# Formal proofs about rewriting using ACL2

José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo and Francisco-Jesús Martín-Mateos

*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*
E-mails: {jruiz,jalonso,mjoseh,fmartin}@cs.us.es

We present an application of the ACL2 theorem prover to reason about rewrite systems theory. We describe the formalization and representation aspects of our work using the first-order, quantifier-free logic of ACL2 and we sketch some of the main points of the proof effort. First, we present a formalization of abstract reduction systems and then we show how this abstraction can be instantiated to establish results about term rewriting. The main theorems we mechanically proved are Newman's lemma (for abstract reductions) and Knuth–Bendix critical pair theorem (for term rewriting).

**Keywords:** theorem proving, ACL2, rewriting, formal verification

## 1. Introduction

Formal, mechanically checked proofs not only provide verification of mathematical results but encourage closer examination and deeper understanding of those results. We report in this paper the status of our work on the application of the ACL2 theorem prover to reason about abstract reductions and term rewriting systems theory; confluence, local confluence, Noetherianity, normal forms and other related concepts have been formalized in the ACL2 logic and some results about abstract reductions and term rewriting have been mechanically proved, including Newman's lemma and Knuth–Bendix critical pair theorem.

ACL2 [8] is both a logic and a mechanical theorem proving system supporting it, developed by J Moore and M. Kaufmann. The ACL2 logic is an existentially quantifier-free, first-order logic with equality. ACL2 is also a programming language, an applicative subset of Common Lisp. The system evolved from the Boyer–Moore theorem prover, also known as Nqthm.

The notion of rewriting or simplification is a crucial component in symbolic computation: simplification procedures are needed to transform complex objects in order to obtain equivalent but simpler objects and to compute unique representations for equivalence classes (see, for example, [5]). Since ACL2 is also a programming language, this work can be seen as a first step to obtain verified executable (and efficient, if possible)

This work has been supported by DGES/MEC: Project TIC2000-1368-CO3-02.

Common Lisp code for components of symbolic computation systems and equational theorem provers. Although a fully verified implementation of such a system is currently impractical, several basic algorithms can be mechanically "certified" and integrated as part of the whole system.

We also show here how a weak logic like the ACL2 logic (no quantification, no infinite objects, no higher order variables, etc.) can be used to represent, formalize, and mechanically prove nontrivial theorems. In this paper, we place emphasis on describing the formalization and representation aspects of our work and we also highlight some of the main points of the proof effort. Due to the lack of space we will skip details of the mechanical proofs and for the same reason some function definitions will be omitted. We urge the interested reader to see the complete development, available on the web at URL `http://www.cs.us.es/~jruiz/acl2-rewr`. This paper is an extended and revised version of [15,17].

The rest of the paper is organized as follows. Sections 1.1 and 1.2 present a brief description of ACL2 and an informal presentation of the theory of abstract reductions and term rewriting, respectively. In section 2 we present a formalization of abstract reductions in the ACL2 logic, including a proof of Newman's lemma. In section 3 we describe the instantiation of the abstract formalization presented in the previous section to the case of term rewriting reductions. We also present a proof of Knuth–Bendix critical pair theorem and a proof of decidability of equational theories described by complete term rewriting systems. Finally, in section 4, we draw some conclusions and discuss future work.

### *1.1. The ACL2 system*

We briefly describe here the ACL2 theorem prover and its logic. The best introduction to ACL2 is [8]. To obtain more background on ACL2, see the ACL2 user's manual in [9]. A description of the main proof techniques used in Nqthm, also used in ACL2, can be found in [3].

#### *1.1.1. The logic*

ACL2 stands for A Computational Logic for Applicative Common Lisp. 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 [19] (we will assume that the reader is 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, and instantiation. By the *principle of definition*, new function definitions (using `defun`) are admitted as axioms only if there exists an ordinal measure in which the arguments of each recursive call (if any) decrease, thus proving its termination. This ensures that no inconsistencies are introduced by new definitions. The theory has a constructive definition of the ordinals up to $\varepsilon_0$, in terms of lists and natural numbers, given by the predicate `e0-ordinalp` and the order `e0-ord-<`. One important rule of inference is the *principle of induction*, that permits proofs by induction on $\varepsilon_0$.

In addition to the definition principle, the *encapsulation principle* (using `encap-sulate`) allows the user to introduce new function symbols by axioms constraining them to have certain properties. To ensure consistency, witness functions having the same properties have to be exhibited. Within the scope of an `encapsulate`, properties stated with `defthm` need to be proved for the witnesses; 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 if they are known to have the same properties (see [10]).

### *1.1.2. The theorem prover*

The ACL2 theorem prover is inspired by Nqthm, but has been considerably improved. The main proof techniques used by the prover are simplification and induction. Simplification is a process combining term rewriting with some decision procedures (linear arithmetic, type set reasoner, etc.). Sophisticated heuristics for discovering an (often suitable) induction scheme is one of the key points in the success of ACL2 and its predecessor. A collection of definitions and proved theorems is usually stored in a certified file of events (a *book* in the ACL2 terminology), that can be included in other books.

The command `defthm` starts a proof attempt, and, if it succeeds, the theorem is stored as a rule (in most cases, a rewriting rule). The theorem prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, in a deeper sense, the system is interactive. Very often, nontrivial proofs are not found by the system in a first attempt and then the user has to guide the prover by adding lemmas and definitions, used in subsequent proofs as rules. Inspection of failed proofs is very useful to find those lemmas needed to "program" the system in order to get the mechanical proof of a nontrivial result. This kind of interaction is called "The Method" by the authors of the system (see [8]). Thus, the role of the user is important: a typical proof effort consists of formalizing the problem in the logic and helping the prover to find a preconceived hand proof by means of a suitable set of rewrite rules. The mechanical proofs of the results presented here were carried out following "The Method".

## *1.2. Abstract reductions and term rewriting systems*

This section provides a short introduction to basic concepts and definitions from rewriting theory used in this paper. A complete description can be found in [1].

An *abstract reduction* is simply a binary relation $\rightarrow$ defined on a set $A$. We will denote as $\leftarrow$, $\leftrightarrow$, $\overset{*}{\rightarrow}$ and $\overset{*}{\leftrightarrow}$ respectively the inverse relation, the symmetric closure, the reflexive-transitive closure and the equivalence closure. The following concepts are defined with respect to a reduction relation $\rightarrow$. An element $x$ is in *normal form* (or *irreducible*) if there is no $z$ such that $x \to z$. We say that $x$ and $y$ are joinable (denoted as $x \downarrow y$) if there exists $u$ such that $x \overset{*}{\to} u \overset{*}{\leftarrow} y$. We say that $x$ and $y$ are *equivalent* if $x \overset{*}{\leftrightarrow} y$.

An important property to study about reduction relations is the existence of unique normal forms for equivalent objects. A reduction relation has the *Church–Rosser property* if every two equivalent objects are joinable. An equivalent property is *confluence*: for all $x, u, v$ such that $u \overset{*}{\leftarrow} x \overset{*}{\to} v$, then $u \downarrow v$. In every reduction relation with the Church–Rosser property there are not distinct and equivalent normal forms. If in addition the relation is *normalizing* (i.e., every element has a normal form, denoted as $x \downarrow$) then $x \overset{*}{\leftrightarrow} y$ iff $x \downarrow = y \downarrow$. Provided normal forms are computable and identity in $A$ is decidable, then the equivalence relation $\overset{*}{\leftrightarrow}$ is decidable, by means of a test for equality of normal forms.

Another important property is termination: a reduction relation is *terminating* (or *Noetherian*) if there is no infinite reduction sequence $x_0 \to x_1 \to x_2 \to \cdots$. Obviously, every Noetherian reduction is normalizing. The Church–Rosser property can be localized when the reduction is terminating. In that case an equivalent property is *local confluence*: for all $x, u, v$ such that $u \leftarrow x \to v$, then $u \downarrow v$. This result is known as **Newman's lemma**.

An important type of reduction relation is defined on the set $T(\Sigma, X)$ of first order terms of a given language, where $\Sigma$ is a set of function symbols, and $X$ is a set of variables. In this context, an *equation* is a pair of terms $l = r$. The reduction relation defined by a set of equations $E$ is defined as: $s \to_E t$ if there exist $l = r \in E$ and a substitution $\sigma$ of the variables in $l$ (the *matching* substitution) such that $\sigma(l)$ is a subterm of $s$ and $t$ is obtained from $s$ by replacing the subterm $\sigma(l)$ for $\sigma(r)$. This reduction relation is of great interest in universal algebra because it can be proved that $E \models s = t$ iff $s \overset{*}{\leftrightarrow}_E t$. This implies decidability of every equational theory defined by a set of axioms $E$ such that $\to_E$ is terminating and locally confluent. To emphasize the use of the equation $l = r$ from left to right as described above, we write $l \to r$ and talk about *rewrite rules*. A *term rewriting system* (TRS) is a set of rewrite rules. Unless denoted otherwise, $E$ is always a set of equations (equational axioms) and $R$ is a term rewriting system.

Local confluence is decidable for finite and normalizing TRSs: joinability has only to be checked for a finite number of pair of terms, called *critical pairs*, accounting for the most general forms of local divergence (see [1] for a precise definition). The **critical pair theorem** states that a TRS is locally confluent iff all its critical pairs are joinable. Thus, the Church–Rosser property of terminating TRSs is a decidable property: it is enough to check if every critical pair has a common normal form. In that case, the TRS is said to be *complete* and can be used to decide its equational theory. If a terminating TRS has a critical pair with different normal forms, there is still a chance to obtain a decision procedure for its equational theory, adjoining that equation as a new terminating rewrite rule. This is the basis for the well-known *completion algorithm* (see [1] for details).

In the sequel, we describe the formalization of these properties in the ACL2 logic and some points of their mechanical proof. For the rest of the paper, when we talk about "prove" we mean "mechanically prove using ACL2".

## 2. Formalizing abstract reductions in ACL2

One possible way to represent abstract reduction relations in the ACL2 logic could be simply to define them as binary Boolean functions, using `encapsulate` to state their properties. Nevertheless, we adopted a slightly different approach, in order to stress the "reduction" point of view: if $x \rightarrow y$, more important than the relation between $x$ and $y$ is the fact that $y$ is obtained from $x$ by applying some kind of transformation or *operator*. In its most abstract formulation, we can view a reduction as a binary function that, given an element and an operator, returns another object, performing a *one-step reduction*. Think for example of equational reductions: elements in that case are first-order terms and operators are the objects constituted by a position (indicating the subterm replaced), an equation (the rule applied) and a substitution (the matching substitution).

Of course not any operator can be applied to any element. Thus, a second component in this formalization is needed: a Boolean binary function to test if it is *legal* to apply an operator to an element. Finally, a third component is introduced: since computation of normal forms requires searching for legal operators to apply, we will need a unary function that when applied to an element returns a legal operator, whenever it exists, or nil otherwise (a *reducibility* test).[1]

The above considerations lead us to formalize the concept of abstract reductions in ACL2, using three partially defined functions: `reduce-one-step`, `legal` and `reducible`. This can be done with the following `encapsulate` (dots are used to omit local events[2] and technical details, as in the rest of the paper):

```
(encapsulate
  ((reduce-one-step (x op) t)
   (legal (x op) t)
   (reducible (x) t))
   ...
   (defthm legal-reducible-1
     (implies (reducible x) (legal x (reducible x))))

   (defthm legal-reducible-2
     (implies (not (reducible x)) (not (legal x op))))
   ...)
```

[1] It is possible to prove some of the theorems presented here without any reference to a reducibility test (for example, Newman's lemma). See the web page.

[2] The specific witness functions definitions are irrelevant to our discussion, since outside the `encapsulate` only the nonlocal properties are used.

The first part of every `encapsulate` is a signature description of the nonlocal functions partially defined. Note that `(reduce-one-step x op)` is the element obtained applying the operator `op` to `x`. The function `legal` is the applicability test, i.e., `(legal x op)` is not `nil` iff it is legal to apply `op` to `x`. And `reducible` is the reducibility test: `(reducible x)` is a legal operator applicable to `x` whenever such operator exists, `nil` otherwise (we are assuming that `nil` does not represent any operator).

The two theorems assumed above as axioms are minimal requirements for every reduction we defined: if further properties (for example, local confluence, confluence or Noetherianity) were assumed, they have to be stated inside the `encapsulate`. This is a very abstract framework to formalize reductions in ACL2. We think that these three functions capture the basic abstract features every reduction has. On the one hand, a procedural aspect: the computation of normal forms, applying operators until irreducible objects are obtained. On the other hand, a declarative aspect: every reduction relation describes its equivalence closure. Representing reductions in this way, we can define concepts like the Church–Rosser property, local confluence or Noetherianity and even prove nontrivial theorems like Newman's lemma, as we will see.

To instantiate this general framework, concrete instances of `reduce-one-step`, `legal` and `reducible` have to be defined and the properties assumed here as axioms must be proved for those concrete definitions. By functional instantiation, results about abstract reductions can then be easily exported to concrete cases (as we will see for the equational case).

## *2.1. Equivalence and proofs*

Due to the constructive nature of the ACL2 logic, in order to define $x \overset{*}{\leftrightarrow} y$, we have to include an argument with a sequence of steps $x = x_0 \leftrightarrow x_1 \leftrightarrow x_2 \cdots \leftrightarrow x_n = y$. This is done by the function `equiv-p` defined in figure 1. `(equiv-p x y p)` is `t` if `p` is an *abstract proof*[3] justifying that $\mathtt{x}\overset{*}{\leftrightarrow}\mathtt{y}$. This means that `p` is a sequence of legal steps connecting `x` and `y`, where each proof step is a structure[4] `r-step` with four fields: `elt1`, `elt2` (the elements related by the step), `direct` (a boolean value indicating if the step is direct or inverse) and `operator`. A proof step is *legal* (as defined by `proof-step-p`) if one of its elements is obtained by applying its operator (which must be legal) to the other element, in the direction indicated by `direct`. Two abstract proofs justifying the same equivalence will be said to be *equivalent*.

The Church–Rosser property and local confluence can be redefined with respect to the form of abstract proofs (sections 2.2 and 2.3). For that purpose, we define (omitted here) functions to recognize proofs with particular shapes (*valleys* and *local peaks*): `local-peak-p` recognizes proofs of the form $v \leftarrow x \rightarrow u$ and `steps-valley` recognizes proofs of the form $v \overset{*}{\rightarrow} x \overset{*}{\leftarrow} u$.

[3] Or simply a *proof* if that terminology does not arise confusion with proofs done using the ACL2 system.
[4] We used the `defstructure` tool developed by B. Brock [4].

```
(defstructure r-step direct operator elt1 elt2)

(defun proof-step-p (s)
  (let ((elt1 (elt1 s)) (elt2 (elt2 s))
        (op (operator s)) (direct (direct s)))
    (and (r-step-p s)
         (implies direct
                  (and (legal elt1 op)
                       (equal (reduce-one-step elt1 op)
                              elt2)))
         (implies (not direct)
                  (and (legal elt2 op)
                       (equal (reduce-one-step elt2 op)
                              elt1))))))

(defun equiv-p (x y p)
  (if (endp p)
      (equal x y)
      (and (proof-step-p (car p))
           (equal x (elt1 (car p)))
           (equiv-p (elt2 (car p)) y (cdr p)))))
```

Figure 1. Definition of proofs and equivalence.

## 2.2. *The Church–Rosser property and decidability*

We describe how we formalized and proved the decidability of an equivalence relation described by a Church–Rosser and normalizing reduction. Valley proofs can be used to reformulate the definition of the Church–Rosser property: a reduction is Church–Rosser iff for every abstract proof there exists an equivalent valley proof. Since the ACL2 logic is quantifier-free, the existential quantifier in this statement has to be replaced by a Skolem function, which we call `transform-to-valley`. The concept of being normalizing can also be reformulated in terms of abstract proofs: a reduction is normalizing if for every element there exists an abstract proof to an equivalent irreducible element. This proof is given by the (Skolem) function `proof-irreducible` (note that we are not assuming Noetherianity yet). Properties defining a Church–Rosser and normalizing reduction are `encapsulated` as shown in figure 2, item `(a)`.

The function `r-equiv` tests if normal forms are equal. The normal form of an element `x` is defined to be the last element of `(proof-irreducible x)`:

```
(defun normal-form (x)
  (last-of-proof x (proof-irreducible x)))
(defun r-equiv (x y)
  (equal (normal-form x) (normal-form y)))
```

```
;;; (a) Definition of Church-Rosser normalizing reduction:

(encapsulate
 ((legal (x op) t) (reduce-one-step (x op) t)
  (reducible (x) t) (transform-to-valley (x) t)
  (proof-irreducible (x) t))
 .....
 (defthm Church-Rosser-property
   (let ((valley (transform-to-valley p)))
     (implies (equiv-p x y p)
              (and (steps-valley valley)
                   (equiv-p x y valley)))))
 .....
 (defthm normalizing
   (let* ((p-x-y (proof-irreducible x))
          (y (last-of-proof x p-x-y)))
     (and (equiv-p x y p-x-y)
          (not (reducible y))))))

;;; (b) Main theorems proved:

(defthm if-C-R--two-ireducible-connected-are-equal
  (implies (and (equiv-p x y p)
                (not (reducible x))
                (not (reducible y)))
           (equal x y)))

(defthm r-equiv-sound
  (implies (r-equiv x y)
           (equiv-p x y (make-proof-common-n-f x y))))

(defthm r-equiv-complete
  (implies (equiv-p x y p) (r-equiv x y))
```

Figure 2. Church–Rosser and normalizing implies decidability.

To prove decidability of a Church–Rosser and normalizing relation, it is enough to prove that `r-equiv` is a complete and sound algorithm deciding the equivalence relation described by the reduction relation. See figure 2, item `(b)`. We also include the main lemma used, stating that there are no distinct equivalent irreducible elements. Note also that soundness is expressed in terms of a Skolem function `make-proof-common-normal-form` (definition omitted), which constructs a proof justifying the equivalence. These theorems are proved quite easily, without much guidance from the user. The main point here is that the induction scheme suggested by the function `equiv-p` (and mechanically generated by the system), turns out to be very useful in proving properties about the relation $\overset{*}{\leftrightarrow}$: it resembles the intuitive idea of "induction on the number of steps".

## 2.3. *Noetherianity, local confluence and Newman's lemma*

A relation is *well founded* on a set $A$ if every nonempty subset has a minimal element. A restricted notion of well-foundedness is built into ACL2, based on the following meta-theorem: a relation on a set $A$ is well-founded iff there exists a function $F : A \to Ord$ such that $x < y \Rightarrow F(x) < F(y)$, where *Ord* is the class of all ordinals. In ACL2, once a relation is proved to satisfy these requirements (and the theorem is stored as a `well-founded-relation` rule), it can be used in the admissibility test for recursive functions. A general well-founded partial order `rel` can be defined in ACL2 as shown in figure 3, item `(a)`. Since only ordinals up to $\varepsilon_0$ are formalized in the ACL2 logic, a limitation is imposed in the maximal order type of well-founded relations that can be represented. Consequently, our formalization suffers from the same restriction.[5]

In figure 3, item `(b)` a general definition of a Noetherian and locally confluent reduction relation is presented.[6] Local confluence is easily expressed in terms of the shape of abstract proofs involved: a relation is locally confluent iff for every local peak proof there is an equivalent valley proof. This valley proof is assumed to be given by a function named `transform-local-peak`. As for Noetherianity, our formalization relies on the following meta-theorem: a reduction is Noetherian if and only if it is contained in a well-founded partial ordering. Thus, the general well-founded relation `rel` previously presented is used to justify Noetherianity of the general reduction relation defined: for every element `x` such that a `legal` operator `op` can be applied to, then applying `op` to `x` using `reduce-one-step`, produces an element less than `x` (with respect to `rel`).

The standard proof of Newman's lemma found in the literature [1], shows confluence by Noetherian induction based on the Noetherian reduction relation. Nevertheless, the formal proof we obtained is different, influenced by our abstract proof approach. It is inspired by the one given by Klop in [11]. In our formalization, we show that the reduction relation has the Church–Rosser property[7] by *defining* a function `transform-to-valley` and proving that for every proof `p`, `(transform-to--valley p)` is an equivalent valley proof. This function is defined to iteratively apply `replace-local-peak` (which replaces a local peak subproof by the equivalent proof given by `transform-local-peak`), until there are no local peaks. This can be seen as a normalization process acting on abstract proofs. See definition in figure 3, item `(c)`.

[5] Nevertheless, no particular properties of $\varepsilon_0$ are used in our proofs, except well-foundedness.

[6] Name conflicts with the functions presented in the previous and next sections are avoided using Common Lisp packages.

[7] Note that we do not need to deal with confluence since the Church–Rosser property, an equivalent concept, is proved with the same effort.

```
;;; (a) Well-founded partial order:

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

 (defthm rel-transitive
   (implies (and (rel x y) (rel y z)) (rel x z))))

;;; (b) Noetherian and locally confluent reduction:

(encapsulate
 ((legal (x op) t) (reduce-one-step (x op) t)
  (reducible (x) t) (transform-local-peak (x) t))
 ....
 (defthm locally-confluent
   (let ((valley (transform-local-peak p)))
     (implies (and (equiv-p x y p) (local-peak-p p))
              (and (steps-valley valley)
                   (equiv-p x y valley)))))

 (defthm Noetherian
   (implies (legal x op) (rel (reduce-one-step x op) x))))

;;; (c) Definition of transform to valley:

(defun transform-to-valley (p)
  (declare (xargs :measure (proof-measure p)
                  :well-founded-relation mul-rel))
  (if (not (exists-local-peak p))
      p
      (transform-to-valley (replace-local-peak p))))

;;; (d) Main theorem proved:
(defthm Newman-lemma
  (let ((valley (transform-to-valley p)))
    (implies (equiv-p x y p)
             (and (steps-valley valley)
                  (equiv-p x y valley)))))
```

Figure 3. Newman's lemma.

The main effort was done to prove the termination of `transform-to-valley` (needed for its admission), showing that in each iteration, some measure of the abstract proof, `proof-measure`, decreases with respect to a well-founded relation, `mul-rel`:

```
(defthm transform-to-valley-admission
  (implies (exists-local-peak p)
           (mul-rel (proof-measure (replace-local-peak p))
                    (proof-measure p)))).
```

The measure `proof-measure` is the list of elements involved in the proof and the relation `mul-rel` is defined to be the *multiset extension* of `rel`. We needed to prove in ACL2 that the multiset extension of every well-founded relation is also well-founded, a result interesting in its own right, and a tool that can be applied in other ACL2 termination proofs [16].

Once `transform-to-valley` is admitted, it is relatively easy to show that it always returns an equivalent proof which is a valley. See figure 3, item `(d)`. The main point here is that both properties are proved using the induction scheme suggested by the function `transform-to-valley`. When proving a conjecture `(:P P X Y)` (where `:P` stands for a property about `P`, `X` and `Y`, in the ACL2 terminology), this induction scheme can be described as:

```
(AND (IMPLIES (AND (EXISTS-LOCAL-PEAK P)
                   (:P (REPLACE-LOCAL-PEAK P) X Y))
              (:P P X Y))
     (IMPLIES (NOT (EXISTS-LOCAL-PEAK P))
              (:P P X Y)))
```

This induction scheme is justified by the well-foundedness of the multiset relation `mul-rel`: it is a proof by induction on `proof-measure`. This is in contrast with the standard proof: the formal proof obtained with this induction scheme treats the theorem as a property of abstract proofs rather than as a property of the elements involved. This reveals the advantage of considering abstract proofs as objects that can be transformed to obtain new proofs.

Having established Newman's lemma and the result described in the previous subsection, we prove decidability of the equivalence relation described by a locally confluent and Noetherian reduction. Note that in proving Newman's lemma we have given a particular "implementation" of `transform-to-valley` and proved as theorems the properties about it assumed as axioms in the previous subsection. The same can be done with `proof-irreducible`: since the reduction is terminating, by means of `reducible` and `reduce-one-step` we can build an abstract proof connecting every element to its normal form. Since we can now establish the hypothesis of the theorem of the previous subsection, we can easily deduce, by functional instantiation, the decidability of the equivalence relation described by a Noetherian and locally confluent reduction. See the web page for details.

## 3. Formalizing rewriting in ACL2

We defined in the previous section a very general formalization of reduction relations. The results proved can be reused for every instance of the general framework. As a major application, we describe in this section how we formalized and reasoned about term rewriting in ACL2.

### *3.1. First-order terms*

Since rewriting is a reduction relation defined on the set of first order terms, we needed to use a library of definitions and theorems formalizing the lattice theoretic properties of first-order terms: in particular, subsumption and unification algorithms were defined and proved correct. This ACL2 library was obtained translating a previous formalization developed by the authors using Nqthm [14], so we will only give here a very brief description. In this library, we represent first-order terms in prefix notation using lists. For example, the term $f(x, g(y), h(x))$ is represented as `'(f x (g y) (h x))`. Every `consp` object can be seen as a term with its `car` as its top function symbol and its `cdr` as the list of its arguments. Variables are represented by atom objects. Substitutions are represented as association lists and equations and rules as dotted pairs of terms.

Since ACL2 mechanizes a logic of total functions, our functions acting on first-order terms are extended in a "natural" way to deal also with Lisp objects not representing terms, although they are not in the intended domain of the functions. This is not a problem: every function defined returns well-formed terms when its arguments are well-formed terms. Furthermore, the *guard verification* mechanism of ACL2 is used to ensure that every execution in Common Lisp of the functions verified does not evaluate on arguments outside the intended domain (see [9] for details about guards).

Most of the functions are defined, using mutual recursion, for terms and for lists of terms at the same time. This kind of definition suggests to the prover an induction scheme very similar to induction on the structure of terms, which, in most cases, turns out to be the right induction scheme. This good behavior of the system's heuristics when choosing induction schemes for a conjecture is crucial in the automation of our proofs.

We give a brief description of some of the functions defined in this library that will be referenced in the sequel. The function `variable-p` recognizes variables. The function `instance` implements the application of a substitution to a term. Especially important in this context are the functions dealing with the tree structure of terms: `position-p` tests if a sequence of integers is a position of a term, `occurrence` returns the subtree at a given position and `replace-term` performs a replacement of a subterm at a given position. For a detailed description of this library, see the web page.

## *3.2. Instantiating the abstract framework*

The very abstract concept of operator can be instantiated for term rewriting reductions. Equational operators are structures with three fields, containing the rewriting rule to apply, the position of the subterm to be replaced and the matching substitution:

```
(defstructure eq-operator rule pos matching).
```

As we said in section 2, every reduction relation is given by concrete versions of `legal`, `reduce-one-step` and `reducible`. In the equational case:

- `(eq-legal term op E)` tests if the `rule` of the operator `op` is in `E`, and can be applied to `term` at the position indicated by the operator (using the `matching` substitution of the operator `op`).
- `(eq-reduce-one-step term op)` replaces the subterm indicated by the position of the operator `op`, by the corresponding instance (using `matching`) of the right-hand side of the `rule` of the operator.
- `(eq-reducible term E)` returns a legal equational operator to apply to `term`, whenever it exists, or `nil` otherwise.

Note that for every fixed set of equations `E`, a particular reduction relation is defined. The equational counterpart of the abstract equivalence `equiv-p` can be defined in a very similar way: `(eq-equiv-p t1 t2 p E)` tests if `p` is a proof (an *equational proof*) of the equivalence of `t1` and `t2` in the equational theory of `E`. Note that the function `eq-equiv-p` implements a proof checker for equational theories, thus formalizing equational deduction in ACL2. Due to the lack of space, we do not give the definitions here. Recall from section 2 that two theorems (assumed as axioms in the general framework) have to be proved to state the relationship between `eq-legal` and `eq-reducible`. We proved them, in order to be able to export results (by functional instantiation) from the abstract case to the equational case:

```
(defthm eq-reducible-legal-1
  (implies (eq-reducible term E)
           (eq-legal term (eq-reducible term E) E)))

(defthm eq-reducible-legal-2
  (implies (not (eq-reducible term E))
           (not (eq-legal term op E))))
```

Term rewriting systems, as defined in [1], are a special case of sets of equations: the left-hand side of the equations cannot be variables and must contain the variables of the right-hand side. We define the function `rewrite-system` (omitted here) to implement this concept.[8]

[8] Nevertheless, the formalization described in this subsection does not assume the set of equational axioms to be a term-rewriting system.

```
(defthm eq-equiv-p-reflexive
  (eq-equiv-p term term nil E))

(defthm eq-equiv-p-symmetric
  (implies (eq-equiv-p t1 t2 p E)
           (eq-equiv-p t2 t1 (inverse-proof p) E)))

(defthm eq-equiv-p-transitive
  (implies (and (eq-equiv-p t1 t2 p E)
                (eq-equiv-p t2 t3 q E))
           (eq-equiv-p t1 t3 (proof-concat p q) E)))

(defthm eq-equiv-p-stable
  (implies (eq-equiv-p t1 t2 p E)
           (eq-equiv-p (instance t1 sigma)
                       (instance t2 sigma)
                       (eq-proof-instance p sigma) E)))

(defthm eq-equiv-p-compatible
  (implies (and (eq-equiv-p t1 t2 p E)
                (position-p pos term))
           (eq-equiv-p (replace-term term pos t1)
                       (replace-term term pos t2)
                       (eq-proof-context p term pos) E)))
```

Figure 4. Congruence: an algebra of proofs.

Formalizing equational reasoning in this way, we proved a number of results about it, as we will describe in the following sections.

## *3.3. Equational theories and an algebra of proofs*

An equivalence relation on first-order terms is a congruence if it is stable (closed under instantiation) and compatible (closed under subterm replacement). Equational consequence, $E \models s = t$, can alternatively be defined as the least congruence relation containing $E$. In order to justify that the above described formalization is appropriate, it would be suitable to prove that, for a fixed `E`, the relation given by `(eq-equiv-p t1 t2 p E)`,[9] is the least congruence containing `E`.

In figure 4 we sketch part of that result, showing that `eq-equiv-p` is a congruence. The ACL2 proof obtained is again a good example of the benefits gained considering proofs as objects that can be transformed to obtain new proofs. Following

[9] Formally speaking, the relation “exists `p` such that `(eq-equiv-p t1 t2 p E)`”.

Bachmair [2], we can define an "algebra" of equational proofs, given by the following operations: `proof-concat` to concatenate proofs, `inverse-proof` to obtain the reverse proof, `eq-proof-instance`, to instantiate the elements involved in the proof and `eq-proof-context` to include the elements of the proof as subterms of a common term. The empty proof `nil` can be seen as a proof constant. Each of these operations corresponds with one of the properties needed to show that `eq-equiv-p` is a congruence. The theorems are proved easily by ACL2, with minor help from the user.

### *3.4. The critical pair theorem*

The main result we have proved is the critical pair theorem: a rewrite system $R$ is locally confluent if every critical pair obtained with rules in $R$ is joinable.

In figure 5, item `(a)` a term-rewriting system `(RLC)` is partially defined assuming the property of joinability of its critical pairs. The partially defined expression `(transform-cp l1 r1 pos l2 r2)` is assumed to obtain a valley proof for the critical pair (if it exists) determined by the rules `(l1 . r1)` and `(l2 . r2)` at the non-variable position `pos` of `l1`. The function `cp-r` computes such a critical pair whenever it exists, or it returns `nil` otherwise. This computation is done after renaming the variables of the rules, in order to get them standardized apart.

To prove the critical pair theorem in our formalization, we have to *define* a function `transform-eq-local-peak` and prove that it transforms every equational local peak proof into an equivalent valley proof. It has a very long definition (omitted here), dealing with all possible cases of local divergences. The final theorem is shown in figure 5, item `(b)`. Note that this theorem can be used to conclude local confluence of any particular rewrite system with joinable critical pairs.

The ACL2 proof of this theorem is the largest proof we developed. As a basis for our formal proof of the local confluence of `(RLC)`, we follow Huet's proof [6]. The proof is obtained as a typical (but very long) interaction with the ACL2 theorem prover. The "algebra" of equational proofs defined in the previous subsection allows us to control the complexity of this ACL2 proof: for example, one first deals with the case in which one of the two rewritings in the equational local peak is performed at the top the term; later on, this result can be translated to a more general case by inclusion in a context.

As in [6], the proof is mainly structured to deal with three cases, according to the relative positions of the subterms where the two rewriting steps (in a local peak) may occur: disjoint rewriting, noncritical overlap and critical overlap. The main proof effort was done to handle noncritical (or variable) overlaps. It is interesting to point that in most textbooks and surveys this case is proved pictorially. Nevertheless, in our mechanical proof turns out to be the most difficult part and it even requires the design of an induction scheme not discovered by the heuristics of the prover. The critical overlap case was easier to prove than the previous case, but we must not forget that in order to reason properly about this case we used our library about first-order terms, where some results about variable renamings and, more important, a verified unification algorithm [14] were developed.

```
;;; (a) TRS with joinable critical pairs:

(encapsulate
 ((RLC () t) (transform-cp (l1 r1 pos l2 r2) t))
 ...
 (defthm RLC-rewrite-system (rewrite-system (RLC)))

 (defthm RLC-joinable-critical-pairs
   (implies
     (and (member (cons l1 r1) (RLC))
          (member (cons l2 r2) (RLC))
          (position-p pos l1)
          (not (variable-p (occurence l1 pos))))
     (let* ((cp-r (cp-r l1 r1 pos l2 r2))
            (valley-cp (transform-cp l1 r1 pos l2 r2)))
       (implies
          cp-r
          (and (eq-equiv-p
                    (lhs cp-r) (rhs cp-r) valley-cp (RLC))
               (steps-valley valley-cp)))))))

;;; (b) Theorem proved:

(defun transform-eq-local-peak (p) ...)

(defthm critical-pair-theorem
   (let ((valley (transform-eq-local-peak p)))
     (implies (and (eq-equiv-p t1 t2 p (RLC))
                   (local-peak-p p))
              (and (steps-valley valley)
                   (eq-equiv-p t1 t2 valley (RLC))))))
```

Figure 5. The critical pair theorem.

## *3.5. Reduction orderings*

In order to formalize termination properties of term rewriting systems we rely on the well-known concept of *reduction ordering*, i.e., well-founded ordering being *stable* (closed under instantiation) and *compatible* (closed under replacement of subterms). We used the following characterization: a term rewriting system $R$ terminates iff there exists a reduction order $\succ$ that satisfies $l \succ r$ for all $l \to r \in R$. In figure 6, encapsulation is

```
(encapsulate
  ((red< (t1 t2) t) (fn-red< (term) t))
   ....
  (defthm red<-well-founded-relation
    (and (e0-ordinalp (fn-red< t1))
         (implies (red< t1 t2)
                  (e0-ord-< (fn-red< t1) (fn-red< t2))))
         :rule-classes :well-founded-relation)

  (defthm red<-stable
    (implies (red< t1 t2)
             (red< (instance t1 sigma)
                   (instance t2 sigma))))

  (defthm red<-compatible
    (implies (and (position-p pos term) (red< t1 t2))
             (red< (replace-term term pos t1)
                   (replace-term term pos t2))))

  (defthm red<-transitive
    (implies (and (red< x y) (red< y z)) (red< x z))))

(defun noetherian-red< (TRS)
  (if (endp TRS) t
    (let ((rule (car TRS)))
      (and (red< (rhs rule) (lhs rule))
           (noetherian-red< (cdr TRS))))))
```

Figure 6. A reduction order `red<`.

used to (partially) define a function `red<`, assumed to be a reduction order. The function `(noetherian-red< TRS)` is defined to test if `red<` justifies termination of `TRS`.

Once `red<` has been assumed to be a reduction ordering and the function `noe-therian-red<` has been defined, we proved that the reduction relation $\to_{\mathtt{R}}$ is terminating, whenever `R` is a TRS such that `(noetherian-red< R)` (this result is needed to export Newman's lemma to the equational case):

```
(defthm R-Noetherian-if-subsetp-of-red<
  (implies (and (noetherian-red< R)
                (eq-legal term op R))
           (red< (eq-reduce-one-step term op) term)))
```

Although the (partial) definition of the reduction ordering `red<` given in figure 6 works well from a theoretical point of view, the main drawback in this formalization of reduction orderings is that it can be difficult to prove that a particular ordering (for example, a path ordering or a Knuth–Bendix ordering [1]) is a reduction ordering, since an ordinal measure `fn-red<` has to be given explicitly.

## *3.6. Complete term rewriting systems and decidability*

As a consequence of the results presented so far, and using functional instantiation, we can formalize and prove decidability of the equational theory described by a complete TRS. In the following we describe the assumptions needed to define a complete TRS.

Again using `encapsulate` we (partially) define a term rewriting system `(RC)` assumed to be complete: `(RC)` is terminating (justified by `red<`) and every critical pair obtained from rules in `(RC)` have a common normal form (see figure 7). In this formalization, the concepts of critical pairs and normal forms are implemented by the functions `cp-r` (described in section 3.4) and `RC-normal-form`, respectively.

The function `RC-normal-form` is defined to compute normal forms with respect to the term rewriting system `(RC)`. It iteratively applies the function `r-reduce` until a normal form is found. The expression `(r-reduce term TRS)`, whose definition we omit here, performs one step of rewriting, whenever it is possible. It traverses `term` to find a subterm subsumed by the left-hand side of a rule in `TRS`. When such a subterm is found, it is replaced by the corresponding instance of the right-hand side of the rule. If it is not found, then `r-reduce` returns `nil` (and therefore `term` is in normal form). Those properties of `r-reduce` were mechanically verified. Note that a verified subsumption algorithm is needed for that purpose.

It is worth pointing that a function computing the normal form of a term with respect to a TRS would not be admitted in the ACL2 logic, since termination is not assured in general. Instead, we assume `(RC)` to be terminating and we define normal form calculation with respect to `(RC)`.[10]

Having assumed the properties of figures 6 and 7, we can define a function `RC-equivalent` (testing equality of normal forms) and then prove that it provides a complete and sound algorithm to decide the equational theory of `(RC)`:

```
(defun RC-equivalent (t1 t2)
  (equal (RC-normal-form t1) (RC-normal-form t2)))

(defthm RC-equivalent-complete
  (implies (eq-equiv-p t1 t2 p (RC))
           (RC-equivalent t1 t2)))
```

[10] Although this definition is suitable from a formal point of view, the main drawback is that that `RC-normal-form` is not executable. Nevertheless, we can define an executable function `(normal-form-n n term R)` that applies (at most) `n` reduction steps to `term` with respect to the TRS `R`. In practice, this can be used to compute normal forms.

```
    (encapsulate
     ((RC () t))
     ...
     (defthm RC-rewrite-system (rewrite-system (RC)))

     (defthm RC-Noetherian-red< (noetherian-red< (RC)))

     (defun RC-normal-form (term)
       (declare (xargs :measure term
                       :well-founded-relation red<))
       (let ((red (r-reduce term (RC))))
         (if red (RC-normal-form (unpack red)) term)))

     (defthm RC-common-n-f-critical-pairs
       (implies
         (and (member (cons l1 r1) (RC))
              (member (cons l2 r2) (RC))
              (position-p pos l1)
              (not (variable-p (occurrence l1 pos))))
         (let ((cp-r (cp-r l1 r1 pos l2 r2)))
           (implies
             cp-r
             (equal (RC-normal-form (lhs cp-r))
                    (RC-normal-form (rhs cp-r))))))))
```

Figure 7. A complete term rewriting system (RC).

```
(defthm RC-equivalent-sound
  (implies (RC-equivalent t1 t2)
           (eq-equiv-p
             t1 t2
             (RC-make-proof-common-n-f t1 t2) (RC))))
```

The proof of the two theorems above is straightforward (although some elaborated) by means of functional instantiation of the previous theorems presented. The following is part of the functional substitution used in this instantiation, associating to the functions describing an abstract reduction the corresponding functions of the equational reduction associated to (RC):

```
...
  (reduce-one-step    eq-reduce-one-step)
  (reducible          (lambda (term)
                             (eq-reducible term (RC))))
  (legal              (lambda (term op)
                             (eq-legal term op (RC))))
  (equiv-p            (lambda (t1 t2 p)
                             (eq-equiv-p t1 t2 p (RC))))
...
```

An important point in this decidability theorem is that the verified decision algorithm `RC-equivalent` does not deal with equational proofs, equational proof steps or equational operators. This is an example of *compositional reasoning*, or how to reason about an implementation by using rules that transform some functions in other functions (often less efficient) that are easier to reason about.

Note that in this case the functions eq-reducible and `eq-reduce-one-step` provides a way to perform one step of rewriting, whenever it is possible: given a term and a TRS, apply `eq-reducible` to obtain an equational operator and, if non-`nil`, apply this operator to the term using `eq-reduce-one-step`. If the TRS is terminating, then this method can be applied iteratively until a normal form is obtained. This definition of normal form is appropriate for reasoning. For example, it turns out to be useful when we define an equational counterpart of `proof-irreducible`, a function obtaining an equational proof connecting every element to its normal form, that is needed to export by functional instantiation the decidability result of section 2.2. Obviously, this normal form calculation can be optimized in several ways. For example, a function computing normal forms neither needs to build an equational operator in every rewriting step nor traverse the terms twice, searching for a legal equational operator, and then applying the reduction step. As we described above, `r-reduce` is a more efficient (although not optimal) version of one-step rewriting. The main point here is that we used the more theoretical version to reason about normal form calculation, which turned out to be simpler. Later on, we proved theorems relating the behavior of `r-reduce` with `eq-reducible` and `eq-reduce-one-step`, showing the equivalence with the improved version of normal form calculation, and then we stated the final version of the theorem using `r-reduce`.

## 4. Conclusions and further work

We have presented an application of the ACL2 system to formalize and reason about rewrite systems theory. This is a case study of using the ACL2 system as a meta-language to formalize properties of object proof systems (abstract reductions and equational logic in this case) in it. Our formalization has the following main features:

- Abstract reduction relations and their properties are stated in a very general framework, as explained in section 2. Functional instantiation is extensively used to export results from the abstract case to the equational case.
- The concepts of abstract proofs and equational proofs are key notions in our work, as it has been pointed repeatedly. Proofs are treated as objects that can be transformed to obtain new proofs and this point of view has great influence both in formalization and reasoning.
- Compositional reasoning is used, verifying some functions by using rewrite rules that transform them in other functions, often less efficient, that are easier to reason about.

We think that the results presented here are important for two reasons. From a theoretical point of view, it is shown how a weak logic can be used to formalize properties of TRSs. From a practical point of view, this is an example of how formal methods can help in the design of symbolic computation systems. Usually, rewriting techniques are applied to the design of proof procedures in automated deduction. We show how benefits can be obtained in the reverse direction: automated deduction used as a tool to "certify" components of symbolic computation systems.

Since ACL2 is also a programming language, computing and proving tasks can be mixed. As a result of this formalization, we obtained a number of basic functions in term rewriting, executable and verified in ACL2; for example, matching, unification, computation of critical pairs or application of reduction steps with respect to a term rewriting system. We verified the guards of all these functions, ensuring in this way that they are executable in any compliant Common Lisp (with the appropriate files loaded).

It should be stressed that proving nontrivial results in a theorem prover like ACL2 is not trivial. A user expert in both the theorem prover and the subject domain is needed (maybe that is the reason why many of the published formal proofs are about formal systems). As claimed in [8], difficulties come from "the complexity of the whole enterprise of formal proofs", rather than from the complexity of ACL2. 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. Nevertheless, proofs can be simpler if a good library of previous results (*books* in the ACL2 terminology) is used. We think our work provides a good collection of books to be reused in further verification efforts.

The proof described here has been structured in three collection of books (see the web page), chronologically developed in the following order (every book needs results from its predecessor):

1. Books about abstract reductions: `abstract-proofs` contains basic definitions and properties about abstract proofs, `confluence` proves the decidability of the equivalence relation described by a Church–Rosser and normalizing reduction, `newman` is the proof of Newman's lemma and `local-confluence` is a proof, by functional instantiation, of decidability of the equivalence relation described by a terminating and locally confluent reduction relation.

2. Books about equational theories and rewriting: `equational-theories` contains the definition and main properties of the equational theory given by a set of equational axioms and `rewriting` develops the notions of reducibility, reduction orderings and one-step rewriting.

3. The proof of the critical pair theorem is in the book `critical-pairs` and decidability of the equational theory of a complete TRS is proved in `kb-decidability`.

Table 1 gives some quantitative information on the proof. The first column contains the name of the book. The next three columns show the number of lines (including comments), the number of definitions and the number of theorems in each book. These numbers can give an idea of the granularity of our proof. We should say that these sizes can be reduced, but sometimes we preferred to split definitions and theorems for the sake of clarity. We also included a fifth column with the number of theorems that needed hints from the user: the rest of the theorems were proved automatically by the system. Together with the number of theorems, this can give an idea of the degree of automation of the proofs. Most of the hints given are for disabling or enabling rules and for using instances of previous theorems.

It is clear from the table that the main proof effort was done to prove Newman's lemma and the critical pair theorem. It should be emphasized also that, although not listed in the table, the books about first-order terms [14] and multiset relations [16] are crucial in our development.

Some related work has been done in the formalization of abstract reduction relations in other theorem proving systems, mostly as part of formalizations on the $\lambda$-calculus. For example, Huet [7] in the Coq system or Nipkow [13] in Isabelle/HOL. A comparison is difficult because our goal was different and, more important, the logics involved are significantly different: ACL2 logic is a much weaker logic than those of Coq or HOL. A more related work is Shankar [18], using Nqthm. Although his work is on the concrete reduction relation of $\lambda$-calculus and he does not deal with the abstract case, some of his ideas are reflected in our work.

Table 1
Quantitative information on the proofs.

| Book | Lines | Definitions | Theorems | Hints |
|---|---|---|---|---|
| `abstract-proofs` | 284 | 16 | 17 | 0 |
| `confluence` | 387 | 12 | 31 | 7 |
| `newman` | 993 | 15 | 53 | 10 |
| `local-confluence` | 464 | 19 | 14 | 6 |
| `equational-theories` | 543 | 11 | 29 | 8 |
| `rewriting` | 720 | 13 | 38 | 9 |
| `critical-pairs` | 2129 | 43 | 112 | 26 |
| `kb-decidability` | 500 | 16 | 18 | 7 |
| Total | 6020 | 145 | 312 | 73 |

To our knowledge, no formalization of term rewriting systems has been done yet and, consequently, the formal proofs of their properties presented here are the first ones we know performed using a theorem prover.

In addition to extend the library of results about term rewriting systems, there are also several ways in which the work presented here can be further developed, mostly devoted to improve efficiency of the verified algorithms and to apply the results to concrete equational theories:

- In order to obtain certified decision procedures for some concrete equational theories, work has to be done to formalize in ACL2 well-known terminating term orderings (recursive path orderings, Knuth–Bendix orderings, etc.). As commented in section 3.5, maybe some problems will arise due to the restricted notion of Noetherianity supported by ACL2.
- The work presented in [12] suggests another application of this work: other theorem provers can be combined with ACL2 in order to obtain mechanically verified decision algorithms for some equational theories.
- Although a fully verified equational reasoning system is currently impractical, it would be desirable to improve the efficiency of the algorithms (for example, using better data structures). Compositional reasoning can be used to reason about these improved algorithms.
- Our original motivation when we began this formalization (and now our goal in the long term) is to obtain a certified completion procedure written in Common Lisp. We think the work presented here is a good starting point.

## References

[1] F. Baader and T. Nipkow, *Term Rewriting and All That* (Cambridge University Press, Cambridge, 1998).

[2] L. Bachmair, *Canonical Equational Proofs* (Birkhäuser, New York, 1991).

[3] R. Boyer and JS. Moore, *A Computational Logic Handbook*, 2nd ed. (Academic Press, New York, 1998).

[4] B. Brock, `defstructure` for ACL2 version 2.0, Technical Report, Computational Logic, Inc. (1997).

[5] B. Buchberger and R. Loos, Algebraic simplification, in: *Computer Algebra, Symbolic and Algebraic Computation. Computing Supplementum 4* (1982).

[6] G. Huet, Confluent reductions: abstract properties and applications to term rewriting systems, Journal of the ACM 27(4) (1980) 797–821.

[7] G. Huet, Residual theory in $\lambda$-calculus: a formal development, Journal of Functional Programming 4 (1994) 475–522.

[8] M. Kaufmann, P. Manolios and JS. Moore, *Computer-Aided Reasoning: An Approach* (Kluwer Academic, Dordrecht, 2000).

[9] M. Kaufmann and JS. Moore, ACL2 Version 2.5 (2000) available at `http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html`.

[10] M. Kaufmann and JS. Moore, Structured theory development for a mechanized logic, Journal of Automated Reasoning 26(2) (2001) 161–203.

[11] J.W. Klop, Term rewriting systems, in: *Handbook of Logic in Computer Science* (Clarendon Press, Oxford, 1992).

[12] W. McCune and O. Shumsky, Ivy: a preprocessor and proof checker for first-order logic, in: *Computer-Aided Reasoning: ACL2 Case Studies* (Kluwer Academic, Dordrecht, 2000) chapter 16.

[13] T. Nipkow, More Church–Rosser proofs, Journal of Automated Reasoning 26(1) (2001) 51–66.

[14] J.L. Ruiz-Reina, J.A. Alonso, M.J. Hidalgo and F.J. Martín, 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.

[15] J.L. Ruiz-Reina, J.A. Alonso, M.J. Hidalgo and F.J. Martín, A mechanical proof of Knuth–Bendix critical pair theorem (using ACL2), in: *FTP'2000 (Third Workshop on First-Order Theorem Proving)*, Technical Report 5-2000, Fachberichte Informatik, Universität Koblenz-Landau (2000).

[16] J.L. Ruiz-Reina, J.A. Alonso, M.J. Hidalgo and F.J. Martín, Multiset relations: a tool for proving termination, in: *Second ACL2 Workshop*, Technical Report TR-00-29, Computer Science Departament, University of Texas (2000).

[17] J.L. Ruiz-Reina, J.A. Alonso, M.J. Hidalgo and F.J. Martín, Formalizing rewriting in the ACL2 theorem prover, in: *AISC'2000 (Fifth International Conference Artificial Intelligence and Symbolic Computation)*, Lecture Notes in Computer Science, Vol. 1930 (Springer, Berlin, 2001) pp. 92–103.

[18] N. Shankar, A mechanical proof of the Church–Rosser theorem, Journal of the ACM 35(3) (1988) 475–522.

[19] G.L. Steele, *Common Lisp the Language*, 2nd ed. (Digital Press, 1990).