VIII
A Prototype Theorem Prover

Reasoning is the ability to make inferences, and automated reasoning is concerned with the building of computing systems that automate this process. *Stanford Encyclopedia of Philosophy*

In this chapter we present a prototype implementation of the systems $\mathrm{SC}_{\mathcal{ALC}}$ and $\mathrm{SC}_{\mathcal{ALCQ}}$. We choose to implement the Sequent Calculi because they represent a first step towards a ND implementations.

The prototype theorem prover was implemented in Maude [18]. So in Section VIII.1 we present the Maude System and language and in Section VIII.2 we describe the prototype implementation.

VIII.1 Overview of the Maude System

This section presents a general overview of the main characteristics of the Maude system and language. A complete description of Maude can be found at [18]. We will only present the aspects of Maude used in our implementation. Moreover, we will not present the theory foundations of Maude in the "Rewriting logic" [47] since our implementation uses the Maude system as an interpreter for the Maude language. We did not explored any possible mapping between description logics and rewriting logic.

Maude's basic programming statements are very simple and easy to understand. They are equations and rules, and have in both cases a simple rewriting semantics in which instances of the lefthand side pattern are replaced by corresponding instances of the righthand side.

Maude programs are organized in modules. Maude modules containing only equations are called functional modules. Modules containing rules are called system module. In both cases, besides equations and rules, modules may contain declarations of sorts (types), operators and variables.

A functional module defines one or more functions by means of equations. Equations are used as simplification rules. Replacement of equals by equals is performed only from left to right as simplification rewriting. A function specification should have a final result and should be unique. Finally, Maude equations can be conditional, that is, they are only applied if a certain condition holds.

A Maude module containing rules and possibly equations is called a system module. Rules are also computed by rewriting from left to right, but they are not equations. Instead, they are understood as local transition between states in a possibly concurrent system. For instance, a distributed banking system can be represented as account objects and messages floating in a "soup". That is, in a multi-set or bag of objects and messages. Such objects and messages in the soup can interact locally with each other according to specific rewrite rules. The systems specified by rules can be highly concurrent and nondeterministic. Unlike for equations, there is no assumption that all rewrite sequences will lead to the same outcome. Furthermore, for some systems there may not be any final states: their whole point may be to continuously engage in interactions with their environment as reactive systems. Note that, since the Maude interpreter is sequential, the concurrent behavior is simulated by corresponding interleavings of sequential rewriting steps. Logically, when rewriting logic was used as a logical framework to represent other logics a rule specifies a logical inference rule, and rewriting steps therefore represent inference steps.

Maude has two varieties of types: sorts, which correspond to well-defined data, and kinds, which may contain error elements. Sorts can be structured in subsort hierarchies, with the subsort relation understood semantically as subset inclusion. This allows support for partial functions, in the sense that a function whose application to some arguments has a kind but not a sort should be considered undefined for those arguments. Furthermore, operators can be subsort-overloaded, providing a useful form of subtype polymorphism.

In Maude the user can specify operators. An operator has arguments (each one has a sort) and a result sort. Each operator has its own syntax, which can be prefix, postfix, infix, or a "mixfix" combination. This is done by indicating with underscores the places where the arguments appear in the mixfix syntax. The combination of user-definable syntax with equations and equational attributes for matching leads to a very expressive capability for specifying any user-definable data. This is one of the main reasons that makes Maude a perfect language/system for prototyping.

Rewriting with both equations and rules takes place by matching a lefthand side term against the subject term to be rewritten. The most simpler matching is syntactic matching, in which the lefthand side term is matched as a tree on the (tree representation of the) subject term. Nevertheless, Maude allows also more expressive matching like "equational matching". when we define operators in Maude we can use attributes like `assoc` (associative) and `comm` (commutative) called equational attributes. For instance, if an operator is defined with both of these attributed, terms having this operator as the principal operator (the most external one), are not matching of trees, but as multi-set, that is, modulo associativity and commutativity. In general, a binary operator declared in a Maude can be defined with any combination of the equational attributes: associativity, commutativity, left-, right-, or two-sided identity, and idempotency.

A Maude system module implements a *rewrite theory* that must be *admissible*, which means that rules should be coherent relative to the equations [18]. If a rewrite theory contains both rules and equations, rewriting is performed modulo such equations. Maude strategy to rewriting terms is to first apply the equations to reach a canonical form, and then do a rewriting step with a rule (in a rule-fair manner). This strategy is complete if we assume coherence. Coherence means that we will not miss possible rewrites with rules that could have been performed if we had not insisted on first simplifying the term to its canonical form with the equations. Maude implicitly assumes this coherence property.

VIII.2 A Prototype Theorem Prover

In this section we present our Maude implementation of $\text{SC}_{\mathcal{ALC}}$ and $\text{SC}^{\square}_{\mathcal{ALC}}$ sequent calculi. We will omit trivial details of the implementation and focus on the important parts. Moreover, it is important to note that this prototype is available for download at `http://github.com/arademaker/SALC` and also includes the implementation of $\text{SC}_{\mathcal{ALCQI}}$ system an its counterpart $\text{SC}^{\square}_{\mathcal{ALCQI}}$. Those implementations are not described here since they do not differ considerably from the presented.

(a) The Logical Language

Due to the flexibility to specify user-definable data in Maude, the definition of the description logics $\mathcal{ALC}$ and $\mathcal{ALCQI}$ syntax was effortless.

The language $\mathcal{ALC}$ is defined in the function module `SYNTAX` below. We have defined sorts for atomic concepts and atomic roles besides the sort for concepts and roles in general. The constants $\top$ and $\bot$ were also specified.

```
fmod SYNTAX is
 inc NAT .

 sorts AConcept Concept ARole Role .
 subsort AConcept < Concept .
 subsort ARole < Role .

 ops ALL EXIST : Role Concept -> Concept .
 ops CTRUE CFALSE : -> AConcept .
 op _&_ : Concept Concept -> Concept [ctor gather (e E) prec 31] .
 op _|_ : Concept Concept -> Concept [ctor gather (e E) prec 32] .
 op ~_  : Concept -> Concept [ctor prec 30] .

 eq ~ CTRUE = CFALSE .
 eq ~ CFALSE = CTRUE .
endfm
```

The syntax for defining operators is:

```
op NAME : Sort-1 Sort-2 ... -> Sort [attr-1 ...] .
```

where `NAME` may contain underscores to identify arguments position in infix notation. The list of sorts before `->` is the arguments and the sort after is the sort of the resultant term.

Since our SC$_{\mathcal{ALC}}$ and SC$_{\mathcal{ALCQI}}$ systems reason over labeled concepts. The next step was to extend the language with labels and some functions over them. A labeled concept ${}^{\forall R,\exists S}\alpha$ is represented by the term `< al(R) ex(S) | A >` where `A` is a constant of the sort `AConcept` and `R` and `S` constants of the sort `ARole`. In the modules below, we show the declarations of all operators but omitted the specification of logical operators `has-quant`, `has-lt` and so on.

```
fmod LABEL is
 inc SYNTAX .

 sorts Label ELabel ALabel QLabel .
 subsorts ELabel ALabel QLabel < Label .

 ops gt lt : Nat Role -> QLabel .
 op ex : Role -> ELabel .
 op al : Role -> ALabel .
endfm
```

The definition below of the operators `neg` and `neg-aux` should be clear but being the first equational specification deserves an explanation. The operator `neg(L)` operates over the list of labels `L` inverting all its quantifiers. In Section III.1, we represent such operation as $\neg L$. We use `neg-aux` to interact over the list accumulating the result in its second argument until the first argument is completely consumed and the second argument returned.

```
fmod LALC-SYNTAX is
 inc LABEL .
 inc LIST{Label} .

 vars L1 L2 : List{Label} .
 vars R : Role .
 var C : Concept .

 sorts Expression LConcept .
 subsort LConcept < Expression .

 op <_|_> : List{Label} Concept -> LConcept [ctor] .

 ops has-quant has-lt has-gt : List{Label} -> Bool .
 ops has-al has-ex : List{Label} -> Bool .
 op neg : List{Label} -> List{Label} .
 op neg-aux : List{Label} List{Label} -> List{Label} .
 ...
 eq neg(L1) = neg-aux(L1, nil) .

 eq neg-aux(L1 al(R), L2) = neg-aux(L1, ex(R) L2) .
 eq neg-aux(L1 ex(R), L2) = neg-aux(L1, al(R) L2) .
 eq neg-aux(nil, L2) = L2 .
endfm
```

It is worth to note that this is not the only way to define `neg` in Maude, the auxiliary function is not necessary at all, but we will use them frequently in our implementation.

Finally, the module `LALC-SYNTAX` declares the sorts `Expression` and `LConcept` (labeled concept). Expressions are labeled concepts but the distinction can be useful for future extensions of the calculi.

(b) The Sequent Calculus

In the function module `SEQUENT-CALCULUS` we implemented the generic data structures that are used by all sequent calculi. The idea is that a proof will be represented as a multi-set ("soup") of goals and messages (operators with sort `State`). Goals are sequents with additional properties to keep the proof structure. Each goal will have an identifier (natural number), the goal origin, the name of the rule used to produce that goal, and the sequent. In this way, our proof is a graph represented as a multi-set of terms with sort `Proof`. The `goals` operator holds a list of natural numbers as its argument, the list of pending goals. The `next` operator is just an auxiliary operator that provides in each proof step the next goal identifier.

```
fmod SEQUENT-CALCULUS is
 inc LALC-SYNTAX .
 inc SET{Expression} .
 inc SET{Label} .
 ...
 sorts Sequent Goal State Proof .
 subsort Goal State < Proof .

 op next  : Nat -> State .
 op goals : Set{Nat} -> State .

 op [_from_by_is_] : Nat Nat Qid Sequent -> Goal [ctor] .

 op nil : -> Proof [ctor] .
 op __ : Proof Proof -> Proof [ctor comm assoc] .

 op _|-_ : Set{Expression} Set{Expression} ->
          Sequent [ctor prec 122 gather(e e)] .

 op _:_|-_:_ : Set{Expression} Set{Expression} Set{Expression}
              Set{Expression} -> Sequent [ctor prec 122 gather(e e e e)] .
 ...
endfm
```

We must also note that we have defined two operators [1] to construct sequents. The operator `_|-_` is the simplest sequent with two multi-set of expression, one on the left (sequent antecedent, possibly empty) and other on the right (sequent succedent, possibly empty), it is used to implement $\mathrm{SC}_{\mathcal{ALC}}$. The operator `_:_|-_:_` is used by the frozen versions of $\mathrm{SC}_{\mathcal{ALC}}$ and $\mathrm{SC}_{\mathcal{ALCQI}}$. The two additional external sets of expressions hold the frozen formulas.

[1]Term constructor in Maude terminology since these operators will never be reduced, they are used to hold data.

Consider the proof of the sequent $\forall R.(A \sqcap B) \Rightarrow \forall R.A \sqcap \forall R.B$ presented in Figure VIII.1. One proof constructed by our system is represented by the term below. The goal 0 is the initial state of the proof, goals 6 and 5 are the initial sequents. Goal 1 is obtained from goal 0 applying the rule ∀-l. The empty argument of `goals(empty)` represent the fact that this proof is complete, there is no remaining goals to be proved.

```
goals(empty) next(7)
[0 from 0 by 'init is < nil | ALL(R, A & B) > |-
                     < nil | ALL(R, A) & ALL(R, B) >]

[1 from 0 by 'forall-l is < al(R) | A & B > |-
                         < nil | ALL(R, A) & ALL(R, B) >]

[2 from 1 by 'and-l is < al(R) | A >, < al(R) | B > |-
                      < nil | ALL(R, A) & ALL(R, B) >]

[3 from 2 by 'and-r is < al(R) | A >, < al(R) | B > |- < nil | ALL(R, A) >]
[4 from 2 by 'and-r is < al(R) | A >, < al(R) | B > |- < nil | ALL(R, B) >]
[5 from 3 by 'forall-r is < al(R) | A >, < al(R) | B > |- < al(R) | A >]
[6 from 4 by 'forall-r is < al(R) | A >, < al(R) | B > |- < al(R) | B >]
```

Figure VIII.1: An example of a proof in the implementation of $\mathrm{SC}_{\mathcal{ALC}}$

VIII.3 The $\mathrm{SC}_{\mathcal{ALC}}$ System

The $\mathrm{SC}_{\mathcal{ALC}}$ system was implemented in a system module. Basically, each rule of the system is a Maude rewriting rule. The rewriting procedure construct the proof bottom-up.

```
mod SYSTEM is
 inc SEQUENT-CALCULUS .

 [rules and equations presented below]

endm
```

The first observation regards the structural rules of $\mathrm{SC}_{\mathcal{ALC}}$. Since the left and right sides of the sequents are sets of formulas, we do not need permutation of contraction rules. We also proved in Section III.4 that the cut rule was not necessary too. Nevertheless, we could lose completeness if we have omitted the weak rules. We need them to allow the promotional rules applications. Moreover, the initial sequent were implemented as an equation rather than as a rule. We used the fact that in Maude all rewriting steps with rules are executed module equational reductions. The implementation of the initial sequents using equations means that a goal detected as initial will be removed from the goals lists right aways.

```
 eq [ X from Y by Q is ALFA, E |- E, GAMMA ] goals((X, XS)) =
    [ X from Y by Q is ALFA, E |- E, GAMMA ] goals((XS))
  [label initial] .

 rl [weak-l] :
  [ X from Y by Q is ALFA, E |- GAMMA ] next(N) goals((X, XS))
  =>
  [ X from Y by Q is ALFA, E |- GAMMA ] next(N + 1) goals((XS, N))
  [ N from X by 'weak-l is ALFA |- GAMMA ] .
```

First we note the difference between rules and equations. They are very similar expected that the former uses `=>` and the later `=` as a term separator.

```
rl [label] : term-1 => term-2 [attr-1,...] .
eq term-1 = term-2 [attr-1,...] .
```

We note that on each rule the goal being rewritten must be repeated in the left and right side of the rule. See weak rule above. If we omit the goal on the right side of the rule we would be removing the goal from the proof. We are actually including new goals on each step, that is, we put new goals in the "soup" of goals.

Reading bottom-up, some rules create more than one (sub)-goal from a goal. This is the case of rule ⊓-r below. Besides that, whenever a rule has some additional proviso, we use Maude *conditional rules* to express the rule proviso in the rule condition. In the rule ⊓-r, the proviso states that in the list of labels of the principal formula all labels must be universal quantified, in SC$_{\mathcal{ALC}}$, this is the same of saying that L cannot contain existential quantified labels (`has-ex(L)`).

```
crl [and-r] :
 [ X from Y by Q is ALFA |- GAMMA, < L | A & B > ]
 next(N) goals((X, XS))
 =>
 next(N + 2) goals((XS, N, N + 1))
 [ X from Y by Q is ALFA |- GAMMA, < L | A & B > ]
 [ N     from X by 'and-r is ALFA |- GAMMA, < L | A > ]
 [ N + 1 from X by 'and-r is ALFA |- GAMMA, < L | B > ]
if not has-ex(L) .
```

The rule condition can consist of a single statement or can be a conjunction formed with the associative connective `/\`. Rule promotional-∃ has two conditions. The first, from left to right, is the rule proviso (all concepts on the left-side of the sequent must have the same most external label), the second is actually just an instantiation of the variable `GAMMA'` with the auxiliary operator `remove-label`. `GAMMA'` will be the right-side of the new sequent (goal) created. `remove-label` iterate over the concepts removing the most external label of them.

```
crl [prom-exist] :
 [ X from Y by Q is < ex(R) L | A > |- GAMMA ]
 next(N) goals((X, XS))
 =>
 next(N + 1) goals((XS, N))
 [ X from Y by Q is < ex(R) L | A > |- GAMMA ]
 [ N from X by 'prom-exist is  < L | A > |- GAMMA' ]
if all-label(GAMMA, ex(R)) = true
 /\ GAMMA' := remove-label(GAMMA, ex(R), empty) .
```

The implementation of the remain rules is straightforward. We have one observation more about the rules above, the argument of `next(N)` gives the next goal identifier. The argument of `goals` holds the list of goals not solved. A derivation with `goals(empty)` in the "soup" is a completed proof of the sequent in the goal with identifier 0.

(a) The SC$\square_{\mathcal{ALC}}$ System Implementation

The system SC$\square_{\mathcal{ALC}}$ is implemented in a very similar way of SC$_{\mathcal{ALC}}$. The main differences are that sequents now have frozen concepts and two additional rules had to be implemented. Concepts that were frozen together will never be unfrozen separated, so that, instead of defining an operator to freeze a concept, we defined a constructor of a set of frozen concepts.

```
mod SYSTEM is
 inc SEQUENT-CALCULUS .
 ...
 op [_,_,_] : Nat Nat Set{Expression} -> Expression .
```

The constructor of frozen set of concepts has three arguments. The first argument is the context identifier (see Section IV.2) created to group the pair of sets of concepts frozen together on the sequent antecedent and succedent. The second argument is the state of the context where 0 means that the context is saved but not reduced yet (context was frozen by weak rule), and 1 means that the context was reduced (context was frozen by frozen-exchange rule). The last argument is the set of frozen concepts.

Almost all rules of SC$\square_{\mathcal{ALC}}$ do not touch in the frozen concepts. This is the case of negation rule below. We note the use of the operator `neg` inverting the list of labels of a concept.

```
rl [neg-l] :
  [ X from Y by Q is FALFA : ALFA, < L | ~ A > |- GAMMA : FGAMMA ]
  next(N) goals((X, XS))
  =>
  next(N + 1) goals((XS, N))
  [ X from Y by Q is FALFA : ALFA, < L | ~ A > |- GAMMA : FGAMMA ]
  [ N from X by 'neg-l is FALFA : ALFA |- GAMMA, < neg(L) | A > : FGAMMA ] .
```

The weak-r rule was implemented as a conditional rewrite rule below. The left and right-side of the sequent in goal `X` were frozen and added to the set of frozen concepts on the left and right side of the sequent in the new goal `N`. The variables `FALFA` and `FGAMMA` match the set of frozen concepts on both sides. The weak-l rule is similar.

```
crl [weak-r] :
  [ X from Y by Q is FALFA : ALFA |- GAMMA, E : FGAMMA ]
  next(N) goals((X, XS))
  =>
  next(N + 1) goals((XS, N))
  [ X from Y by Q is FALFA : ALFA |- GAMMA, E : FGAMMA ]
  [ N from X by 'weak-l is (FALFA, [M:Nat, 0, ALFA]) : ALFA |-
                           GAMMA : (FGAMMA, [M:Nat, 0, (GAMMA, E)]) ]
 if M:Nat  := next-frozen(union(FALFA, FGAMMA)) .
```

The other SC$\square_{\mathcal{ALC}}$ rule that modify the set of frozen concepts in a goal is the frozen-exchange rule. The Maude pattern matching mechanism was very useful in the implementation of this rule. The rule select randomly [2] a

[2]The selection is made by pattern matching of a context module commutative and associative, thanks to the attributes of the operator comma, the constructor of `Set{Expression}` terms.

context (sets of frozen concepts) to unfreeze – `[O:Nat, 0, ES1]` and `[O:Nat, 0, ES2]` – and freeze the set of formulas that are in the current context – `ALFA` and `GAMMA`. The pattern also guarantee that only contexts saved but not already reduced (second argument equals zero) will be selected. The new context created in the goal `N` has the second argument equals one – it is a reduced context. Maude's pattern matching mechanism is very flexible and powerful. On the other hand, this rule does not provide much control over the choice of contexts (set of frozen formulas) that will be unfreeze. This choice can have huge impact in the performance of a proof construction.

```
crl [frozen-exchange] :
 [ X from Y by Q is [O:Nat,0,ES1], FALFA : ALFA |-
                                     GAMMA : FGAMMA, [O:Nat,0,ES2] ]
 goals((X, XS)) next(N)
 =>
 goals((XS, N)) next(N + 1)
 [ X from Y by Q is [O:Nat,0,ES1], FALFA : ALFA |-
                                     GAMMA : FGAMMA, [O:Nat,0,ES2] ]
 [ N from X by 'frozen-exchange is
   ([M:Nat,1,ALFA], FALFA) : ES1 |- ES2 : (FGAMMA, [M:Nat,1,GAMMA]) ]

if M:Nat := next-frozen(union(([O:Nat,0,ES1], FALFA),
                               ([O:Nat,0,ES2], FGAMMA))) .
```

(b) The Interface

The current user interface of the prototype is the Maude prompt. We do not provide any high level user interface yet, although different alternatives exist for it. For example, we could implement the DIG [2] interface using Maude external objects [18]. The system module `THEOREM-PROVER` is the main interface with the prototype. It basically declares some constants of the sort `AConcept` (atomic concepts) and `ARole` (atomic roles) and the operator `th_end`. This operator is a "syntax sugar" to assist the user in the creation of the proof term in its initial state ready to be rewritten.

```
mod THEOREM-PROVER is
 inc SYSTEM .

 ops A B C D E : -> AConcept .
 ops R S T U V : -> ARole .
 op th_end : Sequent -> Goal .

 vars ALFA GAMMA : Set{Expression} .
 var SEQ : Sequent .

 eq th SEQ end =
    [ 0 from 0 by 'init is SEQ ] next(1) goals(0) .
endm
```

The module THEOREM-PROVER includes the module SYSTEM, where SYSTEM can be any of the implemented systems presented in the previous sections.

With the help of the above module we can prove the theorem from Example 1 (1) using two alternatives.

$$\exists child.\top \sqcap \forall child.\neg(\exists child.\neg Doctor) \sqsubseteq \exists child.\forall child.Doctor \qquad (1)$$

We can use the already declared constants assuming $A = Doctor$ and the role $R = child$ or we can declare two new constants in a module that imports THEOREM-PROVER.

```
mod MY-TP is
 inc THEOREM-PROVER .

 op child : -> ARole .
 op Doctor : -> AConcept .
endm
```

In the second case, after entering the module MY-TP in Maude, we could test the proof initialization with the Maude command reduce (red). The command rewrite the given term using only equations. In that case, only the equation of the operator th_end from module THEOREM-PROVER is applied.

```
Maude> red th < nil | EXIST(child, CTRUE) &
                      ALL(child, ~ EXIST(child, ~ Doctor)) > |-
              < nil | EXIST(child, ALL(child, Doctor)) > end .

result Proof: next(1) goals(0)
[0 from 0 by 'init is
   < nil | EXIST(child, CTRUE) & ALL(child, ~ EXIST(child, ~ Doctor)) >
   |-
   < nil | EXIST(child, ALL(child, Doctor)) > ]
```

To construct a proof of a given sequent, we can use Maude `rewrite` or `search` command. The former will return one possible sequence of rewriting steps until a *canonical term* [3] is reached. The latter will search for all possible paths of rewriting steps from the given initial state until the final given state.

Below we present the same sequent with *Doctor* and *child* replaced by A and R respectively. As we can see, due the presence of weak rules and the lack of a strategy to control the applications of the rules, we failed to obtain a proof for a valid sequent using the command `rewrite`.

```
Maude> rew th < nil | EXIST(R, CTRUE) & ALL(R, ~ EXIST(R, ~ A)) > |-
              < nil | EXIST(R, ALL(R, A)) > end .

result Proof: next(3) goals(2)
[0 from 0 by 'init is
     < nil | EXIST(R, CTRUE) & ALL(R, ~ EXIST(R, ~ A)) > |-
     < nil | EXIST(R, ALL(R, A)) >]
[1 from 0 by 'weak-l is empty |- < nil | EXIST(R, ALL(R, A)) >]
[2 from 1 by 'weak-r is empty |- empty]
```

The `rewrite` command explores just one possible sequence of rewrites of a system described by a set of rewrite rules and an initial state. The search command allows one to explore (following a breadth-first strategy) the reachable state space in different ways.

Using the `search` command we can ask for all possible proof trees that can be constructed for a given sequent. Moreover, we can limit the space search with the two optional parameters `[n,m]` where n providing a bound on the number of desired solutions and m stating the maximum depth of the search. The search arrow `=>!` indicates that only canonical final states are allowed, that is, states that cannot be further rewritten. On the left-hand side of the search arrow we have the starting term, on the right-hand side the pattern that has to be reached, in the case below, `P:Proof goals(empty)`.

```
Maude> search [1,20]
         th < nil | EXIST(R, CTRUE) & ALL(R, ~ EXIST(R, ~ A)) >
            |- < nil | EXIST(R, ALL(R, A)) > end
         =>! P:Proof goals(empty) .

P:Proof --> next(10)
[0 from 0 by 'init is
```

[3] A term that cannot be further rewritten.

```
 < nil | EXIST(R, CTRUE) & ALL(R, ~ EXIST(R, ~ A)) > |-
 < nil | EXIST(R, ALL(R, A)) >]
[1 from 0 by 'and-l is < nil | ALL(R, ~ EXIST(R, ~ A)) >,
  < nil | EXIST(R, CTRUE) > |- < nil | EXIST(R, ALL(R, A)) >]
[2 from 1 by 'forall-l is < nil | EXIST(R, CTRUE) >,
  < al(R) | ~ EXIST(R, ~ A) > |- < nil | EXIST(R, ALL(R, A)) >]
[3 from 2 by 'neg-l is < nil | EXIST(R, CTRUE) > |-
  < nil | EXIST(R, ALL(R, A)) >, < ex(R) | EXIST(R, ~ A) >]
[4 from 3 by 'exist-r is < nil | EXIST(R, CTRUE) > |-
  < ex(R) | ALL(R, A) >, < ex(R) | EXIST(R, ~ A) >]
[5 from 4 by 'forall-r is < nil | EXIST(R, CTRUE) > |-
  < ex(R) | EXIST(R, ~ A) >, < ex(R) al(R) | A >]
[6 from 5 by 'exist-r is < nil | EXIST(R, CTRUE) > |-
  < ex(R) ex(R) | ~ A >, < ex(R) al(R) | A >]
[7 from 6 by 'exist-l is < ex(R) | CTRUE > |- < ex(R) ex(R) | ~ A >,
  < ex(R) al(R) | A >]
[8 from 7 by 'prom-exist is < nil | CTRUE > |-
  < ex(R) | ~ A >, < al(R) | A >]
[9 from 8 by 'neg-r is < nil | CTRUE >, < al(R) | A > |- < al(R) | A >]
```

Above, the variable P in the input pattern was bound in the result to the desired proof term, that is, the one with `goals(empty)`. Since P was the only variable in the pattern, the result shows only one binding. In other worlds, search results are bindings for variables in the pattern given after the search arrow.

Distributed with our prototype there is a simple Maude-2-LaTeX proof terms translator developed by Caio Mello. [4] The translator receives as input a term like the one above and return its representation in LaTeX using the LaTeX package `bussproof` [12]. The output in LaTeX is:

$$
\begin{array}{cl}
\top, {}^{\forall R}A \Rightarrow {}^{\forall R}A & \\
\hline
\top \Rightarrow {}^{\exists R}\neg A, {}^{\forall R}A & \neg\text{-r} \\
\hline
{}^{\exists R}\top \Rightarrow {}^{\exists R,\exists R}\neg A, {}^{\exists R,\forall R}A & \text{prom-}\exists \\
\hline
\exists R.\top \Rightarrow {}^{\exists R,\exists R}\neg A, {}^{\exists R,\forall R}A & \exists\text{-l} \\
\hline
\exists R.\top \Rightarrow {}^{\exists R}\exists R.(\neg A), {}^{\exists R,\forall R}A & \exists\text{-r} \\
\hline
\exists R.\top \Rightarrow {}^{\exists R}\exists R.(\neg A), {}^{\exists R}\forall R.A & \forall\text{-r} \\
\hline
\exists R.\top \Rightarrow {}^{\exists R}\exists R.(\neg A), \exists R.\forall R.A & \exists\text{-r} \\
\hline
\exists R.\top, {}^{\forall R}\neg\exists R.(\neg A) \Rightarrow \exists R.\forall R.A & \neg\text{-l} \\
\hline
\exists R.\top, \forall R.(\neg\exists R.(\neg A)) \Rightarrow \exists R.\forall R.A & \forall\text{-l} \\
\hline
\exists R.\top \sqcap \forall R.(\neg\exists R.(\neg A)) \Rightarrow \exists R.\forall R.A & \sqcap\text{-l}
\end{array}
$$

[4]An undergraduate student working at TecMF/PUC-Rio Lab.

(c) Defining Proof Strategies

An automated theorem prover would not be efficient or even useful if we cannot provide strategies for deduction rules applications. Moreover, from Section IV.2 we know that SC$^{\square}_{\mathcal{ALC}}$ deduction rules were designed to be used in a very specific strategy. Maude support two ways to define strategies for rewriting rules application. The first option is the original one, we can use Maude reflection feature to control of rules applications at the metalevel developing a full user-definable internal strategies. The second options is to use the Maude Strategy Language [25].

The strategy language allows the definition of strategy expressions that control the way a term is rewritten. The strategy language was designed to be used at the object level, rather than at the metalevel. There exist a strict separation between the rewrite rules in system modules and the strategy expressions, that are specified in separate strategy modules. Moreover, a strategy is described as an operation that, when applied to a given term, produces a set of terms as a result, given that the process is nondeterministic in general. In the current version of Maude, not all features of the strategy language are available in *Core* Maude. To be more precise, the *Core* Maude does not support recursive strategies. Recursion is achieved by giving a name to a strategy expression and using this name in the strategy expression itself or in other related strategies. Given that limitation, we use the prototype strategy language implementation in *Full* Maude [18].

In our current prototype version we defined the strategy described in Section IV.2 to control SC$^{\square}_{\mathcal{ALC}}$ rules applications. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term. Strategies operators allow the construction of complex strategy expressions.

The strategy `expand` presented below controls how the rules of SC$^{\square}_{\mathcal{ALC}}$ ought to be applied. It can be interpreted as: the system must first try to reduce the given term using one of the promotional rules (the union operator is `|`). If it is successful, the system must try to further transform the result term using ∀-{l,r}, ∃-{l,r}, ⊔-{l,r}, ⊓-{l,r} or ¬-{l,r} (the operator `;` the a concatenation). If neither the promotional rules nor the previous mentioned rules could be applied, one of the weak rules should be tried. If none of the previous rules could by applied, the frozen-exchange rule must be tried.

```
(smod BACKTRACKING-STRAT is

 strat solve : @ Proof .
 strat expand : @ Proof .
 var P : Proof .

 sd expand := (((try(prom-exist | prom-all) ;
               (and-l | and-r | or-l | or-r | forall-l | forall-r |
                exist-l | exist-r | neg-l | neg-r))
               orelse (weak-l | weak-r))
              orelse frozen-exchange) .

 sd solve := if (match P s.t. (is-solution(P))) then
     idle
   else
     expand ; if (match P s.t. (is-ok(P))) then solve else idle fi
   fi .
endsm)
```

The strategy `expand` defines how each proof step will be performed. The `solve` strategy is the complete strategy to construct a proof. It is basically a backtracking procedure, on each step, the system verifies if it has already a solution – using the defined operator `is-solution`. If the term is not a solution, it executes the `expand` step and check if the result term is a valid term, that is, a term still useful to reach to a solution – this is done with the operator `is-ok`. If the term is still valid but not yet a solution it continues recursively.

The implementations of `is-solution` and `is-ok` were done in a separated module. The operator `is-ok` evaluates to false whenever we detected a loop in the proof construction. There are differents loop situations, below we present one of them, when we have a sequent with two equal sets of frozen formulas (contexts).

```
  op is-ok : Proof -> Bool .
  op is-solution : Proof -> Bool .

  eq is-solution(P:Proof goals(empty)) = true .
  eq is-solution(P:Proof) = false [owise] .
  ...
  eq is-ok(P:Proof
          [M from N by RL is FALFA1, [X1, X3, FALFA0],
             [X2, X4, FALFA0] : ALFA |- GAMMA :
             [X1, X3, FGAMMA0], [X2, X4, FGAMMA0], FGAMMA1])
```

```
     = false .

  eq is-ok(P:Proof) = true [owise] .
```

Using the `solve` strategy defined above, we can prove the subsumption from Equation 1 in SC$\square_{\mathcal{ALC}}$. We use the strategy aware command `srew` instead of the `rew`. In additional, since we are not using Full Maude, the command in Maude prompt is inside parentheses.

```
Maude> (srew th empty : < nil | EXIST(R, CTRUE) &
                                ALL(R, ~ EXIST(R, ~ A)) > |-
               < nil | EXIST(R, ALL(R, A)) > : empty end using solve .)

result Proof :
 goals(empty)next(10)
 [0 from 0 by 'init is empty : < nil | EXIST(R,CTRUE) &
                                       ALL(R,~ EXIST(R,~ A))> |-
    < nil | EXIST(R,ALL(R,A))> : empty]
 [1 from 0 by 'and-l is empty : < nil | ALL(R,~ EXIST(R,~ A))>,
    < nil | EXIST(R,CTRUE)> |- < nil | EXIST(R,ALL(R,A))> : empty]
 [2 from 1 by 'forall-l is empty : < nil | EXIST(R,CTRUE)>,
    < al(R)| ~ EXIST(R,~ A)> |- < nil | EXIST(R,ALL(R,A))> : empty]
 [3 from 2 by 'exist-l is empty : < al(R)| ~ EXIST(R,~ A)>,
    < ex(R)| CTRUE > |- < nil | EXIST(R,ALL(R,A))> : empty]
 [4 from 3 by 'exist-r is empty : < al(R)| ~ EXIST(R,~ A)>,
    < ex(R)| CTRUE > |- < ex(R)| ALL(R,A)> : empty]
 [5 from 4 by 'forall-r is empty : < al(R)| ~ EXIST(R,~ A)>,
    < ex(R)| CTRUE > |- < ex(R)al(R)| A > : empty]
 [6 from 5 by 'neg-l is empty : < ex(R)| CTRUE > |- < ex(R)| EXIST(R,~ A)>,
    < ex(R)al(R)| A > : empty]
 [7 from 6 by 'prom-exist is empty : < nil | CTRUE > |- < nil | EXIST(R,~ A)>,
    < al(R)| A > : empty]
 [8 from 7 by 'exist-r is empty : < nil | CTRUE > |- < al(R)| A >,
    < ex(R)| ~ A > : empty]
 [9 from 8 by 'neg-r is empty : < nil | CTRUE >, < al(R)| A > |-
    < al(R)| A > : empty]
```