J.L. RUIZ–REINA, J.A. ALONSO, M.J. HIDALGO AND F.J. MARTÍN–MATEOS [*]

TERMINATION IN ACL2 USING MULTISET RELATIONS [†]

ABSTRACT: We present in this paper a case study of the use of the ACL2 system, describing an ACL2 formalization of multiset relations, and showing how multisets can be used to prove non-trivial termination properties. Every relation on a set A induces a relation on finite multisets over A; it can be shown that the multiset relation induced by a well-founded relation is also well-founded. We prove this property in the ACL2 logic, and use it by functional instantiation in order to provide well-founded relations for the admissibility test of recursive functions. We also develope a macro `defmul`, to define well-founded multiset relations in a convenient way. Finally, we present three examples illustrating how multisets are used to prove non-trivial termination properties in ACL2: a tail-recursive version of a general binary recursion scheme, a definition of McCarthy's 91 function and a proof of Newman's lemma for abstract reduction relations. These case studies show how non-trivial mathematical results can be stated and proved in the ACL2 logic, in spite of its apparent lack of expressiveness.

1 INTRODUCTION

The ACL2 system [Kaufmann *et al.*, 2000; Kaufmann and Moore, 2002] consists of a programming language (an extension of an applicative subset of Common Lisp), a logic describing the programming language and a theorem prover supporting mechanized deduction in the logic. The ACL2 logic is a quantifier-free, first-order logic with equality: its syntax is that of Common Lisp and it includes axioms for propositional logic and for a number of Lisp functions and data types; rules of inference include those for propositional calculus, equality, instantiation and a principle of proof by induction.

ACL2 is usually applied for software or hardware verification [Kaufmann and Moore, 2002]. In this paper, we present a case study showing that also non-trivial mathematical theorems can be formalized and proved in a system like ACL2, with such a restricted logic. In particular, we show how theorems relying on non-trivial termination properties of recursive functions can be formalized and proved in the system.

In ACL2, new function definitions are admitted as axioms only if there exists a measure in which the arguments of each recursive call can be shown to decrease in some well-founded sense. This is what we call the *principle of definition*. This principle ensures that no inconsistencies are introduced by new definitions, and also provides a basis to perform proofs by induction according to the recursive schemes of the functions that appear in a conjecture.

[*]Dpto. Ciencias de la Computación e Inteligencia Artificial, Universidad de Sevilla.

[†]This work has been supported by MCYT: Project TIC2000–1368–C03–02 and FEDER funds.

Fairouz Kamareddine (eds.),
Thirty Five Years of Automating Mathematics *1–29.*
© *2003, Kluwer Academic Publishers. Printed in the Netherlands.*

When submitting to ACL2 a terminating function with a simple recursive scheme, the prover, using some heuristics, is usually able to prove its termination. Nevertheless, if the termination is not trivial, the user often has to explicitly provide a well-founded relation, a measure and some assistance (in the form of lemmas) to get the termination proof. In this paper we present a tool to obtain a class of well-founded relations that can be used to prove non-trivial termination properties.

In [Dershowitz and Manna, 1979], it is proved that every well-founded relation on a set A induces a well-founded relation on the set of finite multisets of elements taken from A. We have formalized this theorem using ACL2, and stated it in an abstract way. This allows to instantiate the theorem to prove well-foundedness of concrete multiset relations. Based on this idea, we have developed a macro `defmul` in order to easily define induced multiset relations. Besides defining the multiset relation induced by a given relation, this macro automatically proves, by functional instantiation, well-foundedness of the defined multiset relation, provided that the original relation is well-founded. Such well-founded multiset relations can then be used in the admissibility test for recursive functions, allowing the user to provide a particular multiset measure in order to prove termination of recursively defined functions.

This paper is structured as follows. The second section presents a brief description of ACL2. The third section presents how we have formalized and proved well-foundedness of multiset relations induced by well-founded relations. The fourth section presents the macro `defmul` and it is shown how it can be used to define multiset well-founded relations. In the fifth section, three case studies of increasing complexity are presented, showing how multisets can be used to prove non-trivial termination properties. The first one is the transformation of a binary recursion scheme into an equivalent tail-recursive scheme. The second one shows admissibility of an iterative version of McCarthy's 91 function. The third one is a proof of Newman's lemma about abstract reduction relations: terminating and locally confluent reduction relations are confluent. Finally, we draw some conclusions.

We will skip details of the mechanical proofs and omit some technical questions such as the hints given to the theorem prover. The complete development is available on the web at `http://www.cs.us.es/~jruiz-/acl2-mul/`. This paper is a revised version of [Ruiz-Reina *et al.*, 2000], presented at the *Second ACL2 Workshop*.

2 AN INTRODUCTION TO THE ACL2 SYSTEM

ACL2 stands for A Computational Logic for an Applicative Common Lisp. The system evolved from the Boyer-Moore theorem prover, also known as Nqthm [Boyer and Moore, 1998]. Roughly speaking, ACL2 is a programming language, a logic and a theorem prover. As a programming language, it is an extension of an applicative subset of Common Lisp [Steele, 1990] (we will assume the reader familiar with this language). The ACL2 logic describes the programming language, with a formal syntax, axioms, rules of inference and a semantic model. ACL2 also provides a theorem prover allowing mechanized reasoning in the logic. Thus, the system constitutes an environment in which functions can be defined and executed, and their properties can be formally specified and proved with the assistance of a mechanical theorem prover.

Let us illustrate these features with an example. The following definition introduces a function `remove-one` which eliminates the first occurrence of an element `x` from a list `l` (whenever it exists):

```
(defun remove-one (x l)
  (if (atom l)
      l
    (if (equal x (car l))
        (cdr l)
      (cons (car l) (remove-one x (cdr l))))))
```

Once defined, the function can be executed on some explicit values, as in any Common Lisp. For example, the expression `(remove-one 3 '(5 3 7 3))` is evaluated to `(5 7 3)`. But we can also state and prove formal properties about the function, using the ACL2 logic. From the logical point of view, the above definition introduces an axiom in the ACL2 logic, equating the term `(remove-one x l)` to the term given by the body of the definition. Using this axiom and the primitive axioms and rules of inference of the ACL2 logic, it is possible, for example, to prove the following property about `remove-one`:

```
(defthm remove-one-no-duplicatesp
  (implies (no-duplicatesp l)
           (no-duplicatesp (remove-one x l))))
```

Note that the same language is used for computing and proving. Here the function `no-duplicatesp` (checking if its argument is a list without repeated elements) is a primitive ACL2 function. The function `implies` is the function corresponding to the propositional implication connective and the variables `x` and `l` are implicitly universally quantified.

The above theorem can be proved by the theorem prover (although it needs some assistance from the user) using mainly induction and rewriting. The command `defthm` starts a proof attempt in the ACL2 theorem prover. If successful, the theorem is stored as a rule (by default, a rewriting rule) that can be used in subsequent proof attempts.

In the following, we will briefly describe the ACL2 logic and its theorem prover. To obtain more background on ACL2, see [Kaufmann *et al.*, 2000] or the user's manual in [Kaufmann and Moore, 2002]. A description of the main proof techniques used in Nqthm, also used in ACL2, can be found in [Boyer and Moore, 1998].

2.1 The logic

The ACL2 logic is a quantifier-free, first-order logic with equality, describing an applicative subset of Common Lisp functions acting on a set of objects called the *ACL2 universe*. This universe is partitioned into five sets of objects: numbers, characters, strings, symbols and ordered pairs. Essentially, these data types are the same than in Common Lisp and they are represented in the same way. There are two special symbols, `t` and `nil`, denoting respectively "true" and "false" (although any object other than `nil` may serve as an indicator of "true").

The syntax of *terms* in the ACL2 logic is that of Common Lisp, and therefore uses prefix notation. Roughly speaking, a term of the ACL2 logic is a constant, a variable symbol or the application of function symbol of arity n to n terms. The logic includes axioms describing the behavior of a subset of applicative Common Lisp functions on the above five data types. For example, there are axioms describing the functions `cons`, `car` and `cdr` (the constructor, the left part and the right part of an ordered pair, respectively).

Rules of inference include those for propositional logic (with propositional connectives `if`, `and`, `or`, `not`, `implies` and `iff`), equality (the predicate `equal`) and instantiation of variables. The logic is quantifier-free and the variables in a term are implicitly universally quantified. The logic also provides a principle of *proof by induction* that allows to prove a conjecture splitting it into cases and inductively assuming instances of the conjecture that are smaller with respect to some well-founded measure.

In ACL2, the primitive notion of well-foundedness is given by a constructive representation of the ordinals up to ε_0, in terms of natural numbers and ordered pairs. Natural numbers are represented by the corresponding ACL2 numbers. Ordered pairs are used to represent non-natural ordinal numbers: roughly speaking, the elements of the list representing an ACL2 ordinal are the ACL2 ordinals (in decreasing order) corresponding to the non-zero powers of ω in its Cantor normal form, and the coefficients of the Cantor normal form are represented by the number of repetitions of the corresponding power of ω. The final `cdr` of the object is the "natural part" of the ordinal. For example, ω is represented as `(1 . 0)`, $\omega^2 + \omega \cdot 3 + 7$ as `(2 1 1 1 . 7)`, ω^ω as `((1 . 0) . 0)` and $\omega^\omega + \omega^{85} + \omega^3 \cdot 2 + 5$ as `((1 . 0) 85 3 3 . 5)`. The ACL2 primitive function `e0-ordinalp` recognizes those ACL2 objects that represent ordinals, and the function `e0-ord-<` defines the usual order between ordinals. This order is assumed to be well-founded on the set of ACL2 ordinals, and it is the only primitive well-founded relation in the ACL2 logic. As we will see in subsection 3.2, it is possible for the user to define other well-founded relations in ACL2.

By the *principle of definition*, new function definitions (using `defun`) are admitted as axioms in the logic only if there exists a well-founded measure in which the arguments of each recursive call (if any) decrease, thus proving its termination. For example, the previous function `remove-one` is admitted in the logic justified by a measure counting the number of "conses" of its second argument.

In addition to the definition principle, the *encapsulation principle* (via `encapsulate`) allows the user to introduce new function symbols by axioms constraining them to have certain properties. To ensure consistency, local 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. For example, the following definition introduces a new function symbol `sel`, of one argument, and constrains `sel` to denote a function that selects an element from every non-empty list:

```
(encapsulate
 ((sel *) => *)

 (local (defun sel (l) (car l)))

 (defthm sel-selects
   (implies (not (atom l)) (member (sel l) l))))
```

The first part of every `encapsulate` describes the *signature* of the functions introduced (in this case, the function `sel` with one argument). In this example, the local witness is the function `car`, proved to verify the non-local property `sel-selects`. Outside the scope of the `encapsulate`, this non-local property is the only property assumed about the function `sel`.

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 [Kaufmann and Moore, 2001] for details).

2.2 The theorem prover

The ACL2 theorem prover is inspired by Nqthm, adapted to the ACL2 logic and considerably improved. The main proof techniques used by ACL2 are simplification and induction[1]. Roughly speaking, when the prover tries to prove a conjecture, it simplifies the formula. If it obtains `t`, then the conjecture is proved. Otherwise, it guesses an (often suitable) induction scheme, and recursively tries to prove the subgoals generated.

Simplification is a process combining term rewriting with some decision procedures (linear arithmetic, type set reasoner, etc.). To rewrite a conjecture, the system uses axioms, definitions and theorems previously proved by the user, stored as rewrite rules. In addition to simplification, one of the key points in the success of ACL2 and its predecessor is the use of sophisticated heuristics for discovering an induction scheme suitable for a proof by induction of a conjecture. This induction scheme is suggested by the recursive functions occurring in the formula.

For example, in the proof attempt of the theorem `remove-one-no--duplicatesp` above, the system tries a proof by induction, since the conjecture cannot be simplified using the current definitions and rewrite rules. The following induction scheme is automatically generated, where `(p L X)` abbreviates the theorem to be proved:

```
(AND (IMPLIES (AND (NOT (ATOM L))
                   (NOT (EQUAL X (CAR L)))
                   (p (CDR L) X))
              (p L X))                     ; induction step
     (IMPLIES (AND (NOT (ATOM L)) (EQUAL X (CAR L)))
              (p L X))                     ; base case
     (IMPLIES (ATOM L) (p L X)))           ; base case
```

This induction scheme is suggested by the recursive definition of the function `remove-one` appearing in the conjecture. It consists of two base cases and one induction step, where the induction hypothesis is `(p (CDR L) X)`, corresponding to the recursive call of `remove-one`.

Although this induction scheme is suitable for proving the theorem, ACL2 fails to prove it in its first attempt. Inspecting the output of the failed proof, it turns out that the proof would succeed if the following lemma would have been known by the system:

```
(defthm member-remove-one
  (implies (not (member x l))
           (not (member x (remove-one y l)))))
```

When submitted by the user, this lemma can be proved by the system automatically. Once proved, the lemma is stored as a rewrite rule. In subsequent proofs this rule will be used to rewrite instances of `(member x`

[1]Although there are other proof techniques like *destructor elimination*, *generalization*, *elimination of irrelevances* or *cross-fertilization.*

`(remove-one y l))` to `nil`, whenever the corresponding instance of the hypothesis in the rule, `(not (member x l))`, holds. With this rule, a second proof attempt of the theorem `remove-one-no-duplicatesp` succeeds.

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. As shown in the above example, very often non-trivial 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[2]. 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 non-trivial result (see [Kaufmann *et al.*, 2000] for a description of how to inspect failed proofs).

This kind of interaction with the system is called "The Method" by the authors of the system. 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". As a result of this interaction, the user obtains a file containing (mainly) definitions and theorems, called a *book* in the ACL2 terminology. A book can be *certified* (all its definitions are admissible and its theorems are proved) and used by other books.

Finally, `defthm` is not the only command that calls the theorem prover. When submitting a definition with `defun`, the prover tries to justify its termination, by proving that some well-founded measure of the arguments decreases in each recursive call. The prover has some heuristics to automatically obtain an ordinal measure needed to justify the termination of a given definition. Although these heuristics are often successful, sometimes, if the termination is not trivial, the user has to explicitly provide a well-founded relation and a measure to get the termination proof, as we will see in the examples of section 5.

3 FORMALIZATION OF MULTISET RELATIONS IN ACL2

3.1 Multisets: definitions and properties

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

[2]The user may also assist the prover by giving some *hints* when submitting the conjecture; for example, indicating the use of some specific lemma instance or providing an induction scheme.

We will use standard set notation to denote multisets. For example, if $A = \{a, b, c\}$, an example of a multiset over A is $M = \{a, b, b, b\}$, an abbreviation of the function $M(a) = 1$, $M(b) = 3$ and $M(c) = 0$. Thus, $\{a, b, b, b\}$ is identical to the multiset $\{b, b, a, b\}$, but distinct from the multiset $\{a, b, b\}$.

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

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

DEFINITION 1. Given a relation $<$ on a set A, the **multiset relation** induced by $<$ on $\mathcal{M}(A)$, denoted as $<_{mul}$, is defined as: $N <_{mul} M$ if there exist $X, Y \in \mathcal{M}(A)$ such that $\emptyset \neq X \subseteq M$, $N = (M \setminus X) \cup Y$ and for all $y \in Y$ there exists $x \in X$ such that $y < x$.

For example, if $A = \{a, b, c, d, e\}$ and $b < a$, $d < c$, from the definition we have $\{a, b, b, b, b, d, d, d, d, d, e\} <_{mul} \{a, a, b, c, d, e\}$ by replacing $X = \{a, c\}$ by $Y = \{b, b, b, d, d, d, d\}$. It can be shown that if $<$ is a strict ordering, then so is $<_{mul}$. In such case we talk about *multiset orderings*.

A relation $<$ on a set A is *terminating* if there is no infinite decreasing[3] sequence $x_0 > x_1 > x_2 \ldots$. An important property of multiset relations on finite multisets is that they are terminating when the original relation is terminating, as stated by the following theorem [Dershowitz and Manna, 1979]:

THEOREM 2. *Let $<$ be a terminating relation on a set A, and $<_{mul}$ the multiset relation induced by $<$ on $\mathcal{M}(A)$. Then $<_{mul}$ is terminating.*

The above theorem provides a tool for showing termination of recursive function definitions, by using multisets: show that some multiset measure decreases in each recursive call comparing multisets with respect to the relation induced by a given terminating relation. In the following subsection, we explain how we formalize Theorem 2 in the ACL2 logic.

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

3.2 Well-founded multiset relations in ACL2

Let us now explain how we formalize the assumption of Theorem 2. As we said before, ACL2 contains the definition of the ordinals up to ε_0, given by the predicates `e0-ordinalp` (recognizing those objects that represent ordinals) and `e0-ord-<` (the order between ordinals). In addition, a restricted notion of terminating relation is built into ACL2 based on the following meta-theorem (axiom of choice needed): a relation $<$ on a set A is terminating 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[4]. In this case, we also say that the relation is *well-founded*. Thus, a general well-founded relation `rel` defined on a set of objects satisfying a property `mp` can be defined in ACL2 as shown below (dots are used to omit the definition of the local witnesses, as in the rest of the paper):

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

The predicate `mp` (called the *measure property*) recognizes the kind of objects that are ordered in a well-founded way by the relation `rel`. The *embedding* function `fn` is an order-preserving function mapping every measure object to an ordinal. The theorem `rel-well-founded-relation-on--mp` above is called the *well-foundedness theorem* for `rel`, `mp` and `fn`. In ACL2, every particular well-founded relation has to be given by means of three functions (a binary relation, a measure property and an embedding function) and the corresponding well-foundedness theorem for such functions. Note that this notion of well-foundedness is restricted: since only ordinals up to ε_0 are formalized in the ACL2 logic, a limitation is imposed on the maximal order type of well-founded relations that can be formalized. Consequently, our formalization suffers from the same restriction.

Once a relation is proved to satisfy a theorem of the above form, the relation can be used in the admissibility test for recursive functions[5]. Note that here `encapsulate` is used to define a general well-founded relation `rel`, without any additional restriction.

[4]Note that we are denoting the relation on A and the ordering between ordinals using the same symbol $<$.

[5]For that purpose, the well-foundedness theorem has to be stored with `:rule-classes :well-founded-relation`. In general, the `:rule-classes` argument of `defthm` specifies how the theorem will be used by the system in the sequel.

Let us now deal with the formalization of multiset relations. We represent multisets in ACL2 as true lists (a *true list* is either `nil` or an ordered pair whose right component is a true list, usually called a *list* in Lisp terminology). Given a predicate `mp` describing a set A, finite multisets over A are described by the following function:

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

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

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

Let us now describe how we define the multiset relation. The definition of $<_{mul}$ given in the preceding subsection is quite intuitive but, due to its many quantifiers, difficult to implement. Instead, we will use a somewhat restricted definition, based on the following theorem (lemma 2.5.6 in [Baader and Nipkow, 1998]):

THEOREM 3. *Let $<$ be a strict ordering on a set A, and M, N two finite multisets over A. Then $N <_{mul} M$ iff $M \setminus N \neq \emptyset$ and for all $n \in N \setminus M$, there exists $m \in M \setminus N$, such that $n < m$.*

From the computational point of view, the main advantage of this alternative definition[6] is that the we do not have to search the multisets X and Y of the original definition because we can take $M \setminus N$ and $N \setminus M$, respectively. Thus, given a defined (or constrained) binary relation `rel`, we define the induced relation on multisets based on this alternative definition:

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

[6]It should be remarked that this equivalence is true only when $<$ is a strict partial ordering. Anyway, this is not a severe restriction. Moreover, well-foundedness of $<_{mul}$ also holds when this restricted definition is used, even if the relation $<$ is not transitive.

```
(defun forall-exists-rel-bigger (l m)
  (if (atom l)
      t
    (and (exists-rel-bigger (car l) m)
         (forall-exists-rel-bigger (cdr l) m))))
(defun mul-rel (n m)
  (let ((m-n (multiset-diff m n))
        (n-m (multiset-diff n m)))
    (and (not (atom m-n))
         (forall-exists-rel-bigger n-m m-n))))
```

Finally, let us see how we can formalize Theorem 2 in the ACL2 logic, stating well-foundedness of the relation `mul-rel`. As said before, in order to establish well-foundedness of a relation in ACL2, in addition to the relation (`mul-rel` in this case), we have to give the measure property and the embedding function, and then prove the corresponding well-foundedness theorem. Since `mul-rel` is intended to be defined on multisets of elements satisfying `mp`, then `mp-true-listp` is clearly the measure property in this case. Let us suppose we have defined a suitable embedding function called `map-fn-e0-ord`. Then Theorem 2 is formalized as follows:

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

In the next subsection we show a suitable definition of `map-fn-e0-ord` and describe some aspects of the ACL2 proof of this theorem.

3.3 A proof of well-foundedness of the multiset relation

In the literature [Dershowitz and Manna, 1979], Theorem 2 is usually proved using König's lemma: every infinite and finitely branched tree has an infinite path. More recently, a constructive formal proof is described in [Persson, 1999] (chapter II), also formalized in the HOL system [Slind, 2000]. Nevertheless, we have to find a different proof in ACL2, since well-foundedness in ACL2 has to be stablished defining an order-preserving function `map-fn-e0-ord` from `mp-true-listp` objects to `e0-ordinalp` objects. Thus, our proof is based on the following result from ordinal theory: given an ordinal α, the set $\mathcal{M}(\alpha)$ of finite multisets of elements of α (ordinals less than α), ordered by the multiset relation induced by the order between ordinals, is order-isomorphic to the ordinal ω^α and the isomorphism is given by the function H where $H(\{\beta_1, \ldots, \beta_n\}) = \omega^{\beta_1} \oplus \ldots \oplus \omega^{\beta_n}$, where $\oplus$ denotes natural addition of ordinals (see, for example, [Levy, 1979]).

As a by-product, an interesting property about multiset well-founded relations can be deduced. Since $\alpha \leq \varepsilon_0$ implies $\omega^\alpha \leq \omega^{\varepsilon_0} = \varepsilon_0$, this means that one can always prove, in the ACL2 logic, well-foundedness of the multiset relation induced by a given well-founded ACL2 relation (i.e., using embeddings in the ordinal ε_0). This is not the case, for example, of lexicographic products, since the maximal ordinal type of a lexicographic product of two ACL2 well-founded relations may be greater than ε_0.

The isomorphism H above suggests the following definition of the embedding function `map-fn-e0-ord`: given a multiset of elements satisfying `mp`, apply `fn` to every element to obtain a multiset of ordinals. Then apply H to obtain an ordinal less than ε_0. If ordinals are represented in ACL2 notation, then the function H can be easily defined, provided that the function `fn` returns always a non-zero ordinal: the function H simply has to sort the ordinals in the multiset and add `0` as the final `cdr`. These considerations lead us to the following definition of the embedding function `map-fn-e0-ord` [7].

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

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

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

(defun map-fn-e0-ord (l)
  (if (not (atom l))
      (insert-e0-ord-< (fn1 (car l))
                       (map-fn-e0-ord (cdr l)))
    0))
```

Once `map-fn-e0-ord` has been defined, let us now deal with the ACL2 mechanical proof of the well-foundedness theorem for `mul-rel`, `mp-true--listp` and `map-fn-e0-ord` as stated at the end of subsection 3.2 by `multiset-extension-of-rel-well-founded`. The part of the theorem establishing that `(map-fn-e0-ord x)` is an ordinal when `(mp-true-listp x)` is not difficult, and can be proved in ACL2 with minor help form the user. The hard part of the theorem is to show that `map-fn-e0-ord` is order-preserving. Here is an informal proof sketch:

[7] Note that the non-zero restriction on `fn` is easily overcome, defining (the macro) `fn1` equal to `fn` except for integers, where `1` is added. In this way `fn1` returns non-zero ordinals for every measure object and it is order-preserving iff `fn` is.

Proof sketch: Let us denote, for simplicity, the functions `fn1` and `map--fn-e0-ord`, as f and f_{mul}, and the relations `rel`, `mul-rel` and `e0-ord-<` as $<_{rel}$, $<_{mul}$ and $<$, respectively. Let M and N be two multisets of `mp` elements such that $N <_{mul} M$. We have to prove that $f_{mul}(N) < f_{mul}(M)$. We can apply induction on the number of elements of N. Note that M can not be empty, and if N is empty the result trivially holds. So let us suppose that M and N are not empty. Let $f(x)$, $f(y)$ be the biggest elements of $f[N]$ and $f[M]$, respectively. Note that $f(x)$ and $f(y)$ are the `car` elements of $f_{mul}(N)$ and $f_{mul}(M)$, respectively (recall that we are dealing with the ACL2 representation of ordinals, so it makes sense to talk about the `car` or the `cdr` of an ordinal). Since $f(x)$ and $f(y)$ are ordinals, three cases may arise:

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

The ACL2 proof we carried out is based on this informal description. First the above induction scheme must be supplied as hint and then lemmas to handle each of the cases generated by the induction scheme have to be proved, leading ACL2 to a mechanical proof very close to the previous proof sketch. See the web page for details.

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

4 THE `DEFMUL` MACRO

We defined a macro `defmul` in order to provide a convenient way to define the multiset relation induced by a well-founded relation, and to prove the corresponding well-foundedness theorem. We explain now how `defmul` is used.

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

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

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

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

The expansion of this macro generates a number of ACL2 forms, leading to the definition of a function `mul-`*my-rel* as the well-founded relation on multisets of elements satisfying the property *my-mp*, induced by the well-founded relation *my-rel*. More specifically:

- the multiset measure property *my-mp*`-true-listp`, the embedding function `map-`*my-fn*`-e0-ord` from multisets to ordinals and the multiset relation `mul-`*my-rel* are defined, in an analogue way to the corresponding functions defined in the previous section, where *my-mp*, *my-fn* and *my-rel* play the roles of `mp`, `fn` and `rel`, respectively.
- the well-foundedness theorem for `mul-`*my-rel*, *my-mp*`-true-listp` and `map-`*my-fn*`-e0-ord` is proved by functional instantiation from the theorem `multiset-extension-of-rel-well-founded` presented in the previous section.

We have divided the results and tools about multisets into two books. The book `multiset.lisp` contains the proof of the theorem `multiset-extension-of-rel-well-founded` shown in subsection 3.3. The book `defmul.lisp` contains the macro definition of `defmul` and includes the `multiset` book. We have also included some rules about multisets in `multiset.lisp`, which helped us to prove the three examples presented in this paper, and we think they are general enough to assist in other cases. See the web page for details.

We expect `defmul` to work without any assistance from the user. After the above call to `defmul`, the well-founded relation `mul-`*my-rel* could be used in the admissibility test for recursive functions to show that the recursion terminates. In the next section, we illustrate the use of well-founded multiset relations in the admissibility test of functions defined in ACL2.

5 CASE STUDIES USING MULTISET RELATIONS

We now describe three case studies where well-founded multiset relations play an important role in the ACL2 proof of non-trivial termination properties. In the first example, we use a multiset relation to show termination of the tail-recursive version of a general binary recursive scheme. In the second example, a multiset relation is used for the admission of an iterative version of McCarthy's 91 function. The third example is a proof of Newman's lemma for abstract reduction systems: every terminating and locally confluent reduction relation has the Church-Rosser property. This last example is part of a larger project developed by the authors in order to formalize some aspects of equational reasoning using ACL2 [Ruiz-Reina *et al.*, 2002].

All the examples show one function whose termination is proved using a well-founded multiset relation and a multiset measure function. When the function is presented for the first time, its code is commented out (using semicolons), to emphasize that a suitable measure has still to be given in order to pass the admissibility test.

5.1 A tail-recursive version of binary recursion

This example is inspired by [Slind, 2000], where the author formalizes several program transformation schemes using the HOL system. One of these examples is the transformation of a general binary tree recursion scheme into iterative form. The example originates from [Wand, 1980] (who presents a hand proof based on continuations) and has also been developed in the PVS system by [Shankar, 1995]. We present now a formalization of this example in ACL2.

To be as generic as possible, we first introduce a number of functions and assumed properties about them, by means of the `encapsulate` of Figure 1. These functions can be seen as "parameters" for the following generic function defined using a binary recursion scheme:

```
(defun binrec (x)
  (declare (xargs :measure (measure x)
                  :well-founded-relation rel-bin))
  (if (basic x)
      (b x)
    (join (binrec (l x)) (binrec (r x)))))
```

```
(encapsulate
 (((basic *) => *) ((b *) => *) ((join * *) => *)
  ((l *) => *) ((r *) => *) ((id-join) => *)((measure *) => *)
  ((rel-bin * *) => *) ((mp-bin *) => *) ((fn-bin *) => *))
 ...
 (defthm join-associative
   (equal (join (join x y) z) (join x (join y z))))

 (defthm join-identity (equal (join (id-join) x) x))

 (defthm mp-bin-measure (mp-bin (measure x)))

 (defthm l-and-r-decreases
   (implies (not (basic x))
            (and (rel-bin (measure (l x)) (measure x))
                 (rel-bin (measure (r x)) (measure x)))))

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

Figure 1. Parameters for the binary recursion scheme

Here the predicate `basic` represents the base case of the recursion, in which case the function `b` is applied. The functions `l` and `r` (for *left* and *right*) are a pair of destructor functions, used to split non-basic inputs into two parts on which to recurse. The results of the recursive calls are combined using a function `join`. We assume that `join` is associative and that `(id-join)` is a left-identity with respect to it.

We assume that the termination of the recursive scheme is justified by a general well-founded relation `rel-bin` and a measure function `measure` (not necessarily an ordinal), formalizing a termination argument as general as possible. That is, the measures of `(l x)` and `(r x)` decrease with respect to the well-founded relation.

Note that the measure and the well-founded relation are explicitly given in the `defun` of `binrec`, by the hints `:measure` and `:well-founded-` `-relation`. This is the way the user provides a particular well-founded relation and measure, when the heuristics of the prover fails to obtain a termination argument for a definition.

The following function `tailrec` implements a tail-recursive scheme that can be shown to be equivalent to `binrec`:

```
;(defun tailrec-it (l v)
;  (cond
;    ((atom l) v)
;    ((basic (car l))
;     (tailrec-it (cdr l) (join v (b (car l)))))
;    (t (tailrec-it
;          (list* (l (car l)) (r (car l)) (cdr l)) v))))

; (defun tailrec (x) (tailrec-it (list x) (id-join)))
```

The main auxiliary function used in the definition of `tailrec` is `tailrec-it`. Intuitively, the first argument of `tailrec-it` is a stack containing the remaining recursive calls and the second argument accumulates the combination of the values of the function `b` acting on the basic elements encountered during the recursive process.

Note that termination of `tailrec-it` is not trivial (since the length of the stack increases in each recursive call), but it can be proved using a multiset relation. The following `defmul` call automatically defines the well-founded multiset relation `mul-rel-bin` induced by `rel-bin` on multisets of elements satisfying `mp-bin`:

```
(defmul (rel-bin rel-bin-well-founded mp-bin fn-bin x y))
```

Now, the relation `mul-rel-bin` can be used as the well-founded relation in the admissibility test for the function `tailrec-it`, with a suitable measure function. This measure is given by the multiset of measures of the elements of the stack, computed by the following function:

```
(defun measure-list (l)
  (if (atom l)
      nil
    (cons (measure (car l)) (measure-list (cdr l)))))
```

We can now prove termination of `tailrec-it`, giving `mul-rel-bin` as well-founded relation and `measure-list` as measure function:

```
(defun tailrec-it (l v)
  (declare (xargs :measure (measure-list l)
                  :well-founded-relation mul-rel-bin))
  (cond
    ((atom l) v)
    ((basic (car l))
     (tailrec-it (cdr l) (join v (b (car l)))))
    (t (tailrec-it
         (list* (l (car l)) (r (car l)) (cdr l)) v))))
```

The proof obligations generated for the admission of this definition are not difficult, and only a very few previous lemmas are needed, in order to prove that the given multiset measure decreases in the recursive calls. See the web page for details. After this definition, the admission of the function `tailrec` (defined as above) is trivial.

Having `tailrec` admitted in the logic, it is also easy to prove the equivalence between `tailrec` and `binrec`. For that purpose, we first prove the main invariant in the computation performed by `tailrec-it`, established by the following events. Note that the function `join-binrec` computes the combination of the results of `binrec` over the elements of a list:

```
(defun join-binrec (l v)
  (if (atom l)
      v
    (join-binrec (cdr l) (join v (binrec (car l))))))

(defthm equal-tailrec-it-join-binrec
  (equal (tailrec-it l v) (join-binrec l v)))
```

It is remarkable that ACL2 proves this last theorem without assistance from the user. As a particular case, and taking into account that `(id-join)` is a left identity for `join`, we finally have the intended result:

```
(defthm equal-tailrec-binrec
  (equal (tailrec x) (binrec x)))
```

Note that this theorem is proved in a generic way, and it can be easily used by functional instantiation to show the equivalence of a particular version of the binary recursive scheme and its transformation into a tail-recursive version. Thus, we obtain a similar degree of generality as in [Slind, 2000] (for example, we can use arbitrary ACL2 well-founded relations). From a practical point of view, this allows to transform every executable function in ACL2 (and therefore executable in Common Lisp) that follows the general binary recursive schema, into an equivalent tail-recursive function, generating a formal proof of that equivalence.

5.2 McCarthy's 91 function

This example is taken from [Dershowitz and Manna, 1979] and shows admissibility of an iterative version of the recursive definition of McCarthy's 91 function. For a detailed treatment (in ACL2) of McCarthy's 91 function and its generalization given by Knuth, we urge the interested reader to consult the work of [Cowles, 2000], where proofs are done over arbitrary archimedian fields. Our intention here is only to show how multisets can help to prove a non-trivial termination property.

The "91 function" is a function acting on integers, originally given by McCarthy by the following recursive scheme:

```
(defun mc (x)
  (declare (xargs :mode :program))
  (cond ((not (integerp x)) x)
        ((> x 100) (- x 10))
        (t (mc (mc (+ x 11))))))
```

This function is defined in `:program` mode, which means that it can be executed but it is logically undefined. See [Cowles, 2000] for a description of ACL2's resistance to accept this definition in logic mode[8]. Instead, we try to define the following iterative version of that recursive scheme:

```
; (defun mc-aux (n z)
;   (cond ((or (zp n) (not (integerp z))) z)
;         ((> z 100) (mc-aux (- n 1) (- z 10)))
;         (t (mc-aux (+ n 1) (+ z 11)))))

; (defun mc-it (x) (mc-aux 1 x))
```

As we will show, the recursive algorithm implemented by `mc-it` is a somewhat complicated way to compute the following function:

```
(defun f91 (x)
  (cond ((not (integerp x)) x)
        ((> x 100) (- x 10))
        (t 91)))
```

The intended behavior of the function `mc-aux` is that in every iterative step `(mc-aux n z)`= `(f91 (f91` $\overset{n}{\ldots}$`(f91 z)))` and therefore `(mc-it x)` = `(f91 x)`. Proving termination of `mc-aux` may be difficult: note the different behavior of the two recursive calls. In [Dershowitz and Manna, 1979], a multiset measure is given to justify termination of the function: every recursive call of `(mc-aux n z)` is measured with the following multiset: {`z`, `(f91 z)`, `(f91 (f91 z))`,..., `(f91 (f91` $\overset{n-1}{\ldots}$`(f91 z)))`}, and multisets are compared with respect to the multiset relation induced by the "greater-than" relation defined for integers equal [9] or less than `111`. In the sequel, we describe how ACL2 is guided to this termination argument.

[8]To prove its termination, the *nested recursion* in the definition of `mc` leads the ACL2 prover to reason about the function before being introduced in the logic. See [Giesl, 1997] for a method to deal with termination proofs of algorithms with nested recursion, and in particular termination of McCarthy's 91 function.

[9]Performing the ACL2 proof, we discovered a minor bug in the proof given in [Dershowitz and Manna, 1979]: it is necessary to consider integers equal or less than `111`, and not only strictly less than `111`.

First, we define the well-founded relation `rel-mc` that will be extended later to a multiset relation. The following sequence of events defines `rel-mc` and stores it as a well founded relation:

```
(defun integerp-<=-111 (x)
  (and (integerp x) (<= x 111)))

(defun rel-mc (x y)
  (and (integerp-<=-111 x) (integerp-<=-111 y) (< y x)))

(defun fn-mc (x)
  (if (integerp-<=-111 x) (- 111 x) 0))

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

Note that in this case, the measure property is `t`, although only integers under `111` are comparable with respect to `rel-mc`[10].

We now define the well-founded multiset relation induced by `rel-mc` on multisets (`true-listp` objects in this case), using the following `defmul` call:

```
(defmul (rel-mc rel-mc-well-founded t fn-mc x y))
```

With this macro call, we have defined the well-founded relation `mul-rel-mc`, allowing us to use it in the admissibility test for the function `mc-aux`, with the measure described above, and implemented by the function `measure-mc-aux`:

```
(defun measure-mc-aux (n z)
  (if (zp n)
      nil
    (cons z (measure-mc-aux (- n 1) (f91 z)))))
```

We can now define the function `mc-aux`, giving `mul-rel-mc` and `measure-mc-aux` as the well-founded relation and measure function to be used, respectively:

```
(defun mc-aux (n z)
  (declare (xargs :measure (measure-mc-aux n z)
```

[10]One could think that `integerp-<=-111` should be the measure property of the well-founded relation, instead of `t`. But there is a subtle difference: the multiset measure we will define may contain elements greater than `111`, although those elements are not comparable w.r.t. `rel-mc`.

```
                 :well-founded-relation mul-rel-mc))
  (cond ((or (zp n) (not (integerp z))) z)
        ((> z 100) (mc-aux (- n 1) (- z 10)))
        (t (mc-aux (+ n 1) (+ z 11)))))
```

The function is admitted with a minor help from the user (surprisingly, only one specific lemma is needed). After this definition we can define the function `mc-it` as above, and show that verifies the original recursion scheme given by McCarthy. Moreover, we can even prove very easily that `mc-it` is equal to `f91` (previously proving a suitable generalization, as sketched above):

```
(defthm mc-it-recursive-schema
  (equal (mc-it x)
         (cond ((not (integerp x)) x)
               ((> x 100) (- x 10))
               (t (mc-it (mc-it (+ x 11)))))))

(defthm mc-it-equal-f91
  (equal (mc-it x) (f91 x)))
```

5.3 Newman's lemma

Abstract reduction systems

Newman's lemma is a result about abstract reduction systems, which plays an important role in the study of decidability of certain equational theories. We give a short introduction to basic concepts and definitions from abstract reductions. See [Baader and Nipkow, 1998] for more details.

Reduction systems are simply an abstract formalization of step by step activities, such as the execution of a computation, the gradual transformation of an object until some normal form is reached, or the traversal of some directed graph. The term "reduction" gives the intuition that an element of less complexity is obtained in every step. Formally speaking, an *abstract reduction* is simply a binary relation $\rightarrow$ defined on a set A, called its *domain.* 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 $\rightarrow$. We say that x and y are *equivalent* if $x \overset{*}{\leftrightarrow} y$. We say that x and y are *joinable* (denoted as $x \downarrow y$) if there exists u such that $x \overset{*}{\rightarrow} u \overset{*}{\leftarrow} y$. An element x is in *normal form* (or *irreducible*) if there is no z such that $x \rightarrow z$.

A reduction relation has the *Church-Rosser property* if every two equivalent elements are joinable. An equivalent property is *confluence*: for all x, u, v such that $u \overset{*}{\leftarrow} x \overset{*}{\rightarrow} v$, then $u \downarrow v$. In a reduction relation with the Church-Rosser property, two distinct elements in normal form cannot be equivalent. A reduction relation is *normalizing* if every element has an equivalent normal form (denoted as $x\downarrow$). Obviously, every terminating (as defined in subsection 3.1) reduction is normalizing. Church-Rosser and normalizing reduction relations have a nice property: provided normal forms are computable and identity in A is decidable, then the equivalence relation $\overset{*}{\leftrightarrow}$ is decidable. This is due to the fact that, in that case, $x \overset{*}{\leftrightarrow} y$ iff $x\downarrow = y\downarrow$, for all $x, y \in A$.

Confluence 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 $u \downarrow v$. The following theorem, named Newman's lemma, states this:

THEOREM 4. **(Newman's lemma)** *Let $\rightarrow$ be a terminating and locally confluent reduction relation. Then $\rightarrow$ is confluent.*

This result simplifies the study of confluence (or equivalently, of the Church-Rosser property) for terminating reduction relations. One only has to deal with joinability of local divergences. This is crucial in the development of completion algorithms for term rewriting systems in order to obtain decision procedures for equational theories [Baader and Nipkow, 1998].

Formalization of Newman's lemma in ACL2

Every reduction relation has two important aspects. On the one hand, it has a declarative aspect, since it describes its equivalence closure. On the other hand, it has a computational aspect, describing a stepwise activity, a gradual transformation of objects until (eventually) a normal form is reached. Thus, if $x \rightarrow y$, the point here is that y is obtained from x by applying some kind of transformation or *abstract operator*. In its most abstract formulation, we can view a reduction as a binary function that, given an element and an operator, returns another element, performing a *one-step reduction*. Of course not any operator can be applied to any element: we need a boolean binary function to test if it is *legal* to apply an operator to an element.

The discussion above leads us to formalize a general abstract reduction relation using three partially defined functions: `q`, `reduce-one-step` and `legal`; `(q x)` defines the domain of the reduction, `(reduce-one-step x op)` represents a one-step reduction applying the operator `op` to `x`, and `(legal x op)` represents a test to check if the operator `op` may be applied to `x`.

We introduce these three functions via `encapsulate`. In order to formalize Newman's lemma, properties are included to assume termination and local confluence of the reduction relation, encoding in this way the assumptions of the theorem we want to prove. This is shown in Figure 2. In the following, we describe in detail the events appearing in it.

Before describing how we formalized termination and local confluence,

```
(encapsulate
 (((rel * *) => *) ((fn *) => *) ((q *) => *)
  ((legal * *) => *) ((reduce-one-step * *) => *)
  ((transform-local-peak *) => *))
 ...
 (defthm local-confluence
   (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 rel-well-founded-relation-on-q
   (and (implies (q x) (e0-ordinalp (fn x)))
        (implies (and (q x) (q y) (rel x y))
                 (e0-ord-< (fn x) (fn y))))
   :rule-classes :well-founded-relation)

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

 (defthm terminating
   (implies (and (q x) (legal x op)
                 (q (reduce-one-step x op)))
            (rel (reduce-one-step x op) x))))
```

Figure 2. Assumptions of Newman's lemma

we show how we can define the equivalence closure of a reduction relation. In order to define $x \overset{*}{\leftrightarrow} y$, we include an extra argument with a sequence of steps $x = x_0 \leftrightarrow x_1 \leftrightarrow x_2 \ldots \leftrightarrow x_n = y$. An *abstract proof* (or simply, a *proof*) is a sequence of legal *proof steps* and each proof step is a structure[11] `r-step` with four fields: `elt1`, `elt2` (the elements connected), `direct` (a boolean value indicating if the step is direct or inverse) and an `operator`:

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

A proof step is *legal* if one of its elements is obtained by applying the (legal) operator to the other, in the direction indicated. The function `proof-step-p` (we omit its definition) implements this concept. The function `equiv-p` implements the equivalence closure of our abstract reduction

[11] We used the `defstructure` tool developed by Bishop Brock [Brock, 1997], which provides records in ACL2 in a similar way to Common Lisp's `defstruct`.

relation: `(equiv-p x y p)` checks if `p` is a proof justifying that $\mathtt{x}\overset{*}{\leftrightarrow}\mathtt{y}$ (with all the involved elements in the domain `q`):

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

Two proofs justifying the same equivalence will be said to be *equivalent*. We hope it will be clear from the context when we talk about abstract proofs objects and proofs in the ACL2 system.

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

To deal with the assumption of local confluence, note that a reduction is locally confluent iff for every local peak proof there is an equivalent valley proof. Therefore, in order to state local confluence of the general reduction relation defined, we assume the existence of a function `transform-local-peak` which returns a valley proof for every local peak proof (assumption `local-confluence` in Figure 2).[12]

Let us now see how can we formalize termination. Our formalization is based on the following meta-theorem: a reduction is terminating if and only if it is contained in a well-founded partial ordering (axiom of choice needed). Thus, let `rel`[13] be a given general well-founded partial order on the set defined by `q` (assumptions `rel-well-founded-relation-on-q` and `rel-transitive` in Figure 2). This well-founded partial order `rel` is used to state termination of the general reduction relation defined, by assuming that every legal reduction step relating elements of the reduction domain always obtains a smaller element with respect to `rel` (assumption `terminating` in Figure 2).

Having formalized the assumptions, in order to prove Newman's lemma we must show confluence of this general reduction relation assumed to be terminating and locally confluent. Instead of confluence, we prove the Church-Rosser property, which is equivalent. Therefore, we must prove that for every proof there exists an equivalent valley proof; that is, *we have to define* a function `transform-to-valley` and prove that `(transform-to-`

[12]Note that the functions `proof-step-p` and `equiv-p` have to be defined as non-local events inside the `encapsulate` (although, for the sake of clarity, we omit their definitions in the figure).

[13]Conflicts with names used in the `multiset.lisp` book are avoided using packages.

-valley p) is a valley proof equivalent to p. This is the statement of Newman's lemma:

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

A suitable definition of transform-to-valley and a proof of this theorem in ACL2 is shown in the following subsection. The hard part of the proof is to show termination of transform-to-valley. It will be done with the help of a well-founded multiset relation.

An ACL2 proof of Newman's lemma

The proof commonly found in the literature [Baader and Nipkow, 1998], is done by well-founded induction on the terminating reduction relation. Due to our formalization of the theorem, our approach is more constructive and is based on a proof given in [Klop, 1992]. We have to define a function transform-to-valley which transforms every proof into an equivalent valley proof. For that purpose, we can use the function transform-local-peak, assumed to transform every local peak proof into a equivalent valley proof. Thus, the function we need is defined to iteratively apply replace-local-peak, (which replaces the first local peak subproof by the equivalent subproof given by transform-local-peak) until there are no local peaks (checked by exists-local-peak). The following is our intended definition of transform-to-valley (we omit here the definition of the functions replace-local-peak and exists-local-peak):

```
; (defun transform-to-valley (p)
;  (if (exists-local-peak p)
;      (transform-to-valley (replace-local-peak p))
;     p))
```

This function is not admitted without help from the user. The reason is that when a local peak in a proof is replaced by an equivalent valley subproof, the length of the proof obtained may be larger than the length of the original proof. Nevertheless, the key point here is that every element involved in the new subproof is smaller (w.r.t. the well-founded relation rel) than the greatest element of the local peak. If we measure a proof as the multiset of the elements involved in it, then replacing a local peak subproof by an equivalent valley subproof, we obtain a proof with smaller measure with respect to the well-founded multiset relation induced by rel. The function proof-measure returns this measure for a given proof: it collects the elt1 elements of every proof step in a proof.

```
(defun proof-measure (p)
  (if (atom p)
      nil
    (cons (elt1 (car p)) (proof-measure (cdr p)))))
```

Using `defmul`, we define the well-founded relation `mul-rel`, induced by the well-founded relation `rel` introduced in the previous subsection:

```
(defmul (rel rel-well-founded-relation-on-q q fn x y))
```

The main result we proved states that the proof measure decreases (with respect to the well-founded relation `mul-rel`) if a local-peak is replaced by an equivalent valley subproof:

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

With this theorem, admission of the function `transform-to-valley` is now possible, giving a suitable indication:

```
(defun transform-to-valley (p)
  (declare
   (xargs :measure (if (steps-q p) (proof-measure p) nil)
          :well-founded-relation mul-rel))
  (if (and (steps-q p) (exists-local-peak p))
      (transform-to-valley (replace-local-peak p))
    p))
```

Note that our original intended definition had to be slightly modified: since `rel` is well-founded on `q`, `mul-rel` is well-founded on multisets of elements satisfying `q`. The function `steps-q` (omitted here) checks whether all the elements appearing in a proof satisfy `q`, thus ensuring that the measure `proof-measure` returns a multisets of elements satisfying `q`. Anyway these modifications do not affect the statement of the final theorem proved.

Once `transform-to-valley` is admitted (which is the hard part of the theorem), the following two theorems are proved, and this trivially implies Newman's lemma as stated at the end of subsection 5.3.

```
(defthm equiv-p-x-y-transform-to-valley
  (implies (equiv-p x y p)
           (equiv-p x y (transform-to-valley p))))

(defthm valley-transform-to-valley
   (implies (equiv-p x y p)
            (steps-valley (transform-to-valley p))))
```

It is remarkable that the induction scheme generated by the system in the proofs of these two theorems is based on the relation `mul-rel`. That is, they are by induction *on the measure of the proofs*, rather than an induction based on the terminating relation `rel` as in the standard proof.

The proof of Newman's lemma is a classical result formalized in most of the main proof checking systems like Coq, Mizar or Isabelle/HOL. A comparison with those developments is difficult because our formulation is different and, more important, the logics involved are significantly different: ACL2 logic is a much weaker logic than those of Coq or HOL. This proof is the most difficult of the three examples presented here. Lemmas have to be proved to simplify the multiset differences appearing in the conjecture generated by the termination proof of `transform-to-valley`. We also provide books proving decidability of the equivalence relation generated by a terminating and locally confluent reduction relation (see the web page for details). To see how this result can be exported to the study of equational theories, see [Ruiz-Reina *et al.*, 2002].

6 CONCLUSIONS

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

We have presented also a macro named `defmul`, implemented to provide a convenient tool to define well-founded multiset relations induced by well-founded relations. This macro allows the definition of these multiset relations in a single step.

Three case studies are presented, to show how this tool can be useful in obtaining proofs of non-trivial termination properties of functions defined in ACL2. The first case study is the definition of a tail-recursive version of a general binary recursion scheme. The second is the admissibility of a definition of McCarthy's 91 function, and a study of its properties. The third is a proof of Newman's lemma for abstract reduction relations.

From the variety of the examples presented, we think that well-founded multiset relations can be used in other situations as well. See additional examples on the web page. We also think that the `defmul` macro is a good example of the use of macros in ACL2 as a mean to "customize" the behavior of the system.

As a general conclusion, the case studies presented here show how non-trivial mathematical results can be stated and proved in the ACL2 logic, in spite of its apparent lack of expressiveness (first-order and quantifier-free). As we said before, some of the examples (Newman's lemma, for instance) have also been formalized in theorem provers systems with more expressive logics. Although sometimes the use of a more restrictive logic means that formalization is more difficult, usually this also means that automation in the proof is increased. But our main reason for choosing ACL2 is that it is a prover for a widely used programming language: deduction and efficient computation can be done in the same system. Although the examples presented here are all of a theoretical nature, they can serve as a basis for the verification of executable Common Lisp functions of practical interest. For example, the formalization of Newman's lemma allows us the verification of decision procedures for equational theories, as described in [Ruiz-Reina *et al.*, 2002].

Finally, we point out some possible topics for future work. First, it is our intention to provide a good ACL2 library of lemmas to handle multisets and their operations. Also, a remark given at the end of section III in [Dershowitz and Manna, 1979], pointing out a heuristic procedure for proving termination of loops using multisets, suggests that this kind of orderings could be applied to a wider class of termination problems and that the search for a suitable multiset measure could be mechanized to some extent. Another application of multiset orderings could be to provide the basis for some formal proofs of termination of term rewriting systems. In particular, it would be interesting to formalize in ACL2 some well-known termination orderings like the recursive path ordering or the Knuth–Bendix ordering [Baader and Nipkow, 1998]. We intend to make further research following these two lines.

BIBLIOGRAPHY

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

[Boyer and Moore, 1998] R. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, 2nd edition, 1998.

[Brock, 1997] B. Brock. `defstructure` for ACL2 version 2.0, 1997. See [Kaufmann and Moore, 2002].

[Cowles, 2000] J. Cowles. Knuth's generalization of McCarthy's 91 function. chapter 17. *Computer Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, 2000.

[Dershowitz and Manna, 1979] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[Giesl, 1997] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–29, 1997.

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

[Kaufmann and Moore, 2002] M. Kaufmann and J S. Moore. ACL2 version 2.7, 2002. `http://www.cs.utexas.edu/users/moore/acl2/`.

[Kaufmann *et al.*, 2000] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[Klop, 1992] J. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[Levy, 1979] A. Levy. *Basic Set Theory*. Springer–Verlag, 1979.

[Persson, 1999] H. Persson. *Type Theory and the Integrated Logic of Programs*. PhD thesis, Chalmers University of Technology, 1999.

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

[Ruiz-Reina *et al.*, 2002] J. Ruiz-Reina, J. Alonso, M. Hidalgo, and F. Martín. Formal proofs about rewriting using ACL2. *Annals of Mathematics and Artificial Intelligence*, 36(3):239–262, 2002.

[Shankar, 1995] N. Shankar. Step towards mechanizing program transformations using PVS. In *MCP'95 (Mathematics of Program Construction, Third International Conference)*, number 947 in Lecture Notes in Computer Science, pages 50–66. Springer–Verlag, 1995.

[Slind, 2000] K. Slind. Wellfounded schematic definitions. In *CADE-17, 17th Conference on Automated Deduction*, number 1831 in Lecture Notes in Computer Science, pages 45–63. Springer–Verlag, 2000.

[Steele, 1990] G.L. Steele. *Common Lisp the Language*. Digital Press, 2nd edition, 1990.

[Wand, 1980] M. Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 1(27):164–80, 1980.