Formalizing rewriting in the ACL2 theorem prover *

José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo and Francisco-Jesús Martín-Mateos
{jruiz,jalonso,mjoseh,fjesus}@cica.es

Departamento de Ciencias de la Computación e Inteligencia Artificial.
Facultad de Informática y Estadística, Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain

Abstract. We present an application of the ACL2 theorem prover to formalize and reason about rewrite systems theory. This can be seen as a first approach to apply formal methods, using ACL2, to the design of symbolic computation systems, since the notion of rewriting or simplification is ubiquitous in such systems. We concentrate here in formalization and representation aspects of the rewriting theory using the first-order, quantifier-free ACL2 logic based on Common Lisp. The main result we mechanically proved is Knuth-Bendix critical-pair theorem.

Keywords: Computer algebra systems and automated theorem provers. Integration of logical reasoning and computer algebra. Formal methods. Logic and symbolic computing. ACL2 theorem prover. Abstract reduction systems. Term rewriting.

1 Introduction

We report in this paper the status of our work on the application of the ACL2 theorem prover to reason about rewrite systems theory: confluence, local confluence, noetherianity, normal forms and other related concepts have been formalized in the ACL2 logic and some results about abstract reduction relations and term rewriting systems have been mechanically proved, including Newman's lemma and Knuth-Bendix critical pair theorem.

ACL2 is both a logic and a mechanical theorem proving system supporting it. The ACL2 logic is a 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.

Formal proofs using a theorem proving environment provides not only formal verification of mathematical theories, but allows to understand and examine their theorems with much greater detail, rigor and clarity. On the other hand, the notion of rewriting or simplification is a crucial component in symbolic computation: simplification procedures are needed to transform complex objects obtaining equivalent but simpler objects and to compute unique representations for equivalence classes (see, for example, [4] or [9]).

Since ACL2 is also a programming language, this work can be seen as a first step to obtain verified executable Common Lisp code for components of symbolic computation systems. Although a fully verified implementation of such a system is currently beyond our possibilities, 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 non-trivial theorems. In this paper, we place emphasis on describing the formalization and representation aspects of our work. Due to the

* This work has been supported by DGES/MEC: Projects PB96-0098-C04-04 and PB96-1345

lack of space, we will skip details of the mechanical proofs. The complete books are available on the web in `http://www-cs.us.es/~jruiz/acl2-rewr/`.

1.1 The ACL2 system

We briefly describe here the ACL2 theorem prover and its logic. For a good introduction written by the authors of the system, see [6]. Readers wishing more background on ACL2 are urged to see the ACL2 user's manual in [7]. A very good description of the main proof techniques used in Nqthm, that are also used in ACL2, can be found in [3].

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 [13] (we will assume the reader familiar with this language). The logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference include those for propositional calculus, equality, 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 decrease. This ensures that no inconsistencies are introduced by new definitions. The theory has a constructive definition of the ordinals up to ε_0, in terms of lists and natural numbers, given by the predicate `e0-ordinalp` and the order `e0-ord-<`. One important rule of inference is the *principle of induction*, that permits proofs by induction on ε_0.

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

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 combination of decision procedures, mainly term rewriting, using the rules previously proved by the user. The command `defthm` starts a proof attempt, and, if it succeeds, the theorem is stored as a rule. The theorem prover is automatic in the sense that once `defthm` is invoked, the user can no longer interact with the system. However, the user can guide the prover by adding lemmas and definitions, used in the proofs as rules. The role of the user is important: a typical proof effort consists in formalize the problem in the logic and help the prover to find a proof by means of a suitable set of rewrite rules.

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 $\leftarrow$, $\leftrightarrow$, $\stackrel{*}{\rightarrow}$ and $\stackrel{*}{\leftrightarrow}$ 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 $x \rightarrow z$. We say that x and y are joinable (denoted as $x \downarrow y$) if exists u such that $x \stackrel{*}{\rightarrow} u \stackrel{*}{\leftarrow} y$. We say that x and y are *equivalent* if $x \stackrel{*}{\leftrightarrow} y$.

An important property to study about reduction relations is 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 $u \stackrel{*}{\leftarrow} x \stackrel{*}{\rightarrow} v$, 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, noted as $x \downarrow$) then $x \stackrel{*}{\leftrightarrow} y$ iff $x \downarrow = y \downarrow$. Provided normal forms are computable and identity in A is decidable, then the equivalence relation $\stackrel{*}{\leftrightarrow}$ is decidable in this case.

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

One important type of reduction relations is defined in the set $T(\Sigma, X)$ of first order terms in a given language, where Σ 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: $s \rightarrow_E t$ if it exists $l = r \in E$ and a substitution σ of the variables in l (the *matching* substitution) such that $\sigma(l)$ is a subterm of l and t is obtained from s by replacing the subterm $\sigma(l)$ by $\sigma(r)$. This reduction relation is of great interest in universal algebra because it can be proved that $E \models s = t$ iff $s \stackrel{*}{\leftrightarrow}_E t$. This implies decidability of every equational theory defined by a set of axioms E such that $\rightarrow_E$ is terminating and locally confluent. To emphasize the use of the equation $l = r$ from left to right as described above, we write $l \rightarrow r$ and talk about *rewrite rules*. A *term rewriting system* (TRS) is a set of rewrite rules. Unless noted otherwise, E is always a set of equations (equational axioms) and R is a TRS.

Local confluence is decidable for finite 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. The **Knuth-Bendix critical pair theorem** states that a TRS is locally confluent iff all its critical pairs are joinable. Thus, Church-Rosser property of terminating TRSs is a decidable property: it is enough to check if every critical pair has a common normal form. If a TRS R has a critical pair with different normal forms, there is still a chance to obtain a decision procedure for the equational theory of R, adjoining that equation as a new rewrite rule. This is the basis for the well-known *completion algorithms*. See [1] for details.

In the sequel, we describe the formalization of these properties in the ACL2 logic and a proof of them using the theorem prover. For the rest of the paper, when we talk about "prove" we mean "mechanically prove using ACL2".

2 Formalizing abstract reductions in ACL2

Our first attempt to represent abstract reduction relations in the ACL2 logic was simply to define them as binary boolean functions, using `encapsulate` to state their properties. Nevertheless, if $x \to y$, 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 in 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 such that when applied to an element, it returns a legal operator, whenever it exists, or `nil` otherwise (a *reducibility* test).

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 technical details, as in the rest of the paper):

```
(encapsulate
 ((legal (x u) t) (reduce-one-step (x u) 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 o))))
...)
```

The first line of every `encapsulate` is a description signature of the non-local functions partially defined. The two theorems assumed above as axioms are minimal requirements for every reduction we defined: if further properties (for example, local confluence) are assumed, they must 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 Church-Rosser property, local confluence or noetherianity and even prove non-trivial 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 (for example, to 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 $x = x_0 \leftrightarrow x_1 \leftrightarrow x_2 \ldots \leftrightarrow x_n = y$. This is done by the function `equiv-p` defined in figure 1. `(equiv-p x y p)` is t if p is a proof justifying that $x \overset{*}{\leftrightarrow} y$. A *proof* [1] is a sequence of legal steps and each proof step is a structure `r-step` with four fields: `elt1`, `elt2` (the elements connected), `direct` (the direction of the step) and `operator`. Two proofs justifying the same equivalence will be said to be *equivalent*. A proof step is *legal* (as defined by `proof-step-p`) if one of its elements is obtained applying the (legal) operator to the other.

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

(defun proof-step-p (s)
  (let ((e1 (elt1 s)) (e2 (elt2 s)) (op (operator s)) (dt (direct s)))
    (and (r-step-p s)
         (implies dt (and (legal elt1  op)
                          (equal (reduce-one-step elt1 op elt2) elt2)))
         (implies (not dt) (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)))))
```

Fig. 1. Definition of proofs and equivalence

Church-Rosser property and local-confluence can be redefined with respect to the form of a proof (subsections 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 $v \overset{*}{\rightarrow} x \overset{*}{\leftarrow} u$

2.2 Church-Rosser property and decidability

We describe how we formalized and proved the fact that every Church-Rosser and normalizing reduction relation is decidable. Valley proofs can be used to reformulate the definition of the Church-Rosser property: a reduction is Church-Rosser iff for every 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 called `transform-to-valley`. The concept of being normalizing can also be reformulated in terms of proofs: a reduction is normalizing if for every element there exists a proof to an equivalent irreducible element. This proof is given by the (Skolem) function `proof-irreducible` (note that we are not assuming noetherianity for the moment). Properties defining a Church-Rosser and normalizing reduction are `encapsulate`d as shown in figure 2, item `(a)`.

The function `equivalent` tests if normal forms are equal. Note that in this context, the normal form of an element `x` is the last element of `(proof-irreducible x)`:

```
(defun normal-form (x) (last-of-proof x (proof-irreducible x)))

(defun equivalent (x y) (equal (normal-form x) (normal-form y)))
```

[1] Do not confuse with proofs done using the ACL2 system.

```
;;; (a) Definition of Church-Rosser and normalizing reduction:
(encapsulate
  ((legal (x u) t) (reduce-one-step (x u) 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 equivalent-sound
  (implies (equivalent x y) (equiv-p x y (make-proof-common-n-f x y))))

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

Fig. 2. Church-Rosser and normalizing implies decidability

To prove decidability of a Church-Rosser and normalizing relation, it is enough to prove that `equivalent` is a complete and sound algorithm deciding the equivalence relation associated with 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 easily, without much guidance from the user. See the web page for details.

2.3 Noetherianity, local confluence and Newman's lemma

A relation is *well founded* in a set A if every non-empty subset has a minimal element. A restricted notion of well-foundedness is built into ACL2, based in the following meta-theorem: a relation in 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 (axiom of choice needed). In ACL2, once a relation is proved to satisfy these requirements, it can be used in the admissibility test for recursive functions. An arbitrary well-founded partial order `rel` can be defined in ACL2 as shown in item `(a)` of figure 3. Since only ordinals up to ε_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. Nevertheless, no particular properties of ε_0 are used in our proofs, except well-foundedness, so we think the same formal proofs could be carried out if higher ordinals were involved.

In item `(b)` of figure 3, a general definition of a noetherian and locally confluent reduction relation is presented. Local confluence is easily expressed in terms of the shape of proofs involved: a relation is locally confluent iff for every local peak proof there is an equivalent valley proof. This valley proof is given by the function `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 (AC). Thus, the general well-founded relation `rel` previously defined is used to justify noetherianity of the general reduction relation defined: for every element `x` such that a `legal` operator `u` can be applied, then `reduce-one-step` obtains an element less than `x` with respect to `rel`.

```
;;; (a) 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 :rewrite))

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

;;; (b) A noetherian and locally confluent reduction relation:
(encapsulate
 ((legal (x u) boolean) (reduce-one-step (x u) element)
  (reducible (x) boolean) (transform-local-peak (x) proof))
....
 (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 u) (rel (reduce-one-step x u) 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 theorems proved:
(defthm transform-to-valley-admission
  (implies (exists-local-peak p)
           (mul-rel (proof-measure (replace-local-peak p))
                    (proof-measure p))))

(defthm newman-lemma
  (let ((valley (transform-to-valley p)))
    (implies (equiv-p x y p)
      (and (steps-valley valley) (equiv-p x y valley)))))
```

Fig. 3. Newman's lemma

The standard proof of Newman's lemma found in the literature (see [1]) shows confluence by noetherian induction based on the reduction relation. The proof we obtained in ACL2 differs from the standard one and it is based on the proof given in [8]. In our formalization, we have to show that the reduction relation has the Church-Rosser property 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 can be defined to iteratively apply `replace-local-peak` (which replaces local peak subproofs by the equivalent proof given by `transform-local--peak`) until there are no local peaks. See definition in item `(c)` of figure 3.

Induction used in the standard proof is hidden here by the termination proof of `transform-to-valley`, needed for admission. The main proof effort was to show that in each iteration, some measure on the proof, `proof-measure`, decreases with respect to a well-founded relation, `mul-rel`. This can be seen as a normalization process acting on proofs. 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 a well-founded relation is also well-founded, a result interesting in its own (see the web page for details). Once `transform-to-valley` is admitted, it is relatively easy to show that it always returns an equivalent valley proof. See item `(d)` of figure 3.

Note that we gave here a particular "implementation" of `transform-to-valley` and proved as theorems the properties assumed as axioms in the previous subsection. The same was done with `proof-irreducible`. Decidability of noetherian and locally confluent reduction relations can now be easily deduced by functional instantiation from the general results proved in the previous subsection, allowing some kind of second-order reasoning. Name conflicts are avoided using Common Lisp packages.

3 Formalizing rewriting in ACL2

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

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 are defined and proved correct. See [11] for details of this work. Some functions of this library will be used in the following. Although definitions are not given here, their names suggest what they do.

The very general concept of operator can be concreted 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 R) tests if the rule of the operator op is in R, and can be applied to term at the position indicated by the operator (using the matching in 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 R) returns a legal equational operator to apply, whenever it exists, or nil otherwise.

Note that for every fixed term rewriting system R a different reduction relation is defined. The rewriting counterpart of the abstract equivalence equiv-p can be defined in an analogue way: (eq-equiv-p t1 t2 p R) tests if p is a proof of the equivalence of t1 and t2 in the equational theory of R. Due to the lack of space, we do not give the definitions here. Recall also 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:

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

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

Formalizing term rewriting in this way, we proved a number of results about term rewriting systems. In the following subsections, two relevant examples are sketched.

3.1 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 inclusion in contexts). Equational consequence, $E \models s = t$, can be alternatively defined as the least congruence relation containing E. In order to justify that the above described representation is appropriate, it would be convenient to prove that, fixed E, the relation given by (eq-equiv-p t1 t2 p E)[2], is the least congruence containing E.

We proved it in ACL2. In figure 4 we sketch part of our formalization showing that eq-equiv-p is a congruence. The ACL2 proof obtained is a good example of the benefits gained considering proofs as objects that can be transformed to obtain new proofs. Following Bachmair [2], we can define an "algebra" of proofs, a set of operations acting on proofs: 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.

3.2 A mechanical proof of Knuth-Bendix critical pair theorem

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

[2] Formally speaking, p have to be understood as existentially quantified

```
(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) (positionp pos term))
       (eq-equiv-p (replace-term term pos t1) (replace-term term pos t2)
                   (eq-proof-context p term pos) E))
```

Fig. 4. Congruence: an algebra of proofs

This result is formalized in our framework and proved guiding the system to the classical proof given in the literature (see [1] for example).

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

In our formalization, to prove the critical pair theorem we have to to define a function `transform-eq-local-peak` and prove that transforms every equational local peak proof to an equivalent valley proof. The final theorem is shown in item `(b)` of figure 5. It was not easy at all to perform the proof. The definition of `transform-eq-local-peak` (omitted) is a very long case distinction: five cases (some of them symmetric) has to be handled, according to the positions where the two rewritings of the equational local peak take place. The main proof effort was done to handle *non-critical* (or *variable*) overlaps. It is interesting to point that in most of textbooks and surveys this case is proved pictorially. Nevertheless, in our proof turned out to be the most difficult part. The complete proof needed more than one hundred lemmas and fifty auxiliary definitions and an extensive use of definitions and results of the library of first order terms, especially properties of the unification algorithm. The interested reader is urged to see the web page.

This theorem and the theorems described in Section 2 for abstract reduction relations were used to prove that equational theories described by a terminating TRS such that every critical pair has a common normal form are decidable. This result (which some authors call the Knuth-Bendix's theorem) is easily obtained by functional instantiation from the abstract case, taking advantage from the fact that the whole formalization is done in the same framework. Note how this last

```
;;; (a) A TRS with joinable critical pairs
(encapsulate
 ((RLC () t) (transform-critical-peak (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))
          (positionp pos l1) (not (variable-p (occurence l1 pos))))
      (let* ((cp-r (cp-r l1 r1 pos l2 r2)))
       (implies cp-r
        (and (eq-equiv-p (lhs cp-r) (rhs cp-r)
                         (transform-critical-peak l1 r1 pos l2 r2) (RLC))
             (steps-valley (transform-critical-peak l1 r1 pos l2 r2))))))))

;;; (b) Theorem proved:
(defun transform-eq-local-peak (p) ...)

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

Fig. 5. Knuth-Bendix critical pair theorem

result can be used to "certify" decisions procedures for equational theories defined by confluent and terminating TRSs.

4 Conclusions and further work

We have presented in this paper a case study of using the ACL2 theorem prover to formalize some basic results concerning reductions relations and term rewriting systems. Our formalization has the following main features:

– Reduction relations and their properties are stated in a very general framework, as explained in section 2.
– The concept of proof is a key notion in our formalization. Proofs are treated as manipulable objects that can be transformed to obtain new proofs.
– Functional instantiation is extensively used as a way of exporting results from the abstract case to the concrete case of term rewriting systems.

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 λ-calculus. For example, Huet [5] in the Coq system or Nipkow [10] 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 [12], using Nqthm. Although his work is on concrete reduction relations from the λ-calculus and he does not deal with the abstract case, some of his ideas are reflected in our work.

To our knowledge, no formalization of term rewriting systems has been done yet and consequently the formal proof of Knuth-Bendix critical pair theorem is the first one we know performed in a theorem prover.

We think the results presented here are important for two reasons. From a theoretical point of view, it is shown a very weak logic can be used to formalize properties of TRSs. From a practical point of view, this is an example on how formal methods can help in the design of symbolic computation systems. Usually, algebraic 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. Although a fully verified computer algebra system is currently beyond our possibilities, future work will be done to obtain verified Lisp code (executable in any compliant Common Lisp) for some basic procedures of term rewriting systems:

- To obtain certified decision procedures for some equational theories (or for the word problem of some finitely presented algebras) work has to be done to formalize in ACL2 well-known termination term orderings (recursive path orderings, Knuth-Bendix orderings, etc.). Maybe some problems will arise due to the restricted notion of noetherianity supported by ACL2.
- Our goal in the long term is to obtain a certified completion procedure written in Common Lisp. Although for the moment this may be far from the current status of our development, we think the work presented here is a good starting point.

References

1. BAADER, F., AND NIPKOW, T. *Term rewriting and all that.* Cambridge University Press, 1998.
2. BACHMAIR, L. *Canonical equational proofs.* Birkhäuser, 1991.
3. BOYER, R., AND MOORE, J. *A Computational Logic Handbook*, 2nd ed. Academic Press, 1998.
4. BUCHBERGER, B., AND LOOS, R. Algebraic simplification. In *Computer Algebra, Symbolic and Algebraic Computation. Computing Supplementum 4.* SV, 1982.
5. HUET, G. Residual theory in λ-calculus: a formal development. *Journal of Functional Programming*, 4 (1994), 475–522.
6. KAUFMANN, M., AND MOORE, J. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering 23*, 4 (1997), 203–213.
7. KAUFMANN, M., AND MOORE, J. http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html. ACL2 Version 2.4, 1999.
8. KLOP, J. Term rewriting systems. *Handbook of Logic in Computer Science* (1992).
9. LE CHENADEC, P. *Canonical forms in finitely presented algebras.* Pitman-Wiley, London, 1985.
10. NIPKOW, T. More Church-Rosser proofs (in Isabelle/HOL). In *13th International Conference on Automated Deduction* (1996), LNAI 1104, Springer-Verlag, pp. 733–747.
11. RUIZ-REINA, J., ALONSO, J., HIDALGO, M., AND MARTÍN, F. 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.
12. SHANKAR, N. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM 35*, 3 (1988), 475–522.
13. STEELE, G. *Common Lisp the Language, 2nd edition.* Digital Press, 1990.