A Theory About First-Order Terms in ACL2 [⋆]

J.-L. Ruiz-Reina, J.-A. Alonso, M.-J. Hidalgo and F.-J. Martín-Mateos
`http://www.cs.us.es/{~jruiz, ~jalonso, ~mjoseh, ~fmartin}`

Departamento de Ciencias de la Computación e Inteligencia Artificial
Escuela Técnica Superior de Ingeniería Informática, Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain

Abstract. We describe the development in ACL2 of a library of results about first-order terms. In particular, we present the formalization of some of the main properties of the complete lattice of first-order terms with respect to the subsumption relation. As a by-product, verified executable implementations are obtained for some basic operations on first-order terms, including matching, renaming, unification and anti-unification. This work can be seen as a basis for further studies about the formal properties of automated reasoning and symbolic computation systems.

In [9, 11] we described a formal ACL2 theory about equational reasoning and term rewriting systems. In that work we used a library of definitions and theorems formalizing the lattice theoretic properties of the set of first-order terms with respect to the subsumption relation. Nevertheless, the results of this library have not been documented yet, except the verification of a unification algorithm [10] (in fact, a preliminary Nqthm version). We think that these results are interesting for themselves, so our intention is to describe them in this paper.

Our purpose when developing these results about first-order terms is twofold. From a theoretical point of view, we proved in ACL2 that the set of first-order terms in a given signature is a well-founded lattice with respect to the subsumption relation. And from a practical point of view, we implemented and formally verified several basic algorithms on terms, including matching, renaming, unification and anti-unification; these algorithms can be executed in any compliant Common Lisp (with the appropriate files loaded).

The usefulness of this library has already been demonstrated: as we said above, our formalization of equational reasoning and rewriting relies heavily on the work presented here. For that reason, we think that it can be seen as a basis for further studies about the formal properties of automated reasoning and symbolic computation systems.

As far as we know, there is no other complete formalization of the lattice properties of terms, but we can mention some related works done in mechanical verification of properties of terms, especially for unification algorithms. Paulson [6] describes the verification of a unification algorithm using LCF. Rouyer [8] does the same using Coq. For some related work in the Boyer-Moore theorem prover, see [4], where Kaufmann presents a formal proof of a generalization algorithm used in PC-Nqthm. In [5], McCune and Shumsky present the ACL2 verification of a checker for resolution/paramodulation proofs generated by the Otter theorem prover; in this work, the needed theory about first-order terms was also formalized.

Due to the lack of space, we do not present details of the proofs here, and some function definitions will be omitted. We urge the interested reader to consult the books provided by the supporting materials (see also [12]), where the proofs are extensively documented. In each section, we will specify the name of the book that contains the definitions and theorems described.

1 Introduction

We present in this section an informal description of the definitions and results we are going to formalize using ACL2.

[⋆] This work has been supported by MCyT project TIC2000–1368–C03–02.

1.1 The complete lattice of first-order terms

A *signature* Σ is a family of sets $\langle \Sigma_n : n \in \omega \rangle$. If $f \in \Sigma_n$, we say that f is a *function symbol of arity* n. Given a signature Σ and a denumerable set X of *variable symbols*, the set of *(first-order) terms* $T(\Sigma, X)$ in the signature Σ is the smallest set containing X such that $f(t_1, \ldots, t_n) \in T(\Sigma, X)$ whenever each $t_i \in T(\Sigma, X)$ and $f \in \Sigma_n$. Note that functions with *variable arity* are permitted. A function $\sigma : X \to T(\Sigma, X)$ is a *substitution* if only a finite set of variables (the *domain* of the substitution) are not mapped to themselves. If $\{x_1, \ldots, x_n\}$ is the domain of a substitution σ, then the substitution is usually denoted as $\{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$; the set $\{\sigma(x_1), \ldots, \sigma(x_n)\}$ is called the *codomain* of the substitution. A substitution σ can be extended to a function from terms to terms in such a way that $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. A term t *matches* a term s if $\sigma(s) = t$ for some substitution σ. In that case, we write $s \leq t$ and say that t is an *instance* of (or is *more specific* than) s or that s *subsumes* (or is *more general than*) t. We also say that σ is a *matching substitution for* s *and* t. Subsumption is a *quasi-ordering* (a reflexive and transitive relation) on the set of terms. The associated equivalence relation is denoted as $\equiv$. A *variable substitution* is a substitution such that its codomain is a subset of X. A *renaming* substitution is a variable substitution such that restricted to its domain is injective. It can be proved that $s \equiv t$ if and only if there exists a renaming substitution θ, such that $s = \theta(t)$ and the variables of t are contained in the domain of θ.

Also, it can be defined a subsumption relation on substitutions: $\sigma \leq \delta$ if there exists a substitution γ such that $\delta = \gamma \circ \sigma$ where $\circ$ stands for functional composition. An equation is a pair of terms, $t_1 \approx t_2$. A substitution σ is a *matcher* (*solution*) of $t_1 \approx t_2$ if $\sigma(t_1) = t_2$ ($\sigma(t_1) = \sigma(t_2)$), and is a matcher (solution) of a system of equations S if it is a matcher (solution) of every member of S. A solution of S is a *most general solution (mgs)* if it subsumes every other solution of S. A *unifier* of t_1 and t_2 is a solution of the system $\{t_1 \approx t_2\}$ and a *most general unifier (mgu)* of t_1 and t_2 is a mgs of that system. If there exists a unifier of two terms we say that the terms are *unifiable*.

We say that a quasi-ordered set $\langle A, \leq \rangle$ is a *lattice* if for every $x, y \in A$ a least upper bound (*lub*) and a greatest lower bound (*glb*) exist. If only the existence of a glb (lub) can be assured, we call this a *lower (upper) semi-lattice*. We say that the lattice is *complete* if a glb and a lub exist for every $B \subseteq A$. A quasi-ordering $\leq$ is well-founded if there is no infinite descending chain $x_1 > x_2 > x_3 > \cdots$, where $<$ is its associated strict partial order. If $\langle A, \leq \rangle$ is a well founded lower semi-lattice, and we take $\top \notin A$ an additional top element, it can be proved that $\langle A \cup \{\top\}, \leq \rangle$ is a complete lattice.

The set of first order terms in a given signature, with an additional top element, is a well-founded complete lattice with respect to the subsumption quasi-ordering (see, for example, [3] or [7]). In the sequel, we describe the formalization of these properties in the ACL2 logic and a proof of them using the theorem prover.

1.2 An overview of the paper

We now give a thorough top-level view of the ACL2 formalization and proofs described in the following sections:

- In section 2 we discuss how we represent terms and substitutions, and the impact of this representation on the mechanization of the proofs. We adopt the point of view of considering every ACL2 object as the representation of a term, and the same is done for substitutions. Emphasis is placed on the style of definitions by mutual recursion on terms and lists of terms, which resembles standard definitions by recursion on the structure of terms, and suggests to the prover induction schemes very close to induction on the structure of terms.
- Section 3 describes the formalization in ACL2 of the subsumption quasi-ordering. For that purpose, we define and verify a matching algorithm. This algorithm is defined by a rule based transformation system acting on systems of equations. Given this verified matching algorithm

we can easily define the subsumption relation between terms. We finally prove that this relation is a well-founded quasi-ordering.

- In section 4, we define equivalence between terms (with respect to subsumption) and the concept of renaming substitutions, proving that two terms are equivalent if and only if there exists a renaming substitution transforming one term in the other. We also define and verify a function that renames the variables of a term, that can be used for standardization apart.
- A proof of the existence of a greatest lower bound (with respect to subsumption) of every pair of terms is described in section 5. This is done by means of the verification of an anti-unification algorithm. Emphasis is done in compositional reasoning techniques used in this proof effort.
- Section 6 describes the verification of a unification algorithm. To state the properties of this algorithm, we previously define and verify the subsumption relation between substitutions and the concept of idempotent substitutions. The unification algorithm is defined and verified in a similar style to the matching algorithm, following a rule-based transformation system. Having verified the unification and renaming algorithms, we can define and verify a function obtaining a least upper bound of every pair of unifiable terms.
- Although the theorems and definitions in sections from 2 to 6 are proved considering that every ACL2 object represents a first-order term, in section 7 we define well-formed terms in a given signature and prove that the above operations are closed with respect to the terms in a signature. We also discuss guard verification of the functions defined.
- In section 8 we compile all the theorems proved, showing that the set of first-order terms in a given signature (plus an additional top term) is a well-founded lattice w.r.t. subsumption. Finally, in section 9 we discuss some conclusions and further work.

2 Representation of first-order terms and substitutions

The definitions and results described in this section appear in the books `basic.lisp` and `terms.-lisp` of the supporting materials. We will represent first-order terms in ACL2 using lists, in prefix notation. We could define a predicate in ACL2 recognizing those objects that represent first-order terms in a given signature. In fact, this is done in section 7, when we talk about guard verification of the functions presented here. But for the moment we will assume, instead, that every ACL2 object represents a term. We only distinguish between variable and non-variable terms:

```
(defun variable-p (x) (atom x))
```

Every `consp` object can be seen as a non-variable term with its `car` as its top function symbol and its `cdr` as the list of its arguments. In this way, every first order term for a given signature can be represented. For example, the term $f(x, g(y), h(x))$ is represented as `(f x (g y) (h x))`. "Non-proper" terms have to be considered with this wider representation, because the list of its arguments can be non-proper. For example, the term `(f x (g y . 1) (h x) . 3)` has the same structure than the above term, but it is a different (non-proper) term.

As we will see, the theorems we proved do not need hypothesis regarding the type of the objects involved; this means that our results are also valid for these non-proper terms, and that we are formalizing a theory that strictly contains first-order terms. Our definitions deal with these non-proper terms in a natural way, although they are not in the intended domain of the functions.

Of course, to ensure that our formalization is correct, we also have to prove "closure" properties for the main operations acting on terms. For example, we proved that the most general instance of two objects representing terms in a given signature is an object representing a term in the same signature. A discussion about this closure properties will be done in section 7.

Substitutions are represented as association lists, binding variables to terms. The following function `val` returns the term that a substitution associates to a variable:

```
(defun val (x sigma)
  (if (endp sigma)
      x
    (if (eql x (caar sigma)) (cdar sigma) (val x (cdr sigma)))))
```

Note that `val` is different from the primitive function `assoc`, because the variables that are not in the domain of a substitution are mapped to themselves. As with terms, we will assume that any ACL2 object represents a substitution (whose functional behavior is described by this function `val`) although we can also define a predicate recognizing those objects that represent substitutions in a given signature, as we will do in section 7. But unlike terms, different ACL2 objects may represent the same substitution, from a functional point of view. Thus, we cannot use `equal` to state the equality of two substitutions; instead, we have to state that they behave in the same way as functions.

The function `apply-subst` defines the application of a substitution to a term (or list of terms) by mutual recursion, using a standard trick. If `flg` is not `nil`, `(apply-subst flg sigma term)` is the term obtained by applying the substitution `sigma` to `term`. The value of `(apply-subst nil sigma term)` is the list of terms obtained by applying `sigma` to the list of terms `term`. We also define the macro `instance` to abbreviate applications of substitution to terms (i.e. with `flg`=`t`):

```
(defun apply-subst (flg sigma term)
  (if flg
      (if (variable-p term)
          (val term sigma)
        (cons (car term)
              (apply-subst nil sigma (cdr term))))
    (if (endp term)
        term
      (cons (apply-subst t sigma (car term))
            (apply-subst nil sigma (cdr term))))))

(defmacro instance (term sigma) `(apply-subst t ,sigma ,term))
```

The definition of `apply-subst` is a typical definition by recursion on the structure of terms and this style of definitions is extensively used in our development. The function is defined for terms and for lists of terms, using a flag `flg` to implement mutual recursion; then, the intended definition on terms is a particular case, when `flg`=`t`. As a consequence, most of our results about terms are also proved for lists of terms. The reason is that this kind of definitions suggests to the prover an induction scheme very close to induction on the structure of terms[1].

Let us illustrate all these questions, describing how we state a theorem verifying the definition of a function implementing the composition of two substitutions. We define the function `composition` as follows:

```
(defun composition (sigma1 sigma2)
  (if (endp sigma2)
      sigma1
    (cons (cons (caar sigma2) (instance (cdar sigma2) sigma1))
          (composition sigma1 (cdr sigma2)))))
```

Given two substitutions `sigma1` and `sigma2`, this function returns an association list representing the functional composition of `sigma1` and `sigma2`. If we want to prove that this definition of

[1] This style of definitions is taken from [4]. An alternative approach, followed in [5], could be to define a main function that operates on lists and an auxiliary function that operates on a single term.

composition meets the property $(\sigma \circ \delta)(t) = \sigma(\delta(t))$, for every term t and substitutions σ and δ, we generalize the property taking into account terms and lists of terms, establishing the following theorem:

```
(defthm composition-of-substitutions-apply
  (equal (apply-subst flg (composition sigma1 sigma2) term)
         (apply-subst flg sigma1 (apply-subst flg sigma2 term))))
```

Two remarks are worth pointing here, regarding the statement of this theorem. First, note that the theorem is true for all ACL2 objects `term`, `sigma1` and `sigma2`, and no additional hypothesis about them are needed. Thus, the definitions of `apply-subst` and `composition` are "consistent" with non-proper terms and substitutions. Second, since the theorem is stated generalizing the result for terms and for lists of terms, the heuristics of the prover lead to a proof attempt with the induction scheme suggested by `apply-subst`. This induction scheme is, in essence, an induction on the structure of `term` (the kind of proof a mathematician would do by hand). With this induction scheme, the theorem is proved without assistance from the user.

3 The subsumption relation

Having defined the notion of instantiation of a term by a substitution, we can define the subsumption relation between terms, and even prove that subsumption is a well-founded quasi-ordering.

3.1 A matching algorithm

Recall that the subsumption relation is defined as "$s \leq t$ if and only if there exists a substitution σ such that $\sigma(s) = t$". Because of our quantifier-free logic, we need to define the subsumption relation between terms in a more constructive way. That is, we define a matching algorithm, that given two terms, finds, if it exists, a substitution that applied to the first term gives the second (i.e., a matching substitution). The algorithm we verify is described by the following set of transformation rules, a modified version of the transformation rules given by Martelli and Montanari for a unification algorithm (section 6):

Bind:	$\{x \approx t\} \cup R; T$	$\Rightarrow_s R; \{x \mapsto t\} \cup T$ **if** $x \in X$ **and** $x \notin \mathit{dom}(T)$
Fail-Bind:	$\{x \approx t\} \cup R; T$	$\Rightarrow_s$ `nil` **if** $x \in \mathit{dom}(T)$ **and** $T(x) \neq t$
Delete:	$\{x \approx t\} \cup R; T$	$\Rightarrow_s R; T$ **if** $x \in \mathit{dom}(T)$ **and** $T(x) = t$
Fail-Var:	$\{f(s_1, \ldots, s_n) \approx x\} \cup R; T$	$\Rightarrow_s$ `nil` **if** $x \in X$
Decompose:	$\{f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)\} \cup R; T$	$\Rightarrow_s \{s_1 \approx t_1, \ldots, s_n \approx t_n\} \cup R; T$
Conflict:	$\{f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m)\} \cup R; T$	$\Rightarrow_s$ `nil` **if** $n \neq m$ **or** $f \neq g$

These rules act on two systems of equations: the first one with the equations to be matched and the second one with the matcher partially computed. To obtain a matcher of a system S, one starts with the pair of systems $S; \emptyset$ and apply the rules (non-deterministically) until `nil` is detected or until the first system is empty. In the first case, there is no matcher for S. In the latter case, the second system is a matcher of S. We implemented this rule-based matching algorithm in ACL2, by means of a function `match-mv`, whose definition appears in figure 1.

The auxiliary function `pair-args` is used to implement the rules **Decompose** and **Conflict**. Given two lists `(`l_1 `...` l_n`)` and `(`m_1 `...` m_k`)`, this function returns the list `((`l_1 `.` m_1`) ... (`l_n `.` m_k`))` if $n = k$, failure otherwise[2]. The function `transform-subs` applies one step of transformation (with respect to $\Rightarrow_s$) to a pair of systems of equations. The function `subs-system` iterates the

[2] For efficiency reasons, the function returns a multi–value with two elements, the first one returning the list of pairs and the second one returning a boolean denoting success or failure.

```
(defun transform-subs (S match)
  (let* ((equ (sel-match S)) (R (eliminate equ S))
         (t1 (car equ)) (t2 (cdr equ)))
    (cond ((variable-p t1)
           (let ((bound (assoc t1 match)))
              (if bound
                  (if (equal (cdr bound) t2)
                      (mv R match t)                       ;;; DELETE
                    (mv nil nil nil))                      ;;; FAIL-BIND
                (mv R (cons (cons t1 t2) match) t))))      ;;; BIND
          ((variable-p t2) (mv nil nil nil))               ;;; FAIL-VAR
          ((eql (car t1) (car t2))
           (mv-let (pairs bool)
                   (pair-args (cdr t1) (cdr t2))
                   (if bool
                       (mv (append empareja R) match t)    ;;; DECOMPOSE
                     (mv nil nil nil))))                   ;;; CONFLICT1
          (t (mv nil nil nil)))))                          ;;; CONFLICT2

(defun subs-system (S match bool)
  (declare (xargs :measure (length-system S)))
  (if (or (not bool) (not (consp S)))
      (mv S match bool)
    (mv-let (S1 match1 bool1)
            (transform-subs S match)
            (subs-system S1 match1 bool1))))

(defun match-mv (S)
  (mv-let (S1 sol1 bool1) (subs-system S nil t) (mv sol1 bool1)))
```

Fig. 1. A matching algorithm

application of these transformation steps, starting with a given pair of systems, until failure is detected or the first system is empty. The function `match-mv` applies `subs-system` starting with an empty system of initial bindings.

Note that `(match-mv S)` returns a multi-value consisting of two values: the second is a boolean indicating success or failure of matching. In case of success, the first value is a matcher of `S`[3]. The following theorems verify that this is indeed the behavior of `match-mv`:

```
(defthm match-mv-soundness
  (implies (second (match-mv S))
           (matcher (first (match-mv S)) S)))

(defthm match-mv-completeness
  (implies (matcher sigma S)
           (second (match-mv S))))
```

[3] Note the use of a multi–value to indicate success or failure in a separated value, in order to distinguish `nil` as failure from the empty substitution `nil`. This is often used in our formalization, sometimes for this reason and sometimes to distinguish `nil` as failure from the term `nil` (which represents a variable).

The proof of these theorems is clearly separated in two parts. First, we show that there are invariants that are preserved in each step of transformation: in this case, these invariants are the set of matchers of the pair of systems and the fact that the second system is a matcher of itself. Second, transformations are terminating: the number of symbols in the first system decreases in every transformation step. Thus if `match-mv` terminates with failure, the initial system of equations has no matchers; otherwise, the substitution obtained is a matcher of itself, and consequently it is a matcher of the initial system.

In a preliminary version of this work, a more classical subsumption algorithm defined recursively on the terms structure was verified, having the same properties as above. Nevertheless, when we verified a rule-based unification algorithm (section 6), we decided to design a a rule-based subsumption algorithm and apply the same techniques to its verification. We think that this rule-based approach turns out to be more suitable for mechanical verification, since the proof effort was reduced by the following facts[4] (see [2]):

– Termination aspects of the algorithm are clearly separated. We can reason about the algorithm before proving its termination.
– Properties of the algorithm are identified as invariants that are preserved in each step of transformation.
– A family of algorithms can be verified with the same effort, getting a clean separation of logic and control.

Let us explain more about the last point. In our definition of `transform-subs`, we use a function `sel-match` to select an equation from the system of equations to be matched. The selected equation determines the transformation rule to apply. In this particular case, `sel-match` tries to detect as soon as possible a failure in the matching process, because we intend to use this matching algorithm in contexts where matching fails most of times (e.g. in term rewriting). But any other selection function would be sound. In fact, we verified the algorithm with a partial definition of the selection function (only assuming, via `encapsulate`, that it selects an element from every non-empty list). The properties of a particular matching algorithm with a particular selection function are then easily proved by functional instantiation.

In the book `matching.lisp` we prove the properties of the general matching algorithm, and in `subsumption.lisp` we define `match-mv` and prove its properties by functional instantiation.

3.2 The subsumption quasi-ordering

As a particular case of `match-mv`, we can define a matching algorithm for pairs of terms `t1` and `t2`, named `subs-mv`, applying `match-mv` to `(list (cons t1 t2))`. For the sake of readability, we define the function `subs` to return the second value of `subs-mv` and the function `matching` to return the first value of `subs-mv`. Then we prove the following theorems, easy consequences of the properties of `match-mv`:

```
(defthm subs-soundness
  (implies (subs t1 t2)
           (equal (instance t1 (matching t1 t2)) t2)))

(defthm subs-completeness
  (implies (equal (instance t1 sigma) t2)
           (subs t1 t2)))
```

[4] Nevertheless, having said the advantages of a rule-based specification, it is also true that some operations acting on terms, like renaming variables or anti-unification of two terms are more naturally implemented by recursive definitions on the structure of the terms.

These two theorems state that `subs` defines the subsumption relation between terms: `(subs t1 t2)` if and only if there exists a substitution (given by `(matching t1 t2)`) such that applied to `t1` gives `t2`. In order to develop a theory independent from a particular implementation of a matching algorithm, it is remarkable that the theory we develop about the subsumption relation will be deduced exclusively from these two properties[5].

3.3 A well-founded quasi-ordering

The first basic fact that we can prove about the subsumption relation is that subsumption is a quasi-ordering:

```
(defthm subsumption-reflexive (subs t1 t1))

(defthm subsumption-transitive
   (implies (and (subs t1 t2) (subs t2 t3)) (subs t1 t3)))
```

Both properties are easily proved using a suitable instance of the theorem `subs-completeness`. For example, for the transitive property, we can prove `(subs t1 t3)` by giving the matching substitution `(composition (matching t2 t3) (matching t1 t2))`. This is a typical use of the completeness property of `subs` in our formalization.

We can also prove that (strict) subsumption is a well-founded ordering. The intuitive idea is that if `t1` strictly subsumes `t2`, then the size (number of symbols) of `t1` is less or equal than the size of `t2`. In case of equal sizes, the number of distinct variables is greater in `t1` than in `t2`, and this number is bounded by the number of variable positions of `t1` (or `t2`). The following function defines a lexicographic ordinal measure on terms reflecting this idea:

```
(defun subsumption-measure (term)
  (cons (1+ (size t term))
        (- (len (variables t term)) (len (make-set (variables t term))))))
```

The functions `size` and `variables` compute the size and the list of variables of a term, respectively, and they are defined recursively on the term structure (for terms and for lists of terms). The function `make-set` eliminate duplicates from a list. If we define `(strict-subs t1 t2)` as `(and (subs t1 t2) (not (subs t2 t1)))`, we can prove the following theorem:

```
(defthm subsumption-well-founded
  (and (e0-ordinalp (subsumption-measure t1))
       (implies (strict-subs t1 t2)
                (e0-ord-< (subsumption-measure t1)
                          (subsumption-measure t2)))))
```

In the meta-logic, this means that the subsumption relation is well-founded. This result, for example, can be used to prove properties of terms by induction based on subsumption. The proof of this theorem is somewhat elaborated and depends on our particular representation of terms as lists. See the book `subsumption-well-founded.lisp` for a detailed description of the proof.

4 Equivalent terms and renamings

We study in this section the equivalence relation induced by the subsumption quasi-ordering and we describe the verification of a function that renames the variables of a term.

[5] We also used a closure property about `subs` and `matching`. See section 7.

4.1 Renamings and renamed terms

The function `renamed` defines the relation $\equiv$ on first-order terms[6]:

```
(defun renamed (t1 t2)
  (if (subs t1 t2) (if (subs t2 t1) t nil) nil))
```

One interesting property of the function `renamed` is that `(renamed t1 t2)` holds if and only if there exists a renaming substitution (with its domain containing the variables of `t1`) such that applied to `t1` obtains `t2`. We describe now the formalization of this theorem. First, the function `renaming` defines the notion of renaming substitution:

```
(defun renaming (sigma)
  (and (variable-substitution sigma)
       (no-duplicatesp (co-domain sigma))))
```

Here `variable-substitution` defines variable substitutions; note also that we use `no-duplicatesp` to implement the notion of injective substitution on its domain. The following theorems prove the relation between `renaming` and `renamed`:

```
(defthm renaming-implies-renamed
  (implies (and (renaming sigma)
                (subsetp (variables t term) (domain sigma)))
           (renamed (instance term sigma) term)))

(defthm renamed-implies-renaming
  (let ((ren (normal-form-subst t (matching t1 t2) t1)))
    (implies (renamed t1 t2)
             (and (renaming ren)
                  (equal (instance t1 ren) t2)))))
```

The first theorem states that if we apply a renaming `sigma` to a term `term` whose variables are contained in its domain, we obtain an equivalent term w.r.t. subsumption. This theorem is easily proved showing that the inverse substitution of `sigma` is a matching substitution for `(instance term sigma)` and `term`.

The second theorem has a more complicated proof. It establishes that for every two equivalent terms `t1` and `t2` there exists renaming substitution such that applied to `t1` obtains `t2`. Note that in this case we cannot deduce that `(matching t1 t2)` is a renaming substitution (at least using only the soundness and completeness theorems about `subs`). But we can transform this substitution in order to obtain a renaming substitution that acts in the same way on `t1`. This transformation is done by `(normal-form-subst t (matching t1 t2) t1)` and, in essence, restricts the substitution to the variables of `t1`, considered without repetitions. See the details of this proof in the book `renamings.lisp`.

Another interesting point about `renamed` is that we can define it as an equivalence relation in ACL2, allowing congruence rewriting with respect to subsumption:

```
(defequiv renamed)

(defcong renamed iff (subs t1 t2) 1)

(defcong renamed iff (subs t1 t2) 2)
```

[6] Recall that we are only assuming the soundness and completeness properties of `subs`. We define the function `renamed` in this way (instead of the more natural `(and (subs t1 t2) (subs t2 t1))`) in order to force a boolean value, needed later to prove `(defequiv renamed)`.

From the theorem prover point of view, these three events allows rewriting of equivalent terms, in contexts where the subsumption relation appears. This congruence-based rewriting turns out to be very useful to mechanize the proof of some theorems, as we will describe in the following subsection.

4.2 Renaming the variables of a term

Later, when we define the most general instance of two terms, we will need to rename the variables of the terms, obtaining equivalent terms with separated variables (this is usually called *standardization apart*). In general, renaming variables of a term is an operation often used in automated reasoning. For that purpose, we define a function that renames the variable of a term to numeric variables (recall that variables are atomic objects). This function, called `number-rename` is presented in figure 2.

```
(defun number-rename-aux (flg term sigma x y)
  (if flg
      (if (variable-p term)
          (let ((find-term (assoc term sigma)))
            (if find-term
                (mv (cdr find-term) sigma)
              (let ((z (if (endp sigma) x (+ y (cdar sigma)))))
                (mv z (cons (cons term z) sigma)))))
        (mv-let (renamed-args renaming-args)
                (number-rename-aux nil (cdr term) sigma x y)
                (mv (cons (car term) renamed-args) renaming-args)))
    (if (endp term)
        (mv term sigma)
      (mv-let (renamed-car renaming-car)
              (number-rename-aux t (car term) sigma x y)
              (mv-let (renamed-cdr renaming-cdr)
                      (number-rename-aux nil (cdr term) renaming-car x y)
                      (mv (cons renamed-car renamed-cdr)
                          renaming-cdr))))))

(defun number-rename (term x y)
  (mv-let (renamed renaming)
          (number-rename-aux t term nil x y)
          renamed))
```

Fig. 2. A renaming algorithm

The expression `(number-rename term x y)` is the term obtained from `term` by replacing its variables by numbers, starting with the number `x`. Every time a new variable is found when traversing `term`, it is replaced by a new variable obtained adding `y` to the last numeric variable assigned. The function `number-rename-aux` implements this recursive process, for terms and for lists of terms; to take into account previous bindings, an extra parameter `sigma` is used. This function `number-rename-aux` returns as a multi–value the renamed term and the renaming substitution finally computed.

The main property of `number-rename` is that (when its third argument is not `0`) it obtains a renamed version of the original term. This is established in the following theorem:

```
(defthm number-renamed-term-renamed-term
  (implies (and (acl2-numberp x) (acl2-numberp y) (not (= y 0)))
           (renamed (number-rename term x y) term)))
```

In the book `renaming.lisp` it is described in detail the proof of this theorem, done by recursion on the term structure. It is remarkable that, since `renamed` is declared to be an equivalence relation, this theorem is stored as a rule to rewrite renamed terms to the original term, if this renamed term appears as an argument of the subsumption relation. This is very useful from the reasoning point of view, and makes easy to manage renamed terms in the proofs of the theorems.

Finally, we state a property that will be useful to reason about the process of standardization apart:

```
(defthm number-rename-standardization-apart
   (implies (and (acl2-numberp x1) (acl2-numberp x2)
                 (< x1 x2) (< y1 0) (< 0 y2))
            (disjointp (variables t (number-rename t1 x1 y1))
                       (variables t (number-rename t2 x2 y2)))))
```

Given two terms `t1` and `t2`, this theorem provides a way to obtain two respective equivalent terms with separated variables: rename `t1` starting in the number `x1`, using a negative increment, and rename `t2` starting in a greater number `x2`, using a positive increment. Later we will use this procedure to standardize apart two terms.

5 Anti-unification: most specific generalizations

In this section we show that the set of first-order terms is a lower semi-lattice with respect to subsumption. For that purpose we define and verify an algorithm that, given two terms, finds a most specific term that subsumes both of them (i.e., a greatest lower bound w.r.t. subsumption). This is what is usually called an *anti-unification* algorithm (because it can be seen as the converse of unification), an important process in machine learning theory.

We can describe anti-unification as follows. Let ϕ an injective function from $T(\Sigma, X) \times T(\Sigma, X)$ to X. We define the binary operation $\wedge_\phi$ in $T(\Sigma, X)$ such that $s \wedge_\phi t = f(s_1 \wedge_\phi t_1, \dots, s_n \wedge_\phi t_n)$, if $s = f(s_1, \dots, s_n)$ and $t = f(t_1, \dots, t_n)$ or $s \wedge_\phi t = \phi(s, t)$ otherwise. It can be proved, as we will show, that $s \wedge_\phi t$ is a most specific generalization of s and t.

In figure 5, we present the definition of a function `anti-unify` implementing an anti-unification algorithm, inspired on the definition of $\wedge_\phi$. Given two terms, this function traverses them, collecting their common structure. The main function is `anti-unify-aux`, defined recursively on the structure of terms (and lists of terms). This function has an extra argument `phi`, an association list binding pairs of terms to numeric variables. The main difference with $\wedge_\phi$ is that this association list is built incrementally: if during the anti-unification process we need to assign a new variable to a pair of terms, the association list is extended by the new binding (new variables are obtained sequentially from the natural numbers); if the same pair of terms is found later, the same variable has to be assigned[7]. The function `anti-unify-aux` returns three values: the first is the term computed, the second is the association list with all the bindings needed during the process, and the third is a boolean value indicating whether the anti-unification was successful or not. Note that, although

[7] Given a pair of terms `p` and an association list `phi`, the function `anti-unify-injection` (in figure 3) implements this process. It returns a multi–value with the numeric variable assigned to `p` and the association list (possibly extended with a new binding). For coherence with the ACL2 signature of `anti-unify-aux`, it also returns the boolean value `t`.

```
(defun anti-unify-injection (p phi)
  (let ((found (assoc-equal p phi)))
    (if found
        (mv (cdr found) phi t)
      (let ((y (if (endp phi) 1 (+ 1 (cdar phi)))))
        (mv y (cons (cons p y) phi) t)))))

(defun anti-unify-aux (flg t1 t2 phi)
  (if flg
      (cond ((or (variable-p t1) (variable-p t2))
             (anti-unify-injection (cons t1 t2) phi))
            ((eql (car t1) (car t2))
             (mv-let (anti-unify-args phi1 bool)
                     (anti-unify-aux nil (cdr t1) (cdr t2) phi)
                     (if bool
                         (mv (cons (car t1) anti-unify-args) phi1 t)
                       (anti-unify-injection (cons t1 t2) phi))))
            (t (anti-unify-injection (cons t1 t2) phi)))
    (cond  ((endp t1) (if (eql t1 t2) (mv t1 phi t) (mv nil nil nil)))
           ((endp t2) (mv nil nil nil))
           (t (mv-let (anti-unify-cdr phi1 bool1)
                      (anti-unify-aux nil (cdr t1) (cdr t2) phi)
                      (if bool1
                          (mv-let (anti-unify-car phi2 bool2)
                                  (anti-unify-aux t (car t1) (car t2) phi1)
                                  (if bool2
                                      (mv (cons anti-unify-car anti-unify-cdr)
                                          phi2
                                          t)
                                    (mv nil nil nil)))
                        (mv nil nil nil)))))))

(defun anti-unify (t1 t2)
  (mv-let (anti-unify phi bool)
          (anti-unify-aux t t1 t2 nil)
          anti-unify))
```

Fig. 3. Anti–unification algorithm

anti-unification never fails for terms, it may fail for lists of terms (for example, if the lists of terms have distinct lengths). Finally, `anti-unify` calls `anti-unify-aux` with `flg`=`t`, and using `nil` as the initial association list. Here are several examples of anti-unifications:

```
ACL2 !>(anti-unify '(f (h y) x (h y)) '(f (g z) (g z) (g z)))
(F 1 2 1)
ACL2 !>(anti-unify '(f (h y) x (h y)) '(g (g z) (g z) (g z)))
1
ACL2 !>(anti-unify '(f (h (k u)) x (h y)) '(f (h u) (g z) (h z)))
(F (H 3) 2 (H 1))
```

The following two theorems establish that `anti-unify` computes a greatest lower bound (i.e., a most specific generalization) of two terms, with respect to the subsumption relation:

```
(defthm anti-unify-lower-bound
  (and (subs (anti-unify t1 t2) t1)
       (subs (anti-unify t1 t2) t2)))

(defthm anti-unify-greatest-lower-bound
   (implies (and (subs term t1)
                 (subs term t2))
            (subs term (anti-unify t1 t2))))
```

In a first attempt, we tried to prove these two theorems reasoning directly about the definition in figure 3. Nevertheless, it turned out surprisingly difficult (in fact, we were not able to prove the theorem `anti-unify-greatest-lower-bound`). The main problem was to deal with the incremental construction of the association list `phi`. Thus, we tried a different approach, applying a technique that we think constitutes an interesting example of compositional reasoning: we prove properties of a simplified version of the algorithm, translating later the proved properties. We now sketch the main points of the proof strategy we followed:

- Instead of reasoning directly about `(anti-unify-aux flg t1 t2 phi)`, we define the expression `(pre-anti-unify-aux flg t1 t2 phi)`. We omit here the definition of `pre-anti--unify-aux` (see the supporting materials), but it is enough to say that is defined in the same way as `anti-unify-aux`, *except that the association list* `phi` *is considered to be fixed during all the recursive process.* That is, every time a variable has to be assigned to a pair of terms, it is supposed that this pair is already bound in `phi`.
- It is considerably easier to reason about `pre-anti-unify-aux` than to reason about `anti--unify-aux`. Nevertheless, these two functions are not equal: in general, they do not compute the same values. But, under some restrictions on `phi`, we can prove that `pre-anti-unify-aux` computes the most specific generalization of two terms (or lists of terms). These restrictions have to ensure that `phi` represents an injective function (with finite domain) from pairs of terms to variables, and that the domain of `phi` contains all the pairs of terms that are encountered during the anti-unification process. The function `injection-p` formalizes this idea:

  ```
  (defun injection-p (phi flg t1 t2)
    (and (alistp phi)
         (no-duplicatesp (co-domain phi))
         (list-of-variables-p (co-domain phi))
         (mv-let (term phi1 bool)
                 (anti-unify-aux flg t1 t2 nil)
                 (subsetp (domain phi1) (domain phi)))))
  ```

  Note that we use the second value returned by `anti-unify-aux` to obtain all the pairs of terms that are assigned a variable during the unification process. Assuming the condition given by `injection-p`, we can prove that `pre-anti-unify-aux` computes a most specific generalization of `t1` and `t2`. This proof is done without too much effort (see the supporting materials).
- Now we can translate the properties about `pre-anti-unify-aux`, obtaining analogous properties about `anti-unify-aux`. First, we prove that if `pre-anti-unify-aux` uses a specific association list (the one computed by `anti-unify-aux`), it computes the same term than `anti-unify-aux`:

  ```
  (defthm anti-unify-aux-pre-anti-unify-aux
    (equal (first (anti-unify-aux flg t1 t2 phi))
           (first (pre-anti-unify-aux
                    flg t1 t2 (second (anti-unify-aux flg t1 t2 phi))))))
  ```

Second, we prove that the association list computed by `anti-unify-aux` verifies the property `injection-p`:

```
(defthm anti-unify-aux-injection-injection-p
  (implies (and (alistp phi)
                (acl2-numberp-increasing-list (co-domain phi)))
           (injection-p
             (second (anti-unify-aux flg t1 t2 phi)) flg t1 t2))))
```

Independently of the meaning of the hypothesis of this theorem, the main point here is that these conditions are trivially satisfied by `nil`, which is the initial association list used in the definition of `anti-unify`.
Finally, the argument is closed: `(anti-unify t1 t2)` is equal to the term computed by `(pre--anti-unify-aux t t1 t2 (second (anti-unify-aux t t1 t2 nil)))`. But this term has been proved to be a most specific generalization of `t1` and `t2` (since the association list used verifies `injection-p`).

All the events shown in this section are in the book `anti-unification.lisp`. We urge the reader to complete the details of the proof, reading the documentation of the book.

6 Unification and most general instances

Up to now, we have proved that the relation `subs` provides to the set of first-order terms a structure of well-founded lower semi-lattice. In the meta-logic, using properties of well-founded lattices, this implies that every pair of terms that have a common instance, have a least upper bound (i.e., a most general common instance). Nevertheless, we are going to prove this fact in a constructive way, defining a function that returns a most general common instance of two given unifiable terms, and to verify its properties.

For that purpose, we will need to define and verify a unification algorithm. This algorithm returns a most general unifier of two given terms, whenever it exists. Unification is a central process in many applications of automated reasoning. In order to establish the properties of a unification algorithm, we also have to define subsumption between substitutions and idempotency. In the following we develop all these questions.

6.1 Subsumption between substitutions.

Recall that the subsumption relation between substitutions is defined as $\sigma \leq \delta \iff \exists \gamma(\gamma \circ \sigma = \delta)$. Our problem now is to give a constructive definition of this concept. Let $V = \{v_1, \ldots, v_n\}$ be a set of variables containing the domain of σ, the domain of δ and the variables of the codomain of σ (the *important variables* of σ and δ). It can be proved that $\sigma \leq \delta$ if and only if there exists a matcher of the system $S = \{\sigma(v_1) \approx \delta(v_1), \ldots, \sigma(v_n) \approx \delta(v_n)\}$. With this idea, we can use our matching algorithm `match-mv` to define a function `subs-subst` implementing the subsumption relation between substitutions:

```
(defun subs-subst (sigma delta)
  (let ((V (important-variables sigma delta)))
    (mv-let (system bool1)
            (pair-args (apply-subst nil sigma V) (apply-subst nil delta V))
            (mv-let (match bool2) (match-mv system) bool2))))
```

The main property of `subs-subst` is stated by the following theorem:

```
(defthm subs-subst-soundness
  (implies (subs-subst sigma delta)
           (equal (instance
                   term
                   (composition (matching-subst-r sigma delta) sigma))
                  (instance term delta))))
```

There are two remarkable points in this theorem. First, the matching substitution for `sigma` and `delta` is given by the expression `(matching-subst-r sigma delta)`, which plays the role of the existentially quantified substitution γ in the definition above. This substitution is a matcher of the system S above, restricted to the variables of V. Second, equality of the substitutions $\gamma \circ \sigma$ and δ is stated from a functional point of view.

We also proved the converse of this theorem, thus proving that `subs-subst` implements the notion of subsumption between substitutions. See the book `subsumption-subst.lisp` for details of both proofs.

6.2 Idempotent substitutions.

Note that substitutions may also be seen as system of equations. In fact, in our formalization they are represented in the same way. Recall that a solution of a system of equations is a unifier of all its equations. The following is the main lemma for expressing the relationship between subsumption and solution of systems, and exploits that substitutions may also be seen as systems:

```
(defthm substitutions-solution-system
  (implies (solution sigma delta)
           (equal (apply-subst flg sigma (apply-subst flg delta term))
                  (apply-subst flg sigma term))))
```

In other words, if σ is a solution of δ, then $\sigma = \sigma \circ \delta$, and, consequently $\delta \leq \sigma$. This means that if δ is a solution of itself, it is the least such solution with respect to subsumption.

A crucial point in the verification of the unification algorithm below is that the substitution returned is *idempotent*. Usually, a substitution σ is defined to be idempotent if $\sigma \circ \sigma = \sigma$. We will adopt an equivalent definition, considering that a substitution is idempotent if its domain is disjoint with the variables of its codomain. Using this alternative formulation, we defined a function `idempotent`. The main property of idempotent substitutions is the following:

```
(defthm idempotence
  (implies (idempotent sigma) (solution sigma sigma))))
```

Thus, with this theorem and the theorem `substitutions-solution-system` above, we can conclude that every idempotent substitution is a most general unifier of itself. This is an important property that will be exploited in the verification of the unification algorithm.

6.3 Mechanical verification of a rule-based unification algorithm

We define a rule-based unification algorithm based on the well-known transformation system of Martelli-Montanari [1].

Delete:	$\{t \approx t\} \cup R; T$	$\Rightarrow_u R; T$
Decompose:	$\{f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)\} \cup R; T$	$\Rightarrow_u \{s_1 \approx t_1, \ldots, s_n \approx t_n\} \cup R; T$
Conflict:	$\{f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m)\} \cup R; T$	$\Rightarrow_u$ `nil` **if** $f \neq g$ **or** $n \neq m$
Orient:	$\{t \approx x\} \cup R; T$	$\Rightarrow_u \{x \approx t\} \cup R; T$ **if** $x \in X$ **and** $t \notin X$
Check:	$\{x \approx t\} \cup R; T$	$\Rightarrow_u$ `nil` **if** $x \in \mathcal{V}(t)$ **and** $x \neq t$
Eliminate:	$\{x \approx t\} \cup R; T$	$\Rightarrow_u \{x \mapsto t\}R; \{x \approx t\} \cup \{x \mapsto t\}T$ **if** $x \in X$ **and** $x \notin \mathcal{V}(t)$

These rules act on two systems of equations: the first one with the equations to be solved and the second one with the unifier partially computed. As with the matching algorithm described in section 3, to compute a most general solution of a system S, we start with the pair of systems $S; \emptyset$ and apply the rules (non-deterministically) until `nil` is found or the first system is empty. In the first case, the system S is unsolvable. In the latter case, the second system is a most general solution of S. To obtain a most general unifier of two terms t_1 and t_2, we start with the pair of systems $\{t_1 \approx t_2\}; \emptyset$.

In a similar way as we implemented the matching algorithm `match-mv`, we define this rule-based unification algorithm, by means of a function `mgu-mv`, whose definition appears in figure 4.

```
(defun transform-mm (S sol)
  (let* ((ecu (sel-unif S)) (R (eliminate ecu S))
         (t1 (car ecu)) (t2 (cdr ecu)))
    (cond ((equal t1 t2) (mv R sol t))                     ;;; DELETE
          ((variable-p t1)
           (if (member t1 (variables t t2))
               (mv nil nil nil)                            ;;; OCCUR-CHECK
             (mv (substitute-syst t1 t2 R)                 ;;; ELIMINATE
                 (cons ecu (substitute-range t1 t2 sol))
                 t)))
          ((variable-p t2)
           (mv (cons (cons t2 t1) R) sol t))               ;;; ORIENT
          ((not (eql (car t1) (car t2)))
           (mv nil nil nil))                               ;;; CONFLICT1
          (t (mv-let (pairs bool)
                     (pair-args (cdr t1) (cdr t2))
                     (if bool
                         (mv (append pairs R) sol t)       ;;; DECOMPOSE
                       (mv nil nil nil)))))))              ;;; CONFLICT2

(defun solve-system (S sol bool)
  (declare (xargs :measure (unification-measure (cons S sol))))
  (if (or (not bool) (not (consp S)))
      (mv S sol bool)
    (mv-let (S1 sol1 bool1)
            (transform-mm S sol)
            (solve-system S1 sol1 bool1))))

(defun mgs-mv (S)
  (mv-let (S1 sol1 bool1) (solve-system S nil t) (mv sol1 bool1)))

(defun mgu-mv (t1 t2) (mgs-mv (list (cons t1 t2))))
```

Fig. 4. A unification algorithm

Note that `mgu-mv` returns two values: the second is a boolean value denoting success or failure. In case of success, the first value is a most general and idempotent unifier of the input terms, as established by the following theorems, proving the correctness of the unification algorithm defined by `mgu-mv`. To improve readability, we define the function `mgu` as the first value of `mgu-mv` and `unifiable` as the second value of `mgu-mv`:

```
(defthm mgu-completeness
  (implies (equal (instance t1 sigma) (instance t2 sigma))
           (unifiable t1 t2)))

(defthm mgu-soundness
  (implies (unifiable t1 t2)
           (equal (instance t1 (mgu t1 t2)) (instance t2 (mgu t1 t2)))))

(defthm mgu-idempotent
  (idempotent (mgu t1 t2)))

(defthm mgu-most-general-unifier
  (implies (equal (instance t1 sigma) (instance t2 sigma))
           (subs-subst (mgu t1 t2) sigma)))
```

Although the proofs are considerably larger than those of properties of subsumption, it benefits from the same rule-based approach advantages:

– An abstract non-deterministic unification algorithm is verified first, without defining a concrete selection strategy.
– The proof is separated in two stages:
  - Prove that certain properties are preserved in each step of transformation: the set of solutions of the systems and the fact that the second system is idempotent.
  - Prove that the transformation rules terminate: we show that a certain lexicographic measure decreases in each step.

  Thus, when `nil` is detected, the initial system of equations is unsolvable. Otherwise, the algorithm finds a idempotent system which is equivalent to the initial one (i.e. it has the same set of solutions). By the properties about idempotent substitutions described in subsection 6.2, this system is a most general solution of itself, and therefore a most general solution of the initial system.
– The function `mgu-mv` and its properties can be obtained by functional instantiation from the abstract non-deterministic algorithm, simply defining a concrete selection strategy (given by the function `sel-unif`, whose definition we omit here), making a clear distinction between logic and control.

Due to the lack of space, we do not comment more about the proof here. For a more complete description of a preliminary proof in Nqthm, see [10]. The book `unification-pattern.lisp` contains the proof of the correctness of the generic non-deterministic unification algorithm, and in the book `unification.lisp` is the proof (by functional instantiation) of the properties about `mgu-mv` given above.

6.4 Most general instance of two terms.

Now we define a function that finds, whenever it exists, a least upper bound (with respect to subsumption) of two given terms. We prove that a most general instance of two terms s and t, if it exists, it is $\sigma(s')$, where s' and t' are two renamed terms, respectively, of s and t, with no variables in common, and σ is a most general unifier of s' and t'. This idea is implemented by the function `mg-instance`:

```
(defun mg-instance-mv (t1 t2)
  (let ((rename-t1 (number-rename t1 0 -1))
        (rename-t2 (number-rename t2 1 1)))
    (mv-let (mgu unifiable)
            (mgu-mv rename-t1 rename-t2)
            (if unifiable (mv (instance rename-t1 mgu) t) (mv nil nil)))))

(defun mg-instance (t1 t2)
  (mv-let (mg-instance bool)
          (mg-instance-mv t1 t2)
    (if bool (number-rename mg-instance 1 1) nil)))
```

Note that the terms `t1` and `t2` are standardized apart, by means of `(number-rename t1 0 -1)` and `(number-rename t2 1 1)`. As it was shown in section 4.2, equivalent terms with no variables in common are obtained in this way. The function `mg-instance-mv` applies the procedure described above and it returns two values. The second is a boolean value indicating failure or success. In case of success, the first value contains the term computed. For the sake of readability, we define `mg-instance`, a function that in case of success of `mg-instance-mv` renames the term with numeric variables; in case of failure it returns `nil` otherwise. Note that in this case there are no confusion between `nil` as term and `nil` as failure (since a renamed term cannot be `nil`). Here there are some examples of most general instances:

```
ACL2 !>(mg-instance '(f x (h y)) '(f (k u) u))
(F (K (H 1)) (H 1))
ACL2 !>(mg-instance '(f x (h x)) '(f (k u) u))
NIL
ACL2 !>(mg-instance '(f u v u v u) '(f x y x x y))
(F 1 1 1 1 1)
ACL2 !>(mg-instance '(f u v u v u) '(f x y x x y z))
NIL
```

The following theorems establishes that `mg-instance` returns a most general common instance of two terms if and only if they have an upper bound. The theorems were easily proved from the properties of the unification and renaming algorithms.

```
(defthm common-instance-implies-mg-instance
  (implies (and (subs t1 term) (subs t2 term))
           (mg-instance t1 t2)))

(defthm mg-instance-upper-bound
  (implies (mg-instance t1 t2)
           (and (subs t1 (mg-instance t1 t2))
                (subs t2 (mg-instance t1 t2)))))

(defthm mg-instance-least-upper-bound
  (implies (and (subs t1 term) (subs t2 term))
           (subs (mg-instance t1 t2) term)))
```

The mechanical proofs of these theorems constitutes a typical example of the use of congruence-rewriting with respect to the `renamed` equivalence relation. Renamed terms are used to separate variables, but the theorem prover rewrites these renamed terms to the original terms, when these terms appear as arguments of `subs`. This turns out to be very useful in the mechanization of the proofs. See the book `mg-instance.lisp` for details.

7 Closure properties and guard verification

The theorems presented in the previous sections show that the functions `anti-unify` and `mg-instance` computes, respectively, a greatest lower bound and a least upper bound (whenever it exists) of two given terms. Our goal is to show that the set of first-order terms *in a given signature* (plus an additional top element) is a well-founded lattice with respect to subsumption. Recall that our theorems do not need hypothesis regarding the "type" of the variables involved. But for "theoretical completeness", we have to prove that those lattice operations are closed in the set of terms in a given signature.

For that purpose, we define a predicate describing those ACL2 objects that represent first-order terms in a signature. The first question is how we represent signatures. To be as general as possible, we define a signature as a general binary function `signat`:

```
(defstub signat (* *) => *)
```

A signature is defined to be a generic function of two arguments. The intended meaning is that receiving as input a function symbol `f` and a natural number `n`, it is returned `t` if the arity of the symbol `f` is `n`, and `nil` otherwise. This allows even to represent infinite signatures and variadic function symbols.

We now define the macro `terms-s-p`, describing ACL2 objects that represent "proper" first-order terms in the general signature described by `signat`. Note that we also require that the variables of a proper term are `eqlablep` objects. Also, the list of arguments of a non-variable proper term has to be a true list.

```
(defun term-s-p-aux (flg x)
  (if flg
      (if (atom x)
          (eqlablep x)
        (if (signat (car x) (len (cdr x)))
            (term-s-p-aux nil (cdr x))
          nil))
    (if (atom x)
        (equal x nil)
      (and (term-s-p-aux t (car x))
           (term-s-p-aux nil (cdr x))))))

(defmacro term-s-p (x) `(term-s-p-aux t ,x))
```

Similarly, `substitution-s-p` can be defined to describe the objects that represent substitutions in a given signature (see `terms.lisp`). Having defined `term-s-p`, we prove that the operations on terms defined in the previous sections are closed in the domain described by `term-s-p`. For example:

```
(defthm number-rename-term-s-p
  (implies (and (term-s-p term) (acl2-numberp x))
           (term-s-p (number-rename term x y))))
```

We proved analogous closure properties for all the operation on terms and substitutions described in this paper. In particular, for the functions `anti-unify` and `mg-instance`, thus showing that the lattice operations are closed in the set of terms in a given signature.

Let us now see how these closure properties turn out to be useful for guard verification. Recall that a function with a verified guard can be executed in any compliant Common Lisp (with the appropriate files loaded) on any arguments the satisfy its guard. The guard of a function is a formula specifying the intended domain of the function. In our functions acting on terms, the intended domain is defined by `term-p`:

```
(defun term-p-aux (flg x)
  (declare (xargs :guard t))
  (if flg
      (if (atom x)
          (eqlablep x)
        (and (eqlablep (car x))
             (term-p-aux nil (cdr x))))
    (if (atom x)
        (equal x nil)
      (and (term-p-aux t (car x))
           (term-p-aux nil (cdr x))))))

(defmacro term-p (x) `(term-p-aux t ,x))
```

Since variables are forced to be `eqlablep`, we can use the more efficient `eql`, instead of `equal`. Since lists of arguments are true lists, we can use `endp` instead of `atom`. Similarly, `substitution-p` is defined to specify those ACL2 objects expected by the functions that deal with substitutions (see `terms.lisp`). We verified the guards of all the executable functions presented in this paper (except those that define concepts).

Guard verification theorems are very similar to the theorems about closure properties. For example, since `mgu-mv` has guard `(and (term-p t1) (term-p t2))`, in order to verify the same guard for `mg-instance`, we have to prove:

```
(defthm number-rename-term-p
  (implies (and (term-p term) (acl2-numberp x))
           (term-p (number-rename term x y))))
```

A remarkable point is that we can easily obtain this guard verification theorems from the analogous closure properties, by functional instantiation. Note that `term-p` can be seen as a function defining the terms in a specific signature. Namely, the signature described by the function `(lambda (x n) (eqlablep x))`. Thus, closure properties and guard verification theorems are proved with the same effort.

8 The subsumption lattice of first order terms.

As a mathematical recreation, we compile in this section the previous properties and prove that, $\langle T(\Sigma, X) \cup \{\top\}, \leq\rangle$ is a well-founded lattice (where $\top$ is an additional top element). See the book `lattice-of-terms.lisp` for details.

We distinguish the ACL2 object `'the-top-term`, representing an additional top element. The following are some of our previous definitions reformulated to take into account this new object:

```
(defmacro the-top-term () ''the-top-term)
(defmacro is-the-top-term (term) `(equal ,term (the-top-term)))

(defun ext-term-s-p (term)
  (or (term-s-p term) (is-the-top-term term)))

(defun s<= (t1 t2)
  (cond  ((is-the-top-term t2) t)
         ((is-the-top-term t1) nil)
         (t (subs t1 t2))))
```

```
(defun glb-s<= (t1 t2)
  (cond ((is-the-top-term t1) t2)
        ((is-the-top-term t2) t1)
        (t (fix-term (anti-unify t1 t2)))))

(defun lub-s<= (t1 t2)
  (cond ((or (is-the-top-term t1) (is-the-top-term t2)) (the-top-term))
        ((mg-instance t1 t2) (fix-term (mg-instance t1 t2)))
        (t (the-top-term))))
```

The function `ext-term-s-p` defines the set $T(\Sigma, X) \cup \{\top\}$ and the function `s<=` defines the subsumption relation in that set. The functions `glb-s<=` and `lub-s<=`, are respectively the greatest lower bound and least upper bound operations in $T(\Sigma, X) \cup \{\top\}$[8].

Finally, the following properties are a compilation of the work presented in this paper and prove that the set of first-order terms in a given signature (plus an additional top element) is a well-founded-lattice with respect to `s<=`[9].

```
(defthm s<=-quasi-ordering
 (and (s<= term term)
      (implies (and (s<= t1 t2) (s<= t2 t3)) (s<= t1 t3))))

(defun measure-s< (term)
  (if (is-the-top-term term)
      '((1 . 0) . 0)
    (subsumption-measure term)))

(defthm s<-well-founded
 (and (e0-ordinalp (measure-s< term))
      (implies (and (s<= t1 t2) (not (s<= t2 t1)))
               (e0-ord-< (measure-s< t1) (measure-s< t2)))))

(defthm glb-s<=-is-a-glb
 (and (s<= (glb-s<= t1 t2) t1) (s<= (glb-s<= t1 t2) t2)
      (implies (and (s<= term t1) (s<= term t2))
               (s<= term (glb-s<= t1 t2)))))

(defthm lub-s<=-is-a-lub
 (and (s<= t1 (lub-s<= t1 t2)) (s<= t2 (lub-s<= t1 t2))
      (implies (and (s<= t1 term) (s<= t2 term))
               (s<= (lub-s<= t1 t2) term))))

(defthm glb-<=-lub-<=-closure-properties
  (implies (and (ext-term-s-p t1) (ext-term-s-p t2))
           (and (ext-term-s-p (glb-s<= t1 t2))
                (ext-term-s-p (lub-s<= t1 t2)))))
```

In fact, the theorems above establishes that the set of all ACL2 objects (seen as first-order terms in a wide sense) is a well-founded lattice with respect to subsumption. And that the set

[8] When `(anti-unify t1 t2)` is a variable, we need to prove that it is different from `'the-top-term`. To avoid problems, the function `fix-term` forces it to be `0`, which is an equivalent term and is obviously distinct from `'the-top-term`. Analogously for `mg-instance`.

[9] Note that the definition of `measure-s<` is the same as `subsumption-measure`, but a higher ordinal (for example, ω^{ω}) is assigned to `'the-top-term`.

of those ACL2 objects representing first-order terms in a given signature is a sublattice of it. It is remarkable that the proof is constructive, and that, more important from a practical point of view, all the functions defined in this section are executable in any compliant Common Lisp. See the book `lattice-of-terms` for details.

9 Conclusion and future work

We have seen a formalization in ACL2 of the lattice-theoretic properties of first-order terms with respect to subsumption. It has been defined the subsumption relation between terms, and showed that subsumption is a well-founded quasi-ordering. We also showed that every pair of terms has a most specific generalization and that every pair of unifiable terms has a most general instance. Thus, we have proved in ACL2 that the set of first-order terms in a given signature (plus an additional top term) is a well-founded lattice with respect to subsumption.

From a practical point of view, as a by-product we obtained verified implementations of some basic functions acting on terms, including matching, renaming of variables, unification and anti-unification. These functions can be executed in any compliant Common Lisp.

Note that we have used a definition of lattice based on a quasi-ordering (subsumption in this case). An alternative equivalent equational definition of lattice could have been used, but we preferred this formulation because the properties proved are more used in the context of automated deduction and rewriting. It would be interesting to know if this alternative definition of lattice would simplify the proof.

Table 1 gives some quantitative information on the proofs. The first column contains the name of the books. The next three columns show the number of lines, the number of definitions and the number of theorems in each book. These numbers can give an idea of the granularity of our proofs. We also included a fifth column with the number of theorems that needed hints from the user. Together with the number of theorems, this can give an idea of the degree of automation of the proofs. It is clear from the table that the main proof effort was done to prove the properties of the unification algorithm.

Book	Lines	Definitions	Theorems	Hints
`basic`	378	22	79	2
`terms`	770	53	76	12
`matching`	325	7	48	8
`subsumption`	295	13	29	18
`subsumption-subst`	327	16	38	13
`renamings`	578	9	64	25
`subsumption-well-founded`	216	3	30	7
`anti-unification`	434	10	37	6
`unification-pattern`	808	7	105	33
`unification`	277	12	24	8
`mg-instance`	159	3	17	11
`lattice-of-terms`	148	17	20	5
Total	4715	172	567	148

Table 1. Quantitative information

As usual in a typical ACL2 formalization, most of the hints given are for enabling or disabling rules and for using instances of previous lemmas. It is remarkable the good behavior of the theorem prover in the automatization of proofs by induction, especially when an induction on the structure of terms is needed. The induction scheme guessed by ACL2 turns out to be suitable in most cases: as we said in section 2, this scheme is suggested by the functions we defined in a mutual recursive style, for terms and for lists of terms.

This library has already been used in a formalization of equational reasoning and term rewriting system [11], and that was our purpose when we developed it. But we believe that it can help in other verification projects to formalize properties of reasoning systems dealing with first-order logic (automated deduction, logic programming, machine learning,...).

This work provides a good example of how computing and proving tasks can be intermixed, and we think that this library can be seen as a basis to build prototypes of reasoning systems that can be (at least partially) verified. Another direction for future work could be the development of more efficient data structures for dealing with term structures. Single-threaded objects in ACL2 seem to be a promising option for that purpose.

References

1. Baader, F. and Snyder, W. Unification theory. *Handbook of Automated Reasoning*, Elvesier Science Publishers, 2001.
2. Gallier, J., and Snyder, W. Designing unification procedures using transformations: A survey. *Bulletin of the EATCS*, 40 (1990), 273–326.
3. Huet, G. *Résolution d'equations dans les langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Univ. Paris VII, 1976 (in french).
4. Kaufmann, M. Generalization in the presence of free variables: A mechanically-checked proof for one algorithm. *Journal of Automated Reasoning 7*, 1 (1991), 109–158.
5. McCune, W., and Shumsky, O. Ivy: A preprocessor and proof checker for first-order logic. In M. Kaufmann, P. Manolios and J S. Moore, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, ch. 16. Kluwer Academic Publishers, 2000.
6. Paulson, L. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5 (1985).
7. Plotkin, G. D. A note on inductive generalization. In *Machine Intelligence, 5.* 1970, pp. 153–163.
8. Rouyer, J. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Tech. Rep. 1795, INRIA Lorraine, 1992 (in french).
9. Ruiz-Reina, J.L. *Una teoría ecuacional acerca de la lógica ecuacional.* PhD thesis, Universidad de Sevilla, 2001 (in spanish).
10. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., and Martín, F.J. Mechanical verification of a rule based unification algorithm in the Boyer-Moore theorem prover. In *AGP'99 Joint Conference on Declarative Programming* (1999), pp. 289–304.
11. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., and Martín, F.J. Formalizing rewriting in the ACL2 theorem prover. In *AISC'2000 (Fifth International Conference Artificial Intelligence and Symbolic Computation)*, LNAI 1930, pp. 92–103. Springer-Verlag, 2001.
12. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., and Martín, F.J. Formalizing equational reasoning and rewriting (using ACL2). URL: `www.cs.us.es/~jruiz/acl2-rewr`, 2001.