Master 2 Research

Handling Theories in Logic Functors for Recomposing Description Logics

Étienne André

July 4, 2007

Superviser : Sébastien Ferré

LIS Team

Institut de Recherche en Informatique et Systèmes Aléatoires
Rennes – France

Contents

1 Introduction **4**

2 Description Logics **6**
2.1 Syntax 6
2.2 Semantics 8
2.3 Reasoning in description logics 8
2.4 Extensions 10
2.5 Complexity 12
2.6 Discussion 12

3 Logics and Logic Functors **14**
3.1 Logics 14
3.1.1 Syntax and Procedures 15
3.1.2 Semantics and Properties 16
3.1.3 Logics 18
3.2 Logic Functors and their Composition 18
3.2.1 Definitions 19
3.2.2 Reconstructing the Description Logic $\mathcal{ALC}$ 21

4 Theories in Logic Functors **23**
4.1 Redefinition of Logic Functors 23
4.2 The case of Terminological Theories 25
4.2.1 Syntax 25
4.2.2 Semantics 25
4.3 *Taxo/0/1*: taxonomies 26
4.3.1 Syntax 27
4.3.2 Semantics 27
4.3.3 Operations 27
4.3.4 Properties 28
4.4 *Domain/1/1*: taxonomies on concrete domains 31
4.4.1 Syntax 31
4.4.2 Semantics 31
4.4.3 Operations 32

4.4.4 Properties 32
4.5 *Prop/1/1*: terminologies 37
4.5.1 Syntax 37
4.5.2 Semantics 37
4.5.3 Operations 37
4.5.4 Properties 38

5 Application to Description Logics **48**
5.1 Reconstructing $\mathcal{ALC}$ with a Terminology 48
5.2 Handling Bibliographic References in XML 49
5.3 Implementation 51

6 Conclusion and Future Works **52**
6.1 Improving the expressivity of Prop 52
6.2 Recomposing the Description Logic $\mathcal{SHOIQ}$ 53
6.3 Handling More General Theories 54

Chapter 1

Introduction

The number and the diversity of electronic documents is dramatically increasing. Thus, how to optimize organization and research in a set of documents? A common solution is to use a query language, e.g. lists of keywords in the case of search engines, or SQL in the case of databases. Logical formalism can be used to express queries and describe data on the one hand, and to define deduction relations between queries and answers on the other hand [Rij87].

Several information processing domains have logic-based components in which logic plays a crucial role: e.g., information systems and information retrieval or logic-based programming [Llo87, MS98]. This is also the case of Logical Information Systems (LIS), in which we use logic to represent object descriptions and queries, to answer queries, and to compute automatically a flexible navigation structure [FR04]. Querying and navigation in LIS is based on the ability to decide whether an object description is *subsumed* by a query or a navigation link. This notion of subsumption is the same as in *description logics* [CLN98], where the subsumption is done with respect to an ontology.

In this document, we will refer to an ontology as a knowledge base describing a specific domain with axioms. Ontologies can be seen as kinds of theories, which are well-known in logics. Terminologies are a kind of ontology where terms are defined by an expression (e.g. a father is a man having at least one child). Taxonomies are simple terminologies allowing to state that terms are more general or specific than other ones (e.g. a research report is a kind of publication).

Several roles take part in the development of logic-based softwares. The Logic Expert develops a theoretical framework, the Application Designer implements the application w.r.t. the needs, the Ontologist develops a domain specific knowledge, and the End User uses the application. Of course, some roles can be incarnated by software (for example, the End User can be a software), and a same person can have several roles (for example, an

Application Designer can also be the Theorist). However, in the general case, a Logic Expert will not develop the Application, and the Application Designer will not have huge theoretical knowledge and will thus not design the theory. As a consequence, we need generic tools allowing these various roles to be separated.

Logic functors allow to build a logic in an easy and generic way, in a similar way as Lego pieces. Thus, the Theorist can develop the logic functors (which requires knowledge in logics), and the Application Designer can compose them (which does not require knowledge in logics) and design its application. However, logic functors do not presently allow the End User to select or define the knowledge base he wants to use. Thus, we show in the following how to add the notion of theory to logic functors, and show as an example how to compose description logics with logic functors, including a theory. The implementation LogFun[1] of logic functors has been modified in order to allow the use of theories.

Description logics are presented in Chapter 2. Logics and logic functors are presented in Chapter 3, and we show in particular how the description logic $\mathcal{ALC}$ can be recomposed with logic functors (Section 3.2.2). Then, we present our main contributions, that is the adaptation of logic functors in order to allow the use of a theory (Chapter 4), and we finally show in Chapter 5 how $\mathcal{ALC}$ can be recomposed with logic functors, allowing the use of a terminology. We then conclude in Chapter 6 by giving orientations for future works.

[1] http://www.irisa.fr/LIS/ferre/logfun/

Chapter 2

Description Logics

Description Logics (DL) are a class of knowledge representation languages which are widely used in information systems to represent the terminological knowledge of various domains. The first DL-based system was KL-ONE, designed in 1985 [BS85]. Famous later DL systems are LOOM (1987) or CLASSIC (1991). Today, description logics are widely used in the Semantic Web, and the $\mathcal{SHOIQ}$ description logic has been recognized as a standard by the W3C to support the OWL-DL and OWL-Lite sub-languages of the Web Ontology Language (OWL). Description logics are also close to our aim in Logic Information Systems as they can be used to describe objects, express queries, and check if objects match queries.

This chapter is based on Amedeo Napoli's introduction to description logics [Nap97].

2.1 Syntax

Description logics are a family of logics allowing to model knowledge with *descriptions*, which can be *concepts*, *roles* or *individuals*. A concept represents a set of individuals, and a role describes a binary relation between individuals. Individuals are *instances* of concepts: definitions of roles and concepts form the *terminology (TBox)*, whereas definitions of individuals are called *facts* or *assertions (ABox)*. The union of a TBox and a ABox is called a *knowledge box (KB)*.

Primitive concepts and roles are introduced by the symbol $\dot{\leq}$, whereas defined concepts and roles are introduced by the symbol $\doteq$.

The syntax[1] of the basic description logic $\mathcal{AL}$ is given on figure 2.1. `C` represents a concept expression, `A` represents a primitive or a defined concept

[1] Various forms of syntax have been used in Description Logic Knowledge Representation Systems. We present here the German syntax, which has been widely adopted for the theoretical discussion [Hor97].

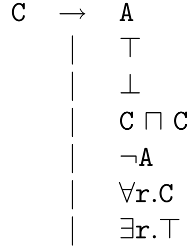

Figure 2.1: Syntax of description logic $\mathcal{AL}$

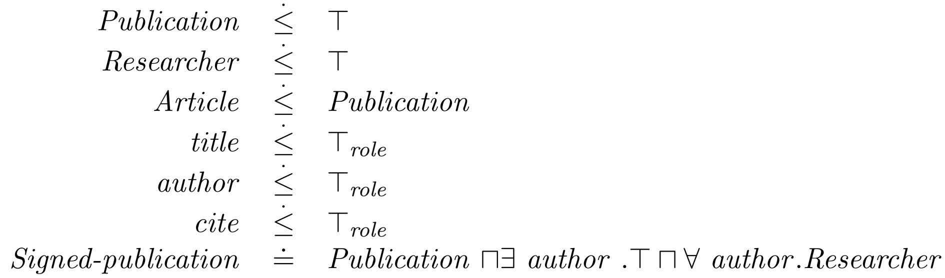

Figure 2.2: Examples of primitive concepts, primitive roles and defined concepts in logic $\mathcal{AL}$

name, and r represents a primitive role name. The $\top$ *(top)* concept represents the most general concept, and includes all possible individuals. The $\bot$ *(bottom)* concept represents the most specific concept, and includes no individual. The constructor $\sqcap$ defines concept conjunction. The constructor $\neg$ represents the negation of concept names. The universal quantification $\forall$r.C limits the codomain of role r; in other words, it considers only the individuals linked through the role r to the individuals that are instance of the concept C. The restricted existential quantification $\exists$r.$\top$ insures at least one individual is linked to another one through the role r. Note that full existential quantification ($\exists$r.C) is not defined in $\mathcal{AL}$.

Various forms of definitions of concepts and roles are allowed. Some logics allow only single terms on the left side, some other allow full expressions on the left side. Some logics allow recursively defined terms, where a term appears on the left and on the right side.

Figure 2.2 presents an example of *TBox* containing primitive concepts, primitive roles and defined concepts in logic $\mathcal{AL}$. *Publication* and *Researcher* are primitive concepts, and are subsumed by $\top$. *Article* is also a primitive concept, and is subsumed by the concept *Publication. title*, *author* and *cite* are primitive roles that associate an article respectively to a title, an author and another article, and are subsumed by $\top_{role}$, which represents the most general role. The concept *Signed-publication* is a defined concept including the set of articles whose all ($\forall$) authors are researchers, and ($\sqcap$) containing

article($\mathtt{WKW06}$)
Researcher(*Wong Kar-Wai*)
title ($\mathtt{WKW06}$, *In the Mood for Description Logics*)
author($\mathtt{WKW06}$, *Wong Kar-Wai*)

Figure 2.3: Examples of assertions in logic $\mathcal{AL}$

$$\begin{array}{rcl} \mathtt{A}^{\mathcal{I}} & \subseteq & \Delta^{\mathcal{I}} \\ \top^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \\ \bot^{\mathcal{I}} & = & \emptyset \\ (\mathtt{C} \sqcap \mathtt{D})^{\mathcal{I}} & = & \mathtt{C}^{\mathcal{I}} \cap \mathtt{D}^{\mathcal{I}} \\ (\neg \mathtt{A})^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus \mathtt{A}^{\mathcal{I}} \\ (\forall \mathtt{r}.\mathtt{C})^{\mathcal{I}} & = & \{x \in \Delta^{\mathcal{I}} \mid \forall y : (x,y) \in \mathtt{r}^{\mathcal{I}} \Rightarrow y \in \mathtt{C}^{\mathcal{I}}\} \\ (\exists \mathtt{r}.\mathtt{C})^{\mathcal{I}} & = & \{x \in \Delta^{\mathcal{I}} \mid \exists y : (x,y) \in \mathtt{r}^{\mathcal{I}} \wedge y \in \mathtt{C}^{\mathcal{I}}\} \end{array}$$

Figure 2.4: Semantic of logic $\mathcal{AL}$

at least one ($\exists$) author.

Figure 2.3 presents an example of *ABox* in the logic $\mathcal{AL}$. We define an article $\mathtt{WKW06}$, a researcher *Wong Kar-Wai*, and associate the title *In the Mood for Description Logics* to the article, and the author *Wong Kar-Wai* to the article $\mathtt{WKW06}$.

2.2 Semantics

In a similar way as in classical logic, a semantic is associated to concepts and roles descriptions. However, in classical logic, formulas are interpreted by truth values, whereas the semantics of description logics is based on sets of individuals. Intuitively, concepts in description logics are interpreted as sets of individuals, and roles are interpreted as binary relations between individuals.

The interpretation function $\cdot^{\mathcal{I}}$ of an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ maps every concept to a subset of $\Delta^{\mathcal{I}}$ following the rules of figure 2.4.

Another way to express semantics of description logic $\mathcal{AL}$ is to translate it into first-order logic (FOL) using the operator π defined in table 2.5 [KM06]. π_x is obtained by simultaneously substituting in the definition of π_y all $x_{(i)}$ for all $y_{(i)}$, respectively, and π_y for π_x. X is a meta variable and is substituted by the actual variable. Thus, description logics can be seen as a subset of first-order logic, using only two variable names.

2.3 Reasoning in description logics

Once concepts and roles are defined, and assertions are made, we are now interested in *reasoning* in description logics. For example, we know that

$$\begin{aligned}
\pi_y(\top, X) &= \top \\
\pi_y(\bot, X) &= \bot \\
\pi_y(A, X) &= A(X) \\
\pi_y(\neg A, X) &= \neg\pi_y(A, X) \\
\pi_y(C \sqcap D, X) &= \pi_y(C, X) \land \pi_y(D, X) \\
\pi_y(\exists r.C, X) &= \exists y : r(X, y) \land \pi_x(C, Y) \\
\pi_y(\forall r.C, X) &= \forall y : r(X, y) \Rightarrow \pi_x(C, Y)
\end{aligned}$$

Figure 2.5: Semantics of $\mathcal{AL}$ by mapping to FOL

`WKW06` is an article, that an article is a publication, and we want to infer that `WKW06` is a publication. Therefore, we need mechanisms to deduce new facts. The most important reasoning mechanisms in description logics are the following ones:

Subsumption Given a knowledge box, the *subsumption test* organizes concepts and roles by generality: intuitively, a concept `D` is subsumed by a concept `C` (written $\mathtt{D} \sqsubseteq \mathtt{C}$) with respect to the KB if `C` is more general than `D`, that is if the set of individuals represented by `C` contains the set of individuals represented by `D` for all interpretations. Thus, $\mathtt{C} \sqsubseteq \mathtt{D}$ iff $\forall i \in I : \mathtt{C}^{\mathcal{I}} \subseteq \mathtt{C}^{\mathcal{I}}$.

Satisfiability The test of satisfiability of a concept `C` w.r.t. a KB checks whether `C` admits instances, e.g. whether there exists at least one interpretation $\mathcal{I}$ such that $\mathtt{C}^{\mathcal{I}} \neq \emptyset$.

Satisfiability of a knowledge box The test of satisfiability of a knowledge box Σ checks whether Σ has a model $\mathcal{I}$, e.g. whether every assertion is satisfied by $\mathcal{I}$.

Instantiation The test of instantiation checks whether an individual `x` is an instance of a concept `C` for a knowledge box Σ, which is written $\Sigma \models \mathtt{C(x)}$. `x` is instance of `C` iff $\forall i \in I : \mathtt{x}^{\mathcal{I}} \in \mathtt{C}^{\mathcal{I}}$.

These four operations are not independent of each other. They all can be reduced to the test of satisfiability. Indeed:

- $\mathtt{C} \sqsubseteq \mathtt{D} \iff \mathtt{C} \sqcap \neg \mathtt{D}$ is not satisfiable,
- $\Sigma \models \mathtt{C(a)} \iff \Sigma \cup \{\neg \mathtt{C(a)}\}$ is not satisfiable.

The translation of subsumption and facts into first-order logics is given on figure 2.6. C and D are concept names, whereas R and S are role names.

$$\begin{array}{rcl}\pi(C \sqsubseteq D) & = & \forall x : \pi_y(C,x) \Rightarrow \pi_y(D,x) \\ \pi(R \sqsubseteq S) & = & \forall x,y : R(x,y) \Rightarrow S(x,y) \\ \pi(C(a)) & = & \pi_y(C,a) \\ \pi(R(a,b)) & = & R(a,b)\end{array}$$

Figure 2.6: Semantics of axioms and subsumption by mapping to FOL

Ext	**Syntax**	**Semantics**	**Translation to FOL**
$\mathcal{C}$	$\neg\mathtt{C}$	$\Delta^{\mathcal{I}} \setminus \mathtt{C}^{\mathcal{I}}$	$\neg\pi_y(C,X)$
$\mathcal{U}$	$\mathtt{C} \sqcup \mathtt{D}$	$\mathtt{C}^{\mathcal{I}} \cup \mathtt{D}^{\mathcal{I}}$	$\pi_y(C,X) \vee \pi_y(D,X)$
$\mathcal{E}$	$\exists\mathtt{r}.\mathtt{C}$	$\{x \in \Delta^{\mathcal{I}} \mid \mathtt{r}^{\mathcal{I}}(x,C) \neq \emptyset\}$	$\exists y : r(X,y) \wedge \pi_x(C,y)$
$\mathcal{N}$	$\leq \mathtt{n}\ \mathtt{r}$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\mathtt{r}^{\mathcal{I}}(x,\top) \leq \mathtt{n}\}$	$\exists y_1,\ldots,y_n : \bigwedge_{1\leq i\leq j\leq n} y_i \not\approx y_j$ $\wedge \bigwedge_{1\leq i\leq n}(\mathtt{r}(X,y_i))$
	$\geq \mathtt{n}\ \mathtt{r}$	$\{x \in \Delta^{\mathcal{I}} \mid \sharp\mathtt{r}^{\mathcal{I}}(x,\top) \geq \mathtt{n}\}$	$\forall y \exists y_1,\ldots,y_n : \mathtt{r}(X,y)$ $\Rightarrow \bigvee_{1\leq i\leq n} y \approx y_i$
$\mathcal{R}$	$\mathtt{r} \sqcap \mathtt{s}$	$\mathtt{r}^{\mathcal{I}} \cap \mathtt{s}^{\mathcal{I}}$	$\pi_y(\mathtt{r}(a,b)) \wedge \pi_y(\mathtt{s}(a,b))$

Figure 2.7: Examples of extensions of logic $\mathcal{AL}$

It is thus easy to recognize that a short article citing no description logic (DL) article is subsumed by an article whose all cited DL articles are interesting, which is written as follows:

$$\begin{array}{rl} & \mathit{Article} \sqcap \mathit{Short} \sqcap \forall \mathit{cite}.(\neg \mathit{DL\text{-}Article}) \\ \sqsubseteq & \mathit{Article} \sqcap \forall \mathit{cite}.(\mathit{DL\text{-}Article} \rightarrow \mathit{Interesting})\end{array}$$

Note that there is no need here to consider the *TBox* of figure 2.2 to prove this goal.

However, in order to prove that the concept *Publication* subsumes the concepts *Article* and *Signed-publication*, we need to take into account the *TBox*, because nothing tells *a priori* that an article is a kind of publication.

2.4 Extensions

The logic $\mathcal{AL}$ defined above represents the minimal basis in description logics. It can be improved by several constructors if needed. Figure 2.7 presents the extensions bringing the negation of non necessarily primitive concepts, union of concepts, full existential quantification, as well as conjunction and cardinality on roles. The notation $\sharp\mathtt{r}$ corresponds to the cardinality of the codomain of role $\mathtt{r}$. By combination of these constructors, it is possible to define new description logics. Note that some extensions imply others: for example, disjunction ($\mathcal{U}$) and typed existential quantification ($\mathcal{E}$) appear in every logic implementing negation ($\mathcal{C}$), and *vice versa*. As a consequence, $\mathcal{ALUEN}$=$\mathcal{ALCN}$.

The description logics made by the composition of the basis $\mathcal{AL}$ and

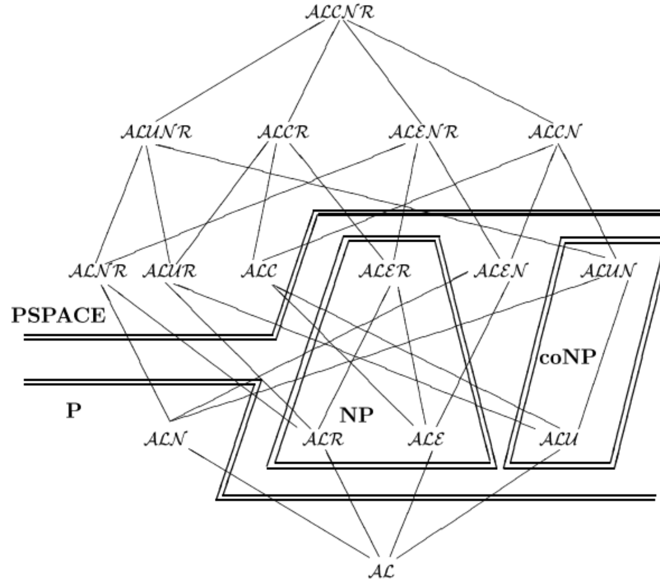

Figure 2.8: Complexity of unsatisfiability and subsumption in $\mathcal{AL}$-languages

$$
\begin{array}{lcl}
\textit{Anonymous-article} & \doteq & \textit{Article} \sqcap \neg \exists \ \textit{author} \\
\textit{Shared-article} & \doteq & \textit{Article} \sqcap \geq 2 \ \textit{author} \\
\textit{DL-Article} & \doteq & \textit{Article} \sqcap \exists \ \textit{cite} \ . \ \textit{DL-Article} \ )
\end{array}
$$

Figure 2.9: Examples of defined concepts in logic $\mathcal{ALCN}$

the described extensions form a lattice. To give an idea, we present on figure 2.8 the description logics made by the composition of the basis $\mathcal{AL}$ and the extensions $\mathcal{C}$, $\mathcal{U}$, $\mathcal{E}$, $\mathcal{N}$ and $\mathcal{R}$ [DLNN97]. We note that, although only 5 extensions are involved, we can compose 16 different logics. However, there exists more extensions, and we can thus compose much more different logics.

To give an example of the expressivity of the possible extensions, we present on figure 2.9 some axioms defined in the $\mathcal{ALCN}$ logic. An anonymous article is an article without any author, a shared article is an article containing at least ($\geq$) 2 authors, and a DL Article is an article citing other DL Articles. Note in this latter case that it is possible to define concepts in a recursive way.

Description logics are not limited to the $\mathcal{AL}$ family. In particular, the description logics $\mathcal{SHOIQ}$ is widely used especially for the semantic web. In the name $\mathcal{SHOIQ}$, $\mathcal{S}$ stands for $\mathcal{ALCR}^+$, where $\mathcal{R}^+$ is the transitivity on roles, $\mathcal{H}$ stands for the role hierarchy, $\mathcal{O}$ allows to use individuals in the concepts, $\mathcal{I}$ stands for inverse roles and $\mathcal{Q}$ stands for the qualified cardinality restrictions on roles. Transitive roles ($\mathcal{R}^+$) allow to consider axioms of the form $\textit{ancestor} \doteq \textit{parent}^+$, e.g. a parent is an ancestor, and the parent of an

ancestor is an ancestor as well. Inverse roles ($\mathcal{I}$) allow to consider axioms of the form *child* $\doteq$ *parent*$^-$. Intuitively, child is the inverse relation of parent. Qualified cardinality restrictions on roles ($\mathcal{Q}$) allows to consider axioms of the form $\geq 2 child.Researcher$, meaning "more than 2 children being researchers".

2.5 Complexity

The logics of the $\mathcal{AL}$ family presented above are all *decidable*, but have different complexities. Thus, it is interesting to reason with as few extensions as necessary, in order to limit the complexity, which is important to take into account when reasoning on huge knowledge data. The complexity is a function of the chosen extensions: figure 2.8 shows the complexity of relations of non-satisfiability and subsumption in description logics of family $\mathcal{AL}$. To summarize, the complexity is polynomial for $\mathcal{AL}$ and $\mathcal{ALN}$, becomes NP-complete or co-NP-complete when adding some extensions, and becomes PSPACE-complete when adding several extensions. $\mathcal{SHOIQ}$ is NExpTime-complete, but some algorithms "behave well" in many typically encountered cases [HS05].

2.6 Discussion

We thus have a big description logics family. The difference of complexity between various description logics implies to choose the less complex description logic w.r.t. the extensions needed, so that the complexity is as low as possible. For every different application, we may need a different description logic. However, the Application Designer may not have enough knowledge in logic to build a new description logics for every application. Thus, we need here generic tools allowing to implement easily a description logic in a generic way, so that an Application Designer can build it without deep knowledge in logic.

We thus could imagine a generic prover where only the needed extensions (e.g. $\mathcal{U}$, $\mathcal{N}$, etc.) would be implemented, as in [dCFG$^+$01]. Indeed, Lotrec is a generic tableau prover for modal and description logics allowing to add the wished tableau proofs rules corresponding to the selected extensions. One advantage is that we can have a prover implemented almost for free, with a complexity adjusted to the needs. However, we note here that, although we can add a new extension almost for free, nothing tells us that the composition with the other extensions will be proved valid; on the contrary, only some compositions are proved valid *a priori* and no guarantee is given for the other ones.

We would need a composition where every extension could be proved independently and composed in a generic way, with an illimited number

of potential combinations. If all combinations are not valid, it would be interesting to have an automatic prover checking if a combination is valid or not, and if not saying which property is missing in order to have the combination valid.

Chapter 3

Logics and Logic Functors

If an Application Designer has to define a customized logic by the means of composing primitive components, these components should be of high-level, so that the resulting logic subsystem can be proved correct. Indeed, if the primitive components are too low-level, proving the correctness of the result is similar to proving the correctness of a program. Thus, we decide to define logical components that are very close to logics themselves. We show in this chapter how a logic can be recomposed in a simple and generic way with these high-level components, which are called logic functors.

3.1 Logics

Logics can be used in applications to express object descriptions, queries and to match objects to queries. For example, logical information systems are based on logics, to express in the same way object descriptions and queries. Thus, checking an object is an answer to a query corresponds to checking if its description is subsumed by the query formula, where the subsumption is defined in an similar way as in description logics.

More formally, a logic is the association of an abstract syntax, a semantics, an implementation, and a type made of a set of properties. Syntax and procedures are what makes up the *signature* of logics, on which all applications must agree in order to satisfy the embeddability of logics.

Now, it is important to attach a semantics to a logic in order to give a meaning to formulas and procedures. Like in description logics, formulas are not interpreted by truth-values, but by sets of interpretations. Note that these interpretations play the same role as individuals in description logics. The semantics is also crucial when defining the properties of procedures, such as the consistency or completeness of the decision procedure.

3.1.1 Syntax and Procedures

Syntax is often defined from a signature of atoms and connectors. However, in order to allow for concrete domains and concrete structures, we choose to have a more general definition of a syntax.

Definition 1 (Syntax) *Given a grammar based on an alphabet* Σ*, an* abstract syntax AS *is the language of* formulas *defined by the grammar as a subset of* Σ^**.*

However, we will reason on the abstract syntactic tree based on the syntax.

Two subsets of syntax are distinguished: *TELL* is the set of formulas used in descriptions, and *ASK* is the set of formulas used in queries. No constraint is put on these two subsets, except they should not be empty: e.g., they can be disjoint or not, they can be equal to AS or not. When they are not defined explicitly, they are equal to the full set of formulas AS. The distinction between *TELL*-formulas and *ASK*-formulas is important for some logic properties, as shown in Section 3.1.2.

Procedures define the interface through which a logic can be used by programs. There is no constraint, except for their types, on what procedures can do. For instance, in the following list, $\top$ is supposed to give the most general formula but nothing forces it to; and its type shows it can be undefined. The proof of some properties is then necessary to ensure a procedure has the intended meaning (see Section 3.1.2).

- $\sqsubseteq$: $AS \times AS \rightarrow bool$ is the decision procedure for the *subsumption* test,
- $\top : AS \uplus \{undef\}$ may give the *top*,
- $\bot : AS \uplus \{undef\}$ may give the *bottom*,
- $\sqcap : AS \times AS \rightarrow AS \uplus \{undef\}$ may give the *conjunction* of 2 formulas,
- $\sqcup : AS \times AS \rightarrow \mathcal{P}(AS)$ gives the *disjunction* of 2 formulas as a set of formulas.

$\sqsubseteq$ represents the subsumption test. Procedures $\top$, $\bot$, $\sqcap$, $\sqcup$ are all defined on the syntax of some logic, and three of them may be (partially) undefined. But they do not necessarily match connectives of the logic, because the connectives of a logic are part of the syntax: e.g., $[0,5] \sqcap [3,6] = [3,5]$ in the logic of integer intervals. Similarly, the syntax and the semantics may define negation or quantifiers, though they are absent from the set of procedures: e.g., in description logics.

3.1.2 Semantics and Properties

Formulas are given a semantics by relating them to *interpretations*, in a similar way as in description logics. These interpretations can be understood as possible objects or individuals, and reciprocally, formulas can be understood as expressing a pattern on interpretations.

Definition 2 (Semantics) *Given a syntax AS, a* semantics *S based on AS is a pair $(I, \models)$, where I is a set of* interpretations*, the* interpretation domain*, and $\models \subseteq (I \times AS)$ is a* satisfaction relation *between interpretations and formulas: $i \models f$ reads "i is a* model *or an* instance *of f". For every formula $f \in AS$, $M(f) = \{i \in I \mid i \models f\}$ denotes the set of all models or instances of formula f.*

Properties are used to relate the behaviour of procedures w.r.t. the semantics of their arguments and results. It is important to understand that, unlike the classical approach, procedures are not given a semantics by definition. For instance the conjunction $\sqcap$ can be only partially defined, and it is free to implement any algorithm. More specifically, nothing tells *a priori* that $M(f \sqcap g) = M(f) \cap M(g)$. This may appear as a weakness, but in fact is a strength when defining logic functors that are not supposed to work in isolation. The natural behaviour of procedure is usually expected on composed logics, but are not always required on intermediate logics.

Before defining a list of properties, we first need to extend the definitions of syntax and decision procedure to sets of formulas.

Definition 3 *Given $\mathcal{P}(X)$ denotes the powerset of X, and $F, G \in \mathcal{P}(AS)$:*

- $AS^* =_{def} \mathcal{P}(AS)$,
- $ASK^* =_{def} \mathcal{P}(ASK)$,
- $TELL^* =_{def} \{F \in \mathcal{P}(\{\bot\} \cup TELL \cup ASK) \mid F \cap (\{\bot\} \cup TELL) \neq \emptyset\}$,
- $F \sqsubseteq^* G =_{def} \bigvee \begin{cases} \exists f \in F, g \in G : f \sqsubseteq g \\ \exists f \in F : f \sqsubseteq \bot \\ \exists g \in G : \top \sqsubseteq g \end{cases}$
- $closed_{\sqcap}(F) =_{def} \forall f \neq f' \in F : f \sqcap f' = undef$,
- $closed_{\sqcup}(G) =_{def} \forall g \neq g' \in G : g \sqcup g' \subseteq G$.
- $open(F, G) =_{def} closed_{\sqcap}(F) \wedge closed_{\sqcup}(G) \wedge F \not\sqsubseteq^* G$.

We give below a list of the most important properties we found useful. Properties are rather technical, because they have been introduced progressively in order to prove global properties on logics and logic functors. This

assumes we have a starting point, i.e. some initial properties we are interested in. In the context of information systems, we need to answer queries. In order to avoid false positives (an answer that should not be an answer) and false negatives (missed answers), *consistency* and *completeness* are required on subsumption. But as our logics are built by composing sub-logics with logic functors, it is important to understand that the properties required on each sub-logics will not be the same, and will depend on each logic functor. For instance, a logic functor may enforce a property even if it is not satisfied by sub-logics. The other way round, some properties may be required on sub-logics, even if it is not so in the composed logic. So, a logic does not need to verify all the following properties.

- $cp_\top =_{def} \top \neq undef \Rightarrow M(\top) = I$,
- $cs_\bot =_{def} \bot \neq undef \Rightarrow M(\bot) = \emptyset$,
- $cs_\sqcap =_{def} \forall f, g \in AS : f \sqcap g \neq undef \Rightarrow M(f \sqcap g) \subseteq M(f) \cap M(g)$,
- $cp_\sqcap =_{def} \forall f, g \in AS : f \sqcap g \neq undef \Rightarrow M(f \sqcap g) \supseteq M(f) \cap M(g)$,
- $cs_\sqcup =_{def} \forall f, g \in AS : \bigcup_{h \in f \sqcup g} M(h) \subseteq M(f) \cup M(g)$,
- $cp_\sqcup =_{def} \forall f, g \in AS : \bigcup_{h \in f \sqcup g} M(h) \supseteq M(f) \cup M(g)$,
- $df =_{def} TELL \subseteq ASK$,
- $cs_\sqsubseteq =_{def} \forall f, g \in AS : f \sqsubseteq g \Rightarrow M(f) \subseteq M(g)$,
- $cp_\sqsubseteq =_{def} \forall f, g \in AS : M(f) \subseteq M(g) \Rightarrow f \sqsubseteq g$,
- $cp'_\sqsubseteq =_{def} \forall d \in TELL, x \in ASK : M(d) \subseteq M(x) \Rightarrow d \sqsubseteq x$,
- $sg' =_{def} \forall d \in TELL : Card(M(d)) = 1$,
- $reduced(F, G) =_{def} closed_\sqcap(F) \wedge closed_\sqcup(G) \Rightarrow$ $(\bigcap_{f \in F} M(f) \subseteq \bigcup_{g \in G} M(g) \Rightarrow F \sqsubseteq^* G)$,
- $reduced(F, G) =_{def} open(F, G) \Rightarrow \bigcap_{f \in F} M(f) \not\subseteq \bigcup_{g \in G} M(g)$,
- $reduced =_{def} \forall F, G \in AS^* : reduced(F, G)$,
- $reduced' =_{def} \forall F \in TELL^*, G \in ASK^* : reduced(F, G)$,
- $reduced.right =_{def} \forall f \in AS, G \in AS^* : reduced(\{f\}, G)$,
- $reduced.bot =_{def} \forall f \in AS : reduced(\{f\}, \emptyset)$.

The properties $cs_{\sqsubseteq}$ to $cp_{\sqcup}$ relate procedures to set operations on the sets of models of the argument formulas. *cs* stands for *consistency*, and *cp* stands for *completeness*. The difference between $cp_{\sqsubseteq}$ and $cp'_{\sqsubseteq}$ is that the latter is limited to the subsumption test between *TELL*-formulas and *ASK*-formulas. While $cp_{\sqsubseteq}$ is generally preferred, only $cp'_{\sqsubseteq}$ is necessary in information systems, as said above. The property *df* is purely about syntax, and says all *TELL*-formulas are also *ASK*-formulas. The property sg' (*sg* stands for *singleton*) says every *TELL*-formula has only one model, which allows these formulas to stand for interpretations, like in some description logics with nominals [HS01], and in epistemic logics [Lev90]. The property *reduced* extends the subsumption completeness $cp_{\sqsubseteq}$ to sets of formulas. Thus, the property $cp_{\sqsubseteq}$ is a weakening of $cp'_{\sqsubseteq}$.

Note that some procedures (conjunction, top, and bottom) can be only partially defined (by using *undef*) if it is convenient, and that their properties apply only on the domain where they are defined. So it is easy to make them satisfy these properties, like consistency and completeness, by keeping them undefined. Reducedness counterbalances this triviality by requiring these procedures to be defined *enough*. And reducedness is required in some logic functors in order to achieve subsumption completeness, which is crucial as said above.

3.1.3 Logics

A *logic* is the association of an abstract syntax, a semantics, a set of procedures, and a type made of a set of property proofs. The class of all logics is denoted by $\mathbb{L}$.

Definition 4 (Logic) *A logic L is a tuple (AS_L, S_L, P_L, T_L), where AS_L is (the* abstract syntax *of) a set of formulas, S_L is a* semantics *based on AS_L, P_L is a set of* procedures*, based on AS_L, and T_L is the* type *of the logic, i.e., a set of proofs of the properties that are satisfied by AS_L and P_L w.r.t. semantics S_L.*

When necessary, the satisfaction relation $\models$ of a logic L will be written $\models_L$, the interpretation domain I will be written I_L, the models $M(f)$ will be written $M_L(f)$, and each operation *op* will be written op_L.

In our implementation LogFun, a logic is a program module made of a type (the abstract syntax), a set of procedures, and an additional procedure that returns the set of properties having a proof (the type). Of course, the semantics is not concretely implemented.

3.2 Logic Functors and their Composition

Logic functors are what is really implemented in LogFun, and logics are the executable result of their composition. Logic functors also have a syn-

tax, a semantics, a set of procedures, and a set of proofs (type), but they are all abstracted over one or several logics that are considered as formal parameters. For instance, the logic functor $Prop(X)$ is the propositional logic, whose atoms have been abstracted by the formal parameter X, and can thus be replaced by more complex formulas. Then the classical propositional logic is reconstructed by applying the Composer to the expression $Prop(Set(Atom))$, where Set is used to make interpretations sets of atoms. The composed logic $Prop(Interval(Int))$ replaces the classical atoms by intervals on integers. In both composed logics, a decision procedure and proofs of consistency and completeness are automatically produced.

3.2.1 Definitions

We formally define the class of logic functors. Given $\mathbb{L}$ the class of logics, the class of logic functors $\mathbb{F}$ includes all functions F taking as a parameter a tuple of logics $(L_1, \ldots, L_n)$ and returning a logic denoted by $F(L_1, \ldots, L_n)$. The notation F/n is used to say that F is a logic functor with arity n, i.e., expecting n logics in order to produce a logic. The special case of logic functors with arity 0 corresponds to the class of logics $\mathbb{L}$.

Let $\mathbb{AS}$ be the class of all syntaxes, $\mathbb{S}$ be the class of all semantics, $\mathbb{P}$ be the class of all sets of procedures, and $\mathbb{T}$ the class of all logic types. Then, the decision procedure of a logic functor is a function from the decision procedures of the logics which are its arguments, to the decision procedure of the resulting logic. The same applies for syntax, semantics, procedures, and type.

Definition 5 (Logic Functor) *A logic functor F is a tuple (AS_F, S_F, P_F, T_F), where:*

- AS_F*:* $\mathbb{AS}^n \to \mathbb{AS}$*, is a function s.t.*
  $AS_{F(L_1,\ldots,L_n)} = AS_F(AS_{L_1}, \ldots, AS_{L_n})$*;*
- S_F*:* $\mathbb{S}^n \to \mathbb{S}$*, is a function s.t.* $S_{F(L_1,\ldots,L_n)} = S_F(S_{L_1}, \ldots, S_{L_n})$*;*
- P_F*:* $\mathbb{P}^n \to \mathbb{P}$*, is a function s.t.* $P_{F(L_1,\ldots,L_n)} = P_F(P_{L_1}, \ldots, P_{L_n})$*;*
- T_F*:* $\mathbb{T}^n \to \mathbb{T}$*, is a function s.t.* $T_{F(L_1,\ldots,L_n)} = T_F(T_{L_1}, \ldots, T_{L_n})$*.*

In order to illustrate this abstract notion of a logic functor, and prepare examples given in next sections, we shortly[1] describe a few logic functors. We classify them according to their role in the composition of logics:

Initiators are 0-ary functors representing various concrete domains

[1]The detailed description of our logic functors takes about 60 pages of definitions, algorithms, theorems and proofs, which are fully available in a research report [FR06].

- *Atom*/0: formulas and interpretations are classical atoms,
- *Int*/0: formulas and interpretations are integers,
- *String*/0: strings and substring patterns (contains, starts with, ...),

Constructors define the structure of formulas and interpretations

- *Sum*/2: the disjoint union of the languages and interpretation domains,
- *Prod*/2: the set product of the languages and interpretation domains,

Abstractors extend the abstract syntax without changing interpretations

- *Prop*/1: closure of the syntax by the 3 boolean connectors (and, or, not)

Adaptors help to ensure some properties are satisfied

- *Set*/1: applies the powerset to the interpretation domain; it reduces the need for the property *reduced* to the weaker property *reduced.right*.

We give some hints about the decision procedure (DP) of these logic functors. The DP of *Atom* and *Int* is simply equality, while the DP of *String* is the "contains" relation. The DP of $Sum(L_1, L_2)$ is roughly to choose the DP of either L_1 or L_2 depending on which logic the argument formulas belong to. In $Prod(L_1, L_2)$, formulas are pairs in $AS_1 \times AS_2$, and each DP is called on the relevant part of each pair of formulas.

The type of each functor is a set of theorems relating the properties of the argument logics to the properties of the produced logic, along with their proofs. For instance, the type $T_{Prop(X)}$ contains the theorem

$cp_{\sqsubseteq} \Leftarrow cs_{\sqcap_X} \wedge cp_{\sqcup_X} \wedge reduced_X$,

which can produce a proof for the completeness of the DP, given proofs for the properties *reduced*, $cs_{\sqcap}$, and $cp_{\sqcup}$ in the argument logic X. The type-checking problem, i.e., the production of property proofs, can be represented by a (possibly recursive) set of equations on types (derived from these theorems), which is solved by classical techniques of fixpoint iteration on finite domains [DP90]. The type-checking is automatically performed by the Composer. The possible recursivity comes from the fact that a logic can be defined in a recursive way.

In the implementation LogFun, logics are ML modules, logic functors are parameterized ML modules [Mac88] (also called *functors*), and the Composer is simply the compiler of the ML programming language (here, Objective Caml[2]).

[2]http://caml.inria.fr/

3.2.2 Reconstructing the Description Logic $\mathcal{ALC}$

We show in this section that the decision procedure of $\mathcal{ALC}$ can be derived automatically.

$$ALC = Prop(Set(Sum(Atom, Prod(Atom, ALC))))$$

This expression defines a logic as a recursive composition of logic functors. It can be coded nearly as such with the library LogFun in the programming language Caml, compiled to produce a module containing the code of the decision procedures (and other procedures), and type-checked w.r.t. properties. The decision procedure of ALC is here proved consistent (property $cs_{\sqsubseteq}$) and complete (property $cp_{\sqsubseteq}$) by combining the properties of the various logic functors. We now derive the syntax and semantics of ALC from its definition, and from the definition of logic functors [FR06].

The abstract syntax of ALC is defined by the grammar rule

$$C \to N \mid (R, C) \mid \neg\, C \mid C\ \wedge\ C \mid C\ \vee\ C \mid 1 \mid 0,$$

where N, and R are from the functor $Atom$, and respectively stand for *atomic concepts*, and *roles*. A formula (R, C) is built by the functor $Prod$, and stands for the concept $\exists R.C$, where C is a complex concept. The concept $\forall R.C$ can be represented by the formula $\neg(R, \neg\, C)$. The five other connectors come from the functor $Prop$, and can be customized to DL notations (e.g. $\top$ instead of 1, and $\bot$ instead of 0). This makes the syntax of ALC equivalent to the syntax of $\mathcal{ALC}$. For readability reasons, we will use the syntactic sugar $\forall r.C$ instead of $\neg\exists r.\neg C$ and $C \to D$ instead of $\neg C \sqcup D$.

The interpretation domain of ALC is $I = \mathcal{P}(N \uplus (R \times I)) = \mathcal{P}(N) \times \mathcal{P}(R \times I)$. This implies that interpretations are trees, whose nodes are labelled by sets of atomic concepts, and edges are labelled by roles. Because $\mathcal{ALC}$ satisfies the tree property, this makes it possible to prove that this semantics is equivalent to the semantics of $\mathcal{ALC}$ [FR06]. For instance, a model of formula (R, C) is a tree node having an edge labelled by R towards a child node that is a model of C.

All this makes ALC a correct implementation of the logic $\mathcal{ALC}$, and provides an executable decision procedure for the subsumption test $\sqsubseteq$.

Example

Now, we can prove a goal such as

$$\begin{array}{ll} & Article \sqcap \exists cite.DL\text{-}Article \sqcap \forall cite.Interesting \\ \sqsubseteq & \exists cite.(DL\text{-}Article \sqcap Interesting) \sqcap \forall cite.(DL\text{-}Article \to Interesting) \end{array}$$

Indeed, an object being an article citing at least one DL article, and whose all citations are interesting is a kind of object citing at least one DL article being interesting, and whose all citations being DL articles are interesting.

Discussion

However, it would be impossible to prove the following goal:

$$\begin{array}{ll} & \textit{DL-Article} \sqcap \forall quote.\textit{Interesting} \\ \sqsubseteq & \exists cite.(\textit{DL-Article} \sqcap \textit{Interesting}) \sqcap \forall cite.(\textit{DL-Article} \rightarrow \textit{Interesting}) \end{array}$$

because nothing tells *a priori* that a DL article is a kind of article citing other DL articles, nor that quoting is a specialization of citing. Even if such rules were known, there would be no possibility to add these axioms to the logic, except by hard-coding them in the *Prop* logic, which is absolutely not the purpose of logic functors, which require genericity.

The problem in the example is that the current subsumption test can not take into account the TBox defined in 2.9 page 11. If we consider $\sqsubseteq_{\mathcal{ALC}+TBox}$ as being the subsumption test of the description logic $\mathcal{ALC}$ with respect to the TBox previously defined, we could then prove:

$$\begin{array}{ll} & \textit{DL-Article} \sqcap \forall quote.\textit{Interesting} \\ \sqsubseteq_{\mathcal{ALC}+TBox} & \exists cite.(\textit{DL-Article} \sqcap \textit{Interesting}) \\ & \sqcap \forall cite.(\textit{DL-Article} \rightarrow \textit{Interesting}) \end{array}$$

Thus, we need a way to model the terminology (TBox) for description logics by a composition of logic functors. This terminology must be modular, as axioms can appear at several levels of the logics (e.g. the terminology on roles will not be the same one as the terminology on concepts). Of course, the recomposition of description logics is not the only purpose of logic functors, and therefore logic functors must allow the use of a theory in general.

Chapter 4

Theories in Logic Functors

We now explain how to add the notion of theory to logic functors. In other words, the operations, including the subsumption test, must be able to take into account a *theory*, that is a set of axioms.

We could first think that the simplest way would be to add a global "terminological functor" (let us call *Term* this fictive functor) that would contain the theory. In this case, adding a terminology to the ALC logic defined in 3.2.2 would be done by replacing the ALC definition by *Term(ALC)*, and then defining various axioms in *Term*. But this is not satisfying, as the axioms will not have the same form depending on which functor they are applied to. Indeed, adding an axiom to a functor (e.g. the *Prod* functor) is different from adding an axiom to another functor (e.g. the *Prop* functor), as they will not have the same form (for example $(a, b) \doteq (c, d)$ in the first case and $a \doteq b \wedge c \vee d$ in the second case), and reasoning can not be done in the same way, e.g. the subsumption test is different for each functor.

Therefore, the interface of logic functors must be modified, including some definitions and properties of Chapter 3.

4.1 Redefinition of Logic Functors

A theory is intended to represent the domain knowledge in an information system, in contrast to object descriptions. A theory th can now be passed to a logic functor after the possible arguments (other logics) are instantiated. In other words, the class of logic functors $\mathbb{F}$ includes all functions F taking as a parameter a tuple of logics $(L_1, \ldots, L_n)$ and returning a function $G = F(L_1, \ldots, L_n)$ taking itself as a parameter a theory th, and returning a logic denoted by $G(th) = F(L_1, \ldots, L_n)(th)$.

We modify Definition 5 in the following way:

Definition 6 (Logic Functor) *A* logic functor *F is a function taking logics $(0..n)$ as parameters, as well as 0 or 1 theory th. A logic functor decomposes itself as a tuple (AS_F, S_F, P_F, T_F) where:*

- *AS_F is a function on syntaxes s.t.*
  $AS_{F(L_1,\ldots,L_n)} = AS_F(AS_{L_1},\ldots,AS_{L_n})$*;*
- *S_F is a function on semantics s.t. $S_{F(L_1,\ldots,L_n)} = S_F(S_{L_1},\ldots,S_{L_n})$;*
- *P_F is a function on procedures s.t. $P_{F(L_1,\ldots,L_n)} = P_F(P_{L_1},\ldots,P_{L_n})$;*
- *T_F is a function on types s.t. $T_{F(L_1,\ldots,L_n)} = T_F(T_{L_1},\ldots,T_{L_n})$.*

Let G be the (partial) application of F to its logic arguments, then either G is a logic, or it is function of a theory.
Given $G = F(L_1,\ldots,L_n) = (AS_G, S_G, P_G, T_G)$, then $G(th)$ is defined in the following way:

$$G(th) = (AS_G(th), S_G(th), P_G(th), T_G(th)),$$

allowing the components of a logic to depend on a theory.

The arity of logic functors is written $/m/n$ were m is the number of logics taken as argument ($m \geq 0$), and n is the need for a theory ($0 \leq n \leq 1$). Note that, in the case where a logic functor does not allow the use of a theory ($n = 0$), this definition remains identical as in Definition 5. Thus, existing functors not allowing the use of the theory are not modified in any way by this change in the global definition of functors.

Three logic functors will be presented in the following.

- *Taxo*, which enables to define taxonomies, is a new logic functor,
- *Domain/1/1*, which enables to define taxonomies on concrete domains, is a new logic functor,
- *Prop*, which enables to define terminologies, is a modification of the previous version of *Prop* without terminology.

Taxo allows to define axioms of the form $f \doteq g$ or $f \dot{\leq} g$ where f and g are simple terms. *Domain/1/1*allows to define axioms of the form $f \dot{\leq} g$, where f are concrete domains formulas or defined terms, and g are defined terms. *Prop* allows to define axioms of the form $f \dot{\leq} g$ where f is a defined term, and g a propositional formula of atoms and defined terms possibly containing f. Those three functors allow to define axioms in a similar way to inclusions and definitions of concepts in description logics. This makes it possible to factorize definitions about syntax and semantics of theories in the case where these theories have the form of a terminology (inclusions and definitions of terms).

4.2 The case of Terminological Theories

Theories are sets of axioms. In the case of terminological theories, axioms are composed of two formulas linked by an operator.

4.2.1 Syntax

Definition 7 (Theory) *Let $AS \in \mathbb{AS}$. The domain of* axioms *is defined as $AX = (AS \times Rel \times AS)$ where Rel is a set of relational operators such as $\doteq$ or $\dot{\leq}$. A* theory *is a set of axioms.*

Intuitively, the relational operator $\doteq$ can be seen as "is equivalent to", and the $\dot{\leq}$ operator as "is a kind of". For example, $Article \ \dot{\leq} \ Publication$ reads "an article is a kind of publication".

4.2.2 Semantics

Let $AS \in \mathbb{AS}$ an abstract syntax. Let $S_\emptyset = (I_\emptyset, \models_\emptyset)$ be a semantics independent of any theory. Let $th \in \mathcal{P}(AS \times Rel \times AS)$ be a theory. We define in this section a function S_t taking as an argument $S_\emptyset$ and th, and returning a new semantics $S = (I, \models)$ that is based on $S_\emptyset$ but takes into account th by considering only interpretations that are consistent with th. So, S_t has type $\mathbb{S} \times \mathcal{P}(AX) \rightarrow \mathbb{S}$.

We first formalize the meaning of th w.r.t. $S_\emptyset$ by extending the satisfaction relation to th and its axioms as follows:

Definition 8 (Semantics of axioms) *Let $f, g \in AS$ and $i \in I_\emptyset$.*

$i \models_\emptyset f \doteq g$ *iff* $i \models_\emptyset f \Leftrightarrow i \models_\emptyset g$

$i \models_\emptyset f \ \dot{\leq} \ g$ *iff* $i \models_\emptyset f \Rightarrow i \models_\emptyset g$

$i \models_\emptyset th$ *iff* $\forall (f \ R \ g) \in th : i \models_\emptyset f \ R \ g$

Definition 9 *The semantics $S = S_t(S_\emptyset, th)$ is defined by $S = (I, \models)$, where:*

- $I = \{i \in I_\emptyset \mid i \models_\emptyset th\}$
- $i \models f \Longleftrightarrow i \models_\emptyset th \wedge i \models_\emptyset f$

In the following, we distinguish two different sets of models for formulas:

- $M_\emptyset(f)$ in $S_\emptyset$,
- $M(f)$ in S.

$M(f)$ takes th into account whereas $M_\emptyset(f)$ does not.

Two lemmas come directly from these definitions, and will be used in the proofs.

Lemma 1 $f \dot{\leq} g \in th \Longrightarrow M(f) \subseteq M(g)$

Proof: $M_{\emptyset}(th) \subseteq M_{\emptyset}(f \dot{\leq} g)$ Def. 8
$M_{\emptyset}(f \dot{\leq} g) = \{i \in I \mid i \models f \Rightarrow i \models g\}$ Def. 8
Assume $i \in M(f)$
$\Longrightarrow i \in M_{\emptyset}(th) \wedge i \in M_{\emptyset}(f)$
$\Longrightarrow (i \models f \Rightarrow i \models g) \wedge i \models f$
$\Longrightarrow i \models g \wedge i \in M_{\emptyset}(th)$
$\Longrightarrow i \in M_{\emptyset}(th) \wedge i \in M_{\emptyset}(g)$
$\Longrightarrow i \in M(g)$ ■

Lemma 2 $f \doteq g \in th \Longrightarrow M(f) = M(g)$

Proof: $M_{\emptyset}(th) \subseteq M_{\emptyset}(f \doteq g)$ Def. 8
$M_{\emptyset}(f \doteq g) = \{i \in I \mid i \models f \Leftrightarrow i \models g\}$ Def. 8
Assume $i \in M(f)$
$\Longrightarrow i \in M_{\emptyset}(th) \wedge i \in M_{\emptyset}(f)$
$\Longrightarrow (i \models f \Leftrightarrow i \models g) \wedge i \models f$
$\Longrightarrow i \models g \wedge i \in M_{\emptyset}(th)$
$\Longrightarrow i \in M_{\emptyset}(th) \wedge i \in M_{\emptyset}(g)$
$\Longrightarrow i \in M(g)$
Assume $i \in M(g)$
$\Longrightarrow i \in M_{\emptyset}(th) \wedge i \in M_{\emptyset}(g)$
$\Longrightarrow (i \models f \Leftrightarrow i \models g) \wedge i \models g$
$\Longrightarrow i \models f \wedge i \in M_{\emptyset}(th)$
$\Longrightarrow i \in M_{\emptyset}(th) \wedge i \in M_{\emptyset}(f)$
$\Longrightarrow i \in M(f)$ ■

We now present the three logic functors *Taxo*, *Domain/1/1* and *Prop*.

4.3 *Taxo/0/1*: taxonomies

Our first purpose is to add the notion of taxonomy to the *Atom/0* logic functor. However, adding axioms to *Atom/0* means a change in its semantics. Indeed, the interpretation domain of *Atom/0* is the set of considered atoms (an interpretation satisfies an atom if both are equal) but, in case of an axiom $a \sqsubseteq b$, the interpretation of a satisfies b, which means that interpretations of *Atom/0* with a terminology will be sets of atoms. That is the reason why we need here a new logic functor, *Taxo/0/1*.

The logic functor *Taxo/0/1* allows to represent taxonomies, e.g. of concepts or of roles in description logics. It takes as argument a theory *th*. For instance, it allows to state that quoting an article is more specific that only citing it, which is represented by the axiom $quote \dot{\leq} cite$.

4.3.1 Syntax

The syntax AS is a set of atoms. A theory th is a set of axioms of the form $f \dot{\leq} g$, where f and g are names. Note that $f \doteq g$ can be decomposed into $f \dot{\leq} g$ and $g \dot{\leq} f$.

4.3.2 Semantics

The semantics of *Taxo/0/1* is $S_{th}(S_\emptyset, th)$ where $S_\emptyset = (I_\emptyset, \models_\emptyset)$. The interpretation domain $I_\emptyset$ is defined by $I_\emptyset = \mathcal{P}(AS)$, and the satisfaction relation $\models_\emptyset$ is defined as follows:

$$i \models_\emptyset a =_{def} a \in i$$

4.3.3 Operations

subsumption: $a \sqsubseteq b =_{def} \bigvee \begin{cases} a = b \\ \exists c : a \dot{\leq} c \wedge c \sqsubseteq b \end{cases}$

The relation $\dot{\leq}$ defined by the axioms of the theory forms a pre-order $\dot{\leq}$ on atoms. Thus, we can define $succ(a)$ as follows:

$succ(a) = \{b \mid a \dot{\leq}^* b\}$

where $\dot{\leq}^*$ is the transitive closure of $\dot{\leq}$.

The notation $a \dot{\leq}^n b$ represents a path from a to b of length n in the theory such that:

$a = x_1 \wedge \forall i \in [1; n-1] : x_i \dot{\leq} x_{i+1} \in th \wedge x_n = b$

$succ(a)$ represents the atoms greater or equal to a with respect to $\dot{\leq}$, that is the atoms implied by a in the theory th.

conjunction:

$$a \sqcap b =_{def} \begin{cases} a & \text{if } a \sqsubseteq b \\ b & \text{if } b \sqsubseteq a \\ undef & otherwise \end{cases}$$

disjunction:

$$a \sqcup b =_{def} \begin{cases} \{b\} & \text{if } a \sqsubseteq b \\ \{a\} & \text{if } b \sqsubseteq a \\ \{a, b\} & otherwise \end{cases}$$

4.3.4 Properties

Lemma 3 $\forall a \in AS : succ(a) \in M_{\emptyset}(th)$

Proof: Assume $succ(a) \notin M_{\emptyset}(th)$

$\Longrightarrow succ(a) \notin \{i \in I \mid \forall f \dot{\leq} g \in th : i \models f \Rightarrow i \models g\}$ — def $M_{\emptyset}(th)$

$\Longrightarrow succ(a) \notin \{i \in I \mid \forall f \dot{\leq} g \in th : f \in i \Rightarrow g \in i\}$ — def. of $\models$

$\Longrightarrow \exists f \dot{\leq} g \in th : f \in succ(a) \not\Rightarrow g \in succ(a)$

$\Longrightarrow \exists f \dot{\leq} g \in th : f \in succ(a) \wedge g \notin succ(a)$ — def. of $\not\Rightarrow$

$\Longrightarrow \exists f \dot{\leq} g \in th : a \dot{\leq}^* f \wedge a \dot{\nleq}^* g$ — def. of $succ$

$\Longrightarrow \exists f \dot{\leq} g \in th : a \dot{\leq}^* g \wedge a \dot{\nleq}^* g$ — def. of $succ$

which leads to a contradiction. ■

- *df*
- *st*

  Proof: $\forall a \in AS : succ(a) \in M_{\emptyset}(th)$ — Lemma 3

  Moreover, $a \in succ(a) \Longrightarrow succ(a) \in M_{\emptyset}(a)$ — def. of $M_{\emptyset}$

  $\Longrightarrow succ(a) \in M_{\emptyset}(th) \cap M_{\emptyset}(a)$

  $\Longrightarrow succ(a) \in M(a)$

  $\Longrightarrow M(a) \neq \emptyset$ ■

- $cs_{\sqsubseteq}$

  Proof: $a \sqsubseteq b =_{def} \bigvee \begin{cases} a = b \\ \exists c : a \dot{\leq} c \in th \wedge c \sqsubseteq b \end{cases}$

  Thus, $a \sqsubseteq b \Longrightarrow a \dot{\leq}^n b$.

  Let us do a recursive proof on the length n of the path from a to b:

  - $n = 0 \Longrightarrow a = b \Longrightarrow M_{\emptyset}(a) = M_{\emptyset}(b) \Longrightarrow M(a) = M(b)$
    $\Longrightarrow M(a) \subseteq M(b)$
  - Assume for every x, y, $x \dot{\leq}^n y \Longrightarrow M(x) \subseteq M(y)$, and prove the subsumption is still consistent for $n + 1$.
    Assume $a \dot{\leq}^{n+1} b$.
    Then $\exists c : a \dot{\leq} c \in th \wedge c \sqsubseteq b$ and there exists a path from c to b of length n in the theory
    $\Longrightarrow \exists c : a \dot{\leq} c \in th \wedge M(c) \subseteq M(b)$ — from recursion hypothesis
    $\Longrightarrow \exists c : M(a) \subseteq M(c) \wedge M(c) \subseteq M(b)$ — lemma 1
    $\Longrightarrow M(a) \subseteq M(b)$

■

- $cp_{\sqsubseteq}$

  Proof: $M(a) \subseteq M(b) \Longrightarrow M_{\emptyset}(th) \cap M_{\emptyset}(a) \subseteq M(b)$
  $\Longrightarrow \forall i \in I : i \models th \wedge i \models a \Rightarrow i \models b$
  $\Longrightarrow \forall i \in I : i \models th \Rightarrow (a \in i \Rightarrow b \in i)$
  $\Longrightarrow succ(a) \models th \Rightarrow (a \in succ(a) \Rightarrow b \in succ(a))$
  $succ(a) \models th$ and $a \in succ(a)$ lemma 3 and def. of *succ*
  $\Longrightarrow b \in succ(a)$
  $\Longrightarrow a \stackrel{\cdot}{\leq}^{*} b$
  $\Longrightarrow a \stackrel{\cdot}{\leq}^{n} b$
  Let us do a recursive proof on the length n of the path from a to b:

  - $n = 0 \Longrightarrow a = b \Longrightarrow a \sqsubseteq b$ definition of $\sqsubseteq$
  - Assume $a \stackrel{\cdot}{\leq}^{n} b \Longrightarrow a \sqsubseteq b$ and prove it for $n+1$.
    $a \stackrel{\cdot}{\leq}^{n+1} b \Longrightarrow \exists c \in AS : a \stackrel{\cdot}{\leq} c \in th \wedge c \stackrel{\cdot}{\leq}^{n} b$
    $\Longrightarrow \exists c \in AS : a \stackrel{\cdot}{\leq} c \in th \wedge c \sqsubseteq b$ recursion hypothesis
    $\Longrightarrow a \sqsubseteq b$ definition of $\sqsubseteq$

  ■

- $cs_{\sqcap}$

  Proof: $def_{\sqcap}(a, b) \Longrightarrow a \sqsubseteq b \vee b \sqsubseteq a$

  Assume $a \sqsubseteq b$ (case $b \sqsubseteq a$ is similar).

  $a \sqsubseteq b \Longrightarrow a \sqcap b = a$
  $\Longrightarrow M(a \sqcap b) = M(a)$

  $a \sqsubseteq b \Longrightarrow M(a) \subseteq M(b)$ $cs_{\sqsubseteq}$
  $\Longrightarrow M(a) \cap M(b) = M(a)$
  $\Longrightarrow M(a \sqcap b) = M(a) \cap M(b)$
  $\Longrightarrow M(a \sqcap b) \subseteq M(a) \cap M(b)$

  ■

- $cp_{\sqcap}$

  Proof: $def_{\sqcap}(a, b) \Longrightarrow a \sqsubseteq b \vee b \sqsubseteq a$

  Assume $a \sqsubseteq b$ (case $b \sqsubseteq a$ is similar).

  $a \sqsubseteq b \Longrightarrow a \sqcap b = a$
  $\Longrightarrow M(a \sqcap b) = M(a)$

  $a \sqsubseteq b \Longrightarrow M(a) \subseteq M(b)$
  $\Longrightarrow M(a) \cap M(b) = M(a)$
  $\Longrightarrow M(a) \cap M(b) = M(a \sqcap b)$
  $\Longrightarrow M(a) \cap M(b) \subseteq M(a \sqcap b)$

■

- $cs_\sqcup$

  Proof:

  - Case $a \sqsubseteq b$
    $a \sqsubseteq b \Longrightarrow a \sqcup b = \{b\} \Longrightarrow \bigcup_{h \in a \sqcup b} = \{b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_\emptyset(h) = M_\emptyset(b)$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_\emptyset(h) \subseteq M_\emptyset(a) \cup M_\emptyset(b)$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M(h) \subseteq M(a) \cup M(b)$

  - Case $b \sqsubseteq a$
    Similar
  - Case $a \not\sqsubseteq b \wedge b \not\sqsubseteq a$
    $a \not\sqsubseteq b \wedge b \not\sqsubseteq a \Longrightarrow a \sqcup b = \{a, b\} \Longrightarrow \bigcup_{h \in a \sqcup b} = \{a, b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_\emptyset(h) = M_\emptyset(a) \cup M_\emptyset(b)$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_\emptyset(h) \subseteq M_\emptyset(a) \cup M_\emptyset(b)$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M(h) \subseteq M(a) \cup M(b)$

■

- $cp_\sqcup$

  Proof:

  - Case $a \sqsubseteq b$
    $a \sqsubseteq b \Longrightarrow a \sqcup b = \{b\} \Longrightarrow \bigcup_{h \in a \sqcup b} = \{b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_\emptyset(h) = M_\emptyset(b)$
    $a \sqsubseteq b \Longrightarrow M_\emptyset(a) \subseteq M_\emptyset(b)$
    $\Longrightarrow M_\emptyset(a) \cup M_\emptyset(b) = M_\emptyset(b)$
    $\Longrightarrow M_\emptyset(a) \cup M_\emptyset(b) \subseteq \bigcup_{h \in a \sqcup b} M_\emptyset(h)$
    $\Longrightarrow M(a) \cup M(b) \subseteq \bigcup_{h \in a \sqcup b} M(h)$

  - Case $b \sqsubseteq a$
    Similar
  - Case $a \not\sqsubseteq b \wedge b \not\sqsubseteq a$
    $a \not\sqsubseteq b \wedge b \not\sqsubseteq a$
    $\Longrightarrow a \sqcup b = \{a, b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} = \{a, b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_\emptyset(h) = M_\emptyset(a) \cup M_\emptyset(b)$
    $\Longrightarrow M_\emptyset(a) \cup M_\emptyset(b) \subseteq \bigcup_{h \in a \sqcup b} M_\emptyset(h)$
    $\Longrightarrow M(a) \cup M(b) \subseteq \bigcup_{h \in a \sqcup b} M(h)$

■

- *reduced.right*
  reduced.bot

  Proof: $M(a) \subseteq \bigcup_{b \in B} M(b)$
  $\Longrightarrow \exists b \in B : M(a) \subseteq M(b)$
  $\Longrightarrow \exists b \in B : a \sqsubseteq b$ $\quad cs_{\sqsubseteq}$
  $\Longrightarrow \{a\} \sqsubseteq^* B.$ ■

4.4 *Domain/1/1*: taxonomies on concrete domains

We now wish to build taxonomies on concrete domains, e.g. on strings or integers. In other words, we want to define axioms on concrete domains. For example, in a context of bibliographic references, we can for example state that the defined term *WongKarWai* is more general than the author names `"Wong Kar Wai"` and `"Kar-Wai, Wong"`, allowing to handle only *WongKarWai* in the requests, and thus avoid to consider the different names of a single author.

The logic functor *Domain/1/1* allows to represent taxonomies on concrete domains. It takes as logic argument a logic A, as well as a theory th.

4.4.1 Syntax

The syntax AS is the union of the abstract syntax of A (written AS_A) and a set T of defined terms, disjoint with AS_A. A theory th is a set of axioms of the form $f \dot{\leq} g$, where f is either a defined term or a formula of AS_A, and g is a defined term. No constraint is put on these axioms, i.e. the same formula can be placed on the left part of several axioms, and a same defined term can appear in the left or right part of several axioms. Cycles can be possibly defined; in this case, all the atoms concerned by the cycle will be considered equivalent.

4.4.2 Semantics

The semantics of *Domain/1/1* is $S_{th}(S_\emptyset, th)$ where $S_\emptyset = (I_\emptyset, \models_\emptyset)$, and:

- the interpretation domain $I_\emptyset$ is defined by $I_\emptyset = \mathcal{P}(T) \times I_A$
- the satisfaction relation $\models_\emptyset$ is defined as follows:
  for every $i = (i_t, i_A) \in I_\emptyset$,

$$\begin{array}{lll} (i_t, i_A) \models_\emptyset a & \textit{iff} & i_A \in M_A(a) \\ (i_t, i_A) \models_\emptyset t & \textit{iff} & t \in i_t \end{array}$$

4.4.3 Operations

subsumption:

$$a \sqsubseteq b =_{def} \begin{cases} a \sqsubseteq_A b & \text{if } a \in AS_A \quad \text{and} \quad b \in AS_A \\ \exists c \in AS_A, \exists d \in T : a \sqsubseteq c \wedge c \dot{\leq} d \wedge d \sqsubseteq b & \text{if } a \in AS_A \quad \text{and} \quad b \in T \\ a = b \vee \exists c \in T : a \dot{\leq} c \wedge c \sqsubseteq b & \text{if } a \in T \quad \text{and} \quad b \in T \\ \mathit{false} & \text{otherwise} \end{cases}$$

The relation $\dot{\leq}$ defined by the axioms of the theory forms a pre-order $\dot{\leq}$ on atoms. Thus, we can define $succ(a)$ as follows:

$succ(a) = \{b \in T \mid a \dot{\leq}^* b\}$

where $\dot{\leq}^*$ is the transitive closure of $\dot{\leq}$.

The notation $a \dot{\leq}^n b$ represents a path from a to b of length n in the theory such that:

$a = x_1 \wedge \forall i \in [1; n-1] : x_i \dot{\leq} x_{i+1} \in th \wedge x_n = b$

$succ(a)$ represents the terms greater or equal to a with respect to $\dot{\leq}$, that is the terms implied by a in the theory th.

conjunction:

$$a \sqcap b =_{def} \begin{cases} a & \text{if } a \sqsubseteq b \\ b & \text{if } b \sqsubseteq a \\ \mathit{undef} & \mathit{otherwise} \end{cases}$$

disjunction:

$$a \sqcup b =_{def} \begin{cases} \{b\} & \text{if } a \sqsubseteq b \\ \{a\} & \text{if } b \sqsubseteq a \\ \{a, b\} & \mathit{otherwise} \end{cases}$$

4.4.4 Properties

Lemma 4 $\forall a \in AS_A : i_A \in M_A(a) \Rightarrow (succ(a), i_A) \in M_\emptyset(th)$

Proof: Assume $(succ(a), i_A) \notin M_\emptyset(th)$

$\Longrightarrow (succ(a), i_A) \notin \{i \in I \mid \forall f \dot{\leq} g \in th : i \models f \Rightarrow i \models g\}$ — def $M_\emptyset(th)$

$\Longrightarrow \exists f \dot{\leq} g \in th : (succ(a), i_A) \models f \not\Rightarrow (succ(a), i_A) \models g$

$\Longrightarrow \exists f \dot{\leq} g \in th : (succ(a), i_A) \models f \wedge (succ(a), i_A) \not\models g$ — def. of $\not\Rightarrow$

- if $f \in AS_A$
  $\Longrightarrow \exists f \dot{\leq} g \in th : i_A \in M_A(f) \wedge g \notin succ(a)$ — def. of $\models$

- if $f \in T$
  $\Longrightarrow (succ(a), i_A) \notin \{i \in I \mid \forall f \dot{\leq} g \in th : f \in i \Rightarrow g \in i\}$ def. of $\models$
  $\Longrightarrow \exists f \dot{\leq} g \in th : f \in succ(a) \not\Rightarrow g \in succ(a)$
  $\Longrightarrow \exists f \dot{\leq} g \in th : f \in succ(a) \wedge g \notin succ(a)$ def. of $\not\Rightarrow$
  $\Longrightarrow \exists f \dot{\leq} g \in th : a \dot{\leq}^* f \wedge a \dot{\not\leq}^* g$ def. of *succ*
  $\Longrightarrow \exists f \dot{\leq} g \in th : a \dot{\leq}^* g \wedge a \dot{\not\leq}^* g$ def. of *succ*

  which leads to a contradiction.

■

- *df*
- $st \Leftarrow st_A$

  Proof:

  - $\forall a \in AS_A : M_A(a) \neq \emptyset$ st_A
    $\Longrightarrow M(a) \neq \emptyset$ def. of $\models$
  - $\forall a \in T, \forall i_A \in I_A : succ(a) \in M_\emptyset(th)$ Lemma 4
    Let $i_A \in I_A$, let $i = (succ(a), i_A)$
    $a \in succ(a) \Longrightarrow i \in M_\emptyset(a)$ def. of $\models$
    $\Longrightarrow i \in M_\emptyset(th) \cap M_\emptyset(a)$
    $\Longrightarrow i \in M(a)$
    $\Longrightarrow M(a) \neq \emptyset$

■

- $cs_\sqsubseteq \Leftarrow cs_{\sqsubseteq_A}$

  Proof:

  - Case $a \in AS_A$ and $b \in AS_A$
    $a \sqsubseteq b \Longrightarrow a \sqsubseteq_A b$
    $\Longrightarrow M_A(a) \subseteq M_A(b)$ $cs_{\sqsubseteq_A}$
    $\Longrightarrow M(a) \subseteq M(b)$
  - Case $a \in T$ and $b \in T$

    Thus, $a \sqsubseteq b \Longrightarrow a \dot{\leq}^n b$.
    Let us do a recursive proof on the length n of the path from a to b:

    * $n = 0$
      $a = b \Longrightarrow M_\emptyset(a) = M_\emptyset(b) \Longrightarrow M(a) = M(b)$
      $\Longrightarrow M(a) \subseteq M(b)$

* Assume for every x, y, $x \dot{\leq}^n y \Longrightarrow M(x) \subseteq M(y)$, and prove the subsumption is still consistent for $n+1$.
  Assume $a \dot{\leq}^{n+1} b$.
  Then $\exists c \in T : a \dot{\leq} c \in th \wedge c \sqsubseteq b$ and there exists a path from c to b of length n in the theory
  $\Longrightarrow \exists c \in T : a \dot{\leq} c \in th \wedge M(c) \subseteq M(b)$ from recursion hypothesis
  $\Longrightarrow \exists c \in T : M(a) \subseteq M(c) \wedge M(c) \subseteq M(b)$ lemma 1
  $\Longrightarrow M(a) \subseteq M(b)$

- Case $a \in AS_A$ and $b \in T$
  $\exists c \in AS_A, \exists c \in T : a \sqsubseteq c \wedge c \dot{\leq} d \wedge d \sqsubseteq b$
  $\exists c \in AS_A, \exists c \in T : a \sqsubseteq c \wedge M(c) \subseteq M(d) \wedge d \sqsubseteq b$ lemma 1
  $\exists c \in AS_A, \exists c \in T : M(a) \subseteq M(c) \wedge M(c) \subseteq M(d) \wedge M(d) \subseteq M(b)$ previous cases
  $\Longrightarrow M(a) \subseteq M(b)$

■

- $cp_{\sqsubseteq} \Leftarrow cp_{\sqsubseteq_A}$

  Proof:

  - Case $a \in AS_A$ and $b \in AS_A$
    Then $M(a) \subseteq M(b) \Longrightarrow M_A(a) \subseteq M_A(b) \Longrightarrow a \sqsubseteq_A b$ $cp_{\sqsubseteq_A}$
    $\Longrightarrow a \sqsubseteq b$ def. of $\sqsubseteq$
  - Else
    $M(a) \subseteq M(b) \Longrightarrow M_\emptyset(th) \cap M_\emptyset(a) \subseteq M(b)$
    $\Longrightarrow \forall i \in I : i \models th \wedge i \models a \Rightarrow i \models b$
    $\Longrightarrow \forall i \in I : i \models th \Rightarrow (i \models a \Rightarrow i \models b)$
    $\Longrightarrow \forall i_A \in I_A : (succ(a), i_A) \models th \Rightarrow ((succ(a), i_A) \models a \Rightarrow (succ(a), i_A) \models b)$
    $\forall i_A \in I_A : (succ(a), i_A) \models th \wedge (succ(a), i_A) \models a$ lemma 4 and def. of *succ*
    $\Longrightarrow \forall i_A \in I_A : (succ(a), i_A) \models b$
    $\Longrightarrow (\forall i_A \in I_A : i_A \in M_A(b)) \vee b \in succ(a)$ def. of $\models$

    * If $\forall i_A \in I_A : i_A \in M_A(b)$
      $\Longrightarrow \forall i_A \in I_A : i_A \in M_A(a) \Rightarrow i_A \in M_A(b)$
      $\Longrightarrow M_A(a) \subseteq M_A(b) \Longrightarrow a \sqsubseteq_A b$ $cp_{\sqsubseteq_A}$
      $\Longrightarrow a \sqsubseteq b$ def. of $\sqsubseteq$

    * If $b \in succ(a) \Longrightarrow a \dot{\leq}^* b \Longrightarrow a \dot{\leq}^n b$
      Let us do a recursive proof on the length n of the path from a to b:

- $n = 0 \Longrightarrow a = b \Longrightarrow a \sqsubseteq b$ definition of $\sqsubseteq$
- Assume $a \stackrel{\cdot}{\leq}^{n} b \Longrightarrow a \sqsubseteq b$ and prove it for $n+1$.
  $a \stackrel{\cdot}{\leq}^{n+1} b \Longrightarrow \exists c \in T : a \stackrel{\cdot}{\leq} c \in th \wedge c \stackrel{\cdot}{\leq}^{n} b$
  $\Longrightarrow \exists c \in T : a \stackrel{\cdot}{\leq} c \wedge c \sqsubseteq b$ recursion hypothesis
  $\Longrightarrow a \sqsubseteq b$ definition of $\sqsubseteq$

■

- $cs_{\sqcap}$

  Proof: $def_{\sqcap}(a, b) \Longrightarrow a \sqsubseteq b \vee b \sqsubseteq a$

  Assume $a \sqsubseteq b$ (case $b \sqsubseteq a$ is similar).

  $a \sqsubseteq b \Longrightarrow a \sqcap b = a$
  $\Longrightarrow M(a \sqcap b) = M(a)$

  $a \sqsubseteq b \Longrightarrow M(a) \subseteq M(b)$ $cs_{\sqsubseteq}$
  $\Longrightarrow M(a) \cap M(b) = M(a)$
  $\Longrightarrow M(a \sqcap b) = M(a) \cap M(b)$
  $\Longrightarrow M(a \sqcap b) \subseteq M(a) \cap M(b)$

■

- $cp_{\sqcap}$

  Proof: $def_{\sqcap}(a, b) \Longrightarrow a \sqsubseteq b \vee b \sqsubseteq a$

  Assume $a \sqsubseteq b$ (case $b \sqsubseteq a$ is similar).

  $a \sqsubseteq b \Longrightarrow a \sqcap b = a$
  $\Longrightarrow M(a \sqcap b) = M(a)$

  $a \sqsubseteq b \Longrightarrow M(a) \subseteq M(b)$
  $\Longrightarrow M(a) \cap M(b) = M(a)$
  $\Longrightarrow M(a) \cap M(b) = M(a \sqcap b)$
  $\Longrightarrow M(a) \cap M(b) \subseteq M(a \sqcap b)$

■

- $cs_{\sqcup}$

  Proof:

  – Case $a \sqsubseteq b$
    $a \sqsubseteq b \Longrightarrow a \sqcup b = \{b\} \Longrightarrow \bigcup_{h \in a \sqcup b} = \{b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_{\emptyset}(h) = M_{\emptyset}(b)$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_{\emptyset}(h) \subseteq M_{\emptyset}(a) \cup M_{\emptyset}(b)$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M(h) \subseteq M(a) \cup M(b)$

- Case $b \sqsubseteq a$
  Similar
- Case $a \not\sqsubseteq b \wedge b \not\sqsubseteq a$
  $a \not\sqsubseteq b \wedge b \not\sqsubseteq a \Longrightarrow a \sqcup b = \{a, b\} \Longrightarrow \bigcup_{h \in a \sqcup b} = \{a, b\}$
  $\Longrightarrow \bigcup_{h \in a \sqcup b} M_{\emptyset}(h) = M_{\emptyset}(a) \cup M_{\emptyset}(b)$
  $\Longrightarrow \bigcup_{h \in a \sqcup b} M_{\emptyset}(h) \subseteq M_{\emptyset}(a) \cup M_{\emptyset}(b)$
  $\Longrightarrow \bigcup_{h \in a \sqcup b} M(h) \subseteq M(a) \cup M(b)$

■

- $cp_{\sqcup}$

  Proof:

  - Case $a \sqsubseteq b$
    $a \sqsubseteq b \Longrightarrow a \sqcup b = \{b\} \Longrightarrow \bigcup_{h \in a \sqcup b} = \{b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_{\emptyset}(h) = M_{\emptyset}(b)$
    $a \sqsubseteq b \Longrightarrow M_{\emptyset}(a) \subseteq M_{\emptyset}(b)$
    $\Longrightarrow M_{\emptyset}(a) \cup M_{\emptyset}(b) = M_{\emptyset}(b)$
    $\Longrightarrow M_{\emptyset}(a) \cup M_{\emptyset}(b) \subseteq \bigcup_{h \in a \sqcup b} M_{\emptyset}(h)$
    $\Longrightarrow M(a) \cup M(b) \subseteq \bigcup_{h \in a \sqcup b} M(h)$
  - Case $b \sqsubseteq a$
    Similar
  - Case $a \not\sqsubseteq b \wedge b \not\sqsubseteq a$
    $a \not\sqsubseteq b \wedge b \not\sqsubseteq a$
    $\Longrightarrow a \sqcup b = \{a, b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} = \{a, b\}$
    $\Longrightarrow \bigcup_{h \in a \sqcup b} M_{\emptyset}(h) = M_{\emptyset}(a) \cup M_{\emptyset}(b)$
    $\Longrightarrow M_{\emptyset}(a) \cup M_{\emptyset}(b) \subseteq \bigcup_{h \in a \sqcup b} M_{\emptyset}(h)$
    $\Longrightarrow M(a) \cup M(b) \subseteq \bigcup_{h \in a \sqcup b} M(h)$

■

- *reduced.right*
  reduced.bot

  Proof: to be done

■

4.5 *Prop/1/1*: terminologies

Prop represents the boolean closure of a given logic by the three boolean connectors. It can also be seen as the usual propositional logic, whose atoms are replaced by the formulas of a logic given as a parameter $A \in \mathbb{L}$.

Prop takes a theory *th* as an argument. Axioms can be introduced as a description ($f \mathrel{\dot{\leq}} g$). The left part f must be a term, that is a name which is not an atom name. The right part g is any formula. Note that this restriction of a single term in the left part is to gain in efficiency. Indeed, it would be possible to allow full expressions on the left part but with a loss of efficiency.

4.5.1 Syntax

The syntax AS of *Prop* is the smallest set of formulas that contains AS_A (custom atoms), 1 (true), 0 (false), T (terms, i.e. classical atoms that can be defined, such that $T \cap AS_A = \emptyset$) and is closed by the application of the unary connector $\neg$ (negation), and the binary connectors $\wedge$ (conjunction) and $\vee$ (disjunction). ASK is the subset of propositions whose atoms are all in ASK_A. $TELL$ is the subset of propositions, whose negative atoms are all in ASK_A, whose positive atoms are all in $TELL_A \cup ASK_A$, and such that every conjunctive clause contains a positive literal in $TELL_A$.

4.5.2 Semantics

The semantics of *Prop* is $S_{th}(S_\emptyset, th)$ where $S_\emptyset = (I_\emptyset, \models_\emptyset)$, and:

- the interpretation domain $I_\emptyset$ is defined by $I_\emptyset = \mathcal{P}(T) \times I_A$
- the satisfaction relation $\models_\emptyset$ is defined as follows:
  for every $i = (i_t, i_A) \in I_\emptyset$,

$$\begin{array}{rll}
(i_t, i_A) \models_\emptyset a & \textit{iff} & i_A \in M_A(a) \\
(i_t, i_A) \models_\emptyset t & \textit{iff} & t \in i_t \\
i \models_\emptyset 1 & \textit{iff} & \textit{true} \\
i \models_\emptyset 0 & \textit{iff} & \textit{false}
\end{array}
\qquad
\begin{array}{rll}
i \models_\emptyset \neg f & \textit{iff} & i \not\models_\emptyset f \\
i \models_\emptyset f \wedge g & \textit{iff} & i \models_\emptyset f \text{ and } i \models_\emptyset g \\
i \models_\emptyset f \vee g & \textit{iff} & i \models_\emptyset f \text{ or } i \models_\emptyset g
\end{array}$$

4.5.3 Operations

Subsumption

$f \sqsubseteq g$ is true iff there exists a proof of the sequent $\vdash \neg f \vee g$ in the sequent calculus of Table 4.1.

In the rules, a and b are atoms, t is a term. L_a is an atom literal, that is an atom or the negation of an atom. L_t is a term literal. $\overline{L}$ denotes the negation of L ($\overline{x} := \neg x$ and $\overline{\neg x} := x$). X is a proposition. β is the disjunction of

$\beta_1 \ldots \beta_n$ and α is the conjunction of $\alpha_1 \ldots \alpha_n$. The tautology 1 is equivalent to the empty conjunction, and the contradiction 0 is equivalent to the empty disjunction. Δ and Δ_1 are always sets of atom literals and term literals, Δ_2 is a set of term literals and Γ is a sequence of propositions. A sequent is of the form $\Delta_1 \mid \Delta_2 \vdash \Gamma$. In some cases, Δ will be used to stand for $\Delta_1 \cup \Delta_2$.

The *Prop* sequent calculus of table 4.1 is originally based on the sequent calculus for the propositional logic [BP95], which contains 3 rules for the elimination of $\vee$, $\wedge$ and $\neg$, and one rule to pass literals from Γ to Δ_1. Then, the axiom-Rule was replaced by 5 rules corresponding to A operations ($\sqsubseteq_A$, $\top_A$, $\bot_A$, $\sqcap_A$, $\sqcup_A$) [FR02]. Then, 5 new rules were added to handle to presence of terms due to the terminology. The *term*-Axiom successfully finishes the branch of the proof if a term and its negation are found in Δ. The *term*-Rule and *term*′-Rule pass a term literal from Γ to Δ_2 if it was not present in Δ before. The *def*-Rule and $\neg def_{\stackrel{.}{\leq}}$-Rule add the definition of a term in Δ_2 to Γ, and move the term to Δ_1, so that it can not be unfolded anymore.

To avoid nontermination due to the possible presence of cycles in the terminology, a tabulation is done by the subsomption when a sequent is in normal form (NF). A goal is in *normal form* if it is of the form $\Delta \vdash$. Moreover, to bring a goal into NF, we only apply the $\neg\neg$-Rule, the β-Rule, the literal-Rule, the α-Rule, the term-Rule and the $\neg$term-Rule until none of these rules is applicable, that is neither unfolding of definitions nor further subsumption test is performed.

Once the goal is in NF, the left part of the sequent ($\Delta_1 \mid \Delta_2$) is saved in a global table accessible from the different interwove subsumption tests. If an identical goal is found in this table, this means the subsumption test contains an infinite recursion, and the subsumption test fails. Else, this goal will be removed from the table at the end of the subsumption test. Note that this tabulation and check is done every time a subsumption test is performed, or one of the *def*-Rule is applied.

Thus, this avoids infinite subsumption tests due to a cyclic terminology.

Other operations

conjunction: $f \sqcap g =_{def} f \wedge g$.

disjunction: $f \sqcup g =_{def} f \vee g$.

tautology: $\top =_{def} 1$.

contradiction: $\bot =_{def} 0$.

4.5.4 Properties

Definition 5 *A sequent* $\Delta \vdash \Gamma$ *is valid iff* $\bigcap_{\delta \in \Delta} M(\delta) \subseteq \bigcup_{\gamma \in \Gamma} M(\gamma)$.

Rule		Condition
$\neg\neg$-Rule:	$\dfrac{\Delta \vdash X, \Gamma}{\Delta \vdash \neg\neg X, \Gamma}$	
literal-Rule:	$\dfrac{\overline{L_a}, \Delta_1 \mid \Delta_2 \vdash \Gamma}{\Delta_1 \mid \Delta_2 \vdash L_a, \Gamma}$	
β-Rule:	$\dfrac{\Delta \vdash \beta_1, \ldots, \beta_n, \Gamma}{\Delta \vdash \beta, \Gamma}$	
α-Rule:	$\dfrac{\Delta \vdash \alpha_1, \Gamma \ \ldots \ \Delta \vdash \alpha_n, \Gamma}{\Delta \vdash \alpha, \Gamma}$	
$\top$-Axiom:	$\neg b, \Delta_1 \mid \Delta_2 \vdash \Gamma$	if $def_{\top_A}$ and $\top_A \sqsubseteq_A b$
$\bot$-Axiom:	$a, \Delta_1 \mid \Delta_2 \vdash \Gamma$	if $def_{\bot_A}$ and $a \sqsubseteq_A \bot_A$
$\sqsubseteq$ -Axiom:	$a, \neg b, \Delta_1 \mid \Delta_2 \vdash \Gamma$	if $a \sqsubseteq_A b$
$\sqcap$-Rule:	$\dfrac{a \sqcap_A b, \Delta_1 \mid \Delta_2 \vdash \Gamma}{a, b, \Delta_1 \mid \Delta_2 \vdash \Gamma}$	if $def_{\sqcap}(a, b)$
$\sqcup$-Rule:	$\dfrac{\neg(a \sqcup_A b), \Delta_1 \mid \Delta_2 \vdash \Gamma}{\neg a, \neg b, \Delta_1 \mid \Delta_2 \vdash \Gamma}$	
term-Axiom:	$t, \neg t, \Delta_1 \mid \Delta_2 \vdash \Gamma$	
term-Rule:	$\dfrac{\Delta_1 \mid \Delta_2, \overline{L_t} \vdash \Gamma}{\Delta_1 \mid \Delta_2 \vdash L_t, \Gamma}$	if $\overline{L_t} \notin \Delta$
term′-Rule:	$\dfrac{\Delta_1 \mid \Delta_2 \vdash \Gamma}{\Delta_1 \mid \Delta_2 \vdash L_t, \Gamma}$	if $\overline{L_t} \in \Delta$
def-Rule:	$\dfrac{t, \Delta_1 \mid \Delta_2 \vdash \neg C, \Gamma}{\Delta_1 \mid \Delta_2, t \vdash \Gamma}$	if $t \dot{\leq} C \in th$
$\neg def_{\dot{\leq}}$-Rule:	$\dfrac{\neg t, \Delta_1 \mid \Delta_2 \vdash \Gamma}{\Delta_1 \mid \Delta_2, \neg t \vdash \Gamma}$	

Table 4.1: Sequent calculus for deduction in propositional logic.

Definition 6 *The expression Trees*$(\Delta \vdash \Gamma)$ *denotes the set of all fully developed trees whose root is the sequent* $\Delta \vdash \Gamma$.

Definition 7 *A sequent* $\Delta \vdash \Gamma$ *is said* open w.r.t. the theory, *denoted by* $open_T(t)$, *iff the term-Axiom can't be applied, that is, there is no* $t \in \Delta$ *such as* $\bar{t} \in \Delta$.

Definition 8 *A fully developed tree t is said* open, *denoted by* $open(t)$, *iff it contains an open sequent (necessary a leaf from definition of an open sequent), i.e.* $\exists A, \overline{B} \vdash \in t : open(A, B) \wedge open_T(A, \overline{B} \vdash)$.

Lemma 9 $\forall i \in I : i \in M(a) \Rightarrow i \notin M(\neg a)$

Proof: $i \in M(a) \Longrightarrow i \in M_\emptyset(a) \cap M_\emptyset(th)$ Definition of M
$\Longrightarrow i \in M_\emptyset(a)$
$\Longrightarrow i \models_\emptyset a$ Definition of $M_\emptyset$
$\Longrightarrow i \not\models_\emptyset \neg a$ Semantics of negation
$\Longrightarrow i \notin M_\emptyset(\neg a)$
$\Longrightarrow i \notin M_\emptyset(\neg a) \cap M_\emptyset(th)$
$\Longrightarrow i \notin M(\neg a)$ ■

Lemma 10 *Contrapositive of lemma 9*
$\forall i \in I : i \in M(\neg a) \Rightarrow i \notin M(a)$

Lemma 11 $\forall i \in I : i \notin M(\neg a) \Rightarrow i \in M(a) \vee i \notin M_\emptyset(th)$

Proof: $i \notin M(\neg a) \Longrightarrow i \notin M_\emptyset(\neg a) \cap M_\emptyset(th)$
$\Longrightarrow (i \in M_\emptyset(th) \wedge i \notin M_\emptyset(\neg a)) \vee i \notin M_\emptyset(th)$
$\Longrightarrow (i \in M_\emptyset(th) \wedge i \in M_\emptyset(a)) \vee i \notin M_\emptyset(th)$ semantics of negation
$\Longrightarrow i \in M(a) \vee i \notin M_\emptyset(th)$ definition of M
■

Lemma 12 $\forall i \in I : i \notin M(a) \Rightarrow i \in M(\neg a) \vee i \notin M_\emptyset(th)$

Proof: $i \notin M(a) \Longrightarrow i \notin M_\emptyset(a) \cap M_\emptyset(th)$
$\Longrightarrow (i \in M_\emptyset(th) \wedge i \notin M_\emptyset(a)) \vee i \notin M_\emptyset(th)$
$\Longrightarrow (i \in M_\emptyset(th) \wedge i \in M_\emptyset(\neg a)) \vee i \notin M_\emptyset(th)$ semantics of negation
$\Longrightarrow i \in M(\neg a) \vee i \notin M_\emptyset(th)$ definition of M ■

Lemma 13 $i_A \in M_A(a) \Leftrightarrow \forall i_t \in 2^T : (i_t, i_A) \in M_\emptyset(a)$

Proof: $i_A \in M_A(a) \Leftrightarrow i_A \models_A a$ def. of M_A
$\Leftrightarrow \forall i_t \in 2^T : (i_A, i_t) \models_\emptyset a$ def. of $\models_\emptyset$
$\Leftrightarrow \forall i_t \in 2^T : (i_A, i_t) \in M_\emptyset(a)$ def. of $M_\emptyset$
■

The following lemma helps ensuring $cp_\sqsubseteq$.

Lemma 14 *Let t be a proof tree for $\vdash \neg f \vee g$ and $\Delta_1 \mid \Delta_2 \vdash \Gamma \in t$ a sequent in this tree.*
Then, $\forall t_1 \in \Delta_1 : (t_1 \dot{\leq} C_1) \in th \Rightarrow$
$\forall i \in I : i \not\models_\emptyset \Delta_1 \mid \Delta_2 \vdash \Gamma \Rightarrow i \models_\emptyset C_1$
where $i \models_\emptyset \Delta_1 \mid \Delta_2 \vdash \Gamma =_{def} \forall \delta \in \Delta : i \models_\emptyset \delta \Rightarrow \exists \gamma \in \Gamma : i \models_\emptyset \gamma$

Proof: By induction on the sequent. The property is trivially true for the root $\vdash \neg f \vee g$, and we prove that each rule preserves the property.

The only non-trivial rule to prove is to show that the *def*-Rule of table 4.1 preserves the property.

- either $t1 = t : C_1 = C, t_1 \dot{\leq} C_1 \in th$
  Assume there is $i \in I : i \not\models_\emptyset t, \Delta_1 \mid \Delta_2, \vdash \neg C, \Gamma$.
  $\Longrightarrow \forall \delta \in \{t\} \cup \Delta_1 \cup \Delta_2 : i \models_\emptyset \delta, \forall \gamma \in \{\neg C\} \cup \Gamma : i \not\models_\emptyset \gamma$
  $\Longrightarrow i \not\models_\emptyset \neg C \Longrightarrow i \models_\emptyset C \Longrightarrow i \models_\emptyset C_1$

- or $t_1 \in \Delta_1$
  Assume there is C_1 such that $t_1 \dot{\leq} C_1 \in th$.
  Assume there is $i \in I$ such that $i \not\models_\emptyset t, \Delta_1 \mid \Delta_2 \vdash \neg C, \Gamma$
  $\Longrightarrow \forall \delta \in \{t\} \cup \Delta_1 \cup \Delta_2 : i \models_\emptyset \delta, \forall \gamma \in \{\neg C\} \cup \Gamma : i \not\models_\emptyset \gamma$
  $\Longrightarrow \forall \delta \in \Delta_1 \cup \Delta_2 \cup \{t\} : i \models_\emptyset \delta, \forall \gamma \in \Gamma : i \not\models_\emptyset \gamma$
  $\Longrightarrow i \not\models_\emptyset \Delta_1 \mid \Delta_2, t \vdash \Gamma$
  $\Longrightarrow i \models_\emptyset C_1$ by hypothesis, as $t_1 \in \Delta_1$, $t_1 \dot{\leq} C_1 \in th$

■

Proof of termination

We prove that the backward-chaining interpretation of the inference system in table 4.1 terminates for all root sequent. For this proof we need a total ordering of sequents.

Every formula being either a literal L, a double negation $\neg\neg X$, a conjunction α or a disjunction β, one defines an integral measure m for every formula in $AS_{prop(A)}$ as follows: $m(\alpha) = 1 + m(\alpha_1) + m(\alpha_2)$, $m(\neg\neg X) = 1 + m(X)$, $m(\beta) = 1 + m(\beta_1) + m(\beta_2)$, and $m(L) = 1$.

This measure is extended to sequences of propositions Γ and to sequences of literals Δ as follows:

- $m(\Gamma) = \sum_{X \in \Gamma} m(X)$,
- $m(\Delta) = \sum_{L \in \Delta} m(L)$.

Let n be the maximal number of possible unfoldings, that is the number of defined terms replaceable by their value. This number is bounded because a replaceable term is stored in Δ_2 and moved to Δ_1 once it is replaced and thus can't be replaced anymore (*def*-Rule, and $\neg def_{\dot{\leq}}$-Rule). Moreover, *term'*-Rule ensures that a term in the right part can't go to the set of replaceable terms Δ_2 if it belongs to the set of already replaced terms Δ_1. n remains equal for all rules except for *def*-Rule, and $\neg def_{\dot{\leq}}$-Rule where it is strictly decreasing.

Thus, we extend our measure to full sequents $\Delta \vdash \Gamma$ as follows:
$m(\Delta \vdash \Gamma) = (n, m(\Gamma), m(\Delta))$.

Finally, sequents are totally ordered according to a lexicographic ordering $<$ on $\mathbb{N}^3$. We observe that for every deduction rule $\frac{Seq_1}{Seq_2}$, $m(Seq_1) < m(Seq_2)$ holds.

So, every proof tree is finite.

However, due to the possible presence of cycles in the structure of logic functors and, as a consequence, in the terminology, non-termination can come from infinite recursive calls to the Prop subsumption test.

But, because of tabulation, a cycle will be detected if the same subsumption test is performed twice. In the worst case, a recursive subsumption test will be done between every possible pair of primitive or defined atoms present either in the terminology or in the two formulas on which the first subsumption test is performed. Thus, termination is ensured.

Note here that this worst case is only a way to ensure termination, and a good ordering of rules should allow to prove a subsumption with a much lower number of recursive subsumption tests.

Proof of properties

- $df \Leftarrow df_A$
- $cs_{\sqsubseteq} \Leftarrow cs_{\sqsubseteq_A} \wedge cp_{\sqcap_A} \wedge cs_{\sqcup_A} \wedge cp_{\top_A} \wedge cs_{\bot_A}$

  Proof:

  - Let us show that the *term*-Rule preserves validity. Assume the sequent $\Delta_1 \mid \Delta_2, \bar{t} \vdash \Gamma$ is valid, then for all $i \in I$
    - either $\exists X \in \Gamma : i \in M(X) \Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2 \vdash t, \Gamma$ is valid
    - or $\exists L \in \Delta_1 : i \notin M(L) \Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2 \vdash t, \Gamma$ is valid
    - or $\exists L \in \Delta_2 : i \notin M(L) \Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2 \vdash t, \Gamma$ is valid
    - or $i \notin M(\bar{t})$
      $\Longrightarrow i \in M(t) \vee i \notin M_\emptyset(th)$ lemmas 11 and 12
      If $i \in M(t)$ the sequent $\Delta_1 \mid \Delta_2 \vdash t, \Gamma$ is valid.
      Else $i \notin M_\emptyset(th)$
      - if $\Delta_1 = \emptyset \wedge \Delta_2 = \emptyset$
        $\Longrightarrow \emptyset \subseteq M(t) \cup \bigcup_{\gamma \in \Gamma} M(\gamma)$
        $\Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2 \vdash t, \Gamma$ is valid.
      - if $\Delta_1 \neq \emptyset \vee \Delta_2 \neq \emptyset$
        $i \notin M_\emptyset(th) \Longrightarrow \exists \delta \in \Delta_1 \cup \Delta_2 : i \notin M_\emptyset(th) \cap M_\emptyset(\delta)$
        $\Longrightarrow \exists \delta \in \Delta_1 \cup \Delta_2 : i \notin M(\delta)$
        $\Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2 \vdash t, \Gamma$ is valid.

The $term'$-Rule can be proved in a similar way.

- Let us show that the *term*-Axiom preserves validity.
  For all $i \in I$,
  $i \in M(t) \Longrightarrow i \notin M(\neg t)$ lemma 9
  $\Longrightarrow M(t) \cap M(\neg t) = \emptyset$
  $\Longrightarrow M(t) \cap M(\neg t) \cap \bigcap_{\delta\in\Delta} M(\delta) = \emptyset$
  $\Longrightarrow M(t) \cap M(\neg t) \cap \bigcap_{\delta\in\Delta} M(\delta) \subseteq \bigcup_{\gamma\in\Gamma} M(\gamma)$
  $\Longrightarrow$ the sequent $t, \neg t, \Delta \vdash \Gamma$ is valid.
- Let us show that $\sqsubseteq$-Axiom is valid:
  $a \sqsubseteq_A b \Longrightarrow M_A(a) \subseteq M_A(b)$ $cs_{\sqsubseteq_A}$
  $\Longrightarrow \forall i_A \in I_A : i_A \in M_A(a) \Rightarrow i_A \in M_A(b)$
  $\Longrightarrow \forall i_A \in I_A, \forall i_t \in 2^T : (i_t, i_A) \in M_A(a) \Rightarrow (i_t, i_A) \in M_A(b)$ lemma 13
  $\Longrightarrow \forall i_A \in I_A, \forall i_t \in 2^T : (i_t, i_A) \in M_\emptyset(a) \Rightarrow (i_t, i_A) \in M_\emptyset(b)$ semantics of $\models_\emptyset$
  $\Longrightarrow \forall i \in I : i \in M_\emptyset(a) \Rightarrow i \in M_\emptyset(b)$
  $\Longrightarrow \forall i \in I : i \in M_\emptyset(a) \cap M_\emptyset(th) \Rightarrow i \in M_\emptyset(b) \cap M_\emptyset(th)$
  $\Longrightarrow \forall i \in I : i \in M(a) \Rightarrow i \in M(b)$ def. of M
  $\Longrightarrow \forall i \in I : i \notin M(a) \vee i \in M(b)$ def. of $\Rightarrow$
  $\Longrightarrow \forall i \in I : i \notin M(a) \vee i \notin M(\neg b)$ Lemma 9
  $\Longrightarrow$ the sequent $a, \neg b, \Delta \vdash \Gamma$ is valid.

  The $\top$-Axiom and $\bot$-Axiom are valid as a corollary: replace a by $\top_A$ for $\top$-Axiom, and b by $\bot_A$ for $\bot$-Axiom, along with $cp_{\top_A}$ and $cs_{\bot_A}$.
- Let us show that the $\sqcap$-Rule preserves validity. Assume that $a \sqcap_A b$ is defined and the sequent $a \sqcap_A b, \Delta_1 \mid \Delta_2 \vdash \Gamma$ is valid, then for all $i \in I$
  * either $\exists X \in \Gamma : i \in M(X) \Longrightarrow$ the sequent $a, b, \Delta_1 \mid \Delta_2 \vdash \Gamma$ is valid
  * or $\exists L \in \Delta_1 \cup \Delta_2 : i \notin M(L) \Longrightarrow$ the sequent $a, b, \Delta_1 \mid \Delta_2 \vdash \Gamma$ is valid
  * or $i \notin M(a \sqcap_A b) \Longrightarrow (i_t, i_A) \notin M(a \sqcap_A b)$
    $\Longrightarrow (i_t, i_A) \notin M_\emptyset(a \sqcap_A b) \cap M_\emptyset(th)$ def. of M
    $\Longrightarrow (i_t, i_A) \notin M_\emptyset(a \sqcap_A b) \wedge (i_t, i_A) \notin M_\emptyset(th)$
    $\Longrightarrow i_A \notin M_A(a \sqcap_A b) \wedge (i_t, i_A) \notin M_\emptyset(th)$ semantics of $\models_\emptyset$
    $\Longrightarrow i_A \notin M_A(a) \cap M_A(b) \wedge (i_t, i_A) \notin M_\emptyset(th)$ $(cp_{\sqcap_A})$
    $\Longrightarrow i_A \notin M_A(a) \wedge (i_t, i_A) \notin M_\emptyset(th)$
    $\Longrightarrow \forall j \in 2^T : (j, i_A) \notin M_\emptyset(a) \wedge (i_t, i_A) \notin M_\emptyset(th)$ lemma 13
    $\Longrightarrow (i_t, i_A) \notin M_\emptyset(a) \wedge (i_t, i_A) \notin M_\emptyset(th)$
    $\Longrightarrow i \notin M_\emptyset(a) \cap M_\emptyset(th)$
    $\Longrightarrow i \notin M(a)$ def. of M
    $\Longrightarrow$ the sequent $a, b, \Delta_1 \mid \Delta_2 \vdash \Gamma$ is valid.

- Let us show that the $\sqcup$-Rule preserves validity. Assume the sequent $\neg(a \sqcup_A b), \Delta_1 \mid \Delta_2 \vdash \Gamma$ is valid, then for all $i \in I$
  * either $\exists X \in \Gamma : i \in M(X) \Longrightarrow$ the sequent $\neg a, \neg b, \Delta \vdash \Gamma$ is valid
  * or $\exists L \in \Delta_1 \cup \Delta_2 : i \notin M(L) \Longrightarrow$ the sequent $\neg a, \neg b, \Delta_1 \mid \Delta_2 \vdash \Gamma$ is valid
  * or $\exists c \in a \sqcup_A b : i \notin M(\neg c) \Longrightarrow \exists c \in a \sqcup_A b : (i_t, i_A) \notin M(\neg c)$
    $\Longrightarrow \exists c \in a \sqcup_A b : (i_t, i_A) \in M(c) \vee (i_t, i_A) \notin M_\emptyset(th)$ lemma 11
    $\Longrightarrow \exists c \in a \sqcup_A b : (i_t, i_A) \in M(c) \vee (i_t, i_A) \notin M_\emptyset(\neg a) \cap M_\emptyset(th)$
    $\Longrightarrow \exists c \in a \sqcup_A b : (i_t, i_A) \in M(c) \vee (i_t, i_A) \notin M(\neg a)$
    $\Longrightarrow \exists c \in a \sqcup_A b : (i_t, i_A) \in M_\emptyset(c) \wedge (i_t, i_A) \in M_\emptyset(th)$
    $\vee (i_t, i_A) \notin M(\neg a)$ def. or M
    $\Longrightarrow \exists c \in a \sqcup_A b : i_A \in M_A(c) \wedge (i_t, i_A) \in M_\emptyset(th)$
    $\vee (i_t, i_A) \notin M(\neg a)$ semantics of $\models_\emptyset$
    $\Longrightarrow i_A \in \bigcup_{c \in a \sqcup_A b} M_A(c) \wedge (i_t, i_A) \in M_\emptyset(th) \vee (i_t, i_A) \notin M(\neg a)$
    $\Longrightarrow i_A \in M_A(a) \cup M_A(b) \wedge (i_t, i_A) \in M_\emptyset(th)$
    $\vee (i_t, i_A) \notin M(\neg a)$ $(cs_{\sqcup_A})$
    $\Longrightarrow \forall j \in 2^T : (j, i_A) \in M_\emptyset(a) \cup M_\emptyset(b) \wedge (i_t, i_A) \in M_\emptyset(th)$
    $\vee (i_t, i_A) \notin M(\neg a))$ lemma 13
    $\Longrightarrow (i_t, i_A) \in M_\emptyset(a) \cup M_\emptyset(b) \wedge (i_t, i_A) \in M_\emptyset(th)$
    $\vee (i_t, i_A) \notin M(\neg a))$
    $\Longrightarrow i \in M(a) \vee i \in M(b) \vee (i_t, i_A) \notin M(\neg a)$ def. of M
    $\Longrightarrow i \notin M(\neg a) \vee i \notin M(\neg b) \vee i \notin M(\neg a)$ lemma 9
    $\Longrightarrow i \notin M(\neg a) \vee i \notin M(\neg b)$
    $\Longrightarrow$ the sequent $\neg a, \neg b, \Delta_1 \mid \Delta_2 \vdash \Gamma$ is valid.
- Let us show that the *def*-Rule preserves validity. Assume the sequent $t, \Delta_1 \mid \Delta_2 \vdash \neg C, \Gamma$ is valid, then for all $i \in I$
  * either $\exists X \in \Gamma : i \in M(X) \Longrightarrow$ the sequent $a, b, \Delta_1 \mid \Delta_2 \vdash \Gamma$ is valid
  * or $\exists L \in \Delta_1 : i \notin M(L) \Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2, t \vdash \Gamma$ is valid
  * or $\exists L \in \Delta_2 : i \notin M(L) \Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2, t \vdash \Gamma$ is valid
  * or $i \notin M(t) \Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2, t \vdash \Gamma$ is valid
  * or $i \in M(\neg C) \Longrightarrow i \notin M(C)$ lemma 10
    $t \dot{\leq} C \Longrightarrow M(t) \subseteq M(C)$ Lemmas 1 and 2
    $\Longrightarrow i \in M(t) \Rightarrow i \in M(C) \Longrightarrow i \notin M(C) \Rightarrow i \notin M(t)$
    $\Longrightarrow i \notin M(t)$
    $\Longrightarrow$ the sequent $\Delta_1 \mid \Delta_2, t \vdash \Gamma$ is valid

  The $\neg def_{\dot{\leq}}$-Rule can be proved in a similar way.
- It is easy to recognize that the inference rules $\neg\neg$-Rule, β-Rule,

α-Rule and literal-Rule also preserve validity, as well as *term*-Axiom. As a consequence, every provable sequent is valid.

Now for any $f, g \in AS$, if $f \sqsubseteq g$ then $\vdash \neg f \vee g$ is a provable sequent
$\Longrightarrow$ this sequent is valid
$\Longrightarrow \bigcap_{\delta\in\emptyset} M(\delta) \subseteq \bigcup_{\gamma\in\{\neg f\vee g\}} M(\gamma) \Longrightarrow M(\neg f \vee g) = I$
$\Longrightarrow (I \setminus M(f)) \cup M(g) = I \Longrightarrow M(f) \subseteq M(g)$.

This proves $cs_{\sqsubseteq}$.

■

- $cp_{\sqsubseteq} \Leftarrow cs_{\sqcap_A} \wedge cp_{\sqcup_A} \wedge reduced_A$
Proof:
  - Now, one proves that $\sqcap$-Rule preserves non-validity. Let us assume that $a \sqcap_A b, \Delta \vdash \Gamma$ is not valid.
  Then $\exists i \in I : (i \in M(a \sqcap_A b) \wedge \forall L \in \Delta : i \in M(L) \wedge \forall X \in \Gamma : i \notin M(X))$
  $\Longrightarrow \exists i \in I : (i \in M(a) \wedge i \in M(b) \wedge \forall L \in \Delta : i \in M(L) \wedge \forall X \in \Gamma : i \notin M(X))$ $\quad cs_{\sqcap_A}$
  $\Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is not valid.
  - Now, one proves that $\sqcup$-Rule preserves non-validity. Let us assume that $\neg(a \sqcup_A b), \Delta \vdash \Gamma$ is not valid.
  Then $\exists i \in I : (\forall c \in a \sqcup_A b : i \in M(\neg c), \forall L \in \Delta : i \in M(L), \forall X \in \Gamma : i \notin M(X))$
  $\Longrightarrow \exists i \in I : \forall c \in a \sqcup_A b : i \in M_\emptyset(\neg c) \cap M_\emptyset(th)$
  $\Longrightarrow \exists i \in I : \forall c \in a \sqcup_A b : i \in M_\emptyset(\neg c) \wedge i \in M_\emptyset(th)$
  $\Longrightarrow \exists i \in I : \forall c \in a \sqcup_A b : i \in M(\neg c) \wedge i \in M_\emptyset(th)$
  $\Longrightarrow \exists i \in I : \forall c \in a \sqcup_A b : i \notin M(c) \wedge i \in M_\emptyset(th)$ (lemma 10)
  $\Longrightarrow \exists i \in I : \nexists c \in a \sqcup_A b : i \in M(c) \wedge i \in M_\emptyset(th)$
  $\Longrightarrow \exists i \in I : i \notin \bigcup_{c\in a\sqcup_A b} M(c) \wedge i \in M_\emptyset(th)$
  $\Longrightarrow \exists i \in I : i \notin M(a) \cup M(b) \wedge i \in M_\emptyset(th)$ $\quad (cp_{\sqcup_A})$
  $\Longrightarrow \exists i \in I : i \notin M(a) \wedge i \notin M(b) \wedge i \in M_\emptyset(th)$
  $\Longrightarrow \exists i \in I : (i \in M(\neg a) \vee i \notin M_\emptyset(th)) \wedge (i \in M(\neg b) \vee i \notin M_\emptyset(th)) \wedge i \in M_\emptyset(th)$ (lemma 12)
  $\Longrightarrow \exists i \in I : (i \in M(\neg a) \wedge i \in M(\neg b)) \vee (i \notin M_\emptyset(th) \wedge i \in M_\emptyset(th))$
  $\Longrightarrow \exists i \in I : i \in M(\neg a) \wedge i \in M(\neg b)$
  $\Longrightarrow \neg a, \neg b, \Delta \vdash \Gamma$ is not valid.
  - Now, one proves that *def*-Rule preserves non-validity. Assume the sequent $t, \Delta_1 \mid \Delta_2 \vdash \neg C, \Gamma$ is not valid.
  Then $\exists i \in I : (i \in M(t), \forall L \in \Delta_1 : i \in M(L), \forall L \in \Delta_2 : i \in M(L), i \notin M(\neg C), \forall X \in \Gamma : i \notin M(X))$
  $\Longrightarrow \exists i \in I : (i \in M(t), \forall L \in \Delta_1 : i \in M(L), \forall L \in \Delta_2 : i \in$

$M(L), \forall X \in \Gamma : i \notin M(X))$
$\Longrightarrow \Delta_1 \mid \Delta_2, t \vdash \Gamma$ is not valid.
The $\neg def_{\dot{\leq}}$-Rule can be proved in a similar way.

- It is easy to check that $\neg\neg$-Rule, α-Rule, β-Rule, and literal-Rule, as well as *term*-Rule and *term'*-Rule also preserve non-validity.
- Now, assume $f \not\sqsubseteq g$. Let t be a fully developed tree of $\vdash \neg f \vee g$.
  Then t is open because $f \not\sqsubseteq g$.
  $\Longrightarrow t$ has an open leaf sequent $s = \Delta_1 \mid \Delta_2 \vdash \Gamma$.
  As s is open, no rule applies, hence $\Delta_2 = \emptyset$ and $\Gamma = \emptyset$.
  Δ_1 is of the form $A, \overline{B}, T_1, \overline{T_2}$, where A are atoms in AS_A, $\overline{B}$ are negations of atoms, T_1 are terms, and $\overline{T_2}$ are negations of terms.
  Because s is open, we have $open_A(A, B)$ and $T_1 \cap T_2 = \emptyset$.
  * $open_A(A, B) \Longrightarrow \bigcap_{a \in A} M_A(a) \not\subseteq \bigcup_{b \in B} M_A(b) \qquad (reduced_A)$
    $\Longrightarrow \exists i_A \in I_A : \forall a \in A : i_A \models_\emptyset a \wedge \forall b \in B : i_A \not\models_\emptyset b$
  * $T_1 \cap T_2 = \emptyset \Longrightarrow \exists i_T = T_1 : \forall t_1 \in T_1 : i_T \models_\emptyset t_1 \wedge \forall t_2 \in T_2 : i_T \not\models_\emptyset t_2$

  Hence $\exists i = (i_T, i_A) : \forall \delta^+ \in A \cup T_1 : i \models_\emptyset \delta^+, \forall \delta^- \in B \cup T_2 : i \not\models_\emptyset \delta^-$
  $\Longrightarrow \exists i \in I : \forall \delta^+ \in A \cup T_1 : i \models_\emptyset \delta^+, \forall \delta^- \in B \cup T_2 : i \models_\emptyset \neg\delta^-$
  $\Longrightarrow \exists i \in I : \forall \delta \in \Delta : i \models_\emptyset \delta \wedge \forall \gamma \in \Gamma : i \not\models_\emptyset \gamma$
  $\Longrightarrow \exists i \in I : i \not\models_\emptyset \Delta_1 \mid \Delta_2 \vdash \Gamma$
  From lemma 14, $\forall t_1 \in T_1 : t_1 \dot{\leq} C_1 \in th \Rightarrow i \models_\emptyset C_1$.
  $\Longrightarrow \forall t_1 \in T_1 : t_1 \dot{\leq} C_1 \in th \Rightarrow i \models_\emptyset C_1 \wedge i \models_\emptyset t_1$
  $\Longrightarrow \forall t_1 \in T_1 : t_1 \dot{\leq} C_1 \in th \Rightarrow i \models_\emptyset t_1 \dot{\leq} C_1$
  Also, $\forall t_2 \notin T_1 : i \not\models_\emptyset t_2 \ (t_2 \notin i_T)$
  $\Longrightarrow \forall t_2 \notin T_1 : t_2 \dot{\leq} C_2 \in th \Rightarrow i \models_\emptyset t_2 \dot{\leq} C_2$ semantics of $\dot{\leq}$
  Hence, $\forall t \dot{\leq} C \in th : i \models_\emptyset t \dot{\leq} C$
  $\Longrightarrow i \models_\emptyset th$
  Hence $\exists i \in I : \forall \delta \in \Delta : i \models_\emptyset th \wedge i \models_\emptyset \delta$
  $\Longrightarrow \exists i \in I : \forall \delta \in \Delta : i \in M_{th}(\delta)$ def. of M_{th}
  $\Longrightarrow \bigcap_{\delta \in \Delta} M_{th}(\delta) \not\subseteq \emptyset$
  $\Longrightarrow \bigcap_{\delta \in \Delta} M_{th}(\delta) \not\subseteq \bigcup_{\gamma \in \emptyset} M_{th}(\gamma)$
  $\Longrightarrow$ the sequent s is non-valid
  $\Longrightarrow$ the sequent $\vdash \neg f \vee g$ is non-valid because all rules preserve the non-validity
  $\Longrightarrow M_{th}(f) \not\subseteq M_{th}(g)$.
  Hence $cp_{\sqsubseteq}$ (the absence of proof entails the non-validity).

■

- $cp'_{\sqsubseteq} \Leftarrow cs_{\sqcap_A} \wedge cp_{\sqcup_A} \wedge reduced'_A$

Proof: Let $f \in \mathit{TELL}, g \in \mathit{ASK}$ and $t \in \mathit{Trees}(\neg f \vee g)$.

From the inference rules, it follows that forall $A, \overline{B} \vdash\in t$, A contains only positive atoms of f and negative atoms of g. Reciprocally, B contains only negative atoms of A and positive atoms of g. Hence B is included in ASK_A, and A is included in $\mathit{TELL}_A \cup \mathit{ASK}_A$. Moreover, from the definitions of syntax and inference rules, we also have that A always contains an atom in TELL_A. In summary
$\forall A, \overline{B} \vdash\in t : (\exists a \in A : a = \bot_A \vee a \in \mathit{TELL}_A) \wedge (\forall a \in A : a \in \mathit{TELL}_A \cup \mathit{ASK}_A \cup \{\bot_A\}) \wedge (\forall b \in B : b \in \mathit{ASK}_A)$
$\Longrightarrow \forall A, \overline{B} \vdash\in t : \mathit{reduced}_A(A, B)$ $(\mathit{reduced}'_A)$
$\Longrightarrow \mathit{open}(t) \Rightarrow \exists A, \overline{B} \vdash\in t : \mathit{open}_A(A, B) \wedge \mathit{reduced}_A(A, B)$
$\Longrightarrow \forall t \in \mathit{Trees}(\vdash \neg f \vee g) : \mathit{open}(t) \Rightarrow \exists A, \overline{B} \vdash\in t : \mathit{open}_A(A, B) \wedge \mathit{reduced}_A(A, B)$
$\Longrightarrow \mathit{cp}(f, g).$ $(\mathit{cs}_{\sqcap_A},\ \mathit{cp}_{\sqcup_A})$ ∎

The following properties can be proved trivially. Indeed, the properties are trivially true when considering $M_\emptyset$. We then consider M, which is only a subset of $M_\emptyset$, and the properties remain true.

- $\mathit{def}_\sqcap \wedge \mathit{cs}_\sqcap \wedge \mathit{cp}_\sqcap$
- $\mathit{def}_\sqcup \wedge \mathit{cs}_\sqcup \wedge \mathit{cp}_\sqcup$
- $\mathit{def}_\top \wedge \mathit{cp}_\top$
- $\mathit{def}_\bot \wedge \mathit{cs}_\bot$
- $\mathit{reduced}(F, G) \Leftarrow \forall f \in F, g \in G : \mathit{cp}_\sqsubseteq(f, g)$
  $\mathit{reduced} \Leftarrow \mathit{cp}_\sqsubseteq$
  $\mathit{reduced}' \Leftarrow \mathit{cp}'_\sqsubseteq$

  Proof:

  Assume we have $F \not\sqsubseteq^* G$, $\mathit{closed}_\sqcap(F)$, $\mathit{closed}_\sqcup(G)$.
  Then neither conjunction on F, nor disjunction on G can be applied. As these operations are totally defined in *Prop*, we have $F = \{f\}$, and $G = \{g\}$.
  Then we also have $f \not\sqsubseteq g$ Definition of $\sqsubseteq^*$
  $\Longrightarrow M(f) \not\subseteq M(g)$ if $\mathit{cp}_\sqsubseteq(f, g)$
  $\Longrightarrow \bigcap_{f \in F} M(f) \not\subseteq \bigcup_{g \in G} M(g).$ ∎

Chapter 5

Application to Description Logics

5.1 Reconstructing $\mathcal{ALC}$ with a Terminology

We show in this section that the decision procedure of $\mathcal{ALC}$ can be derived automatically, including now a terminology. We recall the recomposition of $\mathcal{ALC}$ without the terminology, called ALC, as defined in Section 3.2.2:

$$ALC = \mathit{Prop}(\mathit{Set}(\mathit{Sum}(\mathit{Atom}, \mathit{Prod}(\mathit{Atom}, ALC))))$$

Now, let us compose the description logic $\mathcal{ALC}$ including a terminology:

$$ALC_t = \mathit{Prop}(\mathit{Set}(\mathit{Prod}(\mathit{Atom}, ALC_t)))$$

We note that $\mathit{Sum}(\mathit{Atom}, \ldots)$ has disappeared in the latter case. Indeed, the logic functor *Prop* allows the definition of terms, which here play the role of atoms.

The decision procedure of ALC_t is proved consistent (property $cs_{\sqsubseteq}$) and complete (property $cp_{\sqsubseteq}$) by combining the properties of the various logic functors. We now derive the syntax and semantics of ALC_t from its definition, and from the definition of logic functors, including the functor *Prop* allowing the use of a theory.

The abstract syntax of ALC_t is defined by the grammar rule

$$C \to T \mid (R, C) \mid \neg\, C \mid C \,\wedge\, C \mid C \,\vee\, C \mid 1 \mid 0,$$

where T are terms from *Prop*, and stand for *concept names*, and R are from the functor *Atom*, and stand for *roles*. This makes the syntax of ALC_t equivalent to the syntax of $\mathcal{ALC}$.

Axioms are of the form $t \mathrel{\dot{\leq}} C$, where t is a term and C a complex expression, which enable to specify a terminology.

The interpretation domain of ALC_t is $I = \mathcal{P}(T) \times \mathcal{P}(R \times I)$ where T is the domain of terms and R is the domain of roles. As a convention, we will represent terms with a capital first letter and atoms with a lowercase first letter. Thus, *quote* is an atom and *Article* is a term.

Let us now go back to our example of Section 3.2.2 in logic $\mathcal{ALC}$. We can now define the following axiom in the *Prop* logic:

$$\textit{DL-Article} \;\dot{\leq}\; \textit{Article} \sqcap \exists\ \textit{cite}.\textit{DL-Article}$$

Then, we can prove the following goal

$$\begin{array}{rl} & \textit{DL-Article} \sqcap \forall \textit{cite}.\textit{Interesting} \\ \sqsubseteq & \exists \textit{cite}.(\textit{DL-Article} \sqcap \textit{Interesting}) \sqcap \forall \textit{cite}.(\textit{DL-Article} \rightarrow \textit{Interesting}) \end{array}$$

Let us suppose we now want to provide $\mathcal{ALC}$ with a role hierarchy (extension $\mathcal{H}$ of description logics as defined in 2.4) and concrete domains, e.g. interval of integers and string patterns (e.g. "contains"). Then, we can use *Taxo* instead of *Atom* to represent roles, and add *String* as well as *Interval*(*Int*) to represent concrete domains.

$$ALC'_t = \textit{Prop}(\textit{Set}(\textit{Sum}(\textit{Sum}(\textit{String}, \textit{Interval}(\textit{Int})), \textit{Prod}(\textit{Taxo}, ALC'_t))))$$

The decision procedure of ALC'_t is proved consistent (property $cs_{\sqsubseteq}$) and complete (property $cp_{\sqsubseteq}$) by combining the properties of the various logic functors. We note here that logic functors allow us to define variants of logics almost for free.

We will use the syntax $a..b$ for the interval $[a; b]$ (for example 2002..2006), and the functor *Interval* also allows the use of degenerate intervals (for example, 2002 stands for 2002..2002). The *String* functor allows us to express operations such as `is` (string equality) and `contains` (string inclusion).

We can thus define the following axioms in the *Taxo* logic of ALC'_t:

$$\begin{array}{lcl} \textit{submissionYear} & \dot{\leq} & \textit{year} \\ \textit{publicationYear} & \dot{\leq} & \textit{year} \end{array}$$

And we can then prove the following goal:

$$\begin{array}{rl} & \textit{Article} \sqcap \exists \textit{submissionYear}.2004 \sqcap \exists \textit{publicationYear}.2006 \\ \sqsubseteq & \textit{Publication} \sqcap \exists \textit{year}.2005..2007 \end{array}$$

More examples of ALC'_t are given in the following section.

5.2 Handling Bibliographic References in XML

Let us suppose we have access to a set of bibliographic references expressed in XML [BPSM+06]. For example, BibTex references can easily be converted to XML references, with a simple parser and printer, for example Bib2Xml[1].

[1] http://www-plan.cs.colorado.edu/henkel/stuff/bib2xml/

An example of two bibliographic references expressed in XML is given below:

```
<article>
   <title>In the Mood for Description Logics</title>
   <author>Wong Kar-Wai</author>
   <submission-year>1999</submission-year>
   <publication-year>2000</publication-year>
   <keyword>DescriptionLogic</keyword>
   <keyword>Concept</keyword>
</article>

<book>
   <title>My Logic Functors Nights</title>
   <author>K.W. Wong</author>
   <submission-year>2007</submission-year>
   <keyword>Night</keyword>
   <keyword>LogicFunctor</keyword>
</book>
```

Note that the chosen structure is simple, although both XML and description logics allow more complicated structures. However, for simplicity reasons, we will restrict ourselves to this simple data structure.

To handle such XML documents, we use the logic ALC'_t.

We choose to convert the general type of XML documents into a term (for example *Article*) and the tags `<name>value</name>` into formulas of the form $\exists name.value$. Titles and authors will be converted into strings, years into integers and keywords into terms. In order to perform such a conversion, one needs to implement what we call a *transducer*, which takes as argument an XML file and returns formulas of the logics.

Following these rules, the two bibliographic references will be converted into the following formulas:

$$\begin{aligned} d_1 = \; & Article \\ \sqcap \; & \exists title.\, \texttt{is "In the Mood for Description Logics"} \\ \sqcap \; & \exists author.\, \texttt{is "Wong Kar-Wai"} \\ \sqcap \; & \exists submissionYear.1999 \sqcap \exists publicationYear.2000 \\ \sqcap \; & \exists keyword.DescriptionLogic \sqcap \exists keyword.Concept \end{aligned}$$

$$\begin{aligned} d_2 = \; & Book \\ \sqcap \; & \exists title.\, \texttt{is "My Logic Functors Nights"} \\ \sqcap \; & \exists author.\, \texttt{is "K.W. Wong"} \\ \sqcap \; & \exists submissionYear.2007 \\ \sqcap \; & \exists keyword.Night \sqcap \exists keyword.LogicFunctor \end{aligned}$$

We then define the following axioms in the *Prop* functor:

$$\begin{array}{rcl} \mathit{DescriptionLogic} & \dot{\leq} & \mathit{Logic} \\ \mathit{LogicFunctor} & \dot{\leq} & \mathit{Logic} \\ \mathit{Article} & \dot{\leq} & \mathit{Publication} \\ \mathit{Book} & \dot{\leq} & \mathit{Publication} \end{array}$$

And we define the following axioms in the *Taxo* functor:

$$\begin{array}{rcl} \mathit{submissionYear} & \dot{\leq} & \mathit{year} \\ \mathit{publicationYear} & \dot{\leq} & \mathit{year} \end{array}$$

Thus, we can express the two following requests for $i = 1$ and $i = 2$:

$d_i \sqsubseteq \exists author.$ `contains "Wong"` $\sqcap \exists keyword.Logic \sqcap \exists year.2000..2007$

As one expects, both requests (with d_1 and d_2) can be proved true.

However, the semantics style of logic functors does not limit applications to variants of descriptions logics. Other logics have been built in other styles: e.g., a logic of programing types for retrieving software components, a logic to manage geographic coordinates [BFRQ06] or a logic to manage taxonomies on pictures.

5.3 Implementation

The implementation has been done in Caml, by modifying the existing logic functor *Prop*, as well as adding the new logic functor *Taxo*. This is available in the LogFun implementation. The XML bibliographic references example has also been implemented (including a transducer based on Bib2Xml).

This makes possible the use of a terminology expressed in description logics in Camelis. The Camelis software[2], which is an implementation of a logical information system, allows to manage various kinds of documents, e.g. mp3 or pictures. Camelis is independent of the logic, that is every end user can choose the logic he wants to use. Therefore, taxonomies and terminologies can now be handled in Camelis without any change in the source code of Camelis.

[2]http://www.irisa.fr/LIS/ferre/camelis/

Chapter 6

Conclusion and Future Works

We are now able to handle logic functors with a knowledge base (theory). In particular, we can handle taxonomies on atoms (logic functor *Taxo/0/1*) or on concrete domains (logic functor *Domain/1/1*), and terminologies (logic functor *Prop/1/1*). Various properties have been proved for these three functors, so that the automatic theorem prover can check if their composition with other functors is valid, and if not, why.

This allows us to recompose various logics including description logics such as $\mathcal{ALC}$ or $\mathcal{ALCH}$ ($\mathcal{ALC}$ with a role hierarchy), including their terminology. This is now part of the LogFun implementation. The produced logics can be used independently or embedded in a software such as Camelis.

We now give some perspectives in order to improve the handling of theories in logic functors.

6.1 Improving the expressivity of Prop

The first improvement will be to allow the use of the definition of axioms of the form $f \doteq g$ in the logic functor *Prop*. Presently, it is only possible to define axioms of the form $f \dot{\leq} g$. However, this is a more difficult problem than it could seem to. Indeed, in the proof of $cp_{\sqsubseteq}$, we exhibit an interpretation i which is not a model of the sequent, but which is a model of the theory. We thus have to prove that this i will model every axiom of the form $f \doteq g$, which is a little tricky. In particular, this proof would certainly need to add and prove the property cs_{th}, stating that every theory used is consistent.

We could also possibly consider only the axioms $f \doteq g$, where $f, g \in ASK$.

Moreover, it would be interesting to allow a full expression in the left part of an axiom. However, this would lead to a loss of efficiency in the case

of a big theory. But there is *a priori* no theoretical limit to allow such a thing. Suppose we want to test whether $f \sqsubseteq g$. Instead of proving $\neg f \vee g$ and using the new rules for the axioms in the sequent calculus, we can prove the goal $\neg th \vee \neg f \vee g$ where th stands for the conjunction of axioms of the theory. But one easily understands that, in case of a big theory, this will lead to a loss of efficiency, as the formula to prove will be very big, and the proof tree size will thus increase dramatically with the theory size.

6.2 Recomposing the Description Logic $\mathcal{SHOIQ}$

Another interesting improvement would be to recompose the description logic $\mathcal{SHOIQ}$, which is widely used on the Semantic Web. Indeed, it has been recognized as a standard by the W3C to support the OWL-DL and OWL-Lite sub-languages of the Web Ontology Language (OWL).

In the name $\mathcal{SHOIQ}$, as described in 2.4, $\mathcal{S}$ stands for $\mathcal{ALCR}^+$, where $\mathcal{R}^+$ is the transitivity on roles, $\mathcal{H}$ stands for the role hierarchy, $\mathcal{O}$ allows to use individuals in the concepts, $\mathcal{I}$ stands for inverse roles and $\mathcal{Q}$ stands for the qualified cardinality restrictions on roles. Note that the role hierarchy ($\mathcal{H}$) is already implemented, as shown in section 5.1.

Transitive roles ($\mathcal{R}^+$) allow to consider axioms of the form $ancestor \doteq parent^+$, e.g. a parent is an ancestor, and the parent of an ancestor is an ancestor as well. A solution which could be implemented rather easily would be to consider that transitive roles are defined as follows:

$$\exists \mathtt{r}^+.\mathtt{C} \doteq \exists \mathtt{r}.(\mathtt{C} \sqcup \exists \mathtt{r}^+.\mathtt{C})$$

Thus, we can replace every occurrence of $\exists \mathtt{r}^+.\mathtt{C}$ with X in the goal to prove, and define the following axiom:

$$X \doteq \exists \mathtt{r}.(\mathtt{C} \sqcup X)$$

This could be soon implemented, after some preprocessing on formulas.

This would allow to prove for example that someone having a parent whose parent is tall is more specific than someone having a tall ancestor ($parent^+$), which is expressed by

$$\exists parent.\exists parent.Tall \sqsubseteq \exists parent^+.Tall$$

However, note that the transitivity form $^+$ (e.g. $parent^+$) is not the only one, e.g. one might consider possibly degenerate transitivity with *.

Inverse roles ($\mathcal{I}$) allow to consider axioms of the form $child \doteq parent^-$. Intuitively, child is the inverse relation of parent. No investigation has been done yet in order to integrate this notion to logic functors.

Qualified cardinality restrictions on roles ($\mathcal{Q}$) allow to consider axioms of the form $\geq 2 child.Researcher$, meaning “more than 2 children being researchers”. This has not been studied at all, and could be tricky to integrate to logic functors.

6.3 Handling More General Theories

We considered the case of terminological theories, where axioms are composed of two formulas linked by an operator. This approach is adopted in most description logics. However, we could consider other operators than $\doteq$ or $\dot{\leq}$. Moreover, nothing prevents us from considering different kinds of theories, where axioms could have even more general forms than two formulas linked by an operator. Depending on the possible applications, other cases can be studied.

Acknowledgments

I would like to thank Sébastien Ferré for his useful suggestions and discussions, Olivier Bedel for his review of this report, and my table football partners at Irisa.

Bibliography

[BFRQ06] Olivier Bedel, Sébastien Ferré, Olivier Ridoux, and Erwan Quesseveur. GEOLIS: A logical information system for geographical data. In *Int. Conf. Spatial Analysis and GEOmatics – SAGEO 2006*, 2006.

[BP95] Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[BPSM+06] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (XML) 1.1 (second edition). World Wide Web Consortium, 2006.
http://www.w3.org/TR/2006/REC-xml11-20060816.

[BS85] Ronald J. Brachmann and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–202, April 1985.

[CLN98] Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Description logics for conceptual data modeling. In *Logics for Databases and Information Systems*, pages 229–263, 1998.

[dCFG+01] Luis Fariñas del Cerro, David Fauthoux, Olivier Gasquet, Andreas Herzig, Dominique Longin, and Fabio Massacci. Lotrec The Generic Tableau Prover for Modal and Description Logics, 2001.

[DLNN97] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. *Information and Computation*, 134(1):1–58, 1997.

[DP90] B.A. Davey and H.A. Priestley. Introduction to lattices and order. *Cambridge University Press*, 1990.

[FR02] Sébastien Ferré and Olivier Ridoux. A framework for developing embeddable customized logics. In A. Pettorossi, editor,

Int. Work. Logic-based Program Synthesis and Transformation, LNCS 2372, pages 191–215. Springer, 2002.

[FR04] S. Ferré and O. Ridoux. An introduction to logical information systems. *Information Processing & Management*, 40(3):383–419, 2004.

[FR06] Sébastien Ferré and Olivier Ridoux. Logic functors: A toolbox of components for building customized and embeddable logics. Research report, Irisa, 2006.

[Hor97] Ian Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

[HS01] Ian Horrocks and Ulrike Sattler. Ontology reasoning in the SHOQ(D) description logic. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

[HS05] Ian Horrocks and Ulrike Sattler. A tableaux decision procedure for $\mathcal{SHOIQ}$. In *IJCAI*, pages 448–453, 2005.

[KM06] Yevgeny Kazakov and Boris Motik. A resolution-based decision procedure for $\mathcal{SHOIQ}$. In *IJCAR*, pages 662–677, 2006.

[Lev90] Hector J. Levesque. All i know: a study in autoepistemic logic. *Artif. Intell.*, 42(2-3):263–309, 1990.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Symbolic computation — Artificial Intelligence. Springer, Berlin, 1987.

[Mac88] D. MacQueen. An implementation of standard ML modules. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming, Snowbird, UT*, pages 212–223, New York, NY, 1988. ACM.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[Nap97] Amedeo Napoli. Une introduction aux logiques de descriptions. Research report, Inria, 1997.

[Rij87] C J Van Rijsbergen. A new theoretical framework for information retrieval. *SIGIR Forum*, 21(1-2):23–29, 1987.