Logical Foundations for Reasoning about Transformations of Knowledge Bases

Mohamed Chaabani[1], Rachid Echahed[2], Martin Strecker[3]

[1] LIMOSE, University of Boumerdès, Algeria
[2] Laboratoire d'Informatique de Grenoble
[3] Université de Toulouse / IRIT *

Abstract. This paper is about transformations of knowledge bases with the aid of an imperative programming language which is non-standard in the sense that it features conditions (in loops and selection statements) that are description logic (DL) formulas, and a non-deterministic assignment statement (a choice operator given by a DL formula). We sketch an operational semantics of the proposed programming language and then develop a matching Hoare calculus whose pre- and post-conditions are again DL formulas. A major difficulty resides in showing that the formulas generated when calculating weakest preconditions remain within the chosen DL fragment. In particular, this concerns substitutions whose result is not directly representable. We therefore explicitly add substitution as a constructor of the logic and show how it can be eliminated by an interleaving with the rules of a traditional tableau calculus.

Keywords: Description Logic; Graph Transformation; Programming Language Semantics; Tableau Calculus

1 Introduction

Contribution The question explored in this paper is: What is an adequate formalism for describing modifications of Knowledge Bases (KBs), and how to reason about the effects of these modifications?

Let us fix some terminology: In this paper, a KB is perceived as a graph structure, consisting of nodes and binary relations between these nodes. A KB transformation modifies this graph structure, by inserting or deleting arcs in the graph (and by adding or deleting nodes, but this aspect is not addressed in this paper because it would need mechanisms analogous to memory allocation and deallocation in traditional programming languages).

A KB can be seen as a model of a formula of a particular logical language. For expressive logics, typical transformation problems (see further below) become undecidable. We therefore cut down the problem to rather inexpressive logics, in this case a variant of the Description Logic $\mathcal{ALCQ}$.

A transformation of a KB induces a transformation of predicates true about the KB: If predicate P is true for a KB k, which predicate P' is true for the

* Part of this research has been supported by the *Climt* project (ANR-11-BS02-016).

transformed KB k'? The answer to such a question depends on at least two factors: the language used for defining transformations, and the logical formalism for reasoning about them. Our contribution consists in

- a proposal for a transformation language similar to a traditional imperative language, but endowed with some non-standard constructs (a non-deterministic assignment operator and conditional and loop statements where expressions are replaced by formulas). In spite of these features, transformations are effectively computable. The extension of $\mathcal{ALCQ}$ we use is defined in Sect. 2, the programming language in Sect. 3.
- a sound and decidable Hoare-style calculus for reasoning about transformations written in this language.

The approach is rather standard: we compute weakest preconditions (WPs) by structural recursion over the statements of the transformation language, see Sect. 4. Unfortunately, it turns out that the logic $\mathcal{ALCQ}$ is not directly closed wrt. substitutions that have to be carried out when computing WPs. We therefore add a new constructor for substitutions to the logic, and show how it can be eliminated in a tableau calculus (in Sect. 5).

Related work Reasoning about graph transformations in full generality is hard [7]. Some decidable logics for graph transductions are known, such as MSO [6], but are descriptive, applicable to a limited number of graphs and often do not match with an algorithmic notion of transformation. Some implementations of verification environments for pointer manipulating programs exist [9], but they often impose severe restrictions on the kind of graphs that can be manipulated, such as having a clearly identified spanning tree.

In [4], the authors have introduced a dynamic logic which is very expressive. It has been designed to describe different kinds of elementary knowledge base transformations (addition of new items, addition and deletion of links, etc.). It allows also to specify advanced properties on graph structures which go beyond μ-calculus or MSO logics. Unfortunately, the expressive power of that logic has a price: the undecidability of the logic. The purpose of the present paper is to identify a programming language together with a logic such that the proof problem resulting from the transformation of the KB is decidable. The transformations themselves are not encoded in the logic itself (as in [4]) but in a dedicated imperative language for which we develop a Hoare-style calculus.

Work on (KB) updates [8] seems to approach the problem from the opposite direction: Add facts to a KB and transform the KB at the same time such that certain formulas remain satisfied. In our approach, the modification of the KB is exclusively specified by the program.

The work described in this paper is ongoing, some results are still preliminary. Based on previous work [5], we are in the process of coding the formalism described here in the Isabelle proof assistant [10]. Parts of the coding in this paper are inspired by formalizations in the Isabelle distribution and by [11].

The formal development accompanying this paper will be made available on the web[4], which should also be consulted for proofs.

Before starting with the formal development, let us give an example of the kind of program (see Fig. 1) that we would like to write. Assume a knowledge base with objects of class A and B, and a relation R. The node n is initially connected to at least 3 objects of class A, and all objects it is connected to are of class A or B. Because the number of connections to A is too large, we execute a loop that selects an A-object (let's call it a) that n is connected to, and delete the r-connection between n and a. To compensate, we select an object b of class B and connect n to b. We stop as soon as the number of A-connections of n has reached 2, which is one of the post-conditions we can ascertain. The resulting transformation is depicted in Fig. 2. One also sees that the language is too weak to express other properties, for example that the total number of arcs is preserved by the transformation.

```
vars n, a, b;

/* Pre:  n : (≥ 3 R A) ⊓ (∀ R (A ⊔ B))  */

while ( n : (> 2 R A) ) do {
      /* Inv:  n : (≥ 2 R A) ⊓ (∀ R (A ⊔ B))  */
      select a sth a : A ∧ (n R a);
      delete(n R a);
      select b sth b : B ;
      add(n R b)
   }
/* Post:  n : (= 2 R A) ⊓ (∀ R (A ⊔ B))  */
```

Fig. 1. An example program

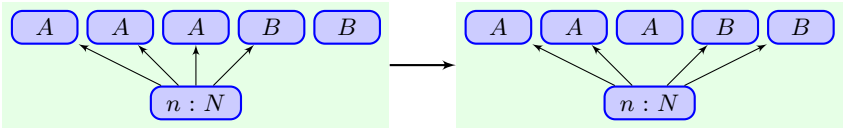

Fig. 2. Resulting transformation

[4] http://www.irit.fr/~Martin.Strecker/Publications/dl_transfo2013.html

2 Logic

Our logic is a three-tier framework, the first level being DL *concepts*, the second level *facts*, the third level *formulas* (Boolean combinations of facts and a simple form of quantification).

Concepts: We concentrate on a DL featuring concepts with simple roles and number restrictions, similar to $\mathcal{ALCQ}$ [2]. For c being the type of concept names and r the type of role names, the data type C of concepts can be defined inductively by:

$$\begin{array}{lll} C ::= \bot & & \text{(empty concept)} \\ \quad | \;\; c & & \text{(atomic concept)} \\ \quad | \;\; \neg\, C & & \text{(negation)} \\ \quad | \;\; C \sqcap C & & \text{(conjunction)} \\ \quad | \;\; C \sqcup C & & \text{(disjunction)} \\ \quad | \;\; (\geq n\ r\ C) & & \text{(at least)} \\ \quad | \;\; (< n\ r\ C) & & \text{(no more than)} \\ \quad | \;\; C[r := RE] & & \text{(explicit substitution)} \end{array}$$

We define the universal concept $\top$ as $\neg\bot$ and write $(\exists\ r\ C)$ for $(\geq 1\ r\ C)$ and $(\forall\ r\ C)$ for $(< 1\ r\ (\neg C))$.

The last constructor, *explicit substitution* [1], is a particularity of our framework, required for a lazy elimination of substitutions that replace, in a concept C, a role name r by a role expression RE. If i is the set of individual variable names, the type RE is defined by

$$\begin{array}{lll} RE ::= r & & \text{(atomic role)} \\ \quad | \;\; r - (i, i) & & \text{(deletion of relation instance)} \\ \quad | \;\; r + (i, i) & & \text{(insertion of relation instance)} \end{array}$$

Please note that concepts implicitly depend on the types c, r and i, which we assume mutually disjoint. A substitution can therefore never affect an individual variable.

A set-theoretic semantics is provided by a domain Δ and an interpretation function $\mathcal{I}$ mapping c to a set of individuals (subsets of Δ), r to a binary relation of individuals (subsets of $\Delta \times \Delta$), and i to individual elements of Δ.

For interpretation of concepts C, negation is inductively interpreted as complement, concept conjunction as intersection and disjunction as union. $\mathcal{I}(\geq n\ r\ C) = \{x \mid card\{y \mid (x, y) \in \mathcal{I}(r) \wedge y \in \mathcal{I}(C)\} \geq n\}$, and analogously for $\mathcal{I}(< n\ r\ C)$. Here, *card* is the cardinality of finite sets (and 0 otherwise).

For interpretation of role expressions RE, we define $\mathcal{I}(r - (i_1, i_2)) = \mathcal{I}(r) - \{(\mathcal{I}(i_1), \mathcal{I}(i_2))\}$, and $\mathcal{I}(r + (i_1, i_2)) = \mathcal{I}(r) \cup \{(\mathcal{I}(i_1), \mathcal{I}(i_2))\}$.

Interpretation update $\mathcal{I}^{[r:=rl]}$ modifies the interpretation $\mathcal{I}$ at relation name r to relation rl, thus $\mathcal{I}^{[r:=rl]}(r) = rl$ and $\mathcal{I}^{[r:=rl]}(r') = \mathcal{I}(r')$ for $r' \neq r$. With this, we can define the semantics of explicit substitution by $\mathcal{I}(C[r := RE]) = \mathcal{I}^{[r:=\mathcal{I}(RE)]}(C)$.

Facts: Facts make assertions about an instance being an element of a concept, and about being in a relation. In DL parlance, facts are elements of an ABox. The type of facts is defined as follows:

$$
\begin{array}{lll}
fact ::= i : C & & \text{(instance of concept)} \\
\quad | \;\; i\ r\ i & & \text{(instance of role)} \\
\quad | \;\; i\ (\neg r)\ i & & \text{(instance of role complement)} \\
\quad | \;\; i = i & & \text{(equality of instances)} \\
\quad | \;\; i \neq i & & \text{(inequality of instances)}
\end{array}
$$

The interpretation of a fact is a truth value, defined by:

- $\mathcal{I}(i : C) = (\mathcal{I}(i) \in \mathcal{I}(C))$
- $\mathcal{I}(i_1\ r\ i_2) = (\mathcal{I}(i_1), \mathcal{I}(i_2)) \in \mathcal{I}(r)$ and $\mathcal{I}(i_1\ (\neg r)\ i_2) = (\mathcal{I}(i_1), \mathcal{I}(i_2)) \notin \mathcal{I}(r)$
- $\mathcal{I}(i_1 = i_2) = (\mathcal{I}(i_1) = \mathcal{I}(i_2))$ and $\mathcal{I}(i_1 \neq i_2) = (\mathcal{I}(i_1) \neq \mathcal{I}(i_2))$

Please note that since concepts are closed by complement, facts are closed by negation (the negation of a fact is again representable as a fact), and this is the main motivation for introducing the constructors "instance of role complement" and "inequality of instances".

Formulas: A formula is a Boolean combination of facts. We also allow quantification over individuals i (but not over relations or concepts), and, again, have a constructor for explicit substitution. We overload the notation $\bot$ for empty concepts and the Falsum.

$$
\begin{array}{lll}
form ::= \bot & & \\
\quad | \;\; fact & & \\
\quad | \;\; \neg form & & \\
\quad | \;\; form \wedge form & | \; form \vee form & \\
\quad | \;\; \forall i.form & | \; \exists i.form & \\
\quad | \;\; form[r := RE] & &
\end{array}
$$

The extension of interpretations from facts to formulas is standard; the interpretation of substitution in formulas is in entire analogy to concepts. As usual, a formula that is true under all interpretations is called *valid*.

When calculating weakest preconditions (in Sect. 4), we obtain formulas which essentially contain no existential quantifiers; we keep them as constructor because they can occur as intermediate result of computations. We say that a formula is *essentially universally quantified* if $\forall$ only occurs below an even and $\exists$ only below an odd number of negations. For example, $\neg(\exists x.\, x : C \wedge \neg(\forall y.\, y : D))$ is essentially universally quantified.

Implication $f_1 \longrightarrow f_2$ is the abbreviation for $\neg f_1 \vee f_2$, and $ite(c, t, e)$ the abbreviation for $(c \longrightarrow t) \wedge (\neg c \longrightarrow e)$, not to be confused with the if-then-else statement presented in Sect. 3.

3 Programming Language

The programming language is an imperative language manipulating relational structures. Its distinctive features are conditions (in conditional statements and loops) that are restricted DL formulas, in the sense of Sect. 2. It has a non-deterministic assignment statement allowing to select an element satisfying a fact. Traditional types (numbers, inductive types) are not provided.

In this paper, we only consider a core language with traditional control flow constructs, but without procedures. Also, it is only possible to modify a relational structure, but not to "create objects" (with a sort of `new` statement) or to "deallocate" them. These constructs are left for further investigation.

The type of statements is defined by:

stmt ::=	`Skip`	(empty statement)
	`select` i `sth` $form$	(assignment)
|	`delrel(`$i\ r\ i$`)`	(delete arc in relation)
|	`insrel(`$i\ r\ i$`)`	(insert arc in relation)
|	*stmt* `;` *stmt*	(sequence)
|	`if` $form$ `then` *stmt* `else` *stmt*	
|	`while` $form$ `do` *stmt*	

The semantics is a big-step semantics with rules of the form $(st, \sigma) \Rightarrow \sigma'$ expressing that executing statement st in state σ produces a new state σ'.

The rules of the semantics are given in the Fig. 3. Beware that we overload logical symbols such as $\exists$, $\wedge$ and $\neg$ for use in the meta-syntax and as constructors of $form$.

The state space σ is a function mapping individual variables to individuals in the semantic domain Δ; concepts to sets of individuals, and so forth. It is therefore identical to an interpretation function $\mathcal{I}$ as introduced in Sect. 2, and it is only in keeping with traditional notation in semantics that we use the symbol σ. We may therefore write $\sigma(b)$ to evaluate the condition b (a formula) in state σ.

Most of the rules are standard, apart from the fact that we do not use expressions, but formulas as conditions. The auxiliary function *delete_edge* modifies the state σ by removing an r-edge between the elements represented by v_1 and v_2. With the update function for interpretations introduced in Sect. 2, one defines

$$delete\_edge\ v_1\ r\ v_2\ \sigma = \sigma^{[r:=\sigma(r)-\{(\sigma(v_1),\sigma(v_2))\}]}$$

and similarly

$$generate\_edge\ v_1\ r\ v_2\ \sigma = \sigma^{[r:=\sigma(r)\cup\{(\sigma(v_1),\sigma(v_2))\}]}$$

The statement `select` v `sth` $F(v)$ selects an element vi that satisfies formula F, and assigns it to v. For example, `select` a `sth` $a : A \wedge (a\ r\ b)$ selects an element a instance of concept A and being r-related with a given element b.

`select` is a generalization of a traditional assignment statement. There may be several instances that satisfy F, and the expressiveness of the logic might not suffice to distinguish them. In this case, any such element is selected, non-deterministically. Let us spell out the precondition of ($SelAssT$): Here, $\sigma^{[v:=vi]}$ is an interpretation update for individuals, modifying σ at individual name $v \in i$ with an instance $vi \in \Delta$, similar to the interpretation update for relations seen before. We therefore pick an instance vi, check whether the formula b would be satisfied under this choice, and if it is the case, keep this assignment.

In case no satisfying instance exists, the semantics blocks, *i.e.* the given state does not have a successor state, which can be considered as an error situation.

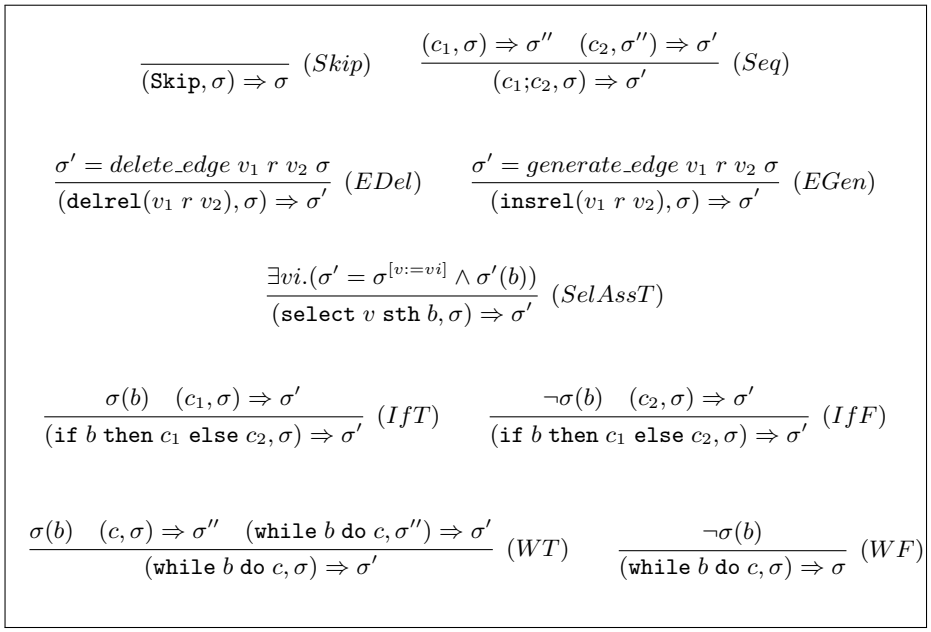

Fig. 3. Big-step semantics rules

Some alternatives to this design choice can be envisaged: We might treat a `select` v `sth` $F(v)$ with unsatisfiable F as equivalent to a `Skip`. This would give us a choice of two rules, one in which the precondition of rule ($SelAssT$) is satisfied, and one in which it is not. As will be seen in Sect. 4, this would introduce essentially existentially quantified variables in our formulas when computing weakest preconditions and lead us out of the fragment that we can deal with in our decision procedure. Alternatively, we could apply an extended type check verifying that select-predicates are always satisfiable, and thus ensure that type-correct programs do not block. This is the alternative we prefer; details still have to be worked out.

4 Weakest Preconditions

We compute weakest preconditions wp and verification conditions vc. Both take a statement and a DL formula as argument and produce a DL formula. For this purpose, while loops have to be annotated with loop invariants, and the `while` constructor becomes: `while` $\{form\}$ $form$ `do` $stmt$. Here, the first formula (in braces) is the invariant, the second formula the termination condition. The two functions are defined by primitive recursion over statements, see Fig. 4.

$$
\begin{array}{l}
wp(\texttt{Skip},\ Q) = Q \\
wp(\texttt{delrel}(v_1\ r\ v2),\ Q) = Q[r := r - (v_1, v_2)] \\
wp(\texttt{insrel}(v_1\ r\ v2),\ Q) = Q[r := r + (v_1, v_2)] \\
wp(\texttt{select}\ v\ \texttt{sth}\ b,\ Q) = \forall v.(b \longrightarrow Q) \\
wp(c_1; c_2,\ Q) = wp(c_1, wp(c_2,\ Q)) \\
wp(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ Q) = ite(b, wp(c_1, Q), wp(c_2, Q)) \\
wp(\texttt{while}\{iv\}\ b\ \texttt{do}\ c,\ Q) = iv \\
\\
vc(\texttt{Skip},\ Q) = \top \\
vc(\texttt{delrel}(v_1\ r\ v2),\ Q) = \top \\
vc(\texttt{insrel}(v_1\ r\ v2),\ Q) = \top \\
vc(\texttt{select}\ v\ \texttt{sth}\ b,\ Q) = \top \\
vc(c_1; c_2,\ Q) = vc(c_1, wp(c_2, Q)) \wedge vc(c_2, Q) \\
vc(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\ Q) = vc(c_1, Q) \wedge vc(c_2, Q) \\
vc(\texttt{while}\{iv\}\ b\ \texttt{do}\ c,\ Q) = (iv \wedge \neg b \longrightarrow Q) \wedge (iv \wedge b \longrightarrow wp(c, iv)) \wedge vc(c, iv)
\end{array}
$$

Fig. 4. Weakest preconditions and verification conditions

Without going further into program semantics issues, let us only state the following soundness result that relates the operational semantics and the functions wp and vc:

Theorem 1 (Soundness). *If $vc(c, Q)$ is valid and $(c, \sigma) \Rightarrow \sigma'$, then $\sigma(wp(c, Q))$ implies $\sigma'(Q)$.*

What is more relevant for our purposes is the structure of the formulas generated by wp and vc, because it has an impact on the decision procedure for the DL fragment under consideration here. Besides the notion of "essentially universally quantified" introduced in Sect. 2, we need the notion of *quantifier-free* formula: A formula not containing a quantifier. In extension, we say that a statement is quantifier-free if all of its formulas are quantifier-free.

By induction on c, one shows:

Lemma 1 (Universally quantified). *Let Q be essentially universally quantified and c be a quantifier-free statement. Then $wp(c, Q)$ and $vc(c, Q)$ are essentially universally quantified.*

5 Decision Procedure

5.1 Overview

We present a decision procedure for verifying the validity of essentially universally quantified formulas. As seen in Lemma 1, this is the format of formulas extracted by wp and vc, and as motivated by the soundness result (Theorem 1),

validity of verification conditions is a precondition for ensuring that a program executes according to its specification.

Given an essentially universally quantified formula e, the rough lines of the procedure for determining that e is valid are spelled out in the following.

Getting rid of quantifiers:

1. Convert e to an equivalent prenex normal form p, which will consist of a prefix of universal quantifiers, and a quantifier-free body: $\forall x_1 \ldots x_n.b$
2. p is valid iff its universal closure $ucl(p)$ (universal abstraction over all free variables of p) is.
3. Show the validity of $ucl(p)$ by showing the unsatisfiability of $\neg ucl(p)$.
4. $\neg ucl(p)$ has the form $\neg\forall v_1 \ldots v_k, x_1 \ldots x_n.b$. Pull negation inside the universal quantifier prefix, remove the resulting existential quantifier prefix, and show unsatisfiability of $\neg b$ with the aid of an extended tableau method.

Computation of prenex normal forms is standard. Care has to be taken to avoid capture of free variables, by renaming bound variables. Free variables are defined as usual; the free variables of a substitution $f[r := r - (v_1, v_2)]$ are those of f and in addition v_1 and v_2 (similarly for edge insertion). We illustrate the problem with the following program fragment *prg*:

```
select a sth a : A ;
select b sth b r a ;
select a sth a r b
```

For a given post-condition Q, we obtain

$$wp(prg, Q) = \forall a.a : A \longrightarrow \forall b.(b\ r\ a) \longrightarrow \forall a.(a\ r\ b) \longrightarrow Q$$

whose prenex normal form $\forall a_1, b, a_2.\ (a_1 : A \longrightarrow (b\ r\ a_1) \longrightarrow (a_2\ r\ b) \longrightarrow Q)$ contains more logical variables than *prg* contains program variables.

Extended tableau method – prerequisites: The tableau method takes a quantifier-free formula f and proves its unsatisfiability or displays a model. We aim at reusing existing tableau methods (such as [3]) as much as possible. The difficulty consists in getting rid of the substitution constructor.

Substitution is compatible with the constructors of formulas:

Lemma 2 (Substitution in formulas).
$\bot[r := re] = \bot$
$(\neg f)[r := re] = (\neg f[r := re])$
$(f_1 \wedge f_2)[r := re] = (f_1[r := re] \wedge f_2[r := re])$
$(f_1 \vee f_2)[r := re] = (f_1[r := re] \vee f_2[r := re])$

The case of formulas which are facts, missing in Lemma 2, will be dealt with separately. This is a consequence of substitution not being compatible with concepts, as will be seen in Sect. 5.2: For a given concept C, there is not necessarily

a concept $C' = C[r := re]$. However, substitutions can be eliminated from facts, by the equations given in Sect. 5.2.

We will refer to the equations in Lemma 2 and those in Sect. 5.2 as *substitution elimination rules*. We say that a substitution in a formula is *visible* if one of these rules is applicable; and that it is *hidden* if none of these rules is applicable. For example, the substitution in $(x : (C_1 \sqcap C_2))[r := re]$ is visible; it is hidden in $(x : (C_1[r := re] \sqcap C_2[r := re]))$ and only becomes visible after application of an appropriate tableau rule, for example of the system $\mathcal{ALCQ}$.

To describe our procedure, we introduce the following terminology: An ABox is a finite set of facts (interpreted as the conjunction of its facts), and a tableau a finite set of ABoxes (interpreted as a disjunction of its ABoxes). We need the following functions:

- *push_subst* takes a formula and applies substitution elimination rules as far as possible;
- $form\_to\_tab$ converts to disjunctive normal form and then performs the obvious translation to a tableau;
- *tab_to_form* takes a tableau and constructs the corresponding formula.

Extended tableau method – procedure: Our method is parameterized by the following interface of an implementation of your favorite tableau calculus:

- a transition system $\mathcal{T} \Longrightarrow \mathcal{T}'$, defining a one-step transformation of a tableau $\mathcal{T}$ to a tableau $\mathcal{T}'$.
- a function *sat* which checks, for tableaux $\mathcal{T}$ that are irreducible wrt. $\Longrightarrow$, whether $\mathcal{T}$ is satisfiable.

From this, we construct a restricted relation $\mathcal{T} \Longrightarrow_r \mathcal{T}'$, which is the same as $\Longrightarrow$ provided that $\mathcal{T}$ does not contain visible substitutions:

$$\frac{\mathcal{T} \Longrightarrow \mathcal{T}' \quad \text{no visible subst in } \mathcal{T}}{\mathcal{T} \Longrightarrow_r \mathcal{T}'}$$

We also define a relation $\Longrightarrow^s$ that pushes substitutions until they become hidden:

$$\frac{\mathcal{T}\text{contains visible subst} \qquad \mathcal{T}' = form\_to\_tab(push\_subst(tab\_to\_form(\mathcal{T})))}{\mathcal{T} \Longrightarrow^s \mathcal{T}'}$$

From these, we define the relation $\Longrightarrow_r^s = (\Longrightarrow_r \cup \Longrightarrow^s)$.

The extended tableau algorithm takes a formula f and computes a $\mathcal{T}_f$ such that $form\_to\_tab(f)(\Longrightarrow_r^s)^* \mathcal{T}_f$. The result of the algorithm is $sat(\mathcal{T}_f)$.

The following lemmas show that $\Longrightarrow_r^s$ is a correct and complete algorithm for deciding the decidability of formulas with substitution provided $\Longrightarrow$ is for substitution-free formulas.

Lemma 3 (Termination). *$\Longrightarrow_r^s$ is well-founded provided $\Longrightarrow$ is.*

To show termination of the extended algorithm, define

– the *substitution size* of a *formula* or `fact` as the sum of the term sizes below its substitutions.
– the substitution size of a *tableau* as the multiset of the substitution sizes of its facts.

Note that application of $\Longrightarrow^s$ leads to a reduction of the substitution size. For a well-founded measure m of $\Longrightarrow$, construct a well-founded measure of $\Longrightarrow^s_r$ as the lexicographic order of the substitution size and m.

Lemma 4 (Confluence). *$\Longrightarrow^s_r$ is confluent provided $\Longrightarrow$ is.*

$\Longrightarrow^s_r$ has no other critical pairs than $\Longrightarrow$.

Lemma 5 (Satisfiability). *$\Longrightarrow^s_r$ preserves satisfiability provided $\Longrightarrow$ does.*

The three auxiliary functions used for defining $\Longrightarrow^s$ do.

5.2 Elimination of Substitutions

We now show how substitutions can be pushed into facts. For lack of space, we cannot treat all constructors.

For facts of the form $x : C$, where C is a concept, we have the cases:

– $(x : \neg C)[r := re]$ reduces to $x : (\neg C[r := re])$
– $(x : C_1 \wedge C_2)[r := re]$ reduces to $x : (C_1[r := re] \wedge C_2[r := re])$
– $(x : C_1 \vee C_2)[r := re]$ reduces to $x : (C_1[r := re] \vee C_2[r := re])$
– $(x : (\geq\ n\ r\ C))[r' := re]$, for $r' \neq r$, reduces to $x : (\geq\ n\ r\ C[r' := re])$, and similarly when replacing $\geq$ by $<$
– $(x : (\geq\ n\ r\ C))[r := r - (v_1, v_2)]$ reduces to

$$\begin{array}{l} \mathit{ite}\ ((x = v_1) \wedge (v_2 : (C[r := r - (v_1, v_2)])) \wedge (v_1\ r\ v_2), \\ \quad (x : (\geq\ \ (n+1)\ r\ (C[r := r - (v_1, v_2)]))), \\ \quad (x : (\geq\ \ n\ r\ (C[r := r - (v_1, v_2)])))) \end{array}$$

  and similarly when replacing $\geq$ by $<$
– $(x : (\geq\ (n+1)\ r\ C))[r := r + (v_1, v_2)]$ reduces to

$$\begin{array}{l} \mathit{ite}\ ((x = v_1) \wedge (v_2 : (C[r := r + (v_1, v_2)])) \wedge (v_1\ (\neg r)\ v_2), \\ \quad (x : (\geq\ \ n\ r\ (C[r := r + (v_1, v_2)]))), \\ \quad (x : (\geq\ \ (n+1)\ r\ (C[r := r + (v_1, v_2)])))) \end{array}$$

  and similarly when replacing $\geq$ by $<$
– $(x : (\geq\ 0\ r\ C))[r := r + (v_1, v_2)]$ reduces to $\top$
– $(x : (<\ 0\ r\ C))[r := r + (v_1, v_2)]$ reduces to $\bot$
– Pathological case $(x : C[sbst_1])[sbst_2]$: lift inner substitution to $(x : C)[sbst_1][sbst_2]$, then apply the above.

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.
2. Franz Baader and Ulrike Sattler. Expressive number restrictions in description logics. *Journal of Logic and Computation*, 9(3):319–350, 1999.
3. Franz Baader and Ulrike Sattler. Tableau algorithms for description logics. In Roy Dyckhoff, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1847 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin / Heidelberg, 2000.
4. Philippe Balbiani, Rachid Echahed, and Andreas Herzig. A dynamic logic for termgraph rewriting. In *5th International Conference on Graph Transformations (ICGT)*, volume 6372 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2010.
5. Mohamed Chaabani, Mohamed Mezghiche, and Martin Strecker. Vérification d'une méthode de preuve pour la logique de description $\mathcal{ALC}$. In Yamine Ait-Ameur, editor, *Proc. 10ème Journées Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, pages 149–163, June 2010.
6. Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic, a language theoretic approach*. Cambridge University Press, 2011.
7. Neil Immerman, Alex Rabinovich, Tom Reps, Mooly Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin / Heidelberg, 2004.
8. Hongkai Liu, Carsten Lutz, Maja Milicic, and Frank Wolter. Foundations of instance level updates in expressive description logics. *Artificial Intelligence*, 175(18):2170–2197, 2011.
9. Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001.
10. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002.
11. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.