Verifying an Applicative ATP Using Multiset Relations*

Francisco J. Martín-Mateos, Jose A. Alonso, María J. Hidalgo, and José L. Ruiz-Reina

Departamento de Ciencias de la Computación e Inteligencia Artificial.
Facultad de Informática y Estadística, Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
{fjesus,jalonso,mjoseh,jruiz}@cica.es

Abstract. We present in this paper a formalization of multiset relations in the ACL2 theorem prover [6], and we show how multisets can be used to mechanically prove non-trivial termination properties. Every relation on a set A induces a relation on finite multisets over A; it can be shown that the multiset relation induced by a well-founded relation is also well-founded [3]. We have carried out a mechanical proof of this property in the ACL2 logic. This allows us to provide well-founded multiset relations in order to prove termination of recursive functions. Once termination is proved, the function definition is admitted as an axiom in the logic and formal mechanized reasoning about it is possible. As a major application of this tool, we show how multisets can be used to prove termination of a tableaux based theorem prover for propositional logic.

Introduction

We present in this paper a formalization of multiset relations in the ACL2 system [6], and we show how these relations can be used to prove non-trivial termination properties, providing a tool for defining relations on finite multisets and showing that, under certain conditions, these relations are well-founded. Such well-founded relations allows the user to provide a particular multiset measure in order to prove termination of a recursively defined function. Termination proofs are required by ACL2 to admit function definitions as axioms in the logic, as a mean to avoid inconsistencies. Once a function definition is admitted, formal mechanized reasoning about it is possible. We illustrate the use of this tool, presenting the termination proof of a Common Lisp definition of a tableaux based theorem prover for propositional logic. This allows us to verify soundness and completeness of this prover.

ACL2 is a programming language, an applicative subset of Common Lisp. ACL2 is also a logic designed to reason about the programs defined in the language. And, finally, ACL2 is a mechanical theorem proving system, supporting

* This work has been supported by DGES/MEC: Projects TIC2000-1368-C03-02 and PB96-1345

formal reasoning in the logic. The system evolved from the Boyer-Moore theorem prover, also known as Nqthm. For an introduction to ACL2, see the tutorials in the ACL2 web page [6]. To obtain more background on ACL2, see [5].

The ACL2 logic is a quantifier-free, first-order logic with equality, describing an applicative subset of Common Lisp. The syntax of terms is that of Common Lisp (we will assume the reader familiar with this language). The logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference include those for propositional calculus, equality, instantiation and induction. By the *principle of definition*, new function definitions (using `defun`) are admitted as axioms only if there exists a measure function taking values on a well-founded set, in which the arguments of each recursive call decrease, ensuring in this way that no inconsistencies are introduced by new definitions. The primitive well-founded set in the logic is the ordinal ε_0. The theory has a constructive definition of the ordinals up to ε_0, in terms of lists and natural numbers, given by the predicate `e0-ordinalp` and the order `e0-ord-<`. For every function definition introduced by the user, ACL2 starts a proof attempt of its termination. In some non trivial cases, the system is not able to prove it by its own and needs help from the user. Thus, it allows the user to provide a particular measure and a well-founded relation.

Multisets provide a powerful way to prove termination in some of these non trivial cases. Multisets are usually defined in an informal way as "sets with repeated elements". Dershowitz and Manna [3] proved that every well-founded relation on a set A induces a well-founded relation on the set of finite multisets of elements taken from A. In the first section of this paper, we present how we have formalized and proved this theorem using ACL2, and stated it in an abstract way. This allows to instantiate the theorem to show well-foundedness of concrete multiset relations. We have also developed a macro `defmul` in order to easily make definitions of induced multiset relations. Besides defining the multiset relation induced by a given relation, this macro performs a mechanical proof, by functional instantiation, of well-foundedness of the defined multiset relation, provided that the given relation is well-founded.

We illustrate our multiset tool, showing how it is used as part of the verification process of a Common Lisp definition of a tableaux based theorem prover for propositional logic. This prover is defined in the second section. In the third section we show that the use of a well founded multiset relation is specially well suited in the termination proof of that definition, and how our `defmul` tool can assist in the automation of the proof. Once termination is proved, one can use the ACL2 logic to reason about the prover and mechanically prove its soundness and completeness. This case study is part of our current work on formalizing properties of deduction systems using ACL2.

Due to the lack of space we will skip details of the mechanical proofs. The complete files with definitions and theorems are available on the web in `http://www-cs.us.es/~fmartin/acl2-tab-prop/`.

1 Formalization of Multiset Relations in ACL2

A *multiset* M over a set A is a function from A to the set of natural numbers. This is a formal way to define "sets with repeated elements". Intuitively, $M(x)$ is the number of copies of $x \in A$ in M. This multiset is *finite* if there are finitely many x such that $M(x) > 0$. The set of all finite multisets over A is denoted as $\mathcal{M}(A)$.

Basic operations on multisets are defined to generalize the same operations on sets, taking into account multiple occurrences of elements: $x \in M$ means $M(x) > 0$, $M \subseteq N$ means $M(x) \leq N(x)$, for all $x \in A$, $M \cup N$ is the function $M + N$ and $M \setminus N$ is the function $M \dot{-} N$ (where $x \dot{-} y$ is $x - y$ if $x \geq y$ and 0 otherwise).

Any ordering defined on a set A induces an ordering on multisets over A: given a multiset, a smaller multiset can be obtained by removing a non-empty subset X and adding elements which are smaller than some element in X. This construction can be generalized to binary relations in general, not only for partial orderings. This is the formal definition:

Definition 1. *Given a relation $<$ on a set A, the* **multiset relation** *induced by $<$ on $\mathcal{M}(A)$, denoted as $<_{mul}$, is defined as $N <_{mul} M$ iff there exist $X, Y \in \mathcal{M}(A)$ such that $\emptyset \neq X \subseteq M$, $N = (M \setminus X) \cup Y$ and $\forall y \in Y\ \exists x \in X,\ y < x$. It can be easily shown that if $<$ is a strict ordering, then so is $<_{mul}$. In such case we talk about* **multiset orderings**.

A relation $<$ on a set A is *terminating* if there is no infinite decreasing[1] sequence $x_0 > x_1 > x_2 \ldots$. An important property of multiset relations on finite multisets is that they are terminating when the original relation is terminating, as stated by the following theorem:

Theorem 1. *(Dershowitz and Manna, [3]). Let $<$ be a terminating relation on a set A, and $<_{mul}$ the multiset relation induced by $<$ on $\mathcal{M}(A)$. Then $<_{mul}$ is terminating.*

The above theorem provides a tool for showing termination of recursive function definitions, by using multisets: show that some multiset measure decreases in each recursive call, comparing multisets with respect to the relation induced by a given terminating relation. In the following subsection, we explain how we formalized theorem 1 in the ACL2 logic.

1.1 Formalization of Well-Founded Multiset Relations in ACL2

Let us deal with formalization of terminating relations in ACL2. A restricted notion of terminating relations is built into ACL2 based on the following meta-theorem: a relation $<$ on a set A is terminating iff there exists a function F :

[1] Although not explicitly, we will suppose that the relations given here represent some kind of "smaller than" relation.

$A \to Ord$ such that $x < y \Rightarrow F(x) < F(y)$, where *Ord* is the class of all ordinals. In this case, we also say that the relation is *well-founded.* Note that we are denoting the relation on A and the ordering between ordinals using the same symbol $<$. Thus, an arbitrary well-founded relation `rel` defined on a set of objects satisfying a property `mp` (measure property) can be defined in ACL2 as shown below:

```
(encapsulate
 ((mp (x) booleanp) (rel (x y) booleanp) (fn (x) e0-ordinalp))
 ...
 (defthm rel-well-founded-relation-on-mp
   (and (implies (mp x) (e0-ordinalp (fn x)))
        (implies (and (mp x) (mp y) (rel x y))
                 (e0-ord-< (fn x) (fn y))))
   :rule-classes :well-founded-relation))
```

By the encapsulation mechanism (using `encapsulate`), the user can introduce new function symbols by axioms constraining them to have certain properties (to ensure consistency, a witness local function having the same properties has to be exhibited). Inside an `encapsulate`, properties stated with `defthm` need to be proved for the local witnesses, and outside, those theorems work as assumed axioms. The functions partially defined with `encapsulate` can be seen as second order variables, representing functions with those properties. A derived rule of inference, functional instantiation, allows some kind of second-order reasoning: theorems about constrained functions can be instantiated with function symbols known to have the same properties.

In this case, we partially define three functions `mp`, `fn` and `rel`, defining a general well-founded relation in ACL2 (dots are used to omit the irrelevant local definitions). The predicate `mp` recognizes the kind of objects (called *measures*) that are ordered in a well-founded way by `rel`. The *embedding* function `fn` is an order-preserving function mapping every measure to an ordinal. Once a relation is proved to satisfy these properties and the theorem is stored as a well-founded relation rule, it can be used in the admissibility test for recursive functions. We call the theorem `rel-well-founded-relation-on-mp` above the *well-foundedness theorem* for `rel`, `mp` and `fn`. In ACL2, every particular well-founded relation (except the primitive relation on ε_0 ordinals) has to be given by means of three functions (a binary relation, a measure predicate and an embedding function), and the proof of the corresponding well-foundedness theorem for such functions.

Let us now deal with formalization of multisets relations. We represent multisets in ACL2 as true lists. Given a predicate `(mp x)` describing a set A, finite multisets over A are described by the following function:

```
(defun mp-true-listp (l)
  (if (atom l)
      (equal l nil)
      (and (mp (car l)) (mp-true-listp (cdr l)))))
```

Note that this function depends on the particular definition of the predicate `mp`. With this representation, different true lists can represent the same multiset: two true lists represent the same multiset iff one is a permutation of the other. Thus, the order in which the elements appear in a list is not relevant, but the number of occurrences of an element is important. This must be taken into account, for example, when defining multiset difference in ACL2 (the function `remove-one`, omitted here, deletes one occurrence of an element from a list, whenever possible):

```
(defun multiset-diff (m n)
  (if (atom n) m (multiset-diff (remove-one (car n) m) (cdr n))))
```

The definition of $<_{mul}$ given in the preceding subsection is quite intuitive but, due to its many quantifiers, computationally complex. Instead, we will use a somewhat restricted definition, based on the following theorem:

Theorem 2. *Let $<$ be a strict ordering on a set A, and M, N two finite multisets over A. Then $N <_{mul} M$ iff $M \setminus N \neq \emptyset$ and $\forall n \in N \setminus M$, $\exists m \in M \setminus N$, such that $n < m$.*

From the computational point of view, the main advantage of this alternative definition is that we do not have to search the multisets X and Y of the original definition because we can take $M \setminus N$ and $N \setminus M$, respectively. It should be remarked that this equivalence is true only when $<$ is a strict partial ordering. Anyway, this is not a severe restriction. Moreover, well-foundedness of $<_{mul}$ also holds when this restricted definition is used, even if the relation $<$ is not transitive, as we will see. Thus, given a defined (or constrained) binary relation `rel`, we define the induced relation on multisets based on this alternative definition:

```
(defun exists-rel-bigger (x l)
  (cond ((atom l) nil)
        ((rel x (car l)) t)
        (t (exists-rel-bigger x (cdr l)))))

(defun forall-exists-rel-bigger (l m)
  (if (atom l)
      t
      (and (exists-rel-bigger (car l) m)
           (forall-exists-rel-bigger (cdr l) m))))

(defun mul-rel (n m)
  (let ((m-n (multiset-diff m n))
        (n-m (multiset-diff n m)))
    (and (consp m-n) (forall-exists-rel-bigger n-m m-n))))
```

Finally, let us see how we can formalize in the ACL2 logic the theorem 1 above, which states well-foundedness of the relation `mul-rel`. As said before, in order to establish well-foundedness of a relation in ACL2, in addition to the relation (`mul-rel` in this case), we have to provide the measure predicate and the embedding function, and then prove the corresponding well-foundedness theorem. Since `mul-rel` is intended to be defined on multisets of elements satisfying `mp`, then `mp-true-listp` is the measure predicate in this case. Let us suppose we have defined a suitable embedding function called `map-fn-e0-ord`. Then theorem 1 is formalized as follows:

```
(defthm multiset-extension-of-rel-well-founded
  (and (implies (mp-true-listp x)
                (e0-ordinalp (map-fn-e0-ord x)))
       (implies (and (mp-true-listp x)
                     (mp-true-listp y)
                     (mul-rel x y))
         (e0-ord-< (map-fn-e0-ord x) (map-fn-e0-ord y))))
  :rule-classes :well-founded-relation)
```

The command `defthm` starts a proof attempt in ACL2. The theorem prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, the user can guide the prover by adding previous lemmas and definitions, in order to carry out a formal proof based on a preconceived hand proof. In the following, we present a suitable definition for the embedding function `map-fn-e0-ord` and the proof sketch we followed to obtain a mechanical proof of the above theorem.

1.2 A Proof of Well-Foundedness of the Multiset Relation

In the literature (for example [3]) Theorem 1 is usually proved using König's lemma: every infinite and finitely branched tree has an infinite path. Nevertheless, we have to find a different proof (and more constructive) in ACL2, defining an order-preserving embedding function `map-fn-e0-ord` from `mp-true-listp` objects to `e0-ordinalp` objects. Thus, our proof is based on the following result from ordinal theory: given an ordinal γ, the set $\mathcal{M}(\gamma)$ of finite multisets of elements of γ, ordered by the multiset relation induced by the order between ordinals, is order-isomorphic to the ordinal ω^γ and the isomorphism is given by the function H where $H(\{\beta_1, \dots, \beta_n\}) = \omega^{\beta_1} + \dots + \omega^{\beta_n}$. This result can be proved using Cantor's normal form of ordinals and its properties.

The isomorphism H above suggests the following definition of the embedding function `map-fn-e0-ord`: given a multiset of elements satisfying `mp`, apply `fn` to every element to obtain a multiset of ordinals. Then apply H to obtain an ordinal less than ε_0. If ordinals are represented in ACL2 notation (see [5]), then the function H can be easily defined, provided that the function `fn` returns always a non-zero ordinal: the function H simply has to sort the ordinals in the multiset and add 0 as the final `cdr`. These considerations lead us to the following definition of the embedding function `map-fn-e0-ord`. Note that the non-zero restriction on `fn` is easily overcome, defining (the macro) `fn1` equal to `fn` except for integers, where `1` is added. In this way `fn1` returns non-zero ordinals for every measure object and it is order-preserving if and only if `fn` is.

```
(defun insert-e0-ord-< (x l)
  (cond ((atom l) (cons x l))
        ((not (e0-ord-< x (car l)))  (cons x l))
        (t (cons (car l) (insert-e0-ord-< x (cdr l))))))

(defun add1-if-integer (x) (if (integerp x) (1+ x) x))

(defmacro fn1 (x) `(add1-if-integer (fn ,x)))

(defun map-fn-e0-ord (l)
  (if (consp l)
      (insert-e0-ord-< (fn1 (car l)) (map-fn-e0-ord (cdr l)))
    0))
```

Once `map-fn-e0-ord` has been defined, let us now deal with the ACL2 mechanical proof of the well-foundedness theorem for `mul-rel`, `mp-true-listp` and `map-fn-e0-ord` as stated at the end of subsection 1.1. The first part of the theorem, which establishes that `(map-fn-e0-ord x)` is an ordinal when `(mp-true-listp x)`, it is not difficult, and can be proved in ACL2 with minor help form the user. The hard part of the theorem is to show that `map-fn-e0-ord` is order-preserving. Here is an informal proof sketch:

Proof sketch:
Let us denote, for simplicity, the functions `fn1` and `map-fn-e0-ord`, as f and f_{mul}, and the relation `rel`, `mul-rel` and `e0-ord-<` as $<_{rel}$, $<_{mul}$ and $<$, respectively. Let M and N be two multisets of `mp` elements such that $N <_{mul} M$. We have to prove that $f_{mul}(N) < f_{mul}(M)$. We can apply induction on the number of elements of N. Note that M can not be empty, and if N is empty the result trivially holds. So let us suppose that M and N are not empty. Let $f(x)$, $f(y)$ be the biggest elements of $f[N]$ and $f[M]$, respectively. Note that $f(x)$ and $f(y)$ are the `car` elements of $f_{mul}(N)$ and $f_{mul}(M)$, respectively. Since $f(x)$ and $f(y)$ are ordinals, three cases may arise:

1. $f(x) < f(y)$. Then, by definition of $<$, we have $f_{mul}(N) < f_{mul}(M)$.
2. $f(x) > f(y)$. This is not possible: in that case x is in $N \setminus M$ and by the multiset relation definition, exists z in $M \setminus N$ such that $x <_{rel} z$. Consequently $f(z) > f(x) > f(y)$. This contradicts the fact that $f(y)$ is the biggest element of $f[M]$.
3. $f(x) = f(y)$. In that case, $x \in M$, since otherwise it would exist $z \in M \setminus N$ such that $x <_{rel} z$ and the same contradiction as in the previous case appears. Let $M' = M \setminus \{x\}$ and $N' = N \setminus \{x\}$. We have $N' <_{mul} M'$ and, in addition, $f_{mul}(N')$ and $f_{mul}(M')$ are the `cdr` of $f_{mul}(N)$ and $f_{mul}(M)$, respectively. Induction hypothesis can be applied here to conclude that $f_{mul}(N') < f_{mul}(M')$ and therefore $f_{mul}(N) < f_{mul}(M)$. $\square$

We carried out this proof in ACL2. The proof effort was not trivial: lemmas to handle each of the cases generated by the above induction scheme have to be proved, obtaining a mechanical proof very close to the previous proof sketch. See the book `multiset.lisp` in the web page for details about the mechanical proof.

Well-foundedness of `mul-rel` has been proved in an abstract framework, without assuming any particular properties of `rel`, `mp` and `fn`, except those concerning well-foundedness. This allows us to functionally instantiate the theorem in order to establish well-foundedness of the multiset relation induced by any given well-founded ACL2 relation. We defined a macro `defmul` in order to mechanize this process of functional instantiation, providing a convenient way to define the multiset relation induced by a given well-founded relation and to declare the corresponding well-founded relation rule. The following section describes the `defmul` macro.

1.3 The `defmul` Macro

Let us suppose we have a previously defined relation *my-rel*, which is known to be well-founded on a set of objects satisfying the measure property *my-mp* and justified by the embedding function *my-fn*. That is to say, the following theorem, using variables *x* and *y*, has been proved (and stored as a well-founded relation rule):

```
(defthm theorem-name
   (and (implies (my-mp x) (e0-ordinalp (my-fn x)))
        (implies (and (my-mp x) (my-mp y) (my-rel x y))
                 (e0-ord-< (my-fn x) (my-fn y))))
   :rule-classes :well-founded-relation))
```

In order to define the (well-founded) multiset relation induced by *my-rel*, we simply write the following macro call:

```
(defmul (my-rel theorem-name my-mp my-fn x y))
```

The expansion of this macro generate a number of ACL2 events. After the above call to `defmul`, the function `mul-`*my-rel* is defined as a well-founded relation on multisets of elements satisfying the property *my-mp*, induced by the well-founded relation *my-rel*, and a proof of the corresponding well-foundedness theorem is carried out, without assistance from the user. From this moment on, `mul-`*my-rel* can be used in the admissibility test for recursive functions to show that the recursion terminates.

2 An Applicative ATP for Propositional Logic

We illustrate the use of the `defmul` tool with a case study: the formal verification of an applicative Common Lisp definition of a tableaux based theorem prover for propositional logic. In this section, we present an ACL2 function implementing the prover; as we will see, termination of this function is not trivial. In the next section we sketch a termination proof in ACL2 using well-founded multisets relations. To build the theorem prover, we closely follow the approach given by M. Fitting in [4].

2.1 Formalization of Propositional Logic and Uniform Notation

We explain now how we have represented propositional formulas in ACL2. Any ACL2 symbol (recognized by the ACL2 function `symbolp`) will represent a propositional symbol. We represent propositional formulas in prefix notation, using lists. The propositional connectives considered are the usual: negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$). If a list represents a propositional formula, its first element is a logic connective, and the rest are the arguments. The following function `propositional-p` recognize those ACL2 objects representing propositional formulas. The functions `arg1` and `arg2` obtain, respectively, the first and the second argument of a formula, if they exist. There are three kinds of propositional formulas: atomic, monary and binary formulas. The functions `atomic-p`, `monary-p` and `binary-p` to identify these formulas. We omit here all these auxiliary functions.

```
(defun propositional-p (x)
  (cond ((monary-p x) (propositional-p (arg1 x)))
        ((binary-p x) (and (propositional-p (arg1 x))
                           (propositional-p (arg2 x))))
        (t (atomic-p x))))
```

Notwithstanding, we will adopt the *uniform notation* approach (see [4]) to deal with the recursive structure of propositional formulas. We classify propositional formulas with the form $(X \circ Y)$ and $\neg(X \circ Y)$ in two categories: those having a conjunctive behaviour, called α-formulas, and those having a disjunctive behaviour, called β-formulas. Each α-formula and β-formula has two components, α_1 and α_2 for the α-formulas and, β_1 and β_2 for the β-formulas. The classification and components are given in the following tables:

α	α_1	α_2
$X \wedge Y$	X	Y
$\neg(X \vee Y)$	$\neg X$	$\neg Y$
$\neg(X \rightarrow Y)$	X	$\neg Y$

β	β_1	β_2
$X \vee Y$	X	Y
$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$X \rightarrow Y$	$\neg X$	Y
$X \leftrightarrow Y$	$X \wedge Y$	$\neg X \wedge \neg Y$
$\neg(X \leftrightarrow Y)$	$X \wedge \neg Y$	$\neg X \wedge Y$

We define the functions `alpha-formula` and `beta-formula` in order to distinguish these two kinds of formulas. To access to their components, we define the functions `component-1` (to obtain α_1 or β_1) and `component-2` (to obtain α_2 and β_2). There are also formulas neither conjunctive nor disjunctive: the double negations and the literals. We define the functions `double-negation` and `literal-p` to recognize them. The component of a double negation $\neg\neg Y$, is the formula Y. We define the function `component-double-neg` to build the component of a double negation. We omit all these definitions here.

The following theorem is a key lemma, needed to classify propositional formulas using the uniform notation. This result gives a new perspective of the concept `propositional-p` and then, a new way of defining functions by recursion on formulas:

```
(defthm uniform-definition-of-propositional-p
   (iff (propositional-p F)
        (or (alpha-formula F)
            (beta-formula F)
            (double-negation F)
            (literal-p F))))
```

2.2 Semantic Tableau as Rules of Transformation

The semantic tableau method is a refutation system. To prove the validity of a formula X, we start with $\neg X$ until we eventually generate a contradiction. From a constructive point of view, the method works with a set of formulas and tries to build a model of that set. If it is not possible to build a model for the formula $\neg X$, then X is valid.

Given a finite tree T, with its nodes labeled with propositional formulas, the method of semantic tableau selects a branch θ and a non-literal formula X in θ. If X is $\neg\neg Y$, then the branch θ is extended adding a new node labeled with Y. If X is an α-formula, then the branch θ is extended adding two nodes labeled with the components α_1 and α_2 of the original formula. If X is a β-formula, then the branch θ is extended adding two branches at the end, each of them with a node labeled with the components β_1 and β_2 of the selected formula. If we denote the result as T^*, we say that T^* is obtained from T using a tableau expansion rule. If a branch does not have non-literal formulas, we can not apply the above process. If a branch have two complementary formulas we say that the branch is *closed*.

A tableau for a set of formulas, $\{A_1, \ldots, A_n\}$, is the one branch tree with n nodes labeled with $A_1, \ldots, A_n$, or any tree T^* obtained from a tableau for the set of formulas $\{A_1, \ldots, A_n\}$, using a tableau expansion rule. It can be proved that a propositional formula X is valid if and only if there exists a tableau for $\{\neg X\}$ with all its branches closed.

To define a function in ACL2 implementing the semantic tableau method, we have to decide how to represent a tableau. This decision can affect on how the function is defined later. A tableau can be seen as a list of branches, and a branch as a list of formulas. In this way, the function that implements the semantic tableau method has to work recursively; that is, it takes a branch, apply it a tableau expansion rule and replace the original for the new one. If the branch considered is closed, it will be discarded and another branch will be analyzed.

This is a recursive process that works on branches: we begin with a branch θ. If θ is closed, we have finished the process successfully. Otherwise, a non-literal formula X is selected from θ. If X is $\neg\neg Y$, Y is added to θ and the process will be applied again to it. If X is a α-formula, the components α_1 and α_2 will be added to θ and the process will be applied again to it. If X is a β-formula, two new branches are built: θ_1 adding the component β_1 to θ and θ_2 adding the component β_2 to θ. The process is applied to θ_1, and, if it succeeds, it will be applied to θ_2. If any of them does not succeed, the process on the original branch θ does not succeed.

In the process described above a non-literal and non-expanded formula must be chosen every time an expansion rule is to be applied. If the formula chosen has been expanded before, then the new branch generated by the process will have repetitions. To avoid this we can mark the expanded formulas or we can eliminate them. We use the second option to simplify the function definition: this is possible because in the tableau method for propositional logic the formulas are used only once.

Therefore, we can define the function associated with the method of semantic tableau, as a function that works with a list of formulas. This function builds new lists from the initial list of formulas, and recursively applies the same process to them. Thus, it can be seen as a transformation system, specified by a set of rules acting on a set of formulas. This kind of rule-based point of view is common to others provers based on transformations acting on set of formulas. The rules used in this case are the following:

1. Double negation rule:

$$\{F_1, \ldots, F_{i-1}, \neg\neg G, F_{i+1}, \ldots, F_n\} \overset{st}{\Longrightarrow} \{F_1, \ldots, F_{i-1}, G, F_{i+1}, \ldots, F_n\}$$

2. α-formula rule:

$$\{F_1, \ldots, F_{i-1}, \alpha, F_{i+1}, \ldots, F_n\} \overset{st}{\Longrightarrow} \{F_1, \ldots, F_{i-1}, \alpha_1, \alpha_2, F_{i+1}, \ldots, F_n\}$$

3. β-formula rule:

$$\{F_1, \ldots, F_{i-1}, \beta, F_{i+1}, \ldots, F_n\} \overset{st}{\Longrightarrow} \{F_1, \ldots, F_{i-1}, \beta_1, F_{i+1}, \ldots, F_n\}$$

$$\{F_1, \ldots, F_{i-1}, \beta, F_{i+1}, \ldots, F_n\} \overset{st}{\Longrightarrow} \{F_1, \ldots, F_{i-1}, \beta_2, F_{i+1}, \ldots, F_n\}$$

2.3 ACL2 Definition of a Semantic Tableau Prover

Based on the above considerations, our ACL2 implementation of a tableau based theorem prover receives as argument a list of formulas that represents a branch of the tableau. If this branch is not closed, a selection function chooses a non-literal formula. The tableau expansion rules are applied to this formula, generating new branches. The function is recursively applied to these new branches. With this idea, we define the ACL2 function `closed-tableau`, implementing the tableau method for propositional logic:

```
(defun closed-tableau (S)
  (declare (xargs :mode :program))
  (cond ((endp S) nil)
        ((closed S) t)
        (t (let ((F (selection S)))
             (cond ((double-negation F)
                    (closed-tableau (add (component-double-neg F)
                                         (remove-one F S))))
                   ((alpha-formula F)
                    (closed-tableau (add (component-1 F)
                                         (add (component-2 F)
                                              (remove-one F S)))))
                   ((beta-formula F)
                    (and (closed-tableau (add (component-1 F)
                                              (remove-one F S)))
                         (closed-tableau (add (component-2 F)
                                              (remove-one F S)))))
                   (t nil))))))
```

Several remarks are due about this definition. First, note that to implement the control of this process we need a selection function, which determines the chosen formula and, consequently, the expansion rule to apply. For this purpose, we consider a function named `selection` (definition omitted) that receives a list of formulas as an argument and returns the first non-literal formula from that list, whenever there exists such formula. Nevertheless, we could have used any function with the following properties:

1. If the argument of `selection` is a list with some non-literal formula, then the function returns a non-literal formula from that list.
2. If the argument of `selection` is a list of literal formulas, then the function returns `nil`.

This function can be executed on every compliant Common Lisp implementation. For example, we have checked the validity of some Urquhart formulas obtaining the following time results:

N	6	8	10	12	14
time (msec)	130	840	5270	29380	156200

One of the base cases of this recursive function appears when the branch has two complementary formulas. In such case we recognize the branch as closed. The function `closed`, omitted here, checks if a list has two complementary formulas. For the recursive calls, we have to build new branches by replacing the non-literal formula chosen with its components. We define the function `add` to add one formula to a branch avoiding repetitions.

3 Termination of `closed-tableau`

The definition of `closed-tableau` is not admitted immediately as an axiom in the ACL2 logic, since the default heuristics of the prover are not able to prove its termination. The termination proof of this function is not trivial: note the different behaviour of the recursive calls for α-formulas and β-formulas; in particular, the α expansion rule obtains a larger set of formulas.

The declaration `(xargs :mode :program)` forces ACL2 to accept this definition without proving its termination. A function definition in `:program` mode is not included as an axiom of the logic (and therefore reasoning about it is not possible) until its termination is proved. Thus, a suitable measure and well-founded relation has still to be explicitly given to the prover. We will use a multiset relation for that purpose, as we explain now.

We can define a measure on formulas, related to the uniform notation, ensuring that the measure of the components of an α-formula, β-formula or double negation, are smaller than the measure of the original formula. We extend the measure given on [1] to include equivalences:

Definition 2. *The* **uniform measure** *of a propositional formula X is given by the function μ:*

1. *If X is atomic, $\mu(X) = 0$*
2. *If $X = \neg Y$, $\mu(X) = 1 + \mu(Y)$*
3. *If $X = Y_1 \circ Y_2$, with $\circ$ distinct of equivalence, $\mu(X) = 2 + \mu(Y_1) + \mu(Y_2)$*
4. *If $X = Y_1 \leftrightarrow Y_2$, $\mu(X) = 5 + \mu(Y_1) + \mu(Y_2)$*

We can easily implement the measure μ in ACL2, defining a function `uniform-measure`, omitted here. The main property of this uniform measure is that it decreases on the components of compound formulas. For example, the property for α-formulas is showed below (analogous properties for double negation and beta formulas are established):

```
(defthm uniform-measure-alpha-formula-decreases
  (implies (alpha-formula F)
    (and (< (uniform-measure (component-1 F))
            (uniform-measure F))
         (< (uniform-measure (component-2 F))
            (uniform-measure F)))))
```

Now we can define a suitable measure for the termination of the function `closed-tableau`. Recall that the argument of this function is a list of formulas, representing a branch of a tableau. The idea is to measure this argument by the list of the uniform measures of each of its formulas. The following function defines this measure:

```
(defun branch-uniform-measure (branch)
  (cond ((endp branch) nil)
        (t (cons (uniform-measure (car branch))
                 (branch-uniform-measure (cdr branch))))))
```

Note that this measure can be seen as a multiset of ordinals (natural numbers). Thus, the multiset relation induced by `e0-ord-<` on multisets of ordinals is a well-founded relation that can be used as the well-founded relation needed to justify termination of `closed-tableau`. We simply make this `defmul` call to define in ACL2 the intended multiset well-founded relation:

```
(defmul (e0-ord-< nil e0-ordinalp e0-ord-<-fn nil nil))
```

After this `defmul` call, the function `mul-e0-ord-<` is automatically defined and proved to be well-founded over multisets of ordinals. We can now verify the termination of the function `closed-tableau`, providing the measure of the arguments and the well-founded relation:

```
(verify-termination closed-tableau
  (declare (xargs :measure (branch-uniform-measure S)
                  :well-founded-relation mul-e0-ord-<)))
```

This call to `verify-termination` generates a proof attempt to show that the measure `branch-uniform-measure` decreases (w.r.t the multiset relation `mul-e0-ord-<`) in every recursive call of the function `closed-tableau`. With the help of some previous lemmas, this proof can be successfully completed in ACL2 (see the web page for details) and the function definition is admitted as an axiom in the logic. This allows formal reasoning about it.

For example, we can define a function to check the validity of a formula, calling the function `closed-tableau` on the list built with the negation of the original formula:

```
(defun tableau-valid-p (F)
  (closed-tableau (list (negation F))))
```

A formal verification of this function is now possible. For example, we can prove in ACL2 the soundness and completeness theorem (see [4]) for this tableau based theorem prover, following the lines of a previous verification work of Boyer and Moore [2], where a tautology checker based on binary decision diagrams is formally verified using Nqthm. Nevertheless, we do not discuss this issue here, since we are concentrating on termination aspects and how multiset relations can help in the task of proving it.

4 Conclusions

We have presented a formalization of multiset relations in ACL2, showing how they can be used as a tool for proving non-trivial termination properties of recursive functions in ACL2. We have defined the multiset relation induced by a given relation and proved a theorem establishing well-foundedness of the multiset relation induced by a well-founded relation. This theorem is formulated in an abstract way, so that functional instantiation can be used to prove well-foundedness of concrete multiset relations. We also presented a macro named `defmul`, implemented to provide a convenient tool to define these concrete multiset well-founded relation.

We initially presented this tool in [8], where we successfully used it to prove several non-trivial termination properties: a tail-recursive version of Ackermann's function, a definition of McCarthy's 91 function and a proof of Newman's lemma for abstract reductions. In this paper we present how this tool can be applied to prove termination of an applicative Common Lisp definition of a tableau-based theorem prover. Proving termination allows us to formally verify the intended properties of the function, namely its soundness and completeness. One interesting aspect of ACL2 is that the functions verified are defined in an applicative subset of Common Lisp, and (under some conditions) they can be executed in any interpreter of that language.

Proving theorems in ACL2 is not a trivial task. A typical proof effort consists of formalizing the problem, and guiding the prover to a preconceived hand proof, by decomposing the proof into intermediate lemmas. If one lemma is not proved in a first attempt, then additional lemmas are often needed, as suggested by inspecting the failed proof. See the web page for a detailed description of the proofs presented in this paper.

The work presented in the second section is part of the ambitious project of providing a mechanically verified set of automated reasoning algorithms for some logics. We have begun with propositional logic and a well-known automated theorem proving technique, semantic tableau. We have seen that the multiset tool plays an unexpected role in the termination proof. This work can be extended to others ATP's for this logic, others logics (first order, equational [7], modal, ...) and applications based on these logics; we think that the multiset tool will be important to develop this project.

References

1. Ben-Ari, M. *Mathematical Logic for Computer Science.* Prentice Hall, 1993.
2. Boyer, R., and Moore, J S. *A Computational Logic.* Academic Press, 1979.
3. Dershowitz, N., and Manna, Z. Proving termination with multiset orderings. In *Annual International Colloquium on Automata, Languages and Programming* (1979), no. 71 in LNCS, Springer-Verlag, pp. 188–202.
4. Fitting, M.C. *First-Order Logic and Automated Theorem Proving*, 2nd edition. Springer-Verlag, New York, 1996.
5. Kaufmann, M., Manolios, P., and Moore, J S. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
6. Kaufmann, M., and Moore, J S. ACL2 version 2.5. `http://www.cs.utexas.edu/users/moore/acl2/`.
7. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., and Martín, F.J. Formalizing rewriting in the ACL2 theorem prover. In *Proceedings of AISC'2000 (Fifth International Conference Artificial Intelligence and Symbolic Computation)*, no. 1930 in LNAI, Springer-Verlag, pp. 92–106.
8. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., and Martín, F.J. Multiset Relations: a Tool for Proving Termination. In *ACL2 Workshop 2000 Proceedings*, Technical Report, TR–00–29 Computer Science Departament, University of Texas.