a game immediately receives an `IsDefence`-annotation, while the corresponding attack the player defends against, receives an `AnsweredAttack`-annotation so that it is not possible to answer it again. This concerns especially rule **SR-ST2I**.

¹⁰*folding* is explained by [OGS09], p. 93–97.

The last annotation `AppliedMoves` holds a list of *abstract moves*. The annotation is added as soon as the first move is applied on the corresponding assertion. After performing other moves on the same assertion, the list of abstract moves is extended accordingly. With these annotations, we prevent the player from repeating a move with the same parameters (excluding the `NewRow`-parameter) on the same assertion after it has been unlocked.

Entity Annotations

Whenever `P` might be able to access a role-filling entity by performing a *defence*, the entity receives a `PAccessible`-annotation. The problem is that `P` might be prohibited to access the entity because of other rules. However, it is important that `P` would be allowed to do so with a defence sooner or later (**SR-ST7cALC**).

The following function defines if a role-filling entity is *accessible* by a player:

```

checkEntityAccessibility
  :: EntityID -> Player -> DialogStep -> [GameNav] -> Bool
— entity to check, player,
— step with updated assertion annotations and games, nav
checkEntityAccessibility
  destination player (dialogue, gColl, eColl, aColl, trace) nav
= not
  (isEmpty (filter (containsParam (RoleFiller (0,0))) pRuleParameters))
— only if at least one role-filler might be accessed with a defence
— that true is returned
where
  step = (dialogue, gColl, eColl, aColl, trace)
  game = getGame dialogue nav
— all roles that have 'entity' as role-filler
  relevantRoles = filter (\role -> (getRelDst role) == destination)
                  (getAllRelationsInGame game)
— all of O's unanswered attacks
  unansweredAttackRows
    = getUnansweredAttacks game (changePlayer player) nav aColl
  unansweredAttacks
    = retrieveSortedAssertionsFromGame game unansweredAttackRows

```

```

— all attacks that claim an entity which is member of 'roleSources'
relevantAttacks = filter
  (\a -> existsSuitableDestination a relevantRoles) unansweredAttacks
— all particle rules that may be applied to the relevant attacks
defendingPRules
  = catMaybes
    (map (getSuitableRule getAllParticleRules game Defending)
      relevantAttacks )
— rule parameters for the defences against the relevant attacks
pRuleParameters
  = map (\x -> (pRuleGetParRequire x) Defending) defendingPRules

```

We first obtain those entities for which our query entity is a role-filler. In this way, we get our *relevant roles*. Next, we retrieve all of the rival's attacks that have not been answered yet. These are our *unanswered attacks*.

We keep those attacks that have a *suitable destination*, i.e. the claimed refining entity must have the queried entity as role-filler. Now an artful trick: we find out which particle rule can be used to defend these remaining attacks and then we request the *parameters* that are required to perform these defences. If there is at least one parameter-group that contains a **RoleFiller** parameter then the queried role-filling entity is *accessible* by the player.

To explain it again in a more formal way: let us suppose that we want to check if an entity ϵ^* is accessible by player X . Then we first get Y 's unanswered attacks A (denoted by unavailable **AnsweredAttack** annotations). Every attack $a \in A$ claims a refining entity ϵ' for which X is compelled to defend himself. We check only these attacks of A for which there is a role-relation from ϵ' to ϵ^* . Then we look if, according to the particle rules, defences are possible for these remaining attacks and obtain the required particle rule parameters. If there is at least one **RoleFiller** parameter, we can be sure that with a defence, X would be able to access ϵ^* .

So, after solving this problem let us move to the next one. As explained before, an entity ϵ^* is *protected* with respect to rule **SR-ST7cALC** if and only if it is *not accessible* by P with respect to **SR-ST7cALC** and O has been forced to generate ϵ^* with a defence.

```

causeEntityProtected
  :: Assertion -> Game -> [GameNav] -> EntityAColl -> [EntityID] -> Bool
-- added assertion, game, nav.,
-- annotations with updated PAccessible annotations, new entities
causeEntityProtected (Defence row entityID O reference exp)
  game nav eColl newEntities
  | entityPAccessible eColl nav entityID = False — accessible by P
  | otherwise = member entityID newEntities
causeEntityProtected _ _ _ _ _ = False

```

This is quite easy as this function receives the assertion that has just been added to the game and also the *new entities* (i.e. those which have not been part of the game before appending the last assertion) as inputs. An entity may only be protected with a defending move performed by **O**. This is covered by the first pattern. In any other case, *False* is returned (second pattern).

We first check if there is a **PAccessible** annotation for the entity `entityID` used by **O**'s defending assertion. In this case, the entity cannot be protected. If there is no such annotation for `entityID` then we have to find out if it is *new*. This means that it has not existed in the game before the move. Additionally, `entityID` must be a *role-filler*, but we do not test this, because every entity introduced by **O** with a *defence* **must** be a role-filler. All other entities **O** uses in defences are claimed by **P** in his attacks and therefore must already exist.

So, if the entity ϵ^* that is used by **O**'s new assertion is not accessible by **P** and is also new then it is protected.

Game Annotations

Let us keep that matter short. First of all, if a player states a *prime formula*, then it is added to the game's entity information. As we have one list for atoms stated by **O** and one for atoms stated by **P** we are then able to check quite easily if the *intersection* of both lists is empty or if it is not. But obviously, we also have to consider the more general entities.

But it is even easier because of rule **SR-ST4cALC**. So, if *P* states an atomic formula, it must have been stated by *O* for the same or a more general entity before and therefore the game is closed anyway. It then receives a **GameClosed**-annotation.

A game is finished if and only if the player whose turn it is next cannot move anymore, i.e. our function `getAllPossibleMoves` returns an empty list. The game then receives a **GameFinished**-annotation.

With both annotations we can find out easily who is the winner of a game (**SR-ST1**). We have talked about that earlier.

3.4.4. Backtrack

Let us finally consider rule **SR-ST3**. It says that *O* may perform a backtrack to any position she has had a choice, if she loses a game. However, this is not enough for us because we also have to enable *P* to do the same (see Section 2.3.5).

Whenever a player has more than one choice, the corresponding game is removed from the tree and copied as much times as we have choices for the player. To each copy, one possible move/assertion is appended. We then add a node to the dialogue at the position where the game has been, indicating the row number where the choice has been made and the player who has made the choice. Afterwards, all new games are attached to that node. This procedure is illustrated in Figure 3.6. Here, `game1.1` is copied and extended with two of *O*'s possible moves (in row 5) and afterwards replaced by a new node referring to both copies.

Of course, the annotations for the original game have to be copied, too. Further, the *keys* containing the navigation path to indicate the game have to be *shifted* so that we can find our new games afterwards.

Now, backtracking is performed quite easily. The losing player just moves upwards to the node that is labelled by his/her name and enters the next possibility. For example, (regarding the situation illustrated in Figure 3.6) if *P* loses `game1.1.1` then he will move upwards through the tree, looking for a node marked with “*player = P*”. If he does not find such a node, he has lost definitely. But in our example, he finds one and goes directly to `game1.2`. In this way, we perform a *Depth-First Search*.

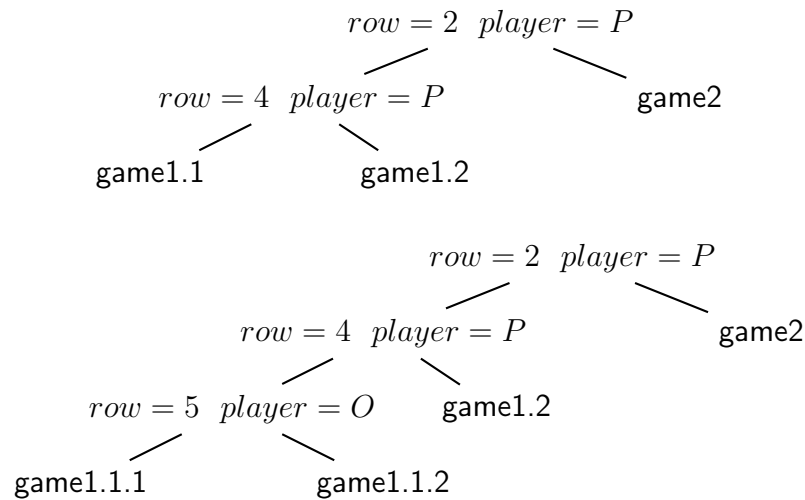


Figure 3.6.: Adding two Possible Moves to a Dialogue at the same Time

3.4.5. Strategies

As explained before, the function `getAllPossibleMoves` obtains both the attacking and defending moves that are possible to be performed by a given player. Now, we have seen in Section 2.3.3 that it is wise for *O* to create new entities whenever possible. This is already implemented, but in addition, it seems to be clever for *P* to follow the most refining entity if he attacks one of *O*'s assertions. Further, while *O* seems to have better chances to move *P* in a worse position if she attacks instead of defending, *P* has better chances when preferring defences (especially because of rule **SR-ST2I**).

Note that if we request attacking moves for *P* with `getPossibleMoves` and one attack is possible for different refining entities, because *P* may move the focus to an arbitrary refining entity, then at the beginning of the returned list, we have those attacks that claim the more general entities (e.g. 1.0), followed by the same attacks for more refining entities (e.g. 1.1, 1.2, ...)

So, as this is not relevant for *O*, in her case, our strategy function only collects the attacking moves and appends the defending ones. For *P*, we do the same but additionally we *reverse* the resulting list. Now the defences are at the beginning and for the attacks which are the same but claim different refining entities, those which claim more refining ones appear before those which claim less refining entities.

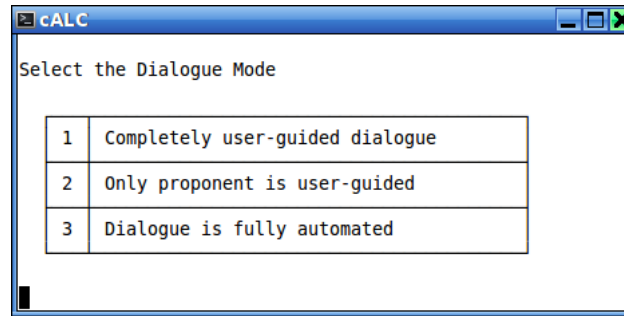


Figure 3.7.: Selecting the Level of Interaction

```

getAllPossibleMoves :: DialogStep -> [GameNav] -> Player -> [Move]
getAllPossibleMoves step nav player
| player == P = defendingMoves ++ (reverse attackingMoves)
| otherwise  = attackingMoves ++ defendingMoves
  where attackingMoves = getPossibleMoves step nav player Attacking
        defendingMoves  = getPossibleMoves step nav player Defending

```

With these strategies, as the players react in a more *clever* way, it is likely that they do not have to backtrack if they have made the correct choice in the first place. The dialogue tree might then be shorter.

3.5. A Simple User-Interface

Finally, let us have a look at a terminal-based user-interface. It is also implemented in Haskell using *monads*¹¹. It is started in *ghci* by typing `menu Nothing` after loading the module `DialogInterface`. Alternatively, you can call the executable binary of the *bin*-folder.

Starting a new dialogue leads us to a screen where we are asked to enter the thesis. It can be written as explained in Section 3.2.2. After that, we choose the level of interaction by typing the corresponding number (Figure 3.7). The first option will start a dialogue which is completely controlled by the user, i.e. the user selects the moves for both players, so no branches are recorded. By contrast, for the second option,

¹¹We use *I/O monads* which are explained by [OGS09], p. 183–188; the I/O system in general on p. 165 ff.

O is controlled by the program. Whenever she loses against the user, she will perform a backtrack. Therefore, all of O's decisions are recorded in the dialogue structure. P must not backtrack.

The last option tests all possible moves until the loser of a game is not able to backtrack, because he/she has no other decision nodes in the dialogue structure that have not been fully exploited. Then it is clear who is the winner.

After selecting the level of interaction, the *option menu* appears (Figure 3.8). Here, we can save a current dialogue to a file, export the tree-structure to a PDF-file or set the *bound*.

The *bound* indicates the maximal length of a game, i.e. the maximum number of assertions assigned to it. If this limit is exceeded, the corresponding branch of the dialogue tree is *pruned* and P will backtrack if running in fully automated mode. A message is also displayed on the screen noting that this has happened.

We type `c` in order to start the dialogue. If we have selected the fully automated mode before, we just have to wait until the program has found the solution. The program then returns to the option menu. In the interactive modes, the current game is shown on the screen and we may decide for the player, whose turn it is next, which move we want him/her to perform.

Figure 3.9 explains how to read the output of the program. It shows a game after some moves have been performed and all the possible moves for the next player to continue. We just select the move we want P to perform by typing the number displayed in the corresponding box. If we type `0`, the dialogue will be interrupted and we return to the option menu. Then, we may select `c` to continue the game or do something else listed in the menu.

At the end, the winner will be displayed.

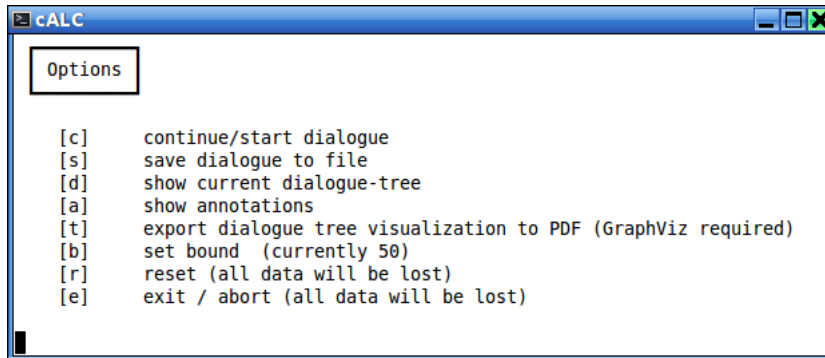


Figure 3.8.: The Option Menu

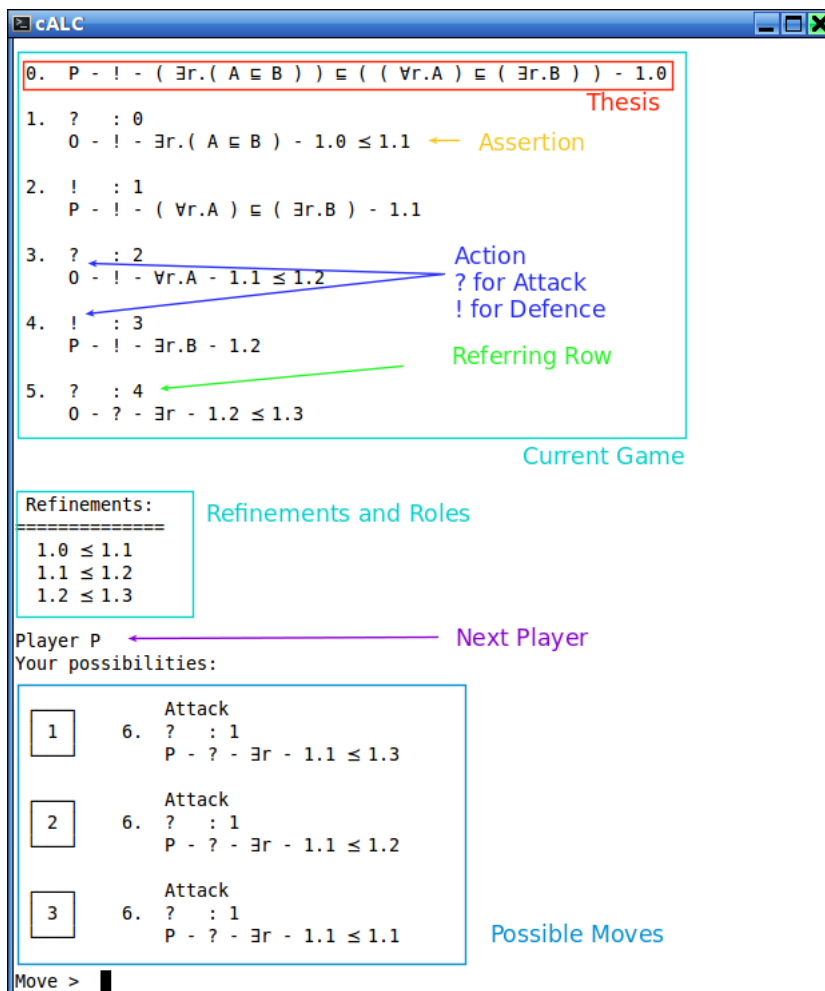


Figure 3.9.: The User is Asked to Select a Move For Player P

4

EVALUATION AND FURTHER STEPS

In this last chapter, we compare the tableau calculus by [Sch] and our dialogue system for *cALC*. We also make some suggestions on how to improve the rules and how to show its soundness.

After that, we discuss the implementation concerning the used programming language, problems that have occurred and possible improvements.

4.1. Testing

Right now, we have no proof showing that the dialogue rules are adequate, i.e. that our ‘algorithm’ is sound and complete. We have performed tests on some simple formulæ. Later, in Section 4.2.4, we make some suggestions on how one might be able to prove soundness and completeness for the dialogues. As we will see, there are some problems showing validity or invalidity of some formulæ, because of resource restrictions concerning our program.

For every formula that is presented here, one can find a dialogue file on the CD that is attached to this work. The files always contain only the initial dialogues with P’s theses. For the dialogues that can be finished, an illustration of the corresponding tree can be found on the CD, too. Whenever we refer to a file, we use the symbol

☞ followed by the file name of the folder `dialogues`. The corresponding PDF files have the same prefix name (`pdf` as suffix) and can be found in the same directory.

4.1.1. Some Basic Intuitionist Features

In classical Description Logic, the formula

$$(C \sqsubseteq D) \sqsubseteq (\neg D \sqsubseteq \neg C)$$

is valid and also in intuitionistic logic (☞ `in01.dlg`). We can test the validity for $c\mathcal{ALC}$ with the tableau calculus with an initial constraint system, e.g. $S_0 = (\mathcal{C}_0, \mathcal{A}_0)$ such that

$$\mathcal{C}_0 = \{x : +(C \sqsubseteq D), \quad x : -(\neg D \sqsubseteq \neg C)\} \quad \mathcal{A}_0 = \{x\}$$

In addition, as C and D may be arbitrary concepts, we can replace them by more complex descriptions, e.g.

$$(\forall r.C \sqsubseteq \exists t.D) \sqsubseteq (\neg \exists t.D \sqsubseteq \neg \forall r.C)$$

(☞ `in02.dlg`). For a dialogue with this expression stated as thesis, the branching is now much more dramatic. This is because P has different choices for the claimed refining entity he may choose for his attacks. Allowing him to repeat attacks after O has stated a prime formula (**SR-ST5I**) even increases the number of possibilities for moves.

As we see in this example, it is not always the best choice for P to follow the most refining entity with his attacks. Let us consider this dialogical game:

	O		P		
1			P - ! - $(\forall r.C \sqsubseteq \exists t.D) \sqsubseteq$ $(\neg \exists t.D \sqsubseteq \neg \forall r.C)$ -	1.0	
2	?1	O - ! - $\forall r.C \sqsubseteq \exists t.D$ -	!2	P - ! - $\neg \exists r.D \sqsubseteq \neg \forall r.C$ -	1.1
3	?2	O - ! - $\neg \exists t.D$ -	!3	P - ! - $\neg \forall r.C$ -	1.2
4	?3	O - ! - $\forall r.C$ -	?3	P - ! - $\exists t.D$ -	1.2 \preceq 1.3
5	?4	O - ? - $\exists t$ -	?2	P - ! - $\forall r.C$ -	1.1 \preceq 1.4
6	?5	O - ? - $\forall r/2.0$ -	?4	P - ? - $\forall r/2.0$ -	1.3 \preceq 1.5
7	!6	O - ! - C -	!6	P - ! - C -	2.0
8	!5	O - ! - $\exists t.D$ -	?8	P - ? - $\exists t$ -	1.4 \preceq <u>1.4</u>
9	!8	O - ! - D -	!5	P - ! - D -	3.0

Table 4.1.: Sometimes it is Better not to Choose the most Refining Entity

In row 8, P claims 1.4 as refining entity (underlined). For him, this is clever, because in row 5, O has already claimed 1.4 so he has to defend with a t -role-filler that is reachable from 1.4. The only way to get it is to make O create it.

According to our implemented strategy, P claims the most refining entities with his attacks. However, O has introduced 1.5 in row 6, so let us look at what would have happened if P had claimed this.

		O		P	
⋮	⋮	⋮		⋮	⋮
8	!5	O - ! - $\exists t.D$	- 1.4	?8	P - ? - $\exists t$ - 1.4 \preceq <u>1.5</u>
9	!8	O - ! - D	- 3.0	?8	P - ? - $\exists t$ - 1.4 \succeq 1.4
10	!9	O - ! - D	- 4.0	!5	O - ! - D - 4.0

In row 9, P may not defend himself against O's attack of row 5, because the t -role-filler's source is entity 1.4. Fortunately, he may repeat his attack, claiming entity 1.4 instead of 1.5 (rule **SR-ST5I** allows him to do so) but with this we have some further steps (and much more branches). To sum up, it is not always the best choice to claim the most refining entity when attacking.

Now let us consider the formula of the beginning the other way round (in03.dlg):

$$(\neg D \sqsubseteq \neg C) \sqsubseteq (C \sqsubseteq D)$$

For classical logic, this is correct, but for $c\mathcal{ALC}$ it is not. Unfortunately, there are so many possibilities for P of how to react in later steps that it is almost impossible to write all games down. Additionally, the games are getting too long so that our implementation prunes the corresponding branches in the dialogue tree after fifty assertions. Using a higher *bound* causes us waiting even longer and the memory runs out.

As we know that the expression is not valid for $c\mathcal{ALC}$, all possibilities have to be checked to show the invalidity in the dialogue. So here, our procedure is not able to handle that formula in a reasonable way, although it is a quite simple one.

Peirce's Law

Peirce's Law is not valid in intuitionist logic, but in classical logic (see [RK05], p. 373). A 'descriptive version' looks thus (\mathbb{S} peirce.dlg):

$$((C \sqsubseteq D) \sqsubseteq C) \sqsubseteq C$$

Unfortunately, there is a *cycle* in this case. The problem is the *no delaying tactic rule* (**SR-ST5I**) which allows repetitions for P after O has stated a prime formula. The dialogue looks thus:

		O		P	
1				P - ! - ((C \sqsubseteq D) \sqsubseteq C) \sqsubseteq C	- 1.0
2	?1	O - ! - (C \sqsubseteq D) \sqsubseteq C	- 1.0 \preceq 1.1	?2 P - ! - C \sqsubseteq D	- 1.1 \preceq 1.1
3	?2	O - ! - C	- 1.1 \preceq 1.2	?3 P - ! - C \sqsubseteq D	- 1.1 \preceq 1.2
4	?3	O - ! - C	- 1.2 \preceq 1.3	?4 P - ! - C \sqsubseteq D	- 1.1 \preceq 1.3
\vdots	\vdots	\vdots		\vdots	\vdots

Table 4.2.: Peirce's Law Causes Cycles

As O creates new refining entities with her attacks, P may follow her because with her attack, O also states a prime formula each time, so that is why the cycle starts.

It is interesting that Peirce's Law terminates in the tableau of [Sch], while other formulæ do not. We talk about cycles in Section 4.2.3 again.

4.1.2. IK1 to IK5

Let us now try the descriptive versions of the intuitionistic axioms IK1 to IK5 (see Section 1.2.3). We will also check FS1¹, FS2², FS4*³ and FS6.

¹The \top in FS1 has been replaced by $\neg(A \sqcap \neg A)$ so that we have no problems with our particle rules.

²We split FS2 into two single expressions and try to show both.

³We only show $(\exists r.A \sqcup \exists r.B) \sqsubseteq \exists r.(A \sqcup B)$ because the other way round is already covered by IK4.

These are our formulæ⁴ with the corresponding file names:

IK1	$\forall r.(A \sqsubseteq B) \sqsubseteq (\forall r.A \sqsubseteq \forall r.B)$	☞ ik1.dlg
IK2	$\forall r.(A \sqsubseteq B) \sqsubseteq (\exists r.A \sqsubseteq \exists r.B)$	☞ ik2.dlg
IK3	$\neg \exists r.(A \sqcap \neg A)$	☞ ik3.dlg
IK4	$\exists r.(A \sqcup B) \sqsubseteq (\exists r.A \sqcup \exists r.B)$	☞ ik4.dlg
IK5	$(\exists r.A \sqsubseteq \forall r.B) \sqsubseteq \forall r.(A \sqsubseteq B)$	☞ ik5.dlg
FS1	$\forall r.(\neg(A \sqcap \neg A))$	☞ fs1.dlg
FS2	$\forall r.(A \sqcap B) \sqsubseteq (\forall r.A \sqcap \forall r.B)$	☞ fs2a.dlg
	$(\forall r.A \sqcap \forall r.B) \sqsubseteq \forall r.(A \sqcap B)$	☞ fs2b.dlg
FS4*	$(\exists r.A \sqcup \exists r.B) \sqsubseteq \exists r.(A \sqcup B)$	☞ fs4b.dlg
FS6	$\exists r.(A \sqsubseteq B) \sqsubseteq (\forall r.A \sqsubseteq \exists r.B)$	☞ fs6.dlg

We have already seen some of these dialogues in Section 2.3.5. Fortunately, for all dialogues, our program terminates and correct results are returned. You find the dialogue trees on the CD. Again, the high branching factor in some examples is striking. This is especially the case for dialogues for which P loses.

4.1.3. Other Formulæ

Let us try some other formulæ. As we can see easily, the expression

$$((A \sqcap C) \sqcup (A \sqcap B)) \sqsubseteq (A \sqcap B)$$

is not valid (even not for classical \mathcal{ALC}). So, P has to lose the dialogue (☞ misc01.dlg) if our system is correct. Indeed, he does, but when starting the program, we see that hundreds of possibilities are generated. But finally, the correct result is returned (O wins). Unfortunately, the tree is so large that GraphViz is not able to create a PDF file. This is the reason why it is not on the CD.

Other valid formulæ such as

$$((A \sqcap C) \sqcup (A \sqcap B)) \sqsubseteq A$$

⁴The \perp in IK3 has been replaced by $A \sqcap \neg A$ so that we have no problems with our particle rules.

([misc02.dlg](#)) or a more complex variant like

$$((\forall r.A \sqcup \forall t.C) \sqcup (\forall r.A \sqcap \forall e.B)) \sqsubseteq \forall r.A$$

([misc03.dlg](#)) are no big problem. The expression

$$((C \sqsubseteq D) \sqcap C) \sqsubseteq D$$

([misc04.dlg](#); adapted from [RR98], p. 5) is also valid and does not cause trouble. By contrast, a more complex version like

$$((\forall r.C \sqsubseteq \exists v.D) \sqcap \forall r.C) \sqsubseteq \exists v.D$$

([misc05.dlg](#)) does, because of the immense amount of possibilities for P. Note that here, we have this problem the first time for a *valid* formula. However, the dialogue can be written down quite easily. The problem is that our program does not perform a good strategy for P. A tree showing the validity by omitting P's 'wrong' decisions can be found on the CD ([misc050.pdf](#)).

4.1.4. Conclusion

Our tests provide some limited confidence that our procedure works correctly but there are often problems concerning the amount of possibilities of moves that can be performed by P, especially if the thesis is not valid. One reason is that P may repeat his attacks whenever O states a prime formula. Another problem concerns *cycles* that might occur because of rule **SR-ST5I**.

4.2. Dialogues in Scope

We now consider some advantages and disadvantages to tableau-based reasoning as it has been shown in Section 1.2.5. Afterwards, we propose some suggestions on how to improve dialogue-based reasoning and also on how one could prove soundness and completeness.

4.2.1. Advantages Compared to the Tableau-Based Algorithm

First of all, the tableau algorithm comes with three rules that we do not cover in the dialogues. These are (\rightarrow_{R-}) that is covered by **SR-ST7cALC**, $(\rightarrow_{\leq+})$ that is covered by P's *attacking particle rules* and $(\rightarrow_{R\perp+})$ that might be needed to close the tableau but as we do not cope with \top and \perp in the dialogues directly, we do not need it here. Further, we also do not need the *active set* because it is also implied by rule **SR-ST7cALC** and its protected entities (though we do not have a proof for this).

As explained before, the global semantics for the dialogues are kept in the structural rules, while the local semantics are declared by the particle rules which are *symmetric*, i.e. they are the same for P and O. By contrast, the tableau only holds one set of rules, but it has rules that are applicable either for *positive* constraints or for *negative* ones. The semantics are hidden inside the preconditions and the active set.

Another point is the independence of structural and particle rules in the dialogues. We can modify them easily and generate new logics by extending or altering them. As stated in Section 2.4, we could for example introduce transitive or reflexive roles by replacing a structural rule.

4.2.2. Disadvantages Compared to the Tableau-Based Algorithm

As we have just seen, the dialogue can have an immense branching factor if all of P's moving possibilities are considered. This often results in a problem, as memory or time resources might be exceeded quite quickly. We do not have this problem in the tableau because it is not important in which order the rules are applied as we always get the same result (proven by [Sch]).

Another big disadvantage are the formulations of the structural rules. As they are informal, it is sometimes hard to interpret them. It is possible to misunderstand them and do the wrong thing. Further, it is harder to check the correctness of informal rules than it is for formal ones.

We have also seen that cycles might occur for some formulæ (e.g. Peirce's Law) that do not cause cycles in the tableau algorithm. This is also a problem that needs to be fixed.

4.2.3. Solving Problems

Branching and Cycles

The main reason for the high branching is rule **SR-ST5I**. The problem is that P may attack O's formulæ again and again, whenever she states a prime. Sometimes it even causes cycles. So, we could adjust the rule so that it makes stricter restrictions on possibilities for repetitions. However, this is a delicate issue, because after the rule has been altered we do not know if the system is 'still' correct.

We have also seen that it might be helpful to improve P's strategy concerning the claimed entities in his attacks. This concerns especially the implementation. Better strategies for P's behaviour should reduce the runtime if the thesis is valid.

For the cycle detection, it might be better to introduce one more rule that describes how cycles can be revealed and that prevents P to perform problematic moves. There are also formulæ for which the tableau algorithm does not terminate, e.g.

$$(\exists r.(C \sqcup D) \sqcap \neg \exists r.C \sqcap \neg \exists r.D) \sqsubseteq \perp$$

(see [MS09], p. 226 ff). [HS99] describe *blocking* mechanisms for transitive and inverse roles in tableau algorithms that might be useful for this. A problem is that we do not have a global view of a dialogue, as we do not have a set representing the ABox that contains all formulæ. That is why it seems that cycle detection is not so easy for dialogical games.

Further Possible Improvements

It could be helpful to embed guidelines about how to deal with TBoxes in the structural rules because that matter is not recorded so far.

Further, whenever we have used \top and \perp , we have always replaced them by $\neg(A \sqcap \neg A)$ and $A \sqcap \neg A$. However, for a formula $C \sqsubseteq \perp$ (with C being an arbitrary concept description), this method is not recommendable. Just imagine that C is a concept description containing A as atomic concept. Otherwise, we would still have some extra moves to dissolve $A \sqcap \neg A$. That is why it would be helpful to have particle rules for \top

and \perp and (if necessary) also structural rules that handle the corresponding (global) semantics.

4.2.4. How to Show Soundness and Completeness

As we know from [Sch], the tableau algorithm for $c\mathcal{ALC}$ is sound and complete. Now, to prove soundness and completeness for our dialogues, it is enough to show that every dialogue can be translated to a constraint system and every constraint system can be translated to a dialogue. Still, this is a complex process because we have to consider every single rule in detail.

We have seen several times that termination is not guaranteed. As suggested before, this might be achieved by altering or adding structural rules.

4.3. The Implementation in Scope

Finally, let us have one more look at the implementation. We first discuss the suitability of Haskell for implementing the dialogue-based reasoner. Afterwards, we give some suggestions on how the implementation can be improved.

4.3.1. Experiences with Haskell

It is remarkable that you can declare functions and especially data types using the *data-* and *type-*constructs very quickly in Haskell. Simple functions can be created intuitively making use of *pattern matching* and *guards*. Further, mapping- and folding functions in combination with *anonymous functions* make it possible to apply functions on a higher amount of data in a fast and easy way.

Concerning testing, it is quite easy to find errors because of the absence of global variables. As we know, the functions return the same output whenever we provide the same input parameters. This makes it easy to find and correct errors.

Now, complex functions could become confusing, e.g. when using long *let-*constructs (see for example the function `getPossibleMoves` from Section 3.4.2).

Problems occur if we alter a type structure after having implemented some functions that cope with it. Then we also have to adjust the functions, i.e. input patterns and output values, and this can take much time, depending on the amount of functions that are using this type.

Implementing user-interfaces is not so easy as it is with many procedural or object-oriented languages, especially if we want to build a graphical user-interface (GUI), because in Haskell, we do not have global variables and therefore we are not able to store GUI-elements separately. For terminal-based interfaces, we have to deal with monads. There are other restrictions compared to procedural languages, e.g. we cannot make use of non-recursive loops.

An interesting issue is that the function `getChar` that receives a character from the user and returns it, works perfectly in the Haskell interpreter GHCi, but after compiling, it leads to the problem that the program continues only after the user has pressed the return-key. But then, he has already entered the second character that is forwarded and accepted by the next occurrence of `getChar`.⁵

However, in my opinion, Haskell has been a good choice for the implementation especially because of the possibilities concerning the data structures and mapping- and folding functions.

4.3.2. Possible Improvements

The implementation can be improved especially by adding new features like a graphical user-interface that visualizes the dialogues or games in a better way. One could also try to improve the flexibility e.g. by making it possible to export particle and structural rules to external configuration files that can be loaded into the program. For example, such a thing is possible with the reasoner described and implemented by [Ehr96].

Apart from this, it seems to be reasonable to add a strategy-function that might be based on heuristics and that sorts possible moves depending on the player and the current game. This would probably improve the performance as it has already been explained in previous sections.

⁵See also discussion:
`why-isnt-my-io-executed-in-order`

<http://stackoverflow.com/questions/2706635/>

Further, it might be possible to parallelize the search process. For example, if P has to make a decision, each new branch of the dialogue tree can be processed by one thread, as every branch contains the complete game and therefore they can be handled independently from each other.

4.4. Final Conclusion

We have introduced a dialogue-based proof system for the constructive Description Logic $c\mathcal{ALC}$. It has some disadvantages to the tableau-based algorithm, especially because of the extreme branching factor. But on the other hand, it emphasizes the semantics and provides another view of the proofs. For example, with respect to an auditing process (see [MS08]), one could regard the proponent as the *auditee* claiming that everything is all right in his business, while the *auditor* is represented by the opponent who wants to check the correctness.

Improving the rules for the dialogues might solve many of the problems we have seen in this chapter, so there is a motivation for further research on this topic.

Appendix

A

RULES

A.1. Particle Rules for $c\mathcal{ALC}$

These are the final particle rules for $c\mathcal{ALC}$ -dialogues which are also used for the implementation.

	\sqcap		\sqcup		\neg
Assert	$X - ! - C \sqcap D - \epsilon$		$X - ! - C \sqcup D - \epsilon$		$X - ! - \neg C - \epsilon$
Attack	$Y - ? - L - \epsilon'$	$Y - ? - R - \epsilon'$	$Y - ? - \sqcup - \epsilon'$		$Y - ! - C - \epsilon'$
Defend	$X - ! - C - \epsilon'$	$X - ! - D - \epsilon'$	$X - ! - C - \epsilon'$	$X - ! - D - \epsilon'$	—

	\sqsubseteq	$\forall r$	$\exists r$
Assert	$X - ! - C \sqsubseteq D - \epsilon$	$X - ! - \forall r.C - \epsilon$	$X - ! - \exists r.C - \epsilon$
Attack	$Y - ! - C - \epsilon'$	$Y - ? - \forall r/\epsilon'^* - \epsilon' \dagger$	$Y - ? - \exists r - \epsilon'$
Defend	$X - ! - D - \epsilon'$	$X - ! - C - \epsilon'^*$	$X - ! - C - \epsilon'^* \ddagger$

Entity description:

- ϵ' is a refining entity of ϵ , i.e. $\epsilon \preceq \epsilon'$.
- ϵ'^* is a r -role-filler of ϵ' .

[†]for any r -filling entity ϵ'^* of ϵ' that Y chooses

[‡]for any r -filling entity ϵ'^* of ϵ' that X chooses

A.2. Structural Rules for $cALC$

These are the final structural rules for $cALC$ -dialogues that are also used for the implementation.

(SR-ST0) (*starting rule*): “Expressions are numbered and alternately uttered by P and O. The thesis is uttered by P. All even numbered expressions including the thesis are P-labelled, all odd numbered expressions are O moves. Every move below the thesis is a reaction to an earlier move with another player label and performed according to the particle and the other structural rules.”

[RK05], p. 372

(SR-ST1) (*altered winning rule*): A dialogical game is closed if and only if it contains two copies of the same prime formula, one stated by X and the other one by Y . Otherwise it is open. A dialogical game is finished if no other move is allowed by the (other) structural and particle rules of the game. The player who stated the thesis wins the game if and only if it is closed and finished and the last move of the game has been performed by this player. The player who started the dialogue as a challenger wins if the game is finished and if he/she has performed the last move of the game.

(SR-ST2I) (*intuitionist ROUND closing rule*): “In any move, each player may attack a (complex) formula asserted by his partner or he may defend himself against the last not already defended attack. Defences may be postponed as long as attacks can be performed. Only the latest open attack may be answered: if it is X 's turn at position n and there are two open attacks m, l such that $m < l < n$, then X may not at position n defend himself against m .”

[RK05], p. 372

(SR-ST3) (*strategy branching rule*): “At every propositional choice (i.e., when O defends a disjunction, reacts to the attack against a conditional or attacks a conjunction), O will motivate the generation of two [dialogical games] differentiated only by the expressions produced by the choice. O will move into a second [dialogical game] [if and only if] he loses the first chosen one. No other move will generate new [games].”

[RK05], p. 372

(SR-ST4cALC) (*formal use of prime formulæ for cALC*): only O may introduce prime formulæ. P cannot use a prime formula O did not utter first for the same entity or an entity which is refined by the entity P wants to make an assertion about. O can introduce a new prime formula anytime he wants, according to the other rules.

(SR-ST5I) (*intuitionist no delaying tactics rule*): “P may perform a repetition of an attack if and only if O has introduced a new prime formula which can now be used by P.”

(original rule (SR-ST5) from [RK05], p. 373)

(SR-ST6cALC) (*formal rule for entities*): O may introduce a new entity anytime the other rules let him do so. P cannot introduce a new entity, and his choices when changing the focus of an individual are restricted to individuals which are refinements, direct role-fillers or role-fillers of refinements of the entity in focus.

(SR-ST7cALC) (*coupling rule for existential quantifications*): If P forces O to introduce a new role-filler by an attack, then the entity introduced by O’s answer is protected against further attacks, i.e. no formulæ stated for that entity may be attacked, unless it *would* be possible for P to access that entity with a defence.

B

NOTATION AND REPRESENTATION OF CONCEPTS

B.1. Infix Notation for Concept Descriptions

This table provides a list of constructs that are used to declare expressions, i.e. concept descriptions, in the implemented reasoner.

Description		Representation
Atomic Concept	A	Atom <Atom>
And / Intersection	$C \sqcap D$	And <Concept> <Concept>
Or / Union	$C \sqcup D$	Or <Concept> <Concept>
Implication / Subsumption	$C \sqsubseteq D$	Impl <Concept> <Concept>
Not / Complement	$\neg C$	Not <Concept>
Value Restriction	$\forall r.C$	All <RType> <Concept>
Existential Quantification	$\exists r.C$	Some <RType> <Concept>

Table B.1.: Infix Notation for Concept Descriptions

This is a recursive definition. Any concept description that can be again constructed

with the definition, is indicated by the token $\langle \text{Concept} \rangle$. The token $\langle \text{RType} \rangle$ represents an arbitrary role-name that is represented by a string of characters (e.g. "navigates"). Atomic concept names are represented by $\langle \text{Atom} \rangle$ what is also represented by a string (e.g. "Captain").

B.2. Concept-Representation in ASCII-Terminals

There are terminals that are not able to display some Unicode Symbols. This table shows how concept descriptions are represented then. They have a modal-logical style:

Concept Description	Representation
A	A
$C \sqcap D$	C \wedge D
$C \sqcup D$	C \vee D
$C \sqsubseteq D$	C \rightarrow D
$\neg C$	~C
$\forall r.C$	[r] C
$\exists r.C$	<r> C

Table B.2.: Alternative Concept-Representation in ASCII-Terminals

Note that the refinement-symbol \preceq is represented by ' \langle '.



CONTENT OF THE CD

C.1. Content

The CD that comes with this work contains the implementation (source-code and binaries) of the dialogue-based reasoner for Microsoft[®] Windows[®] and Linux

Further, some files containing initial dialogues that can be loaded into the reasoner are included together with some representations of the dialogue trees (see Section 4.1 for more details). Finally, a digital version of this work is also available as a PDF-file.

Table C.1 gives an overview of the content and the location where to find it on the CD.

Content	Location
Binaries for Linux	reasoner/linux/bin/
Binaries for Windows [®]	reasoner/win/bin/
Dialogue Files and Trees	dialogues/
Installation Instructions	doc/readme.txt
Module Documentation	doc/modules/
Program Manual	doc/manual.txt
Source-Code for Linux	reasoner/linux/src/
Source-Code for Windows [®]	reasoner/win/src/
Thesis	doc/thesis.pdf

Table C.1.: Content of the CD

C.2. Module Descriptions

Table C.2 shows the Haskell modules that contain the implementation and explains the contents. Note that some of them have already been used for a tableau-based prover.

<i>Module</i>	<i>Description</i>
<code>Annotation</code>	Generic module for annotations; used for tableaux and dialogues
<code>Assertion</code>	Defines all kinds of assertions, i.e. attacks, defences, theses; and simple types as <code>Action</code> , <code>Player</code> , <code>Reference</code> and <code>Row</code>
<code>DialogAnn</code>	Defines and provides access to annotations for assertions, entities and games
<code>DialogInterface</code>	Provides a terminal-based user interface
<code>Dialogue</code>	Handles the tree-structure of a dialogue that holds all games
<code>DialogueEngine</code>	Provides functions for <code>DialogStep</code> and defines backtracks
<code>DialogueImage</code>	Provides functions that transform a dialogue to a string of the DOT-language and that calls DOT for creating illustrations of the dialogue tree
<code>Entity</code>	This module records atomic formulae in list structures that are assigned to a certain entity ID
<code>Expression</code>	This module handles concept descriptions
<code>Game</code>	Manages assertions and relations of a dialogical game
<code>Interface</code>	Generic module for user-interfaces handling both interaction for the tableau-based and dialogue-based proofs
<code>Main</code>	Needed to compile the project
<code>ParticleRule</code>	Contains definitions for all particle rules
<code>StructuralRule</code>	This module defines all structural rules and the data types <code>DialogStep</code> and <code>Trace</code>
<code>Types</code>	Atomic types and other operators
<code>World</code>	Originally used for the tableau-based <i>cALC</i> -prover. Collects expressions according to their enveloping constraints. One list is used for $x : +C$ -constraints, one for $x : -C$ -constraints and the last one for $x : -_R D$ -constraints.

Table C.2.: Module Descriptions

BIBLIOGRAPHY

- [Baa09] Franz Baader. Description Logics. In *Reasoning Web: Semantic Technologies for Information Systems, 5th International Summer School 2009*, volume 5689 of *Lecture Notes in Computer Science*, pages 1–39. Springer–Verlag, 2009.
- [BCM⁺05] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook - Theory, Implementation and Applications*. Cambridge Univ. Press, 2005.
- [BE99] Jon Barwise and John Etchemendy. *Language, Proof and Logic*. CSLI Publications, 1999.
- [BFFF07] Loris Bozzato, Mauro Ferrari, Camilo Fiorentini, and Guido Fiorino. A constructive semantics for ALC. In *Workshop on Description Logics*, pages 219–226, 2007.
- [Bla01] Patrick Blackburn. Modal logic as dialogical logic. *Synthese*, (127):57–93, 2001.
- [BMPS⁺91] Ronald J. Brachman, Deborah L. McGuinness, Peter F. Patel-Schneider, Lori A. Resnick, Lori Alperin Resnick, and Alexander Borgida. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. In *Principles of Semantic Networks*, pages 401–456. Morgan Kaufmann, 1991.
- [BS01] Franz Baader and Ulrike Sattler. An Overview of Tableau Algorithms for Description Logics. *Studia Logica*, 69:5–40, 2001.
- [BvBW07] Patrick Blackburn, Johan F. A. K. van Benthem, and Frank Wolter. *Handbook of Modal Logic*. Studies in Logic and Practical Reasoning. Elsevier, 1st edition, 2007.
- [CDL99] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Reasoning in Expressive Description Logics with Fixpoints Based on Automata on Infite Trees. In *16th Int. Joint Conf. on Artificial Intelligence*, pages 84–89, 1999.

- [Che76] Peter Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
- [DLNN97] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Werner Nutt. The complexity of concept languages. *Information and Computation*, 134:1–58, 1997.
- [Ehr96] Jürgen Ehrensberger. Ein System für Dialogische Logik. Master’s thesis, Friedrich-Alexander Universität Erlangen-Nürnberg, 1996.
- [EMGR⁺01] Hartmut Ehrig, Bernd Mahr, Martin Große-Rhode, Felix Cornelius, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag, 2nd edition, 2001.
- [FFF10] Mauro Ferrari, Camilo Fiorentini, and Guido Fiorino. BCDL: Basic constructive description logic. *Journal of Automated Reasoning*, (44):371–399, 2010.
- [FS80] G. Fischer-Servi. Semantics for a class of intuitionistic modal calculi. *Italian Studies in the Philosophy of Science*, pages 59–72, 1980.
- [GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning - Theory and Practice*. Elsevier, 2004.
- [HRdP10] Edward Hermann Haeusler, Alexandre Rademaker, and Valeria de Paiva. Using intuitionistic logic as a basis for legal ontologies rade. In *Legal Ontologies and Artificial Intelligence Techniques (LOAIT)*, July 2010.
- [HS99] Ian Horrocks and Ulrike Sattler. A description logic with transitive and inverse roles and role hierarchies. *J. of Logic and Computation*, 9(3):385–410, 1999.
- [Kei09] Laurent Keiff. *Dialogical Logic*. Stanford Encyclopedia of Philosophy. <http://plato.stanford.edu/entries/logic-dialogical/>, March 2009. downloaded Dec, 14th 2010.
- [MdP05] Michael Mendler and Valeria de Paiva. Constructive CK for contexts. In L. Serafini and P. Bouquet, editors, *Context Representation and Reasoning*, volume 13 of *CEUR Proceedings*, 2005.
- [Min00] Grigori Mints. *A Short Introduction to Intuitionistic Logic*. Kluwer Academic Publishers, 2000.
- [MS08] Michael Mendler and Stephan Scheele. Constructive Description Logic cALC as a Type System for Semantic Streams in the Domain of auditing. In *Logics for Agents and Mobility (LAM’08)*, 2008.

- [MS09] Michael Mendler and Stephan Scheele. Towards constructive DL for abstraction and refinement. *Journal of Automated Reasoning*, (44):207–243, 2009.
- [MS10] Michael Mendler and Stephan Scheele. Cut-free gentzen calculus for multimodal CK. *Information and Computation*, to appear, 2010.
- [OGS09] Bryan O’Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O’Reilly, 1st edition, 2009.
- [PR05] David Pym and Eike Ritter. A game semantics for reductive logic and proof-search. In *GaLoP 2005: Games for Logic and Programming Languages*, pages 107–123, 2005.
- [Pri01] Graham Priest. *An Introduction to Non-Classical Logic*. Cambridge Univ. Press, 2001.
- [PS86] G. Plotkin and C. Stirling. A framework for intuitionistic modal logics. *Theoretical Aspects of Reasoning about Knowledge*, 1986.
- [RK05] Shahid Rahman and Laurent Keiff. *Logic, Thought and Action*, chapter On How to be a Dialogician, pages 359–408. Springer-Verlag, 2005.
- [RL99] Fethi Rabhi and Guy Lapalme. *Algorithms: A Functional Approach*. Addison-Wesley, 2nd edition, 1999.
- [RR98] Helge Rückert and Shahid Rahman. Dialogische Modallogik (für T, B, S4 und S5). Technical report, Universität des Saarlandes, November 1998.
- [Sch] Stephan Scheele. *Constructive Description Logic*. PhD thesis, Otto-Friedrich Universität Bamberg. To Appear.
- [Sim94] Alex K. Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.
- [SSS91] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.
- [Wij90] Duminda Wijesekera. Constructive Modal Logic I. *Annals of Pure and Applied Logic*, 50:271–301, 1990.

Ich erkläre hiermit gemäß § 27 Abs. 2 APO, dass ich die vorstehende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum

Unterschrift