Formal verification of a proof procedure for the description logic $\mathscr{ALC}$

Mohamed Chaabani Mohamed Mezghiche
LIMOSE, University of Boumerdes
Boumerdes, Algeria
chaabani@umbb.dz mohamed.mezghiche@yahoo.fr

Martin Strecker
IRIT (Institut de Recherche en Informatique de Toulouse)
Université de Toulouse
strecker@irit.fr

Description Logics (DLs) are a family of languages used for the representation and for reasoning about the knowledge base of an application domain, in a structured and formal manner.To achieve this objectif, sevral provers have been implemented such as RACER and FACT++, but these provers themselves have not been certified. In order to insure the soundness of derivations in these DLs, it is necessary to verify formally the deductions applied by these reasoners. Formal Methods offer powerful tools for the specification and verification of proof procedures, among them methods of proving properties such as soundness, completeness and termination of a proof procedure.

In this paper, we present the definition of a proof procedure for the Description Logic $\mathscr{ALC}$, based on a tableau method. We then prove the soundness, completeness and termination of our reasoner with the proof assistant Isabelle. The proof proceeds in two phases, by first establishing these properties on an abstract, set theoretic level, and by instantiating them with an implementation based on lists.

1 Introduction

In this paper, we present a definition of a prover for the description logic $\mathscr{ALC}$ [18] which is based on the method of semantic tableau [5]. We ensure the validity of our method and its implementation by the proof of the properties of its soundness, its completeness and its termination. These proofs are performed using the Isabelle/HOL proof assistant.

Description Logics (DL) are formalisms widely used in several areas such as the Semantic Web and ontology construction. Several description logics, eg, $\mathscr{SHOIQ}$ [12] and $\mathscr{SHOIN}$ [3] are more expressive extensions of $\mathscr{ALC}$. $\mathscr{SHOIN}$, which is considered as a formal model of Web Ontology Language OWL-DL, is used in several provers such as FaCT++ [19] and RACER [10] .

Only a formal proof of the validity of the reasoning process applied to DLs can ensure the correctness of derivations of properties in these logics. This is far from being the case for provers available today. The inference engines of provers like RACER and FaCT++ have not yet been certified. In [13], the authors give a detailed description of the various problems posed by incomplete and incorrect provers yet widely used.

Of the many methods used as decision procedures for DLs, the semantic tableau method is the most common. Indeed, FaCT and RACER use it in their reasoning process. We propose in this paper a definition of a prover based on a semantic tableau method for the description logic $\mathscr{ALC}$. We will also present the formal proof in the Isabelle/HOL proof assistant of soundness, completeness and termination of the prover.

Adel Bouhoula, Tetsuo Ida and Fairouz Kamareddine (Eds.):
Symbolic Computation in Software Science 2012 (SCSS2012)
EPTCS 122, 2013, pp. 51–61, doi:10.4204/EPTCS.122.5

Work close to our formalization of proof procedures are reported in [16], [20] and [17]. The first describes the formalization of a prover for first-order logic in the Isabelle/HOL proof assistant. The second presents a formalization in the Coq proof assistant of some modal logics, and the third a formalization of a prover for LTL. Please note that DLs can be considered as specific modal logics. Closest to our work comes [11], which describes the formalization of $\mathscr{ALC}$ in the PVS proof assistant. Our formalization simplifies the termination argument, see Section 8, and allows to extract a prover directly executable in a " standard" programming language (Caml). This paper builds on and extends our previous work described in [7], which provides an abstract, non-executable proof procedure for $\mathscr{ALC}$, that extension is presented in [8].

This paper is organized as follows: In Sections 2, 3 and 4, we detail the description logics. In Section 5, we present the procedure of semantic tableau for $\mathscr{ALC}$ and the Sections 6 and 7, we describe the formalization of this method and present the proof of soundness and completeness properties in the Isabelle/HOL proof assistant. In the rest of the article, we present an implementation of a method of proof for the logic $\mathscr{ALC}$ and the proof of its soundness and its termination.

The development described here [1] was carried out in the environment of the Isabelle/HOL proof assistant [14] whose logic HOL is a classical logic. However, the background of the development is largely independent of Isabelle and could easily be simulated in other proof assistants.

2 Description logics

Description Logics [1, 2, 4, 15] are a family of knowledge representation languages which can be used to represent knowledge of an application domain in a structured and formal way. A fundamental characteristic of these languages is that they have a formal semantics. Description logics are used for various applications. Among them are: The representation of ontologies [3], natural language processing [9] and representation of the semantics of UML class diagrams [6].

We recall that description logics have as a common basis $\mathscr{AL}$ enriched with different extensions: The description logic $\mathscr{ALC}$, subject of this work, adds negation to $\mathscr{AL}$. Other extensions add the transitive closure of roles, number restrictions on roles, the notion of sub-roles etc.. The formulas C of $\mathscr{ALC}$ logic, called *concepts*, are constructed inductively by the following grammar:

```
C ::=   | A          (atomic concept)
        | ⊤          (universal concept  Top)
        | ⊥          (empty concept  Bottom)
        | ¬ C        (negation)
        | C ⊓ C      (conjunction)
        | C ⊔ C      (disjunction)
        | ∀ r. C     (universal quantifier)
        | ∃ r. C     (existential quantifier)
```

Here, $A \in NC$ is an atomic concept name, and $r \in NR$ is a role name. A role is a binary relation between instances of a concept. $\forall$ and $\exists$ are (multi-)modal operators, similar to $\Box$ and $\Diamond$ in traditional modal logics.

3 Syntax of $\mathscr{ALC}$

We now give details of the formal definition of the logic $\mathscr{ALC}$. The type of roles, defined by:

[1] http://www.irit.fr/CMS-DRUPAL7/AcadieProjects/dl_verified

datatype *'nr role = AtomR 'nr*

It only has a single constructor, but is easily expandable to accommodate more complex logic. Type definitions are parameterized by the type of role names *'nr* and atomic concepts *'nc*. Following the grammar of Section 2, here is the type definition of $\mathscr{ALC}$-concepts:

datatype *('nr, 'nc) concept =*
AtomC 'nc
| Top
| Bottom
| NotC ((('nr, 'nc) concept)
| AndC ((('nr, 'nc) concept) (('nr, 'nc) concept)
| OrC ((('nr, 'nc) concept) (('nr, 'nc) concept)
| AllC ('nr role) ((('nr, 'nc) concept)
| SomeC ('nr role) ((('nr, 'nc) concept)

4 Semantic of $\mathscr{ALC}$

Concepts are interpreted as subsets of a domain of interpretation $\Delta_{\mathscr{I}}$ and roles as subsets of the product $\Delta_{\mathscr{I}} \times \Delta_{\mathscr{I}}$. An interpretation $\mathscr{I}$ is essentially a couple $(\Delta_{\mathscr{I}}, .^{\mathscr{I}})$ where $\Delta_{\mathscr{I}}$ is called the domain of interpretation and $.^{\mathscr{I}}$ is an interpretation function that maps an atomic concept A to subset $A^{\mathscr{I}}$ of $\Delta_{\mathscr{I}}$ and a role r to subset $r^{\mathscr{I}}$ of $\Delta_{\mathscr{I}} \times \Delta_{\mathscr{I}}$. Its extension to other concept constructors is defined, in mathematical notation, as follows:

$$\begin{array}{rcl} \top^{\mathscr{I}} & = & \Delta_{\mathscr{I}} \\ \bot^{\mathscr{I}} & = & \emptyset \\ (C \sqcap D)^{\mathscr{I}} & = & C^{\mathscr{I}} \cap D^{\mathscr{I}} \\ (C \sqcup D)^{\mathscr{I}} & = & C^{\mathscr{I}} \cup D^{\mathscr{I}} \\ (\neg C)^{\mathscr{I}} & = & \Delta_{\mathscr{I}} - C^{\mathscr{I}} \\ (\forall r.C)^{\mathscr{I}} & = & \{x \in \Delta_{\mathscr{I}} / \forall y : (x,y) \in r^{\mathscr{I}} \rightarrow y \in C^{\mathscr{I}}\} \\ (\exists r.C)^{\mathscr{I}} & = & \{x \in \Delta_{\mathscr{I}} / \exists y : (x,y) \in r^{\mathscr{I}} \wedge y \in C^{\mathscr{I}}\} \end{array}$$

The type *domtype* is the type of elements of the interpretation domain. Then The interpretation is defined as follows:

record *('ni, 'nr, 'nc) Interp =*
idomain :: domtype set
interp-c :: 'nc ⇒ domtype set
*interp-r :: 'nr ⇒ (domtype * domtype) set*
interp-i :: 'ni ⇒ domtype

The interpretation of roles in Isabelle is given by:

fun *interpR :: ('ni,'nr,'nc) Interp ⇒ 'nr role ⇒ (domtype * domtype) set* **where**
interpR i (AtomR b) = (interp-r i) b

The interpretation of concepts is described by the function:

fun *interpC :: ('ni, 'nr, 'nc) Interp ⇒ ('nr, 'nc) concept ⇒ domtype set* **where**
interpC i Bottom = {}
| interpC i Top = UNIV
| interpC i (AtomC a) = interp-c i a
| interpC i (AndC c1 c2) = (interpC i c1) ∩ (interpC i c2)
| interpC i (OrC c1 c2) = (interpC i c1) ∪ (interpC i c2)

| *interpC i* (*NotC c*) = − (*interpC i c*)
| *interpC i* (*AllC r c*) = {*x* . ∀*y*. ((*x*,*y*) ∈ (*interpR i r*) ⟶ *y* ∈ (*interpC i c*)) }
| *interpC i* (*SomeC r c*) ={ *x*. ∃*y*. ((*x*,*y*) ∈(*interpR i r*) ∧ *y* ∈ (*interpC i c*))}

An interpretation $\mathscr{I}$ is a model of the concept C if $C^{\mathscr{I}} \neq \emptyset$. As given by the following Isabelle definition:

definition *is-model* :: ('*ni*, '*nr*, '*nc*) *Interp* ⇒ ('*nr*, '*nc*) *concept* ⇒ *bool*
where *is-model i c* ≡ (*interpC i c*) ≠ {}

A concept C is satisfiable if there exists an interpretation $\mathscr{I}$ such that $\mathscr{I}$ is a model of C. This definition is written in Isabelle:

definition *satisfiable-def* :: ('*nr*, '*nc*) *concept* ⇒ *bool*
where *satisfiable-def c* ≡ ∃ *i*. *is-model i c*

5 Semantic tableau rules

Our rules handle *Abox* (corresponding to a branch in a tableau), which are sets of facts. A fact may be of the form *x*: *C* for an individual *x* and concept *C*, or *R x y*, for individuals *x*, *y* and role *R*. It can therefore be defined by:

datatype ('*ni*,'*nr*,'*nc*) *fact* =
Inst ('*ni*) (('*nr*,'*nc*) *concept*)
| *Rel* ('*nr role*) ('*ni*) ('*ni*)

type-synonym('*ni*,'*nr*,'*nc*) *abox* = (('*ni*,'*nr*,'*nc*) *fact*) *set*

It is now easy to define an interpretation that satisfies a fact and an *Abox*:

fun *satisfies-fact* :: ('*ni*,'*nr*,'*nc*) *Interp* ⇒ ('*ni*,'*nr*,'*nc*) *fact* ⇒ *bool*
where *satisfies-fact icr* (*Inst x c*) = ((*interp-i icr x*) ∈ (*interpC icr c*))
|*satisfies-fact icr* (*Rel r x y*)= ((*interp-i icr x*, *interp-i icr y*) ∈ (*interpR icr r*))

definition *satisfiable-abox* :: (('*ni*,'*nr*,'*nc*) *abox*) ⇒ *bool*
where *satisfiable-abox Ab* = (∃ *i*. (∀ *f*∈ *Ab*. *satisfies-fact i f*))

We can now describe the rules of the decision procedure. A rule is a relationship between two *Abox*, the *Abox* before and after the application of the rule:

type-synonym ('*ni*,'*nr*,'*nc*)*rule* = (('*ni*,'*nr*,'*nc*)*abox*)⇒ (('*ni*,'*nr*,'*nc*)*abox*)⇒ *bool*

This same format is applicable to simple rules, described later, and the composite rules. This homogeneous format is useful for writing and verifying tactics. For example, we show the rule for the constructor *AndC*:

inductive *Andrule* :: ('*ni*,'*nr*,'*nc*) *rule* **where**
mk-andrule: ⟦*Inst x* (*AndC c1 c2*)∈ *b1*; ¬ ((*Inst x c1*)∈ *b1* ∧ (*Inst x c2*) ∈ *b1*);
b2 = {*Inst x c2*} ∪ {*Inst x c1*} ∪ *b1*⟧ ⟹ *Andrule b1 b2*

It expresses that an instance of the concept (*AndC c1 c2*)] must be located in the *Abox* before applying the rule (this is the condition of applicability), the concept has not been decomposed, and the application of the rule adds the sub-concepts *c1* and *c2*. Of course, this rule is highly non-deterministic, since it does not indicate which instance of a conjunction rule will be applied. Making the calculation more deterministic is one of the goals of the Section 8.

Rule	Condition	Negative Appl Cond	Action
$\rightarrow_{\sqcap}$	$x : C_1 \sqcap C_2 \in \mathscr{A}$	$x : C_1$ and $x : C_2$ are not both in $\mathscr{A}$	$\mathscr{A} := \mathscr{A} \cup \{x : C_1, x : C_2\}$
$\rightarrow_{\sqcup}$	$x : C_1 \sqcup C_2 \in \mathscr{A}$	neither $x : C_1$ nor $x : C_2$ in $\mathscr{A}$	$\mathscr{A} := \mathscr{A} \cup \{x : C_1\}$ or $\mathscr{A} := \mathscr{A} \cup \{x : C_2\}$
$\rightarrow_{\forall}$	$x : \forall rC \in \mathscr{A}$	$r\ x\ y \in \mathscr{A}$ but $y : C \notin \mathscr{A}$	$\mathscr{A} := \mathscr{A} \cup \{y : C\}$
$\rightarrow_{\exists}$	$x : \exists rC \in \mathscr{A}$	$\neg\exists y$ such that $r\ x\ y$ and $y : C$ are both in $\mathscr{A}$	$\mathscr{A} := \mathscr{A} \cup \{z : C, r\ x\ z\}$ Where z is a new variable

Table 1: The decomposition rules for the method of semantic tableaux for $\mathscr{ALC}$

For space reasons, we cannot present all the rules in detail. They are reproduced in Table 1.

The constructor *SomeC* requires special attention. As indicated in Table 1, the application of rule requires the use of a new variable.What first comes to mind is to postulate the existence of this variable with an existential quantifier in the precondition of the rule. However, this non-deterministic existential choice would be impossible to implement by any specific generator function, or lead to a very complex notion of correspondence of abstract and implemented states. We therefore parameterize the rule with the generator function *gen*, which is also used in the implementation (see Section 8).

inductive *Somerule-gen*:: $((\prime ni,\prime nr,\prime nc)abox \Rightarrow \prime ni) \Rightarrow (\prime ni,\prime nr,\prime nc)abox$
$\Rightarrow (\prime ni,\prime nr,\prime nc)abox \Rightarrow bool$ **where**
mk-Somerule-gen: $[\![(Inst\ x\ (SomeC\ r\ c1)) \in b1; \forall\ y.\ \neg((Rel\ r\ x\ y) \in b1 \wedge (Inst\ y\ c1) \in b1); z = gen\ b1; b2 = insert\ (Rel\ r\ x\ z)\ (insert\ (Inst\ z\ c1)\ b1)]\!]$
$\Longrightarrow Somerule\text{-}gen\ gen\ b1\ b2$

In summary, our rules are:

definition *list-alc-rules*::$(((\prime ni,\prime nr,\prime nc)abox) \Rightarrow \prime ni) \Rightarrow ((\prime ni,\prime nr,\prime nc)rule)list$
where *list-alc-rules gen* = [*Andrule*, *Orrule*, *Allrule*, *Somerule-gen gen*]

From these elementary rules, we can construct composite rules by application of rule constructors, such as the following:

fun *disj-rule* :: $(\prime ni,\prime nr,\prime nc)$ *rule* $\Rightarrow$ $(\prime ni,\prime nr,\prime nc)$ *rule* $\Rightarrow$ $(\prime ni,\prime nr,\prime nc)$ *rule*
where *disj-rule r1 r2* = $(\lambda\ a\ b.\ r1\ a\ b\ \vee\ r2\ a\ b)$

It allows to define by recursion the function *disj-rule-list* that converts a list of rules in a rule. Finally, we define the rule

definition *alc-rule* :: $((\prime ni,\prime nr,\prime nc)\ abox) \Rightarrow \prime ni) \Rightarrow (\prime ni,\prime nr,\prime nc)$ *rule*
where *alc-rule gen* = *disj-rule-list* (*list-alc-rules gen*)

6 Soundness

The first central property of a system of rules is soundness. A rule is called sound, if its conclusion is satisfiable then its premise is satisfiable:

definition *sound* :: $(\prime ni,\prime nr,\prime nc)$ *rule* $\Rightarrow$ *bool*
where *sound r*== $\forall$ *A1 A2. r A1 A2* $\longrightarrow$ *satisfiable-abox A2* $\longrightarrow$ *satisfiable-abox A1*

It is easy to show that the elementary rules preserve soundness, and the disjunction of rules:

lemma *disj-rule-sound* [*simp*]: *sound r1* $\Longrightarrow$ *sound r2* $\Longrightarrow$ *sound* (*disj-rule r1 r2*)

or the transitive closure:

lemma *tranclp-rule-sound* [*simp*]: *sound r* $\Longrightarrow$ *sound* (*r*^++)

Also the proof of soundness of the various rules offers no surprises:

lemma *alcrule-sound* [*simp*]: *sound* (*alc-rule gen*)

7 Completeness

To prove completeness, we first define the notion of complete rule. A rule is complete if the satisfiability of *Abox A1* implies that there exists at least one satisfiable *Abox A2* obtained by rule application from *A1*.

definition *complete* ::('*ni*,'*nr*,'*nc*) *rule* $\Rightarrow$ *bool*
  where *complete r* == $\forall$ *A1*.$\exists$ *A2*. *satisfiable-abox A1* $\longrightarrow$ (*r A1 A2*)
    $\longrightarrow$ *satisfiable-abox A2*

We can show this property for each rule. For the rule $\rightarrow_{\sqcap}$, we obtain:

lemma *and-complete* [*simp*]: *complete Andrule*

An *Abox* is *contradictory* if it contains a contradiction (clash), i.e, $x : C$ and $x : \neg C$ or $x : \bot$.

fun *contains-clash* :: ('*ni*,'*nr*,'*nc*) *abox* $\Rightarrow$ *bool*
  where *contains-clash AB* =
  ($\exists$ *x c*. ((*Inst x c*) $\in$ *AB* $\wedge$ (*Inst x* (*NotC c*)) $\in$ *AB*) $\vee$ ((*Inst x Bottom*) $\in$ *AB*))

The fundamental property of the correctness of the tableau algorithm is that if the *Abox* is closed (contains a clash) then it is unsatisfiable:

lemma *content-clash-not-satisfiable*:[[*contains-clash AB*;*satisfiable-abox AB*]] $\Longrightarrow$ *False*

An *Abox* is saturated for a rule if the rule is not applicable to it.

definition *saturated* :: ('*ni*,'*nr*,'*nc*) *abox* $\Rightarrow$ ('*ni*,'*nr*,'*nc*) *rule* $\Rightarrow$ *bool*
  where *saturated AB1 r* $\equiv$ ($\forall$ *AB2*. $\neg$ (*r AB1 AB2*))

Finally, if an *Abox A* is saturated and not contradictory, then it is satisfiable. In this case, there is an interpretation that satisfies *A*, which is called the *canonical interpretation* $\mathscr{I}_A$, whose components are defined as follows:

1. The interpretation domain $\Delta_{\mathscr{I}_A}$ is the set of all individuals included in A
2. For each concept name P we define $P_{\mathscr{I}_A} = \{x | (x : P) \in A\}$
3. For each role name r we define $r_{\mathscr{I}_A} = \{(x,y) | r(x,y) \in A\}$

Now the goal is to prove that if an *Abox* is not contradictory and saturated, then it is satisfiable by the canonical interpretation.

lemma *canon-interp-sat-fact*:[[*inj i*; $\neg$*contains-clash AB*; *saturated AB* (*alc-rule gen*);
    is-Normal-Abox AB; *f* $\in$ *AB*]] $\Longrightarrow$ *satisfies-fact* (*canon-interp i AB*) *f*

8 Implementation

In this section, we propose an implementation, i.e. an executable proof procedure for the description logic $\mathscr{ALC}$. It is based on lists as data structure to implement the *Abox*. The tableau is encoded as a list of *Abox*. The different rules are defined as functions whose argument is list *abox-impl* and return a list of *Abox* (Tableau).

The *Abox* is implemented as a list of facts (*abox_impl*).

type-synonym $('ni,'nr,'nc)$ *abox-impl* $=$ $(('ni,'nr,'nc)$ *fact*$)$ *list*

The implementation of a rule is defined as a function that transforms an *Abox_impl* to a list of *Abox_impl*, just called a tableau.

type-synonym $('ni,'nr,'nc)$*rule-impl*$=$
$('ni,'nr,'nc)$*abox-impl*$\Rightarrow('ni,'nr,'nc)$*abox-impl list*

The type of abstraction of an (*Abox_impl*) to an *Abox* is given by:

type-synonym $('ni,'nr,'nc)$*abstraction*$=('ni,'nr,'nc)$*abox-impl*$\Rightarrow('ni,'nr,'nc)$*abox*

Tableau is simply a list of *Abox_imp*.

type-synonym $('ni,'nr,'nc)$ *tableau* $=('ni,'nr,'nc)$ *abox-impl list*

8.1 Implementing Rules

Each rule is encoded as a pair consisting of the condition of applicability and the given action *(Condition, Action)*. The type of condition is given by:

type-synonym $('ni,'nr,'nc)$*appcond*$=('ni,'nr,'nc)$*abox-impl*$\Rightarrow('ni,'nr,'nc)$*fact*$\Rightarrow$ *bool*

The type of the action of a rule is defined by:

type-synonym $('ni,'nr,'nc)$ *action* $=$
$('ni,'nr,'nc)$*abox-impl* $*$ $('ni,'nr,'nc)$*fact* $*$ $('ni,'nr,'nc)$*abox-impl*
$\Rightarrow('ni,'nr,'nc)$*abox-impl list*

The rule is implemented in Isabelle by the function:

datatype $('ni,'nr,'nc)$ *srule* $=$ *Rule* $('ni,'nr,'nc)$*appcond* $*$ $('ni,'nr,'nc)$*action*

Once the data structures and format rules are defined, we can encode each rule. For this, we determine for each rule the condition of its applicability and the action of this rule. For example the rule $\rightarrow_{\sqcap}$ is coded as follows: The condition of applicability of the rule $\rightarrow_{\sqcap}$ is given by the following function:

fun *appcond-and* :: $('ni, 'nr, 'nc)$ *appcond*
where *appcond-and Ab-i* (*Inst x* (*AndC c1 c2*)) $=$
$(\neg$(*list-ex* (*is-x-c-inst x c1*) *Ab-i*) $\vee$ $\neg$(*list-ex* (*is-x-c-inst x c2*) *Ab-i*))
|*appcond-and Ab-i* - $=$ *False*

The function `is_x_c_inst` means: *is-x-c-inst x c f* $= (f =$ *Inst x c*$)$
The action provided by the application of this rule is:

fun *action-and* :: $('ni,'nr,'nc)$ *action*
where *action-and* (*prefix*, (*Inst x* (*AndC c1 c2*)), *suffix*) $=$
[[*Inst x c1*, *Inst x c2*] @ *prefix* @ [*Inst x* (*AndC c1 c2*)] @ *suffix*]
| *action-and* - $=$ []

prefix denotes the elements of the list before the fact *Inst x* (*AndC c1 c2*) and *suffix* denotes the elements after the fact. The rule $\rightarrow_{\sqcap}$is implemented by:

definition *and-srule* :: $('ni,'nr,'nc)$*srule*
where *and-srule* == *Rule* (*appcond-and*,*action-and*)

Its application is defined in Isabelle by:

definition *and-rule*::$('ni,'nr,'nc)$ *rule-impl* **where** *and-rule* ≡ *apply-srule and-srule*

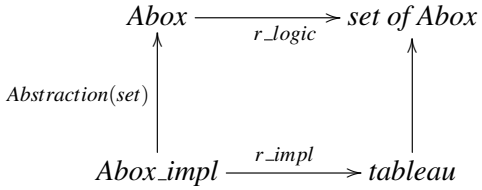
To generalize our implementation for all $\mathscr{ALC}$ rules, we define a list of implemented rules.

definition *list-alc-rules-impl-gen* ::
$((('ni,'nr,'nc)$ *abox-impl*$) \Rightarrow 'ni) \Rightarrow ('ni$::*alloc*, $'nr$, $'nc)$ *rule-impl list* **where**
list-alc-rules-impl-gen gen == [*and-rule*, *or-rule*, *all-rule*, *some-rule-gen gen*]

8.2 Soundness

We have shown in section 6 the proof of soundness property of rules on the abstract level. The following result allows us to demonstrate the soundness of our implementation.

The idea can be illustrated by the following diagram. Above, we see the proof on an abstract level: the application of an abstract rule (*r-logic*), applied to an *Abox*, generates a set of *Abox* successors. The implementation of *Abox* by lists gives a *Abox_impl*, whose abstraction with a *set* gives a set of facts. The application of a rule for this implementation (*r-impl*) must provide a list of *Abox* implementations that have the same abstraction.



More formally, the definition in Isabelle:

definition *sound-rule-impl*::
$('ni,'nr,'nc)$*abstraction*⇒$('ni,'nr,'nc)$*rule*⇒$('ni,'nr,'nc)$*rule-impl*⇒*bool* **where**
sound-rule-impl abstr r r-impl≡ ∀ *ai ai'*.$(ai' \in$*set*$($*r-impl ai*$)) \longrightarrow r($*abstr ai*$)($*abstr ai'*$)$

8.3 Termination

Termination is an important property of rewriting systems. A standard method for proving termination of a rewrite system is to exhibit a well-founded ordering on terms, such that if A_1 is rewritten to A_2 then $A_1 \gg A_2$.

Formally, for proving termination, we associate to each implementation of a *Abox* a measure. If this measure decreases with every rule in a well-founded ordering, termination is assured. In our case, a measure is a function of type *Abox_imp* $\rightarrow T$ for T a domain equipped with a well founded relation $\gg$ in T which we define in the sequel.

We introduce the constructors necessary to define the measure:

The function *sizeC* calculates the size of a concept, it is defined as the number of constructors of this concept.

- The measure in our case is a multiset of pairs of natural numbers.

- For each axiom (`fact`) of *Abox_imp*, we associate a pair, depending on the structure of the axiom and the *Abox _imp*.
- The measure of the element that is applicable must decrease, without affecting the measure of other elements.

We now define the measure of the axioms (this is the function *meas_comp* defined in the following). If the axiom is:

- A relation $x\ r\ y$, we associate the pair $(0,0)$
- An instance $x : D$. Depending on the structure of D, there are three cases:

1. If D is an *Atom* $(x : A)$ or *Negation* $(x : \neg A)$, we associate the value $(0,0)$;
2. If D is a conjunction, disjunction or existential quantifier:
   - If the corresponding rule is applicable on the axiom, we associate the pair $(sizeC(D),0)$,
   - else $(0,0)$;
3. If D is a universal quantifier $(D = \forall r.C)$ we associate the pair $(Comp_1, Comp_2)$ such that:
   (a) $Comp_1 = sizeC(D)$;
   (b) $Comp_2 = Comp_{21} + Comp_{22}$ such that:

- $Comp_{21}$ is the number of applicability of the rule $\rightarrow_\forall$, ie the number of $x\ r\ y$ in *Abox_imp* such that $y : C$ is not in the *Abox_imp*. It is noted here that $Comp_{21}$ decreases if the rule $\rightarrow_\forall$ is applicable, but if we apply the rule $\rightarrow_\exists$ on another fact, this measure may increase. For this, we add the component $Comp_{22}$ which ensures that this remains constant by the application of another rule;
- $Comp_{22}$ is the number of $\exists$-*terms* reducible and hidden (in the sense that they appear in the structure of the concept) in the *Abox*. This value decreases if the rule $\rightarrow_\exists$ is applied and remains constant or decreases if another rule is applied.

In the end we can prove that measure is well founded.

lemma *wf-measure-abox-impl-order*: *wf measure-abox-impl-order*

9 Conclusion

In this paper we have presented a definition of a reasoner validated for the description logic $\mathscr{ALC}$ based on the method of semantic tableaux. This formalization in Isabelle and development is based on several modules:

- Specifying the syntax and semantics of $\mathscr{ALC}$,
- Coding of *Abox* and the formalization of the transformation rules of semantic tableaux,
- The proof of the soundness and completeness of semantic tableaux,
- The proof of termination requires the definition of a measure for each *Abox*. We have shown that this measure decreases for each application of a rule.
- The implementation of *Abox*, tableaux and rules transformation
- Defining a strategy of proof for $\mathscr{ALC}$

- Finally, the extraction of an executable, certified reasoner in the Caml language.

We will consider several extensions of this work, among which are:

- Extensions of this work to more expressive logics used in the semantic Web, such as $\mathcal{SHOIQ}$ and $\mathcal{SHOIN}$.
- Sets of refinements by other more efficient data structures as lists, and in particular the use of indexing techniques to speed up testing unsatisfiability of a table (" clash") or to identify the applicable rules.

References

[1] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi & Peter F. Patel-Schneider (2007): *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, doi:10.1017/CBO9780511711787.

[2] Franz Baader & Philipp Hanschke (1991): *A schema for integrating concrete domains into concept languages*. In: *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI'91)*, pp. 452–457.

[3] Franz Baader, Ian Horrocks & Ulrike Sattler (2005): *Description Logics as Ontology Languages for the Semantic Web*. In: *Mechanizing Mathematical Reasoning*, pp. 228–248, doi:10.1007/978-3-540-32254-2_14. Available at `http://www.informatik.uni-trier.de/~ley/db/conf/birthday/siekmann2005.html#BaaderHS05`.

[4] Franz Baader & Ulrike Sattler (1999): *Expressive Number Restrictions in Description Logics*. *Journal of Logic and Computation* 9(3), pp. 319–350, doi:10.1093/logcom/9.3.319. Available at `http://lat.inf.tu-dresden.de/research/papers/1999/BaaderSattler-JLC-99.ps.gz`.

[5] Franz Baader & Ulrike Sattler (2001): *An Overview of Tableau Algorithms for Description Logics*. *Studia Logica* 69(1), pp. 5–40, doi:10.1023/A:1013882326814. Available at `http://lat.inf.tu-dresden.de/research/papers/2001/BaaderSattler-StudiaLogica.ps.gz`.

[6] Daniela Berardi, Diego Calvanese & Guiseppe De Giacomo (2001): *Reasoning on UML Class Diagrams using Description Logic Based Systems*. In: *Proc of the KI 2001 Workshop on Applications of Description Logics, CEUR Electronic Workshop Proceedings, http://ceur-ws.org/Vol-44*.

[7] Mohamed Chaabani, Mohamed Mezghiche & Martin Strecker (2009): *Formalisation de la logique de description $\mathcal{ALC}$ dans l'assistant de preuve Coq*. In L. Bellatrache, G. Kassel & P. Thiran, editors: *Proc. 3es Journées francophones sur les ontologies*, pp. 139–147.

[8] Mohamed Chaabani, Mohamed Mezghiche & Martin Strecker (2010): *Vérification d'une méthode de preuve pour la logique de description $\mathcal{ALC}$*. In: *Proc. 10ème Journées Approches Formelles dans l'Assistance au Développement de Logiciels*, pp. 149–163.

[9] Detlef Fehrer, Ullrich Hustadt, Manfred Jaeger, Andreas Nonnengart, Hans Jürgen Ohlbach, Renate A. Schmidt, Christoph Weidenbach & Emil Weydert (1994): *Description Logics for Natural Language Processing*. In Franz Baader, Maurizio Lenzerini, Werner Nutt & Peter F. Patel-Schneider, editors: *International Workshop on Description Logics '94, Document* D-94-10, DFKI, Bonn, Germany, pp. 80–84.

[10] Volker Haarslev & Ralf Möller (2001): *RACER System Description*. In: *IJCAR '01: Proc. First International Joint Conference on Automated Reasoning*, Springer-Verlag, pp. 701–706, doi:10.1007/3-540-45744-5_59.

[11] María-José Hidalgo, José-Antonio Alonso, Joaquín Borrego-Díaz, Francisco-Jesus Martin-Mateos & José-Luis Ruiz-Reina (2007): *A formally verified prover for the ALC description logic*. In: *20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007, Lecture Notes in Computer Science* 4732, pp. 135–150, doi:10.1007/978-3-540-74591-4_11. Available at `http://www.cs.us.es/~mjoseh/pub/2007-TPHOLs.pdf`.

[12] Ian Horrocks & Ulrike Sattler (2007): *A Tableau Decision Procedure for $\mathcal{SHOIQ}$*. J. of Automated Reasoning 39(3), pp. 249–276, doi:10.1007/s10817-007-9079-9. Available at `download/2007/HoSa07a.pdf`.

[13] Marko Luther, Thorsten Liebig, Sebastian Böhm & Olaf Noppens (2009): *Who the Heck is the Father of Bob?* In: *6th Annual European Semantic Web Conference (ESWC2009)*, pp. 66–80. Available at `http://data.semanticweb.org/conference/eswc/2009/paper/222`.

[14] Tobias Nipkow, Lawrence Paulson & Markus Wenzel (2002): *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283, Springer, doi:10.1007/3-540-45949-9.

[15] Werner Nutt, Francesco M. Donini, Maurizio Lenzerini & Daniele Nardi (1997): *The complexity of concept languages*. Inf. Comput. 134(1), pp. 1–58, doi:10.1006/inco.1997.2625.

[16] Tom Ridge & James Margetson (2005): *A mechanically verified, sound and complete theorem prover for FOL*. In Joe Hurd & Tom Melham, editors: *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, Lecture Notes in Computer Science 3603, doi:10.1007/11541868_19.

[17] Alexander Schimpf, Stephan Merz & Jan-Georg Smaus (2009): *Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL*. In Tobias Nipkow & Christian Urban, editors: *22nd Intl. Conf. Theorem Proving in Higher-Order Logics (TPHOLs 2009)*, Lecture Notes in Computer Science 5674, Springer, Munich, Germany, doi:10.1007/978-3-642-03359-9_29. Available at `http://www.loria.fr/~merz/papers/tphols2009.html`.

[18] Manfred Schmidt-Schaubß & Gert Smolka (1991): *Attributive concept descriptions with complements*. Artif. Intell. 48(1), pp. 1–26, doi:10.1016/0004-3702(91)90078-X.

[19] Dmitry Tsarkov & Ian Horrocks (2006): *FaCT++ Description Logic Reasoner: System Description*. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006), Lecture Notes in Artificial Intelligence 4130, Springer, pp. 292–297, doi:10.1007/11814771_26.

[20] Paulien de Wind (2001): *Modal Logic*. Master's thesis, Vrije Universiteit Amsterdam.