A mechanical proof of Knuth-Bendix critical pair theorem (using ACL2) *

J.-L. Ruiz-Reina, J.-A. Alonso, M.-J. Hidalgo and F.-J. Martín
{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

1 Introduction

In this paper, we describe a mechanical proof of Knuth-Bendix critical pair theorem, carried out using the ACL2 theorem prover. ACL2 is both a logic and a mechanical theorem proving system supporting it, which evolved from Nqthm. The ACL2 logic is an existentially quantifier-free fragment of first-order logic with equality. ACL2 is also a programming language, an applicative subset of Common Lisp.

This work is a part of a larger project attempting to formalize theories about equational reasoning in the ACL2 logic, including abstract reduction relations, first-order terms and term rewriting systems, as described in [11]. ACL2 is used here as a metalanguage, in order to formalize properties of an object proof system (equational logic) in it.

As far as we know, this is the first formal proof of Knuth-Bendix critical pair theorem performed with a theorem prover. We think the results presented here are important for two reasons: from a theoretical point of view, it is shown how a very weak logic can be used to formalize and reason about non-trivial properties of equational reasoning. From a practical point of view, this is a first step to obtain mechanically verified decision procedures for some equational theories. As a by-product, "certified" compliant Common Lisp code is obtained for some basic algorithms used in rewriting theory (like subsumption, unification and computation of normal forms).

Due to the 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/`.

2 The ACL2 system

We briefly describe here the ACL2 theorem prover and its logic. The best introduction to ACL2 is [5]. See also [6], for an overview of the system. To obtain more background on ACL2, see the ACL2 user's manual in [7]. A description of the main proof techniques used in Nqthm, that are also used in ACL2, can be found in [3].

The ACL2 logic is a quantifier-free, first-order logic with equality, describing an applicative subset of Common Lisp. The syntax of terms is that of Common Lisp (we will assume the reader familiar with this language). The logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference include those for propositional calculus, equality, and instantiation. By the *principle of definition*, new function definitions (using `defun`) are admitted as axioms

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

only if there exists an ordinal measure in which the arguments of each recursive call decrease, ensuring in this way 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.

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

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.). 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, in a sense, the prover is interactive: the system's behaviour is influenced by the database of stored rules. In a typical user interaction, one guides the prover to a preconceived proof, by adding lemmas and definitions to be used in subsequent proofs.

3 Formalization of the critical pair theorem in ACL2

In this section we explain how we formalized the statement of the critical pair theorem in the ACL2 logic. The reader is assumed familiar with term rewriting systems (TRS) theory. A good introduction to that field can be found in [1] (notations and definitions used here are borrowed from that book). The critical pair theorem is stated as follows:

Theorem (Knuth and Bendix [8]): *Let R a term rewriting system such that $\rightarrow_R$ is terminating. Then $\rightarrow_R$ has the Church-Rosser property iff for every critical pair (s,t) of R the normal forms of s and t are equal.*

This theorem is a basic result in order to mechanize equational deduction and to develop *completion algorithms* to obtain decision procedures for equational theories (see [1] for details). In the sequel, we describe the formalization of this theorem in the ACL2 logic. One of the implications of the theorem is easy to prove: since Church-Rosser reductions provide unique normal forms for equivalent elements, critical pairs of a Church-Rosser TRS have common normal forms. Although we also proved this result in ACL2, we will focus in this paper on the reverse implication, which is the hard part of the theorem.

3.1 Equational theories

Since rewriting is a reduction relation defined on first-order terms, we need to reason about them in ACL2. 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. In order to make easy variable renamings, we consider an exception to this rule: `consp` objects with the symbol `var` in its car are also considered as variables (for example, `'(var x 1)`). The function `variable-p` recognizes variables. Substitutions are represented as association lists and equations and rules as dotted pairs of terms. The function `instance` implements the application of a substitution to a term. A number of functions acting on first-order terms were defined and theorems about them were proved. Especially important in this context are the functions dealing with the tree structure of terms: `positionp` 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 (see [1] for details).

Given a set of equations E, in order to formalize the relation $\overset{*}{\leftrightarrow}_E$ in ACL2, we first concentrate on the *one-step* reduction relation noted as $\rightarrow_E$. Our first attempt was to formalize the relation $\rightarrow_E$ as a binary boolean function defined on first-order terms. Nevertheless, if $t_1 \rightarrow_E t_2$, more important than the relation between t_1 and t_2 is the fact that t_2 can be obtained by applying a transformation (or *operator*) to t_1. Thus $\rightarrow_E$ can be seen as a binary function that given a term and an equational operator, returns another term, performing a *one-step reduction*. Equational operators are represented by structures with three fields containing the rewriting rule (equation) to apply, the position of the subterm to be rewritten and the matching substitution:

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

Of course, not every equational operator can be applied to every term: the left hand side (`lhs`) of the rule has to subsume the occurrence at the given position. The function `eq-legal` tests if a given operator can be applied to a term:

```
(defun eq-legal (term op E)
  (let ((pos (pos op)) (rule (rule op)) (sigma (matching op)))
    (and (eq-operator-p op) (member rule E) (positionp pos term)
         (equal (instance (lhs rule) sigma) (occurrence term pos)))))
```

The function `eq-reduce-one-step` applies an equational operator (which must be *legal*) to a term (replacing the indicated subterm by the corresponding instance of the right hand side (`rhs`) of the equation):

```
(defun eq-reduce-one-step (term op)
  (replace-term
     term (pos op) (instance (rhs (rule op)) (matching op))))
```

These two functions allow us to define the relation $s \overset{*}{\leftrightarrow}_E t$. Due to the constructive nature of the ACL2 logic, we have to include an argument with a sequence of steps $s = t_0 \leftrightarrow_E t_1 \leftrightarrow_E t_2 \ldots \leftrightarrow_E t_n = t$. In figure 1, we define the function `(eq-equiv-p t1 t2 p E)`. This function returns `t` if `p` is a proof justifying that $t_1 \overset{*}{\leftrightarrow}_E t_2$ and `nil` otherwise. A *proof* [1] is a sequence of legal *proof steps* and each proof step is a structure `r-step` with four fields: `elt1`, `elt2` (the terms connected), `direct` (the direction of the step) and `operator`. A proof step is *legal* (as defined by `eq-proof-step-p`) if one of its elements is obtained applying the (legal) `operator`

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

to the other. Two proofs justifying the same equivalence will be said to be *equivalent*. Note that the function `eq-equiv-p` implements a proof checker for equational theories, since $E \models s = t$ iff $s \overset{*}{\leftrightarrow}_E t$, thus formalizing equational deduction in ACL2.

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

(defun eq-proof-step-p (s E)
  (let ((t1 (elt1 s)) (t2 (elt2 s)) (op (operator s)) (dt (direct s)))
   (and (r-step-p s)
        (implies dt (and (eq-legal t1 op E)
                         (equal (eq-reduce-one-step t1 op) t2)))
          (implies (not dt) (and (eq-legal t2 op E)
                                 (equal (eq-reduce-one-step t2 op) t1))))))

(defun eq-equiv-p (t1 t2 p E)
  (if (endp p) (equal t1 t2)
    (and (eq-proof-step-p (car p) E) (equal t1 (elt1 (car p)))
         (eq-equiv-p (elt2 (car p)) t2 (cdr p) E))))
```

Fig. 1. Definition of proofs and equational theories

Church-Rosser property and local confluence can be redefined with respect to the form of a proof . 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 $s \leftarrow_E u \rightarrow_E t$ and `steps-valley` recognizes proofs of the form $s \overset{*}{\rightarrow}_E u \overset{*}{\leftarrow}_E t$.

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. Nevertheless, the formalization given in this subsection does not assume the set of equational axioms to be term rewriting systems.

3.2 Termination and reduction orderings

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

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 *measure* function $F : A \rightarrow Ord$ 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. 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 figure 2, the `encapsulate` mechanism is used to (partially) define a function `red<`, assumed to be a reduction order (dots are used to omit technical details, as in the rest of the paper). The function `(noetherian-red< TRS)` is defined to test if `red<` justifies termination of `TRS`. Note that an ordinal measure `fn-red<` is used to justify well-foundedness of `red<`. This property is stored by the system as a `:well-founded-relation` rule, which allows to use it in the admissibility test for recursive functions. In our case, a function that computes normal forms with respect to a terminating TRS will be admitted using that rule (section 4). Well-foundedness of `red<` will be also crucial to instantiate Newman's lemma.

```
(encapsulate
  ((red< (t1 t2) booleanp) (fn-red< (term) e0-ordinalp))
   ....
  (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 (positionp 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))))))
```

Fig. 2. A reduction order

3.3 The critical pair theorem

Using `encapsulate` we (partially) define a term rewriting systems `(RC)` assuming to have the properties in the hypothesis of the critical pair theorem: `(RC)` is terminating (justified by `red<`) and every critical pair obtained from rules in `(RC)` have a common normal form. See figure 3. In this formalization, the concepts of normal forms and critical pairs are implemented by the functions `RC-normal-form` and `cp-r`, 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 function `(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. An additional theorem about `r-reduce` was also needed, in order to prove termination of `RC-normal-form`: if `(r-reduce term (RC))` does not return `nil`, it returns a term less than `term` with respect to the well founded relation `red<` (see section 4).

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

The function `(cp-r l1 r1 pos l2 r2)` computes the critical pair (if it exists) determined by the rules `l1`→`r1` and `l2`→`r2` at position `pos` of `l1`. Before computing the critical pair, the rules are renamed to get their variables standardized apart. To reason properly about this function we needed to develop some results about variable renamings. And, more important, a verified unification algorithm was required.

```
(encapsulate
 ((RC () terminating-rewrite-system-with-common-n-f-critical-pairs))
 ...
 (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 (make-rule l1 r1) (RC)) (member (make-rule l2 r2)(RC))
                 (positionp 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))))))))
```

Fig. 3. A terminating TRS with common normal form critical pairs

Having assumed the properties of figures 2 and 3, in order to prove Knuth-Bendix critical pair theorem we have to show that `(RC)` is a term rewriting system with the Church-Rosser property: in the terminology of proofs, this means that for every proof in `(RC)` there exists an equivalent valley proof. Due to the absence of existential quantification in the ACL2 logic, we have to define a function `RC-transform-eq-proof` and prove that, given a proof `p` justifying $t_1 \overset{*}{\leftrightarrow}_{\texttt{(RC)}} t_2$, then `(RC-transform-eq-proof p)` returns an equivalent valley proof. This is the main theorem we proved:

```
(defthm kb-critical-pair-theorem
   (implies (eq-equiv-p t1 t2 p (RC))
            (and (eq-equiv-p t1 t2 (RC-transform-eq-proof p) (RC))
                 (steps-valley (RC-transform-eq-proof p)))))
```

The definition of `RC-transform-eq-proof` is omitted here due to the lack of space (see the web page for details). It has an important component: a function

`RC-transform-eq-peak` transforming every equational local peak proof to an equivalent valley proof, thus showing local confluence of `(RC)` (see section 4). The function `RC-transform-eq-proof` is defined to apply iteratively `RC-transform-eq-peak` until the proof obtained has no local peaks (i.e., it is a valley proof). Showing that this definition of `RC-transform-eq-proof` terminates is difficult. Note that once the definition is admitted, this can be seen almost as a proof of **Newman's lemma**: terminating and locally confluent reduction relations have the Church-Rosser property. In fact, we used a previously developed ACL2 library of results about abstract reductions relations including Newman's lemma, applied here as a particular case.

4 Some comments about the proof

First-order terms: Previous to the work presented here, we developed a library of definitions and theorems (*books* in ACL2 terminology) about first-order terms. These books were translated from a previous formalization done using Nqthm, where the lattice-theoretic properties of terms were proved (see [10] for details).

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 can be used to ensure that every execution in Common Lisp of the functions verified does not evaluate on arguments outside the intended domain (see [5] for details).

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

Abstract reductions and Newman's lemma: An *abstract reduction* [1] is simply a binary relation, and equational reductions are a particular case of abstract reductions. As part of our project to formalize properties of equational reasoning, we developed books proving results about abstract reduction relations. Concepts like Church-Rosser property, local confluence or noetherianity were defined in an abstract framework.

One of the main theorems in this library is Newman's lemma. We use this result in our proof of the critical pair theorem. This previous work about abstract reductions appears in [11], where we describe the formalization of abstract reduction relations in the ACL2 logic, a proof of Newman's lemma (among other results) and how the `encapsulate` mechanism is used to export these results from the abstract case to the equational case. See also the web page.

Reducibility and one-step rewriting: To instantiate our results from the abstract case to the equational case, we need to define a function `eq-reducible` such that `(eq-reducible term R)` returns a legal equational operator, whenever it exists, and `nil` otherwise [11]. We omit the definition of `eq-reducible` here, but these are the theorems we proved stating its main property:

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

(defthm not-eq-reducible-nothing-legal
  (implies (not (eq-reducible term R))
           (not (eq-legal term op R)))
```

Having defined `eq-reducible` and `eq-reduce-one-step`, this 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. Indeed, this is the definition of normal form we used for reasoning. However, this definition is only useful for theoretical purposes: obviously, the normal form calculation can be optimized in several ways. For example, a function computing normal forms neither need to build an equational operator in every rewriting step nor traversing the terms twice, searching a legal equational operator, and then applying the reduction step.

We defined a more efficient (although not optimal) version of one-step rewriting, named `r-reduce`, as we said in subsection 3.3. 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 equivalence with the improved version, and then we stated the final version of the theorem with it.

Reduction orderings: Once `red<` has been assumed to be a reduction ordering and function `noetherian-red<` has been defined (figure 2), we proved that the reduction relation $\rightarrow_{\texttt{R}}$ is terminating, whenever `R` is a TRS such that `(noetherian-red< R)`:

```
(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)))
```

This lemma is essential to prove termination of the function `RC-normal-form` defined in figure 3. The proof is almost automatic, using stability and compatibility of `red<`. The lemma is also needed to export Newman's lemma to the equational case.

Although the (partial) definition of the reduction ordering `red<` given in figure 2 works well from a theoretical point of view, we think that 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. Future work will be done in this direction.

Local confluence: Having the properties about subsumption and unification verified and Newman's lemma as a result in libraries previously developed, the main proof effort was done to prove local confluence of the term rewriting system `(RC)`. As outlined in subsection 3.3, in order to prove local confluence is enough to define a function `RC-transform-eq-peak` acting on proofs, and prove that it obtain an equivalent valley proof for every equational local peak proof.

As a basis for our formal proof of local confluence of `(RC)`, we follow Huet's proof given in [4]. The proof is obtained as a typical (but very long) interaction with the ACL2 theorem prover: the user guides the prover by adding lemmas and definitions used later as rewriting rules. Most of lemmas are proved using only simplification and induction.

An important feature of our formalization is to consider the object proofs (equational proofs) as elements that can be transformed to obtain new proofs. Following Bachmair [2], we can define an "algebra" of equational proofs, a set of operations acting on proofs: concatenation of proofs, reverse proofs, instantiation of the elements involved in a proof and inclusion of the elements of a proof as subterms of a common term (inclusion in a context). 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. See [11] for a description of this issue.

This "algebra" of equational proofs allows us to control the complexity of our ACL2 proofs: 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 [4], the proof is mainly structured to deal with three cases, according to the relatives positions of the subterms where the two rewriting steps (in a local peak) may occur:

- *Disjoint rewriting.* This is the easiest case to prove, although an induction hint has to be given to the prover in order to reason properly about replacement in disjoint positions of a term.
- *Non-critical overlap.* 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 mechanical proof turned out to be the most difficult part. For example, proving Proposition 3.6 of [4] was hard. It was even needed to design an induction scheme not discovered by the heuristics of the prover.
- *Critical overlap*: The critical overlap case was easier to prove than the previous case, but we must not forget that this case relies heavily on previously verified properties of a unification algorithm.

About the proof effort: It is difficult to give a measure of our proof effort, since different collections of books were used, and not every of them were developed exclusively for the work presented here. We think that our formalization of the critical pair theorem is a good example of integration of different theories: books about lists, arithmetic, first-order terms, abstract reductions, equational theories and term rewriting were developed separately and combined together to prove the theorem. Although books about first-order terms [10] and abstract reductions [11] are important contributions to prove the critical pair theorem, we concentrate here on that part of the work mainly devoted to the proof of the theorem.

The proof described here has been structured in four books, chronologically developed in the following order (every book needs results from its predecessor):

1. Definition and main properties of the equational theory given by a set of equational axioms: `equational-theories.lisp` (this book also contains some results not needed for the proof of the critical pair).

2. Definitions and basic properties of term rewriting systems: `rewriting.lisp`. The notions of reducibility, reduction orderings and one-step rewriting are formalized in this book.
3. The proof of the critical pair lemma: `critical-pairs.lisp`. Local confluence of every TRS with joinable critical pair is proved.
4. The proof of Knuth-Bendix critical pair theorem: `knuth-bendix.lisp`. This final part of the development uses Newman's lemma by functional instantiation, besides the theorems in the previous book, to obtain the theorem as stated in subsection 3.3.

Book	Lines	Definitions	Theorems	Hints
`equational-theories`	543	11	29	8
`rewriting`	720	13	38	9
`critical-pairs`	2129	43	112	26
`knuth-bendix`	614	23	24	10
Total	4006	90	203	53

Fig. 4. Quantitative information on the proof

Figure 4 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, 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 or for using instances of previous theorems.

It is clear from the table that the main proof effort was done to prove local confluence of `(RC)` (the book `critical-pair.lisp`). This is, up to now, even the largest book we developed in our current project (for example, definition and verification of unification needed 8 definitions and 96 theorems, and the proof of Newman's lemma needed 22 definitions and 73 theorems).

5 Conclusions and further work

We have presented a formalization and a mechanical proof of the Knuth-Bendix critical pair theorem, developed in the ACL2 system. This is an example (one more) of how a restricted logic like the ACL2 logic (a quantifier-free first-order logic with equality) can be used to formalize and prove non-trivial theorems. Related work had been done by Shankar [12], where the Boyer-More logic is used as a metalanguage to formalize Gödel's incompleteness theorem and Church-Rosser theorem for lambda-calculus, and Nqthm is used to prove those results. To our knowledge, no other formal proof of Knuth-Bendix critical pair theorem had been performed by a theorem prover.

Although a fully verified equational theorem prover is currently beyond our possibilities, this can be seen as an approach to "certify" some of its components. For

example, the guard verification mechanism in ACL2, can be used to obtain verified compliant Common Lisp code for some basic procedures in term rewriting: subsumption, unification, normal form computation and critical pairs. A recent work using ACL2 [9], opens another possible application: ACL2 functions can be combined with non-ACL2 programs to check properties of their outputs.

Since equational theories described by a terminating and Church-Rosser TRS are decidable, this work opens a possibility to obtain mechanically verified decision procedures (executable in Common Lisp) for some equational theories. An open problem is to prove termination (in the ACL2 logic) of concrete TRSs. Work has to be done to formalize in ACL2 well-known termination term orderings (RPO, KBO, etc.). Maybe some problems will arise due to the restricted notion of well-foundedness (ordinal types below ε_0) 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 S. *A Computational Logic Handbook*, 2nd ed. Academic Press, 1998.
4. Huet, G. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM* (1980).
5. Kaufmann, M., Manolios, P., and Moore, J S. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
6. Kaufmann, M., and Moore, J S. 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 S. http://www.cs.utexas.edu/users/moore/acl2/acl2-doc.html. ACL2 Version 2.4, 1999.
8. Knuth, D., and Bendix, P. Simple word problems in universal algebras. In *Computational problems in abstract algebras*, J Leech, Ed. Pergamon Press, 1970, pp. 263–297.
9. McCune, W., and Shumsky, O. Ivy: A preprocessor and proof checker for first-order logic. In *Computer-Aided Reasoning: ACL2 Case Studies*, M. Kaufmann, P. Manolios, and J S. Moore, Eds. Kluwer Academic Publishers, 2000, ch. 16.
10. Ruiz-Reina, J.-L., Alonso, J.-A., Hidalgo, M.-J., and Martín, F.-J. Mechanical verification of a rule based unification algorithm in the Boyer-Moore theorem prover. In *AGP'99 Joint Conference on Declarative Programming* (1999), pp. 289–304.
11. Ruiz-Reina, J.-L., Alonso, J.-A., Hidalgo, M.-J., and Martín, F.-J. Formalizing rewriting in the ACL2 theorem prover. To be presented at *AISC'2000 (Fifth International Conference Artificial Intelligence and Symbolic Computation*), 2000.
12. Shankar, N. *Metamathematics, Machines, and Godel's Proof.* Cambridge University Press, 1994.