Formalisation de la logique de description $\mathcal{ALC}$ dans l'assistant de preuve Coq

Mohamed Chaabani
IRIT, Université Paul Sabatier
118 route de Narbonne
F-31062 Toulouse
France
chaabani@irit.fr

Mohamed Mezghiche
LIFAB,Université de Boumerdès
Faculté des sciences
Boumerdès, Algérie
mohamed-mezghiche@umbb.dz

Martin Strecker
IRIT, Université Paul Sabatier
118 route de Narbonne
F-31062 Toulouse
France
strecker@irit.fr

ABSTRACT

Le langage d'ontologie Web (Web Ontology Language OWL) est un langage utilisé pour le web sémantique. OWL est basé sur les logiques de description (LD), une famille de langages adaptés pour la représentation et le raisonnement sur des connaissances d'un domaine d'application d'une façon structurée et formelle. Le web sémantique est actuellement l'un des champs d'application des méthodes formelles, dont l'objectif est d'assurer leur fiabilité. Un point essentiel de l'application de ces méthodes formelles est la preuve de validité des raisonnements dans des LDs, comme celle de la terminaison, l'adéquation (soundness) et la complétude d'un raisonneur. Dans ce papier, on présente une spécification formelle de la syntaxe et de la sémantique de $\mathcal{ALC}$, qui est considérée comme un représentant typique d'une large gamme de LDs. On prouve pour cette logique les propriétés d'adéquation, de complétude et de terminaison dans l'assistant de preuve Coq.

Keywords

Logiques de Description, méthodes formelles, OWL-DL, Tableau sémantique, Terminaison, Complétude, Adéquation

1. INTRODUCTION

L'objectif de ce travail est de fournir une formalisation de la logique de description $\mathcal{ALC}$ et une verification formelle de la correction de l'algorithme du tableau sémantique pour le test de la satisfiabilité dans l'assistant de preuve Coq.

Les logiques de description sont des formalismes largement utilisés dans plusieurs domaines tels que le web sémantique et la construction d'ontologies. $\mathcal{ALC}$ est considérée comme une base pour plusieurs logiques de description plus expressives comme par exemple $\mathcal{SHOIQ}$[17] et $\mathcal{SHOIN}$[3]. Cette dernière est la base formelle du langage d'ontologie Web OWL-DL et supportée par plusieurs raisonneurs tels que FaCT++ [24] et RACER [14].

JFO 2009 December 3-4, 2009, Poitiers, France

Parmi plusieurs méthodes utilisées comme des procédures de décision pour les LDs, la méthode des tableaux[5] est la plus répandue. En effet, Fact++ et RACER utilisent cette méthode pour faire des raisonnements.

Les systèmes de preuves formelles ont une grande utilité dans la vérification et la validation des applications, surtout dans les domaines critiques tel que le Web sémantique. La certification de ces applications nécessite la manipulation d'une large variété d'objets mathématiques qui sont implantés dans la bibliothèque du système de preuves.

Il y a des travaux connexes à notre formalisation: Ridge et Margetson [22] décrivent la formalisation d'un prouver pour la logique du premier ordre dans l'assistant de preuve Isabelle/HOL. Plus proche de nos travaux est une formalisation de certaines logiques modales [9] en Coq – les LD peuvent être perçues comme des logiques modales spécifiques. Un groupe de chercheurs a, indépendamment de notre formalisation, proposé un codage [15, 16] de la logique de description $\mathcal{ALC}$ dans l'assistant de preuve PVS. Notre formalisation en diffère surtout en offrant une simplification de l'argument de terminaison de la procédure de tableau (voir § 9). En plus, nous espérons pouvoir utiliser les facilités d'extraction de code exécutable fournies par Coq pour dériver un raisonneur certifié de notre formalisation.

Cet article est organisé comme suit: Dans la section 2, nous présentons brièvement le système de preuve Coq. Les logiques de description sont détaillées dans les sections 3, 4 et 5. Dans la section 6, nous présentons le tableau sémantique pour $\mathcal{ALC}$ et dans le reste de l'article, nous décrivons la formalisation et la preuve des propriétés du tableau, à savoir, l'adéquation, la complétude et la terminaison dans Coq.

2. L'ASSISTANT DE PREUVE COQ

Coq [23, 7] est un assistant de preuve interactif, similaire en esprit avec d'autres systèmes de preuve comme PVS [21], HOL [19] et Isabelle [20]. Coq est basé sur le Calcul des Constructions Inductif (CCI), une logique d'ordre supérieur intuitionniste, mettant ainsi en pratique l'isomorphisme de Curry-Howard dont les types sont vus comme des propositions et les termes comme des preuves. Coq fournit des mécanismes pour écrire des définitions et pour faire des preuves formelles d'une manière interactive. Il permet ainsi d'extraire des programmes fonctionels à partir de la preuve.

Le système de types de Coq est assez complexe. On n'aura pas besoin de cette complexité pour la formalisation qui sera décrite dans la suite, et effectivement, on pourrait faire le

même développement aussi dans l'un des autres assistants de preuve cités plus haut. On introduira les éléments de Coq nécessaires pour la compréhension de cet article au fur et à mesure.

3. LES LOGIQUES DE DESCRIPTION

Les logiques de description [1, 2, 4, 12] forment une famille de langages de représentation de connaissances qui peuvent être utilisés pour représenter les connaissances d'un domaine d'application d'une façon structurée et formelle. Une caractéristique fondamentale de ces langages est qu'ils ont une sémantique formelle. Les logiques de description sont utilisées pour de nombreuses applications. Parmi elles on cite les domaines suivants:

- Le web sémantique: représentation d'ontologies [3] et recherche d'information...
- Traitement automatique des langues [13]
- L'ingénierie logicielle: représentation de la sémantique des diagrammes de classe UML [6].

Les logiques de description utilisent les notions de concept, de rôle et d'individu. Les concepts correspondent à des classes d'individus, les rôles sont des relations entre ces individus.

EXEMPLE 1. *Cet exemple décrit les relations entre membres d'une famille:*

- *Concepts : Femme , Homme , Personne ...*
- *Rôles : enfant-de , père-de ...*
- *Instances : Pierre , Marie ....*

Un concept et un rôle possèdent une description structurée définie à partir d'un certain ensemble de constructeurs.

Dans les logiques de description on distingue deux niveaux de traitement:

- Niveau terminologique Tbox : le niveau générique (globale) vrai dans tous les modèles et pour tous individus;

EXEMPLE 2. *Cet exemple décrit une Tbox:*

$$\begin{array}{rcl} Femelle & \sqsubseteq & \top \sqcap \neg Male \\ Male & \sqsubseteq & \top \sqcap \neg Femelle \\ Animal & \equiv & Male \sqcup Femelle \\ Humain & \sqsubseteq & Animal \\ Femme & \equiv & Humain \sqcap Femelle \\ Homme & \equiv & Humain \sqcap \neg Femelle \end{array}$$

- Niveau assertionnel *Abox* : fournit des instances des concepts et des rôles.

EXEMPLE 3. *Cet exemple représente une* Abox.

$$\begin{array}{c} pierre : Homme \\ marie : Femme \\ epouse\_de(pierre, marie) \end{array}$$

Dans la section § 5.3, on introduit une définition formelle de *Abox*.

3.1 Les familles de DL

Les logiques de description ont une base commune enrichie de différentes extensions:

- $\mathcal{AL}$ *(Attributive Language)*: c'est le language de base défini à partir des éléments syntaxiques suivants:
  - A: Concept atomique
  - $\top$: Le concept universel Top
  - $\bot$: Le concept vide Bottom
  - $\neg A$: Négation d'un concept atomique
  - $C \sqcap D$: Conjonction de concepts
  - $\forall r.C$: Quantificateur universel
  - $\exists r$: Quantificateur existentiel non typé

  D'autres langages, plus expressifs, peuvent être définis en rajoutant d'autres constructeurs au langage $\mathcal{AL}$, à savoir :
- $\mathcal{ALU} = \mathcal{AL} \cup \{C \sqcup D\}$: Disjonction de concepts;
- $\mathcal{ALE} = \mathcal{AL} \cup \{\exists r.C\}$: Quantificateur existentiel typé
- $\mathcal{ALC}$ *(Attributive Language with Complement)*: c'est la logique la plus importante, elle constitue la base de toute les logiques de description "expressifs".
  $\mathcal{ALC} = \mathcal{AL} \cup \{\neg C\}$: Ici, C est un concept primitif ou défini. On note qu'on peut coder la disjonction resp. la quantification existentielle à l'aide de la négation et la conjonction resp. la quantification universelle, pour obtenir $\mathcal{ALU}$ resp. $\mathcal{ALE}$ comme sous-logiques de $\mathcal{ALC}$.
- $\mathcal{ALN} = \mathcal{AL} \cup \{\leq nr, \geq nr\}$: Les restrictions de nombres, désignées par la lettre $\mathcal{N}$ et notées par $\leq nr$ (restriction à moins de n) et $\geq nr$ (restriction à plus de n) où n représente un entier positif.

4. SYNTAXE

Nous allons maintenant entrer plus dans les détails de la définition formelle de la logique $\mathcal{ALC}$: sa syntaxe dans cette section et sa sémantique dans § 5. Pour faciliter la lecture, nous présentons quelques notions tout d'abord dans une notation mathématique traditionnelle, pour ensuite présenter la notation utilisée dans Coq.

Soit NC un ensemble de noms de concept et NR un ensemble de noms de rôle. L'ensemble de $\mathcal{ALC}$-concept est construit par induction selon la grammaire suivante :

C, D `::=`
| A `(concept atomique)`
| $\top$ `(concept universel Top)`
| $\bot$ `(concept vide Bottom)`
| $\neg\ C$ `(négation)`
| $C\ \sqcap\ D$ `(conjonction)`
| $C\ \sqcup\ D$ `(disjonction)`
| $\forall\ R.C$ `(quantificateur universel)`
| $\exists\ R.C$ `(quantificateur existentiel)`

où $A \in NC$ et $R \in NR$.

Dans l'objectif de rendre générique notre formalisation, on propose de paramétrer cette dernière par des types génériques. Ici, NC et NR sont des types quelconques où l'égalité est décidable.

La spécification corespondante en Coq est:

```
Parameter NC NR : Set.
```

Dans cette formalisation on considère que les rôles sont atomiques. On définit donc les rôles comme un type inductif avec un seul constructeur.

```
Inductive role : Set :=  AtomR : NR -> role.
```

La définition des concepts en Coq est donnée comme suit:

```
Inductive concept : Set :=
   AtomC : NC -> concept
 | Top : concept
 | Bottom : concept
 | NotC : concept -> concept
 | AndC : concept ->concept -> concept
 | OrC : concept ->concept ->concept
 | AllC : role -> concept -> concept
 | SomeC :role -> concept->concept
```

Pour réduire la complexité des preuves (surtout les preuves inductives), on s'intéresse à limiter au minimum le nombre des constructeurs du langage. Par exemple, les constructeurs de l'implication et de l'équivalence peuvent être redéfinis comme suit:

```
Definition ImplyC c1 c2 :=
   OrC (NotC c1) c2.
Definition EquivC c1 c2 :=
   AndC (ImplyC c1 c2) (ImplyC c2 c1).
```

5. SÉMANTIQUE

A l'instar de la logique classique, une sémantique est associée aux descriptions des concepts et des rôles: les concepts sont interprétés comme des sous-ensembles d'un domaine d'interprétation $\Delta_\mathcal{I}$ et les rôles comme des sous-ensembles du produit $\Delta_\mathcal{I} \times \Delta_\mathcal{I}$.

5.1 Interprétation

Une interprétation $\mathcal{I}$ est essentiellement un couple $(\Delta_\mathcal{I}, .^\mathcal{I})$ où $\Delta_\mathcal{I}$ est appelé domaine d'interprétation et $.^\mathcal{I}$ est une fonction d'interprétation qui associe à un concept C un sous-ensemble $C^\mathcal{I}$ de $\Delta_\mathcal{I}$ et à un rôle r un sous-ensemble $r^\mathcal{I}$ de $\Delta_\mathcal{I} \times \Delta_\mathcal{I}$. En notation mathématique, elle est définie comme suit:

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

Pour des raisons de typage, dans Coq, l'interprétation est définie comme un record dépendant, contenant le domaine d'interprétation et les fonctions de interprétation de concepts, de rôles et d'instances et dont on présentera les éléments dans la suite:

```
Record Interp : Type := {
       D:Set;
       interp_c : NC -> Ensemble D;
       interp_r : NR ->  Relation D;
       interp_inst : NI-> D}.
```

Étant donnée une interprétation `i`, l'interprétation des concepts est définie par récursion sur la structure des concepts[1]:

```
Fixpoint interpC (i:Interp)(c:concept){struct c}:
     Ensemble (D i) :=
  match c with
  | Bottom     => Empty_set
  | Top        => Full_set
  | AtomC a    => interp_c i a
  | AndC c1 c2 =>
        Intersection (interpC i c1) (interpC  i c2)
  | OrC c1 c2  => Union (interpC i c1)(interpC  i c2)
  | NotC c     => Complement (interpC i c)
  | AllC r c   => fun x =>forall y,
    (interpR _ (interp_r i) r x y) -> (interpC i c y)
  | SomeC r c  => fun x =>exists y,
    (interpR _ (interp_r i) r x y) ∧ (interpC  i c y)
end.
```

L'interprétation des rôles est définie par la fonction suivante:

```
Fixpoint interpR (D: Set)(interp_r:NR -> Relation D)
                 (r:role): Relation D :=
    match r with (AtomR b) => interp_r b
end.
```

Pour la logique $\mathcal{ALC}$, il n'est pas nécessaire de représenter l'interprétation des rôles par une fonction récursive, mais si on veut faire une extension pour autre logique où il y a la conjonction des rôles comme $\mathcal{ALCR}$ ou les rôles inverses, le codage par une fonction récursive est exigé.

Le type NI représente les noms d'instance:

```
Parameter NI : Set.
```

5.2 Satisfiabilité et subsomption

Un concept C est satisfiable s'il existe une interprétation $\mathcal{I}$ telle que $C^\mathcal{I} \neq \emptyset$. Cette définition est traduite en Coq comme suit:

```
Definition concept_satisfiable c :=
  exists i,~ ((interpC i c)= Empty_set(D i)).
```

Un concept C est insatisfiable ssi pour toute interprétation $\mathcal{I}, C^\mathcal{I} = \emptyset$.

```
Definition concept_unsatisfiable c :=
  forall i, interpC i c = Empty_set (D i).
```

Un concept D est subsumé par un concept C, noté $D \sqsubseteq C$, ssi pour toute interpretation $\mathcal{I}$, $D^\mathcal{I} \subseteq C^\mathcal{I}$. Cette définition est codée en Coq comme suit:

```
Definition subsumed c1 c2 := forall i,
  Included (D i) (interpC i c1) (interpC i c2).
```

Parmi les lemmes qu'on peut démontrer, on cite:

- Le concept vide (bottom) est un concept insatisfiable
- Le concept Top subsume tout autre concept.

[1]Remarque notationnelle: on accède à un composant d'un record par application fonctionnelle du sélecteur. Ainsi, le composant `D` de `i` s'écrit `D i`

- Un concept c_1 est subsumé par c_2, ssi $c_1 \sqcap \neg c_2$ n'est pas satisfiable.

```
Lemma sub_to_unsatis: forall c1 c2,
  subsumed c1 c2 <->
      concept_unsatisfiable (AndC c1 (NotC c2)).
```

5.3 Base de connaissances

Dans les logiques de description, la représentation de connaissances s'articule autour de deux niveaux: Les *Tbox* (*terminological box*), qui permettent de raisonner uniquement sur des concepts, et les *Abox* (*assertional box*), qui introduisent un raisonnement sur des individus. Ces derniers forment la notion la plus fondamentale car on peut ramener les *Tbox* aux *Abox*. C'est pour cela que nous nous concentrons uniquement sur ces derniers.

Les *Abox* contiennent un ensemble d'assertions sur les individus, comme les assertions d'appartenance et des assertions de rôle. Un *axiome assertionnel* (aussi appelé *fait*) est soit de la forme $x : C$ ("x appartient au concept C") ou $x \neq y$ ("x et y sont des individus différents") ou $x\ r\ y$ ("les individus x et y sont reliés par le rôle r"). Ces notions sont traduites en Coq comme suit:

```
Inductive fact: Set :=
  inst : NI -> concept -> fact
| rel  : NI-> NI -> role -> fact
| dif  : NI-> NI -> fact.
```

On étend maintenant la notion d'interprétation $\mathcal{I}$ aux faits d'une *Abox*:

- $\mathcal{I}$ satisfait l'axiome assertionnel $x : C$ ssi $x^{\mathcal{I}} \in C^{\mathcal{I}}$
- $\mathcal{I}$ satisfait l'axiome assertionnel xry ssi $(x^{\mathcal{I}}, y^{\mathcal{I}}) \in r^{\mathcal{I}}$
- $\mathcal{I}$ satisfait l'axiome assertionnel $x \neq y$ ssi $x^{\mathcal{I}} \neq y^{\mathcal{I}}$

Dans Coq, la notion de satisfaibilité est donnée par:

```
Definition inter_satisfies_fact i (Aa:fact) :=
  match Aa with
| inst ni c =>
   In (D i) (interpC i c) (interp_inst i ni)
| rel ni1 ni2 r =>
   interpR (D i) (interp_r i) r
      (interp_inst i ni1) (interp_inst i ni2)
| dif ni1 ni2 =>
   interp_inst i ni1<>interp_inst i ni2
end.
```

Une *Abox* est définie comme un ensemble fini d'axiomes assertionnels (fact). Pour la représentation des ensembles finis, on utilise la bibliothèque `Fset` de l'assistant de preuve Coq: donc les éléments de cet ensemble sont des faits dont l'égalité est décidable.

Le système des modules de Coq est similaire à celui de Caml. On distingue trois catégories :

- un *module* est une collection d'objets Coq.
- une *signature* (interface) correspond au type d'un module.
- un *foncteur* (module paramétré) est une fonction qui prend des modules en arguments et produit un module en résultat.

On définit un foncteur `Fact_as_DT` (fait comme un type décidable) qui respecte la signature `DecidableType` de la bibliothèque Coq.

```
Module Fact_as_DT <: DecidableType.
  Definition t := fact.
  Definition eq := @eq t.
  Definition eq_refl := @refl_equal t.
  Definition eq_sym := @sym_eq t.
  Definition eq_trans := @trans_eq t.
  Lemma  eq_dec :forall x y:t, {eq x y}+{~eq x y}.
  Proof.
  unfold eq.
  repeat decide equality .
  Qed.
End Fact_as_DT.
```

Ensuite on définit le type *Abox* qui est un ensemble fini (`Fset`) de faits par l'instanciation du foncteur `WSfun` par le module `Fact_as_DT`:

```
Declare Module Import  fsetsfact : WSfun Fact_as_DT.
Notation Abox := fsetsfact.t.
```

Une interprétation $\mathcal{I}$ est un modèle pour une *Abox* A si tous ses axiomes sont satisfaits par $\mathcal{I}$.

```
Definition is_model_Abox i (A:Abox ): Prop :=
forall Aa, In Aa A -> inter_satisfies_fact i Aa.
```

Une *Abox* A est satisfiable si elle possède un modèle.

```
Definition abox_satisfiable (A :Abox) : Prop :=
                     exists i, is_model_Abox i A.
```

De la même manière que pour les faits, on définit le foncteur `NI_as_DT`.

```
Module NI_as_DT <: DecidableType.
  Definition t := NI.
  Definition eq := @eq t.
  Definition eq_refl := @refl_equal t.
  Definition eq_sym := @sym_eq t.
  Definition eq_trans := @trans_eq t.
  Lemma  eq_dec :forall x y :t,{eq x y} + {~eq x y}.
  Proof.
  unfold eq.
  trivial.
 Qed.
End NI_as_DT.
```

`Setni` représente le type des ensembles finis des instances:

```
Declare Module fsetsni: WSfun NI_as_DT .
Notation Setni := fsetsni.t.
```

6. TABLEAUX SÉMANTIQUES POUR $\mathcal{ALC}$

Un tableau sémantique est une procédure qui permet de construire une interprétation qui satisfait l'assertion d'un concept donné. Les dérivations peuvent être établies par l'application d'un ensemble de règles de décompositions. Initialement, on commence par une *Abox* avec une assertion de la forme $a : C$, où le concept C doit être en forme normale négative (*negation normal form*). On applique des règles de décomposition jusqu'à obtenir un tableau où aucune règle n'est applicable. Notre approche est donc applicable à n'importe quel tableau initial qui peut contenir un nombre quelconque (mais fini) de faits.

6.1 Forme normale négative (NNF)

Soit C un concept arbitraire dans $\mathcal{ALC}$. On dit que C est en forme normale négative ssi $\neg$ apparaît seulement devant les noms de concept. Dans la suite, on construit une fonction qui calcule la forme normale négative d'un concept quelconque, et un prédicat qui teste qu'un concept est en forme normale.

```
Fixpoint trans_neg (a:concept) : concept :=
  match a with
| AtomC x   => NotC (AtomC x)
| Top       => Bottom
| Bottom    => Top
| NotC c    => c
| AndC c d  => OrC (trans_neg c) (trans_neg d)
| OrC c d   => AndC (trans_neg c) (trans_neg d)
| SomeC r c => AllC r (trans_neg c)
end.
```

```
Fixpoint NNF (a :concept) : concept :=
  match a with
| NotC c1    => trans_neg (NNF c1)
| AtomC x    => AtomC x
| Top        => Top
| Bottom     => Bottom
| AndC c1 c2 => AndC (NNF c1) (NNF c2)
| OrC c1 c2  => OrC (NNF c1) (NNF c2)
| AllC  r c  => AllC r (NNF c)
| SomeC r c  => SomeC r (NNF c)
end.
```

La fonction `NNF` calcule la forme normale négative d'un concept. Cette fonction préserve la sémantique, i.e, la sémantique d'un concept C est égale à la sémantique de de sa forme normale:

```
Lemma semNFCC : forall i c ,
     interpC i c  =  interpC i (NNF c).
```

Un prédicat qui vérifie qu'un concept est en forme normale est défini comme suit:

```
Fixpoint isNF (a:concept): Prop :=
  match a with
| AtomC x    => True
| Top        => True
| Bottom     => True
| NotC c1    => exists x , c1 = (AtomC x)
| AndC c1 c2 => isNF c1 ∧ isNF c2
| OrC c1 c2  =>   isNF c1 ∧  isNF c2
| AllC  r c  => isNF c
| SomeC r c  => isNF c
end .
```

Les deux fonctions présentées ci-dessus sont équivalentes, la première est calculatoire, tandis que la deuxième est déclarative. Le lemme suivant démontre l'équivalence entre ces deux fonctions.

```
Lemma isnf_eq_nnf: forall c, isNF c <-> (NNF c) = c.
```

Un axiome assertionnel (`fact`) est en forme normale négative ssi ses concepts sont en forme normale négative.

```
Definition is_fact_normal (f:fact) :=
      match f with
      |  inst x c  => isNF c
      |  _ => True
end.
```

Une *Abox* est en forme normale négative ssi tous ses axiomes assertionnels sont en forme normal négatif.

```
Definition  is_Normal_Abox (ab:Abox):Prop :=
   forall f:fact, In f ab -> is_fact_normal f .
```

6.2 Les règles de décomposition (transformation)

Soit C un concept sous forme `NNF` . Afin de tester la satisfiablité de ce dernier, l'algorithme de tableau démarre avec une *Abox* $A = \{x : C\}$ et applique les règles de transformation (décomposition) qui préservent la consistance jusqu'à ce qu'aucune règle ne puisse être appliquée. Les règles de transformation sont résumées dans la Table 1.

Les règles de décomposition sont considérées comme des règles de réécritures, dont chaque règle est codée indépendamment des autres règles, et on verra dans § 7 que ceci mène à une preuve modulaire de la correction des règles.

Il faut interpréter les règles de la Table 1 comme suit: Chaque règle a une condition d'applicabilité. Celle-ci coïncide, dans notre cas, avec le constructeur de la racine du concept, et c'est de celui-ci que la règle dérive son nom. L'application d'une règle est sujette à des conditions d'applicabilité négatives: Dans le cas de $\rightarrow_{\sqcap}$, par exemple, il faut s'assurer que les deux "sous-concepts" $x : C_1$ et $x : C_2$ ne sont pas encore tous les deux présents dans le tableau actuel $\mathcal{A}$. Si les conditions sont satisfaites, on exécute une action, qui consiste généralement à rajouter des éléments au tableau actuel.

Dans le système Coq, chaque règle est définie comme un prédicat binaire, codé comme prédicat inductif avec un seul constructeur. Regardons maintenant ces définitions de plus près.

La règle $\rightarrow_{\sqcap}$ est déterministe et ne s'applique qu'une seule fois sur un fait:

```
Inductive Andrule (b1:Abox) (b2:Abox): Prop :=
  mk_andrule: forall x c1 c2,
     In (inst x (AndC c1 c2)) b1 ∧
      ~ (In (inst x c1)b1 ∧ In (inst x c2)b1)
        -> b2 = add(inst x c2) (add(inst x c1) b1)
          -> Andrule b1 b2.
```

La règle $\rightarrow_{\sqcup}$ est une règle indéterministe. Pour que l'on puisse la coder, on la rend déterministe par l'introduction de deux règles: La règle $\rightarrow_{\sqcup L}$ (*Left*) introduit l'élément gauche de la disjonction tandis que la règle $\rightarrow_{\sqcup R}$ (*Right*) introduit l'élément droit de la disjonction.

```
Inductive Orruleleft (b1:Abox) (b2:Abox): Prop :=
  mk_Orruleleft: forall x c1 c2,
       In (inst x (OrC c1 c2)) b1 ∧
        ~ In (inst x c1) b1 ∧ ~ In (inst x c2) b1
              -> b2 = add (inst x c1) b1
              -> Orruleleft b1 b2.
```

Règle	Condition	Condition d'applic. negative	Action
$\rightarrow_\sqcap$	$x : C_1 \sqcap C_2 \in \mathcal{A}$	$x : C_1$ et $x : C_2$ ne sont pas tous deux dans $\mathcal{A}$	$\mathcal{A} := \mathcal{A} \cup \{x : C_1, x : C_2\}$
$\rightarrow_\sqcup$	$x : C_1 \sqcup C_2 \in \mathcal{A}$	ni $x : C_1$ ni $x : C_2$ dans $\mathcal{A}$	$\mathcal{A} := \mathcal{A} \cup \{x : C_1\}$ ou $\mathcal{A} := \mathcal{A} \cup \{x : C_2\}$
$\rightarrow_\forall$	$x : \forall rC \in \mathcal{A}$	$x\ r\ y \in \mathcal{A}$ mais $y : C \notin \mathcal{A}$	$\mathcal{A} := \mathcal{A} \cup \{y : C\}$
$\rightarrow_\exists$	$x : \exists rC \in \mathcal{A}$	$\nexists y$ tq. $x\ r\ y$ et $y : C$ soient tous deux dans $\mathcal{A}$	$\mathcal{A} := \mathcal{A} \cup \{z : C, x\ r\ z\}$ où z est une nouvelle variable

Table 1: Les règles de décomposition pour la méthode des tableaux sémantique pour $\mathcal{ALC}$

```
Inductive Orruleright (b1:Abox) (b2:Abox): Prop :=
  mk_Orruleright: forall x c1 c2,
       In (inst x (OrC c1 c2)) b1 ∧
        ~ In (inst x c1) b1 ∧ ~ In (inst x c2) b1
            -> b2 = add (inst x c2) b1
            -> Orruleright b1 b2.
```

Pour un fait, la règle $\rightarrow_\forall$ peut être appliquée plusieurs fois. Il est toutefois possible qu'elle n'est pas applicable à une étape sur un élément de la forme $x : \forall rC$ mais qu'elle le devient après l'application d'une autre règle qui génère un élément de la forme $x\ r\ y$.

```
Inductive Allrule (b1:Abox) (b2:Abox): Prop :=
  mk_Alrule: forall x y r c1,
     In (inst x (AllC  r c1)) b1 ∧
       In (rel x y r) b1∧ ~ In(inst y c1) b1
        -> b2 = add (inst y c1) b1
         -> Allrule b1 b2.
```

Pour formaliser la règle $\rightarrow_\exists$ on doit construire l'ensemble des instances dans l'*Abox*, afin de générer une nouvelle variable. Cette variable ne doit pas avoir une occurrence dans l'*Abox*. La fonction `set_ni_in_abox` retourne l'ensemble des instances qui apparaissent dans l'*Abox*.

```
Definition construct_set_in_fact (f:fact)(A:Setni):
                  Setni := match f with
 | inst ni c  =>
       fsetsni.add  ni  A
 | rel  ni1 ni2 r =>
       fsetsni.add ni1 (fsetsni.add ni2 A)
 | dif  ni1 ni2  =>
       fsetsni.add ni1 (fsetsni.add ni2 A)
 end.

Definition set_ni_in_abox (AB:Abox): Setni :=
     (fold construct_set_in_fact AB fsetsni.empty).
```

La règle $\rightarrow_\exists$ peut maintenant être définie par:

```
Inductive Somerule (b1:Abox) (b2:Abox): Prop :=
   mk_Somerule: forall x z r c1,
     In (inst x (SomeC r c1)) b1 ∧
     (forall y,
      ~ (In (rel x y r) b1 ∧ In (inst y c1) b1)) ∧
        ~ (fsetsni.In z (set_ni_in_abox b1))
        -> b2 = add (rel x z r) (add (inst z c1) b1)
         -> Somerule b1 b2.
```

7. PREUVE DE LA CORRECTION (SOUNDNESS)

Pour prouver la correction de l'algorithme du tableau sémantique, il est nécessaire de démontrer la préservation de la propriété de la satisfiabilité: Si une *Abox* A se transforme en *Abox* B, et si B est satisfiable, alors A est aussi satisfiable. On définit le prédicat d'adéquation comme suit:

```
Definition sound (r:Abox -> Abox -> Prop) :=
     forall AB1 AB2,
             r AB1 AB2
             -> abox_satisfiable AB2
             -> abox_satisfiable AB1.
```

On peut prouver l'adéquation de chaque règle de $\mathcal{ALC}$. Par exemple, l'énoncé du lemme d'adéquation de la règle $\rightarrow_\sqcap$ est:

```
Lemma and_sound: sound Andrule.
```

Le principe de la preuve est similaire pour les autres règles, pour chaque preuve, si AB_2 est satisfiable, donc elle possède un modèle et ce modèle satisfait aussi AB_1.

La formalisation qu'on propose est une formalisation modulaire, dont chaque règle est codée indépendamment des autres règles, les preuves sont aussi modulaires. L'objectif est d'écourter les preuves et de les rendre lisibles, ainsi pour pouvoir réutiliser cette formalisation pour une logique plus expressive.

On définit la liste des règles pour la logique $\mathcal{ALC}$ comme suit:

```
Definition ALC_rules :=
         Andrule::Orruleleft::Orruleright
                     ::Allrule::Somerule::nil.
```

Cette représentation peut éventuellement être valable pour d'autres logiques moins expressives ou encore plus expressives. Il suffit selon le cas réduire ou étendre la liste des règles.

Une *Abox* AB_1 se transforme en *Abox* AB_2, ssi une règle parmi les règles de cette liste est appliquée. La fonction `disj_rule` transforme une liste de règles en une règle qui est la disjonction des règles individuelles:

```
Fixpoint disj_rule (l:list (Abox -> Abox -> Prop)):
   (Abox -> Abox -> Prop) := match l with
| nil   => (fun AB1 AB2 => False)
| r::rs => (fun AB1 AB2 => (r AB1 AB2) ∨
            (disj_rule rs AB1 AB2))
end.
```

On prouve si toute règle dans la liste des règles est adéquate, alors la disjonction de ces règles est aussi adéquate:

```
Lemma sound_extend_disj:
  forall (l:list (Abox ->Abox ->Prop)),
     (forall r, List.In r l -> sound r)
                      -> sound (disj_rule l).
```

Soit $\rightarrow_r^*$ la fermeture réflexive transitive d'une relation r.

```
Definition rstar_rule := Rstar Abox  .
```

On peut démontrer le lemme suivant :

Si une règle $\rightarrow_r$ est adéquate alors $\rightarrow_r^*$ est aussi adéquate, i.e, Si $A \rightarrow_r^* B$, et si B est satisfiable alors A est aussi satisfiable.

```
Lemma sound_extend_rstar:
    forall r, sound r -> sound (rstar_rule r).
```

Pour établir une preuve de correction de toutes les règles, on définit le prédicat `ALC_rule` (écrit $\rightarrow_{\mathcal{ALC}}$) qui représente la disjonction des règles de transformation de la logique $\mathcal{ALC}$.

```
Definition ALC_rule := disj_rule ALC_rules.
Lemma rule_sound: sound ALC_rule.
```

8. PREUVE DE LA COMPLÉTUDE (COMPLETENSS)

Pour prouver la complétude, il est nécessaire d'introduire les deux définitions suivantes: Une *Abox* est *complète* si aucune règle de transformation n'est applicable.

```
Definition complet AB := forall AB1,
                                ~ ALC_rule AB AB1.
```

Une *Abox* est *contradictoire* si elle contient une contrainte contradictoire (clash), i.e, $x : C$ et $x : \neg C$ ou $x : \bot$.

```
Definition contains_clash (AB:Abox): Prop :=
     exists x, exists c,
      (In (inst x c) AB ∧ In (inst x (NotC c)) AB)
    ∨ (In (inst x Bottom) AB).
```

Une étape de calcul consiste a appliquer une règle sur une *Abox*, on calcule donc le "successeur" d'une *Abox* en applicant une règle de $\mathcal{ALC}$.

```
Definition succ AB1 AB2: Prop := ALC_rule AB2 AB1.
```

La propriété fondamentale de la correction de l'algorithme du tableau est que si l'*Abox* est fermée (contient un clash) alors elle est insatisfiable:

```
Lemma content_clash_not_satisfiable: forall AB,
      contains_clash AB -> ~ abox_satisfiable AB.
```

Enfin, si une *Abox* A est complète et non contradictoire, alors elle est satisfiable. Dans ce cas, il existe une interprétation qui satisfait A, qui est appelée l'*interprétation canonique* $\mathcal{I}_A$, dont les composants sont définis comme suit:

1. Le domaine d'interprétation $\Delta_{\mathcal{I}_A}$ de $\mathcal{I}_A$ est l'ensemble de tous les individus inclu dans A
2. Pour chaque nom de concept P on définit $P_{\mathcal{I}_A} = \{x | (x : P) \in A\}$
3. Pour chaque nom de rôle r on définit $r_{\mathcal{I}_A} = \{(x, y) | r(x, y) \in A\}$

La description correspondante en Coq est:

```
Definition
  set_ni_left_atom A (AB:Abox): Ensemble NI :=
          fun x => In (inst x (AtomC A)) AB.

Definition
  rel_ni_atom_role R (AB:Abox): Relation NI :=
          fun x y => In (rel x y (AtomR R)) AB.
```

L'interprétation canonique est donc définie comme suit:

```
Definition  can_in (AB : Abox): Interp :=
       Build_Interp (fun c => set_ni_left_atom c AB)
                    (fun R => rel_ni_atom_role R AB)
                                (fun x => x).
```

Maintenant, l'objectif, est de démontrer que, si une *Abox* est non contradictoire et complète, alors elle est satisfiable par l'interprétation canonique.

```
Lemma complet_not_contr_is_satisfiable:
   forall AB1, is_Normal_Abox AB1
                -> complet AB1
                 -> ~ contains_clash AB1
                  -> abox_satisfiable AB1.
```

9. PREUVE DE LA TERMINAISON

La terminaison est une propriété importante des systèmes de réécriture. Une méthode classique pour prouver la terminaison d'un système de réécriture est d'exhiber un ordre bien fondé sur les termes, tel que, si A_1 se réécrit en A_2 alors $A_1 \gg A_2$.

Formellement, on doit prouver que la relation successeur est une relation d'ordre bien fondée. Nous anticipons ici le résultat que nous développons dans les sections suivantes:

THÉORÈME 1. *wellfounded succ.*

Pour prouver ce théorème, on associe à chaque *Abox* une mesure. Si cette mesure décroît pour toute règle dans un ordre bien fondé, la terminaison est assurée. Dans notre cas, une mesure est une fonction de type $Abox \rightarrow T$ pour un domaine T équipé d'une relation bien fondée $\ll$ dans T que nous définissons dans § 9.3.

La preuve du théorème se fait à l'aide d'un lemme qui dit que la réécriture d'un *Abox* A_1 en A_2 fait décroître la mesure:

LEMME 1.

$$\forall A_1 A_2, (A_1 \rightarrow_{\mathcal{ALC}} A_2) \rightarrow (Mesure(A_2) \ll Mesure(A_1))$$

9.1 L'ordre bien fondé multi-ensembliste

Il est bien connu que, si un ordre est bien fondé alors aussi l'extension de l'ordre aux multi-ensembles est bien fondé [11, 10]. Informellement, un multi-ensemble est un ensemble qui peut contenir plusieurs fois le même élément. Formellement, un multi-ensemble M d'éléments dans un ensemble E est une fonction de E dans l'ensemble des entiers naturels, qui fait correspondre à chaque élément x de E le nombre $M(x)$ d'occurrences de x dans M.

Un multi-ensemble est fini si son support est fini, i.e. l'ensemble des x avec image non nulle est fini. On note $M(A)$ l'ensemble des multi-ensembles finis sur A. Soit $>$ un ordre strict sur A. L'extension multiensemble $>_{mult}$ sur $M(A)$ est définie par: $M >_{mult} N$ si il existe les multi-ensembles X, Y et Z vérifiant:

- $Y \neq \emptyset$
- $M = X \cup Y$
- $N = X \cup Z$
- $\forall z \in Z, \exists y \in Y, y > z$

Nous nous sommes appuyés sur une formalisation de l'ordre multi-ensembliste dans l'assistant Coq élaborée dans le projet CoLoR [18, 8].

EXEMPLE 4. *Un exemple de multi-ensemble est* $\{\{1,2,3,3\}\}$ *qui contient 4 éléments, dont 2 fois l'élément 3.*
Le multi-ensemble $\{\{1,4\}\} >_{mult} \{\{1,2,3,3\}\}$*: le nombre 4 du premier multi-ensemble a été remplacé par les nombres 2, 3, 3 qui sont tous plus petit que 4.*

9.2 L'ordre lexiographique

Soient $>_A$ et $>_B$ des ordres stricts sur les ensembles A et B respectivement. L'ordre lexiographique sur la paire $A \times B$ est défini comme:
$(a,b) >_{lex} (a',b') \Leftrightarrow a >_A a' \vee (a = a' \wedge b >_B b')$

EXEMPLE 5. *Dans l'ensembe N*

- $(5,6) >_{lex} (4,9)$
- $(5,6) >_{lex} (5,5)$

9.3 Une mesure pour l'algorithme de $\mathcal{ALC}$

On introduit les constructeurs nécessaires pour la définition de la mesure:
La fonction `sizeC` calcule la taille d'un concept, elle est défini comme:

```
Fixpoint sizeC (c :concept): nat :=
 match cwith
  AtomC _    => 1
 |Top        => 1
 |Bottom     => 1
 |NotC c1    => 1 + sizeC c1
 |AndC c1 c2 => 1 + sizeC c1 + sizeC c2
 |OrC  c1 c2 => 1 + sizeC c1 + sizeC c2
 |AllC r c1  => 1 + sizeC c1
 |SomeC r c1 => 1 + sizeC c1
end.
```

La définition de la taille d'un fait est dérivée de la définition précédente:

```
Definition sizef f :=
  match f with
   |inst n c => sizeC c
   |_ => 0
end.
```

On définit un prédicat inductif d'applicabilité pour chaque règle sur un fait, par exemple, le prédicat `and_applicable`qui correspond à la règle $\rightarrow_\sqcap$:

```
Inductive and_applicable (f:fact)(AB:Abox): Prop :=
  App_and :forall x c1 c2,
    (inst x (AndC c1 c2) = f) ∧ In f AB ∧
     ~ (In (inst x c1) AB  ∧  In (inst x c2) AB)
        -> and_applicable f AB.
```

Le lemme suivant démontre que l'application de règle $\rightarrow_\sqcap$ est décidable.

```
Lemma and_applicable_or_not: forall f AB,
   {and_applicable f AB} + {~ and_applicable f AB}.
```

Le calcul d'une mesure suit le même schéma pour toutes les règles:

- La mesure dans notre cas est un multi-ensemble de couples de nombres naturels.
- A chaque axiome (`fact`) de l'*Abox*, on associe un couple, en fonction de la structure de l'axiome et de l'*Abox*.
- La mesure de l'élément qui est applicable doit décroître, sans qu'elle influe sur la mesure des autres éléments

On définit maintenant la mesure sur les axiomes (c'est la fonction `mes_comp` d'en bas). Si l'axiome est:

- Une relation $x\ r\ y$, on lui associe le couple $(0,0)$
- De la forme $x \neq y$, on lui associe $(0,0)$
- Une instance $x : D$. Selon la structure de D, il y'a trois cas:
  1. Les cas où D est un *Atome* $(x : A)$ ou *Négation* $(x : \neg A)$, on lui associe la valeur $(0,0)$;
  2. Le cas où D est une conjonction, disjonction ou quantificateur existentiel:
     - Si la règle est applicable sur l'axiome, on lui associe le couple $(size(D), 0)$,
     - Sinon $(0,0)$;
  3. Le cas où D est un quantificateur universel $(D = \forall r.C)$ on lui associe le couple $(Comp_1, Comp_2)$ tel que:
     (a) $Comp_1 = size(D)$;
     (b) $Comp_2 = Comp_{21} + Comp_{22}$ dont:
     - $Comp_{21}$ est le nombre d'applicabilité de la règle $\rightarrow_\forall$, càd le nombre de $x\ r\ y$ dans l'*Abox* tel que $y : C$ n'est pas dans l'*Abox*. On constate ici que $Comp_{21}$ décroît si la règle $\rightarrow_\forall$ est applicable, mais si on applique la règle $\rightarrow_\exists$ sur un autre fait, cette mesure peut augmenter. Pour cela, on ajoute le composant $Comp_{22}$ qui assure que cette mesure reste constante par l'application d'une autre règle;
     - $Comp_{22}$ est le nombre de $\exists$-*termes* réductibles et cachés dans l'*Abox*. Cette valeur décroît si la règle $\rightarrow_\exists$ est appliquée et reste constante ou décroît si une autre règle est appliquée.

On commence par des preuves de décidabilité.

```
Definition is_dec (p:Prop) := p+~p.
Definition eq_dec (T :Set) := forall x y:T, is_dec (x=y).

Lemma eq_dec_concept : eq_dec concept.

Lemma eq_dec_role : eq_dec role.
```

Le lemme `eq_dec_role` prouve que l'égalité des rôles est décidable. Le type de `eq_dec_role` dans Coq est `Set`, on construit le terme `eq_role` équivalent de type booléen (`bool`).

```
Definition eq_role r1 r2 := if eq_dec_role r1 r2
                            then true else false.
```

Et de la même manière pour les instances.

```
Definition eq_ni x y := if NI_as_DT.eq_dec x y
                        then true else false .
```

Après cette discussion de spécificités de Coq, nous présentons maintenant les fonctions qui sont nécessaires pour définir formellement la mesure d'un *Abox*.

Étant donné un fait f, un rôle r, une instance x, et un ensemble des instances A, la fonction `construct_set_of_y_in_rel` ajoute y dans A si le fait est $x\ r\ y$, sinon elle retourne A.

```
Definition
construct_set_of_y_in_rel x r(f:fact)(A:Setni):Setni:=
 match f with
 |rel x1 y r1 => if (eq_ni x1 x) && (eq_role r r1)
                 then fsetsni.add y A else A
 | _ => A
 end.
```

La fonction `set_y_in_abox` retourne l'ensemble des y de $x\ r\ y$ dans l'*Abox*.

```
Definition set_y_in_abox x r (AB:Abox): Setni :=
  fold (construct_set_of_y_in_rel x r) AB
                               fsetsni.empty.
```

Exemple 6. *Soit l'*abox*:*

$$AB = \{x : \forall r.C,\ x\ r\ y,\ x\ r\ z,\ x\ r\ u,\ u : c\}.$$

l'ensemble `set_y_in_abox` $x\ r\ AB = \{y\ |x\ r\ y \in AB\} = \{y\ , z\ , u\ \}$.

Dans la suite on construit l'ensemble des y tel que $y : c$ est dans l'*abox*.

```
Definition construct_set_of_y_in_concept
  c (f:fact)(A:Setni): Setni:= match f with
 | inst y c1 => if (eq_dec_concept c c1)
            then fsetsni.add y A else A
 |  _ => A
 end.

Definition set_y_c_in_abox c (AB:Abox): Setni :=
fold (construct_set_of_y_in_concept c) AB
                               fsetsni.empty.
```

Exemple 7. *Soit l'*abox*:*

$$AB = \{x : \forall r.C,\ x\ r\ y,\ x\ r\ z,\ x\ r\ u,\ u : c\}.$$

l'ensemble des y tel que $y : c$ *est dans AB:*
`set_y_c_in_abox` $c\ AB = \{y\ |y : c \in AB\} = \{u\}$.

On définit le composant $Comp_{21}$ comme la cardinalité de la différence entre les deux ensembles définis avant.

```
Definition set_rel x c r AB := fsetsni.cardinal
fsetsni.diff
  (set_y_in_abox x r AB)(set_y_c_in_abox c AB).
```

Exemple 8. *Soit l'*abox*:*

$$AB = \{x : \forall r.C,\ x\ r\ y,\ x\ r\ z,\ x\ r\ u,\ u : c\}.$$

Dans ce cas: `set_rel` $x\ c\ r\ AB = |\{y\ , z\}| = 2$.

Dans la suit on construit le composant $Comp_{22}$.

On commence par la construction de l'ensemble de $\exists-$termes qui occurent dans un concept:

```
Fixpoint some_term (x:NI) r c: Abox :=
   match c with
 AtomC _    => empty
|Top        => empty
|Bottom     => empty
|NotC c1    => some_term  x r c1
|AndC c1 c2 =>
    union (some_term x r c1) (some_term x r c2)
|OrC c1 c2  =>
    union (some_term x r c1) (some_term x r c2)
|AllC  _ c1 => some_term x r c1
|SomeC r1 c1=>
 if eq_role r r1
 then add (inst x (SomeC r1 c1))(some_term x r c1)
 else some_term x r c1
end.
```

Exemple 9. *Soit* c_1, c_2 *et* c_3 *des concepts atomiques et un concept* $c = \exists rc1\ \sqcap\ \exists r(c_2 \sqcup c_3)\ \sqcup\ \forall rc_3$.
L'ensemble de $\exists-$*termes qui occurent dans le concept c est:* `some_term` $x\ \ r\ \ c = \{x : \exists rc_1,\ x : \exists r(c_2 \sqcup c_3)\}$.

```
Definition set_some_in_fact x r f :=
            match f with
             | inst  y c => some_term x r c
             |_ =>  empty
     end.

Definition Construct_set_some x r f A :=
            union (set_some_in_fact x r f) A.

Definition set_of_some_term  x r AB :=
     fold (Construct_set_some x r) AB empty.
```

Exemple 10. *Soit l'*abox*:*
$AB = \{x\ :\ (\exists rc1\ \sqcap\ \exists r(c_2 \sqcup c_3)\ \sqcup\ \forall rc_3)\ ,\ x : \exists rc_4\}$
L'ensemble `set_of_some_term` $x\ \ r\ \ AB = \{x\ :\ \exists rc_1,\ \ x\ :\ \exists r(c_2 \sqcup c_3),\ x\ :\ \exists rc_4\}$.

```
Definition some_term_reducible  AB f :=
    match some_applicable_or_not_2 f AB with
                      | left _  => singleton f
                      | right _ => empty
       end.

Definition Construct_set_some_r AB f A :=
    union (some_term_reducible AB f) A.

Definition set_of_some_term_r x r AB:=
 fold (Construct_set_some_r AB)
             (set_of_some_term x r AB) empty.
```

Exemple 11. *Pour l'*abox*:*
$AB = \{x : (\exists rc1 \sqcap \exists r(c_2 \sqcup c_3)\ \sqcup\ \forall rc_3)\ ,\ x : \exists rc_4,\ xry,\ y : c1\}$
l'ensemble de $\exists-$*termes réductibles qui occurent dans AB est:* `set_of_some_term_r` $x\ r\ AB = \{x\ :\ \exists r(c_2 \sqcup c_3),\ \ x\ :\ \exists rc_4\}$.

Pour un fait f et une *Abox* AB, la mesure de f dans AB est définie comme suit:

```
Definition mes_comp AB f :=
match f with
|inst x c => match c with
   |AndC c1 c2 =>
       match and_applicable_or_not f AB with
                   | left _   => (sizef f, 0)
                   | right _  => (0,0)
                                 end
   |OrC c1 c2 =>
       match or_applicable_or_not f ABwith
                 | left _   => (sizef f, 0)
                 | right _  => (0,0)
                                  end
   |SomeC r c1 =>
        match some_applicable_or_not f AB with
                   | left _   => (sizef  f, 0)
                   | right _  => (0,0)
                                  end
   |AllC r c1 => (sizef f, set_rel x c1 r AB +
                cardinal (set_of_some_term_r x r AB))

    | _ =>(0,0)
          end
|_ => (0,0)
 end.
```

EXEMPLE 12. *Soit c_1, c_2 , c_3 et c_3 des concepts atomiques et l'*abox*:*

$AB = \{x : c1 \sqcap (c2 \sqcup c3),\ x : c2 \sqcup c3,\ x : c2,\ y : \forall rc_3,\ yrz\}$.

- *Le fait $x : c1 \sqcap (c2 \sqcup c3)$ est réductible, sa mesure est :* `mes_comp` $(x : c1 \sqcap (c2 \sqcup c3))$ *AB = (5,0).*
- *La mesure du fait $x : c2 \sqcup c3$ est:* `mes_comp` $(x : c2 \sqcup c3)$ *AB =(0,0).*
- *La mesure du fait $y : \forall rc_3$ est:* `mes_comp` $(x : c2 \sqcup c3)$ *AB =(2,1).*

On construit la mesure d'une *Abox* de la façon suivante:

```
Definition
   construct_Multiset (A:Abox)(f:fact) M1: Multiset
        := insert (mes_comp A f) M1.

Definition Measure2 A B: Multiset :=
   (fold (construct_Multiset A) B MSetCore.empty).

Definition Measure_Abox AB := Measure2 AB AB.
```

EXEMPLE 13. *Soit c_1, c_2 , c_3 et c_3 des concepts atomiques et l'*abox*:*

$AB = \{x : c1 \sqcap (c2 \sqcup c3),\ x : c2 \sqcup c3,\ x : c2,\ y : \forall rc_3,\ yrz\}$.

La mesure de AB est le multi-ensemble:
`Measure_Abox` $AB = \{\{$ *(5,0), (0,0), (0,0), (2,1), (0,0)*$\}\}$.

9.4 La preuve de terminaison pour $\mathcal{ALC}$

Pour chaque application d'une règle, on doit montrer que la mesure décroît. À titre d'exemple, le lemme suivant démontre cette propriété pour la règle $\rightarrow_\sqcap$.

```
Lemma and_decrease: forall s1 s2, Andrule s2 s1
     -> MultisetLT gtA (Measure s1) (Measure s2).
```

Une fois que cette propriété est démontrée pour chaque règle, on peut généraliser cette propriété sur la relation `succ`:

```
Lemma succ_decrease: forall s1 s2,
  succ s1 s2
   -> MultisetLT gtA (Measure_Abox s1) (Measure_Abox s2).
```

La relation $<_{mul}$ est une relation bien fondée:

```
Lemma wf_multiLT: well_founded (MultisetLT gt).
```

Enfin, on peut déduire facilement que la relation `succ` est une relation d'ordre bien fondé:

```
Lemma wf_succ: well_founded succ.
```

10. CONCLUSION

Dans cet article, nous avons présenté une formalisation de la logique de description $\mathcal{ALC}$ qui adopte une approche modulaire. Cette formalisation en Coq s'articule sur plusieurs modules:

- la spécification de la syntaxe et de la sémantique de $\mathcal{ALC}$,
- le codage de l'*Abox* et la formalisation des règles de transformation du tableau sémantique,
- la preuve de l'adéquation du tableau sémantique,
- la preuve de la complétude du tableau sémantique,
- la preuve de la terminaison du tableau qui nécessite la définition d'une mesure pour chaque *Abox*. Nous avons montré que cette mesure décroît pour chaque application d'une règle.

Nous envisagerons plusieurs extensions pour ce travail, parmi lesquelles on peut citer:

- des extensions de la logique $\mathcal{ALC}$ pour nous rapprocher des logiques plus expressives utilisées dans le Web sémantiques, telles que $\mathcal{SHOIQ}$ et $\mathcal{SHOIN}$.
- l'extraction d'un raisonneur exécutable certifié.
- des stratégies de preuve qui préservent la complétude de la procédure, mais qui en augmentent l'efficacité.
- dans le même sens, on peut viser à concevoir des structures de données qui permettent une optimisation de l'application des règles. Ainsi, dans notre implantation actuelle, il nous faut traverser un tableau à plusieurs reprises pour tester si une règle est applicable.

11. REFERENCES

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

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

[3] F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. In *Mechanizing Mathematical Reasoning*, pages 228–248, 2005.

[4] F. Baader and U. Sattler. Expressive number restrictions in description logics. *Journal of Logic and Computation*, 9(3):319–350, 1999.

[5] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69(1):5–40, 2001.

[6] D. Berardi, D. Calvanese, and G. D. Giacomo. 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*, 2001.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[8] F. Blanqui, S. Coupet Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR: a Coq library on rewriting and termination. In *Eighth International Workshop on Termination - WST 2006*, Seattle United States, 2006.

[9] P. de Wind. Modal logic. Master's thesis, Vrije Universiteit Amsterdam, May 2001.

[10] N. Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3:69–116, 1987.

[11] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[12] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In *KR*, pages 151–162, 1991.

[13] D. Fehrer, U. Hustadt, M. Jaeger, A. Nonnengart, H. J. Ohlbach, R. A. Schmidt, C. Weidenbach, and E. Weydert. Description logics for natural language processing. In F. Baader, M. Lenzerini, W. Nutt, and P. F. Patel-Schneider, editors, *International Workshop on Description Logics '94*, volume D-94-10 of *Document*, pages 80–84, Bonn, Germany, 1994. DFKI.

[14] V. Haarslev and R. Möller. Racer system description. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 701–706, London, UK, 2001. Springer-Verlag.

[15] M.-J. Hidalgo, J.-A. Alonso, J. Borrego-Díaz, F.-J. Martin-Mateos, and J.-L. Ruiz-Reina. A formally verified prover for the alc description logic. In *20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 135–150, 2007.

[16] M.-J. Hidalgo, J.-A. Alonso, F.-J. Martin-Mateos, and J.-L. Ruiz-Reina. Constructing formally verified reasoners for the image description logic. In *Proceedings of the 3rd International Workshop on Automated Specification and Verification of Web Systems (WWV 2007)*, volume 200 of *ENTCS*, 2008.

[17] I. Horrocks and U. Sattler. A tableau decision procedure for $\mathcal{SHOIQ}$. *J. of Automated Reasoning*, 39(3):249–276, 2007.

[18] A. Koprowski. *Termination of Rewriting and Its Certification*. PhD thesis, Eindhoven University of Technology, Sept. 2008.

[19] M.J.C. Gordon. HOL: A proof generating system for higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, Boston, 1988.

[20] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[21] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[22] T. Ridge and J. Margetson. A mechanically verified, sound and complete theorem prover for fol. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, Aug. 2005.

[23] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008.

[24] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.