Proof Pearl: A Formal Proof of Higman's Lemma in ACL2*

Francisco J. Martín-Mateos, José L. Ruiz-Reina,
José A. Alonso, and Mariá J. Hidalgo

Computational Logic Group,
Dept. of Computer Science and Artificial Intelligence,
University of Seville, E.T.S.I. Informática,
Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain
http://www.cs.us.es/{~fmartin, ~jalonso, ~mjoseh, ~jruiz}

Abstract. In this paper we present a formalization and proof of Higman's Lemma in ACL2. We formalize the constructive proof described in [10] where the result is proved using a termination argument justified by the multiset extension of a well-founded relation. To our knowledge, this is the first mechanization of this proof.

1 Introduction

In [8] we presented a formal proof of Dickson's Lemma in ACL2 [6]. This result was needed to prove the termination of a Common Lisp implementation of Buchberger's algorithm for computing Gröbner basis of polynomial ideals [9]. After finishing this work our attention was addressed to similar results already present in the literature [12,15,10].The last one presents a constructive proof of Higman's Lemma similar to the one presented in [8]. Thus, the interest to automatize this proof of Higman's Lemma in ACL2 is multiple: first, the proof has a similar structure to the proof of Dickson's Lemma developed by the authors, and similar techniques are used; second, it is the first (to our knowledge) automatization of this proof, complementing thus the work presented in [10]; third, Dickson's Lemma could be proved in ACL2 as a consequence of this theorem; and finally it could give some advice about how to prove Kruskal's Theorem in ACL2, which is a fundamental theorem in the proof of termination of some well-known term orderings [1].

The ACL2 logic is a subset of first-order logic with a principle of proof by induction. The proof we present here is based on the constructive proof presented in [10], where the result is proved using a termination argument justified by the multiset extension of a well-founded relation. In the mechanization of this proof, we use a tool for defining multiset well-founded relations in ACL2 in an automated way, a tool that we used previously in other formalizations [13] and that can now be reused.

* This work has been supported by project TIN2004-03884 (Ministerio de Educación y Ciencia, Spain) and FEDER founds.

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 358–372, 2005.

Higman's Lemma is a property about embeddings of strings. Previously to present the result we introduce some notation. Let Σ be a set, and let Σ^* denote the set of finite strings over Σ.

Definition 1. *Let $\preceq$ be a binary relation on Σ. The* induced embedding relation *$\preceq^*$ on Σ^* is defined as follows: $s_1 s_2 \cdots s_m \preceq^* t_1 t_2 \cdots t_n$ if there exists indices $j_1 < j_2 < \ldots < j_m \leq n$ such that $s_i \preceq t_{j_i}, \forall i$.*

If $s \preceq t$ ($u \preceq^* w$) we usually say that s (u) is less than t (w) or t (w) is bigger than s (u). The relation with respect to which an element is less or bigger than other is usually obvious in the context.

Definition 2. *We say that a relation on Σ is a* quasi-order *if it is reflexive and transitive. Given a quasi-order $\preceq$ defined on Σ, we say that $\preceq$ is a* well-quasi-order *if for every infinite sequence*[1] *$\{s_k : k \in \mathbb{N}\}$ of elements of Σ there exist indices $i < j$ such that $s_i \preceq s_j$.*

Higman's Lemma establishes a sufficient condition for well-quasi-orders on strings.

Theorem 1. (Higman's Lemma). *If $\preceq$ is a well-quasi-order on Σ then $\preceq^*$ is also a well-quasi-order on Σ^*.*

Given the well-quasi-order $\preceq$ on Σ, it is not difficult to prove that $\preceq^*$ is a quasi-order on Σ^*. Thus, we will center our attention on the well-quasi-order property: for every infinite sequence of strings $\{w_k : k \in \mathbb{N}\}$ there exists indices $i < j$ such that $w_i \preceq^* w_j$.

As we said above, the proof presented here is based on [10], and it essentially builds a well-founded measure that can be associated to the initial segments of a sequence of strings and that decreases whenever a string in the sequence is not bigger than any of the previous strings.

2 Formalizing the Proof in ACL2

The ACL2 logic is a first-order logic with equality, describing an applicative subset of Common Lisp. The syntax of terms is that of Common Lisp and the logic includes axioms for propositional logic and for a number of Lisp functions and data types. Rules of inference of the logic include those for propositional calculus, equality and instantiation. One important rule of inference is the *principle of induction*, that permits proofs by well-founded induction on the ordinal ε_0. The theory has a constructive definition of the ordinals up to ε_0, in terms of lists and natural numbers, given by the predicate `o-p` ($o \in \varepsilon_0 \equiv$ `o-p`(o)) and the order ($o_1 <_{\varepsilon_0} o_2 \equiv$ `o<`(o_1,o_2).) Although this is the only built-in well-founded relation, the user may define new well-founded relations from that.

[1] An infinite sequence of elements of Σ is a function $s : \mathbb{N} \to \Sigma$. As usual, we write s_k instead of $s(k)$ and by abuse of notation, we often identify the sequence with its range $\{s_k : k \in \mathbb{N}\}$.

By the *principle of definition*, new function definitions are admitted as axioms only if there exists a measure in which the arguments of each recursive call decrease with respect to a well-founded relation, ensuring in this way that no inconsistencies are introduced by new definitions. Usually, the system can prove automatically this property using a predefined ordinal measure on Lisp objects and the relation `o<`. Nevertheless, if the termination proof is not trivial, the user has to explicitly provide a measure on the arguments and a well-founded relation ensuring termination.

The `encapsulate` mechanism [7] allows the user to introduce new function symbols by axioms constraining them to have certain properties (to ensure consistency, a witness local function having the same properties has to be exhibited). Inside an `encapsulate`, the properties stated need to be proved for the local witnesses, and outside, they work as assumed axioms. This mechanism behaves like a universal quantifier over a set of functions abstractly defined with it.

A derived rule of inference, called *functional instantiation*, provides a higher-order-like mechanism by allowing to instantiate the function symbols of a previously proved theorem, replacing them with other function symbols or lambda expressions, provided it can be proved that the new functions satisfy the constraints of the old ones.

The ACL2 theorem prover mechanizes the ACL2 logic, being particularly well suited for obtaining automatized proofs based on simplification and induction. For a detailed description of ACL2, we refer the reader to the ACL2 book [5].

For the sake of readability, the ACL2 expressions in this paper are presented using a notation closer to the usual mathematical notation than its original Common Lisp syntax; when it is necessary we show the correspondence between the ACL2 and the mathematical notation. Some of the functions are also used in infix notation.

2.1 Formulation of Higman's Lemma

First we formalize in the ACL2 logic the context in which Higman's Lemma will be established. We consider a unary predicate `sigma-p` to check the membership to Σ ($s \in \Sigma \equiv$ `sigma-p`(s)) and a binary predicate `sigma-<=` representing the well-quasi-order $\preceq$ on the set Σ ($s \preceq t \equiv$ `sigma-<=`(s,t)). These functions are abstractly defined by means of the `encapsulate` mechanism. In this case the assumed properties about `sigma-p` and `sigma-<=` are the following[2]:

ASSUMPTION: `sigma-<=-reflexive`
$s \in \Sigma \rightarrow s \preceq s$

ASSUMPTION: `sigma-<=-transitive`
$s_1, s_2, s_3 \in \Sigma \wedge s_1 \preceq s_2 \wedge s_2 \preceq s_3 \rightarrow s_1 \preceq s_3$

In the following we use $\overline{\Sigma}$ to denote the set of finite sequences of elements of Σ and we use the overline notation to identify the elements of $\overline{\Sigma}$. We characterize

[2] The local witnesses are irrelevant to our description of the proof.

the well-quasi-order property of $\preceq$ in the following way: there exists an ordinal measure on $\overline{\Sigma}$ such that the measure of any element of $\overline{\Sigma}$ is bigger than the measure of any extension of this sequence with an element s such that there is no elements in the sequence less than s. As it is pointed out in [10]: "*Classically, this is easily gotten from the well-quasi-order-ness of* $\preceq$*, but constructively we must have this as an assumption (...) After a moment's reflection, it should be obvious to the reader that* this *is the constructive equivalent to the classical notion of well-quasi-order.*"

The embedding is introduced in the previous `encapsulate` by means of the function `sigma-seq-measure`. To state the properties assumed about this function we also need two concepts whose definitions are based on `sigma-p` and `sigma-<=` functions: the membership to $\overline{\Sigma}$ ($\overline{s} \in \overline{\Sigma} \equiv$ `sigma-seq-p`$(\overline{s})$) and the presence of an element in a finite sequence $\overline{s}$ less than an element t (`exists-sigma-<=`$(\overline{s},t)$). The assumed properties are the following:

ASSUMPTION: `sigma-seq-measure-o-p`
$\overline{s} \in \overline{\Sigma} \rightarrow$ `sigma-seq-measure`$(\overline{s}) \in Ord$

ASSUMPTION: `sigma-seq-measure-neq-0`
$\overline{s} \in \overline{\Sigma} \rightarrow$ `sigma-seq-measure`$(\overline{s}) \neq 0$

ASSUMPTION: `sigma-<=-well-quasi-order-characterization`
$\overline{s} \in \overline{\Sigma} \wedge\ t \in \Sigma \wedge \neg$`exists-sigma-<=`$(\overline{s},t)$
$\rightarrow$ `sigma-seq-measure`$(\mathtt{cons}(t,\overline{s})) <_{\varepsilon_0}$ `sigma-seq-measure`$(\overline{s})$

The first property ensures that the function `sigma-seq-measure` returns an ACL2 ordinal when its argument is a string, and the second property ensures that this ordinal is not 0[3]. The last property is the constructive characterization of the well-quasi-order-ness of $\preceq$. It must be noticed that the elements of $\overline{\Sigma}$ are represented in ACL2 by means of lists of elements of Σ in reverse order; that is, the ACL2 representation of the finite sequence $\{s_1, \ldots, s_n\}$ is the list $(s_n \ \ldots \ s_1)$. This is because an element t is more easily added in front of a sequence $\overline{s}$ using `cons`.

The elements of Σ^* are also represented in ACL2 by means of lists, but in this case the order of the elements is preserved. So, the representation of the string $s_1 s_2 \ldots s_m$ is the list $(s_1\ s_2\ \ldots\ s_m)$. The membership to Σ^* is checked by the function `sigma-*-p` ($w \in \Sigma^* \equiv$ `sigma-*-p`(w)). We use a different function name, but its definition is equal to the definition of `sigma-seq-p`. The relation $\preceq^*$ in Σ^* is formalized with the following function ($w_1 \preceq^* w_2 \equiv$ `sigma-*-<=`(w_1,w_2)):

DEFINITION:
`sigma-*-<=`$(w_1,w_2) \Leftrightarrow$
if `endp`(w_1) **then t**

[3] This is a technical detail that could have been avoided adding 1 to the finite ordinals returned by `sigma-seq-measure`.

elseif endp(w_2) **then nil**
elseif car(w_1) $\in \Sigma \wedge$ car(w_1) $\in \Sigma$
then if car(w_1) $\preceq$ car(w_2)
then sigma-*-<=(cdr(w_1),cdr(w_2))
else sigma-*-<=(w_1,cdr(w_2))
else nil

It should be noted that this definition is algorithmic and different from the non-deterministic declarative Definition 1; but it is not difficult to prove that both definitions are equivalent. This function checks the property $w_1 \preceq^* w_2$ looking for the first elements, from the left side of w_2, bigger than the elements of w_1.

To formalize Higman's Lemma in the ACL2 logic we consider a unary function f, representing an infinite sequence of strings. We use again the encapsulate mechanism to abstractly define this function. In this case, the assumed property about f is the following:

ASSUMPTION: f-returns-strings
$i \in \mathbb{N} \rightarrow$ f(i) $\in \Sigma^*$

Here, the encapsulate mechanism behaves like a universal quantifier over the function abstractly defined with it. So, any theorem proved about this function is true for any function with the above property, by means of functional instantiation (see [5] for details). This is the case for the ACL2 formalization of Higman's Lemma: as the infinite sequence of strings is abstractly defined via encapsulate, the properties that we will prove about it are valid for any infinite sequence of strings.

Let us now define the functions needed to state Higman's Lemma. The function get-sigma-*-<=-f has two arguments, a natural number j and a string w, and it returns the biggest index i such that $i < j$ and f(i) $\preceq^* w$ whenever such index exists (nil otherwise):

DEFINITION:
get-sigma-*-<=-f(j,w) =
if $j \in \mathbb{N}$ **then if** $j = 0$ **then nil**
elseif f($j-1$) $\preceq^* w$ **then** $j-1$
else get-sigma-*-<=-f($j-1$,w)
else nil

Finally, the following function higman-indices receives as input an index k and uses get-sigma-*-<=-f to recursively search a pair of indices $i < j$ such that $j \geq k$ and f(i) $\preceq^*$ f(j):

DEFINITION:
higman-indices(k) =
if $k \in \mathbb{N}$ **then let** i **be** get-sigma-*-<=-f(k,f(k))
in if $i \neq$ **nil then** $\langle i, k \rangle$
else higman-indices($k+1$)
else nil

Let us assume for the moment that we have proved that the function `higman-indices` terminates and that this definition has been admitted by the system. Then the following property is easily proved as a direct consequence of the definitions of the functions involved:

THEOREM: `higman-lemma`
$[k \in \mathbb{N} \wedge \texttt{higman-indices}(k) = \langle i, j \rangle] \rightarrow [i < j \wedge \texttt{f}(i) \preceq^* \texttt{f}(j)]$

This theorem ensures that for any infinite sequence of strings $\{\texttt{f}(k) : k \in \mathbb{N}\}$, there exists $i < j$ such that $\texttt{f}(i) \preceq^* \texttt{f}(j)$ (and the function `higman-indices` explicitly provides those indices). Thus, it is a formal statement of Higman's Lemma in ACL2.

The hard part is the termination proof of the function `higman-indices`. For that purpose, we have to explicitly provide to the system a measure on the input argument and prove that the measure decreases with respect to a given well-founded relation in every recursive call. We present the details in the next subsections.

2.2 A Well-Founded Measure

Before giving a formal definition of the termination measure, we give some intuition by means of an example. Let us consider the set of natural numbers, $\mathbb{N}$, and the reflexive transitive closure of the following relation: $n \preceq n+2$ for all n. In this relation the even numbers are ordered as usual, as well as the odd numbers, but there is no relation between even and odd numbers. It is easy to prove that this relation is a well-quasi-order.

An embedding from $\overline{\mathbb{N}}$ in ordinals characterizing the well-quasi-order-ness of $\preceq$ could be the following: given a finite sequence of natural numbers $\overline{s}$, the ordinal associated is $\omega \cdot 2$ if $\overline{s}$ is empty; $\omega + n$ if n is the last even (odd) number in $\overline{s}$ and there is no odd (even) numbers in $\overline{s}$; and $n + m$ if n and m are respectively the last even number and the last odd number in $\overline{s}$.

Let $\{f_k : k \in \mathbb{N}\}$ be an infinite sequence of strings over $\mathbb{N}$ with $f_0 = 3 \cdot 2$. Note that any string bigger than f_0 is of the form $x_1 ... x_i x y_1 ... y_j y z_1 ... z_k$ with $3 \preceq x$ and $2 \preceq y$. Thus, the strings w such that $f_0 \not\preceq^* w$ could be described as follows:

- Any string $x_1 ... x_n$ with $3 \not\preceq x_i, \forall i$. We represent this set of strings by $\Pi_1 = \langle -, 3 \rangle$.
- Any string $x_1 ... x_n x y_1 ... y_m$ with $3 \not\preceq x_i, \forall i$, $3 \preceq x$ and $2 \not\preceq y_j, \forall j$. There are three components in these strings: $x_1 ... x_n$ with $3 \not\preceq x_i$, that we represent by $\pi_1 = \langle -, 3 \rangle$; x with $3 \preceq x$, that we represent by $\pi_2 = \langle 3 \rangle$; and $y_1 ... y_m$ with $2 \not\preceq y_j$, that we represent by $\pi_3 = \langle -, 2 \rangle$. So the representation of this set of strings is $\Pi_2 = \pi_1 \pi_2 \pi_3 = \langle -, 3 \rangle \langle 3 \rangle \langle -, 2 \rangle$.

We will refer to these representations of set of strings as *patterns*. As it can be seen in the previous example, a pattern could have components that we will call *simple patterns*. Π_1 is a pattern with only one simple pattern $\langle -, 3 \rangle$; and Π_2 is a pattern with three simple patterns π_1, π_2 and π_3. We will use the capital

Greek letter Π (possibly with subscripts) to represent patterns, and the small Greek letter π (possibly with subscripts) to represent simple patterns. The set of strings represented by a pattern Π or a simple pattern π will be denoted $\mathcal{S}(\Pi)$ and $\mathcal{S}(\pi)$ respectively.

Let us suppose that $f_1 = 1 \cdot 2$. Note that $f_1 \in \mathcal{S}(\Pi_1)$ since every element of f_1 is not bigger than 3. Let us see now how we can obtain from Π_1 a set of patterns describing the set of strings w such that $f_0 \not\preceq^* w$ and $f_1 \not\preceq^* w$ (we will call this operation the *reduction* of Π_1 with respect to f_1). The pattern Π_1 should be replaced in such a way that any string bigger than f_1 is removed from the set represented by the pattern. Since the strings bigger than f_1 are of the form $x_1...x_i x y_1...y_j y z_1...z_k$ with $1 \preceq x$ and $2 \preceq y$, then the set of strings $w \in \mathcal{S}(\Pi_1)$ such that $f_1 \not\preceq^* w$ could be described as follows:

- Any string $x_1...x_n$ with $3 \not\preceq x_i, \forall i$ (the string is in $\mathcal{S}(\Pi_1)$) and $1 \not\preceq x_i, \forall i$. We represent this set of strings by $\langle -, 3, 1\rangle$.
- Any string $x_1...x_n x y_1...y_m$ with $3 \not\preceq x_i, x, y_j, \forall i, j$ (the string is in $\mathcal{S}(\Pi_1)$), $1 \not\preceq x_i, \forall i$, $1 \preceq x$ and $2 \not\preceq y_j, \forall j$. There are three components in these strings: $x_1...x_n$ with $3, 1 \not\preceq x_i, \forall i$, represented by $\langle -, 3, 1\rangle$; x with $3 \not\preceq x$ and $1 \preceq x$, represented by $\langle 1, 3\rangle$; and $y_1...y_m$ with $3, 2 \not\preceq y_j, \forall j$, represented by $\langle -, 3, 2\rangle$. So the representation of this set of strings is $\langle -, 3, 1\rangle\langle 1, 3\rangle\langle -, 3, 2\rangle$.

Now, let us see how to deal with a more complicated pattern: let suppose that $f_2 = 1 \cdot 5 \cdot 3$; this string is in $\mathcal{S}(\Pi_2)$ since $1 \in \mathcal{S}(\pi_1)$, $5 \in \mathcal{S}(\pi_2)$ and $3 \in \mathcal{S}(\pi_3)$. The pattern Π_2 should be reduced in such a way that any string bigger than f_2 is removed. In this case if $w = w_1 w_2 w_3 \in \mathcal{S}(\Pi_2)$ with $w_i \in \mathcal{S}(\pi_i)$ and $1 \preceq^* w_1$, $5 \preceq^* w_2$ and $3 \preceq^* w_3$, then $f_2 \preceq^* w$. Thus, if we ensure that some component w_i is not bigger than the correspondent component in f_2, then $f_2 \not\preceq^* w$. Taking this into account, the strings $w \in \mathcal{S}(\Pi_2)$ such that $f_2 \not\preceq^* w$ are described as follows:

- Any string $w_1 w_2 w_3$ with $w_i \in \mathcal{S}(\pi_i), \forall i$ and $1 \not\preceq^* w_1$. The set of strings in $\mathcal{S}(\pi_1)$ whose elements are not bigger than 1 is represented by the pattern $\langle -, 3, 1\rangle$. So the representation of this set of strings is $\langle -, 3, 1\rangle\langle 3\rangle\langle -, 2\rangle$.
- Any string $w_1 w_2 w_3$ with $w_i \in \mathcal{S}(\pi_i), \forall i$ and $5 \not\preceq^* w_2$. The set of strings in $\mathcal{S}(\pi_2)$ whose elements are not bigger than 5 is represented by the pattern $\langle 3, 5\rangle$. So the representation of this set of strings is $\langle -, 3\rangle\langle 3, 5\rangle\langle -, 2\rangle$.
- Any string $w_1 w_2 w_3$ with $w_i \in \mathcal{S}(\pi_i), \forall i$ and $3 \not\preceq^* w_3$. The set of strings in $\mathcal{S}(\pi_3)$ whose elements are not bigger than 3 is represented by the pattern $\langle -, 2, 3\rangle$. So the representation of this set of strings is $\langle -, 3\rangle\langle 3\rangle\langle -, 2, 3\rangle$.

We now explain how we formalize these constructions in ACL2. There are two types of simple patterns: the first one is $\langle -, s_1, \ldots, s_n\rangle$(`(nil` s_n `...` s_1`)` in ACL2), representing any string $t_1 \ldots t_m$ such that $s_i \not\preceq t_j, \forall i, j$; the second one is $\langle s, s_1, \ldots, s_n\rangle$ (`((`s`)` s_n `...` s_1`)` in ACL2), representing any string t such that $s \preceq t$ and $s_i \not\preceq t, \forall i$ (π is a simple pattern $\equiv$ `simple-pattern-p`(π)). In both cases we say that $s_1, \ldots, s_n$ is the sequence with respect to which the simple pattern is defined. Note that this sequence is represented in ACL2 in reverse order.

The following function checks the membership of a 1-length string s to the set of strings described by a simple pattern π:

DEFINITION:
`member-simple-pattern`$(s,\pi) =$
  if `consp`(`car`(π)) **then** `caar`(π) $\preceq s \wedge \neg$`exists-sigma-<=`(`cdr`(π),s)
  else $\neg$`exists-sigma-<=`(`cdr`(π),s)

Simple patterns represent the components in which a new string could be split ensuring that it is not bigger than any previous string in the sequence. Thus, a pattern is a string of simple patterns (Π is a pattern $\equiv$ `pattern-p`(Π)). Given $\Pi = \pi_1 \ldots \pi_n$, a string w is in the set of strings described by Π ($w \in \mathcal{S}(\Pi)$) if there exist strings $w_1, \ldots, w_n$, such that $w = w_1 \ldots w_n$ and $w_i \in \mathcal{S}(\pi_i), \forall i$. The function `member-pattern`$(w,\Pi)$ returns a pair `(res val)` where `res` indicates if $w \in \mathcal{S}(\Pi)$ and, if it is the case, `val` is the list of components `(`w_1 `...` w_n`)` justifying this.

DEFINITION:
`member-pattern`$(w,\Pi) =$
  if `endp`(w) **then if** `endp`(Π) **then** $\langle$**t**, **nil**$\rangle$
                  elseif `consp`(`caar`(Π)) **then** $\langle$**nil**, **nil**$\rangle$
                  else let $\langle$`res`, `val`$\rangle$ **be** `member-pattern`(w,`cdr`(Π))
                    in if `res` **then** $\langle$**t**, `cons`(**nil**, `val`)$\rangle$
                      else $\langle$**nil**, **nil**$\rangle$
  elseif `endp`(Π) **then** $\langle$**nil**, **nil**$\rangle$
  elseif `consp`(`caar`(Π))
      let `res1` **be** `member-simple-pattern`(`car`(w),`car`(Π))
        $\langle$`res2`, `val2`$\rangle$ **be** `member-pattern`(`cdr`(w),`cdr`(Π))
      in if `res1` $\wedge$ `res2` **then** $\langle$**t**, `cons`(`list`(`car`(w)), `val2`)$\rangle$
        else $\langle$**nil**, **nil**$\rangle$
  else let $\langle$`res1`, `val1`$\rangle$ **be** `member-pattern`(w,`cdr`(Π))
      in if `res1` **then** $\langle$**t**, `cons`(**nil**, `val1`)$\rangle$
        else let `res2` **be** `member-simple-pattern`(`car`(w),`car`(Π))
          $\langle$`res3`, `val3`$\rangle$ **be** `member-pattern`(`cdr`(w),Π)
        in if `res2` $\wedge$ `res3`
            then $\langle$**t**, `cons`(`cons`(`car`(w), `car`(`val3`)), `cdr`(`val3`))$\rangle$
          else $\langle$**nil**, **nil**$\rangle$

As we said above, given a string w and a pattern Π such that $w \in \mathcal{S}(\Pi)$, we define the reduction of Π with respect to w as a set of patterns representing the strings in $\mathcal{S}(\Pi)$ not bigger than w. Let us describe the reduction process beginning with the reductions of a simple pattern π:

- If $\pi = \langle s, s_1, \ldots, s_n \rangle$ then $w \in \Sigma$, $s \preceq w$ and $s_i \not\preceq w, \forall i$. In this case the pattern $\langle s, s_1, \ldots, s_n, w \rangle$ represents the set of strings t such that $s \preceq t$, $s_i \not\preceq t, \forall i$ and $w \not\preceq t$, that is, the strings in $\mathcal{S}(\pi)$ that are not bigger than w.
- If $\pi = \langle -, s_1, \ldots, s_n \rangle$ then $w = t_1 \ldots t_m$ such that $s_i \not\preceq t_j, \forall i, j$. In this case we obtain the following patterns after reducing π:

- $\langle -, s_1, \ldots, s_n, t_1 \rangle$: This pattern represents any string w whose elements are not bigger than $s_i, \forall i$ and t_1.
- $\langle -, s_1, ..., s_n, t_1 \rangle \langle t_1, s_1, ..., s_n \rangle \langle -, s_1, ..., s_n, t_2 \rangle$: This pattern represents any string $w_1 t w_2$ such that, the elements of w_1 are not bigger than $s_i, \forall i$ and t_1; t is an element of Σ bigger than t_1 and not bigger than $s_i, \forall i$; and the elements of w_2 are not bigger than $s_i, \forall i$ and t_2.
- $\langle -, s_1, ..., s_n, t_1 \rangle \langle t_1, s_1, ..., s_n \rangle \langle -, s_1, ..., s_n, t_2 \rangle \langle t_2, s_1, ..., s_n \rangle \langle -, s_1, ..., s_n, t_3 \rangle$
- And so on.

The patterns obtained in the second case are all disjoint because the first one represents strings whose elements are not bigger than t_1, the second one represents strings with an element bigger than t_1 followed by elements not bigger than t_2, the third one represents strings with an element bigger than t_1 followed by an element bigger than t_2 and this followed by elements not bigger than t_3, and so on. This disjointness property is preserved by the reduction process of general patterns.

The function `reduce-simple-pattern`(w,π) computes the reductions of the simple pattern π with respect to the string w (assuming that $w \in \mathcal{S}(\pi)$). Let us recall that the sequence with respect to which a simple pattern is defined is represented in ACL2 in reverse order, allowing easy additions of new elements to it:

DEFINITION:
`reduce-simple-pattern`(w,π) =
  if `endp`(w) **then nil**
  elseif `consp`(`car`(π))
    then `list`(`list`(`cons`(`car`(π),`cons`(`car`(w),`cdr`(π))))))
  else `cons`(`list`(`cons`(**nil**,`cons`(`car`(w),`cdr`(π)))),
    `cons2-list-cdr`(`cons`(**nil**,`cons`(`car`(w),`cdr`(π))),
      `cons`(`list`(`car`(w)),`cdr`(π)),
      `reduce-simple-pattern`(`cdr`(w),π)))

where the function `cons2-list-cdr` behaves schematically in the following way:

`(cons2-list-cdr 'x 'y '(`l_1 `...` l_n`)) = '((`x y `.` l_1`) ... (`x y `.` l_n`))`

As we have discussed in the example, the reduction of a pattern depends on its components. Given a pattern $\Pi = \pi_1 \ldots \pi_n$ and a string $w \in \mathcal{S}(\Pi)$, there exist strings $w_1 \ldots w_n$ such that $w = w_1 \ldots w_n$ and $w_i \in \mathcal{S}(\pi_i), \forall i$. The reduction of Π with respect to w is the set of patterns $\pi_1 \ldots \pi_{i-1} \pi'_1 \ldots \pi'_m \pi_{i+1} \ldots \pi_n$ for every index i and every pattern $\pi'_1 \ldots \pi'_m$ obtained by reducing the simple pattern π_i with respect to w_i. The function `reduce-simple-pattern-list` computes the reduction of a list of simple patterns (the components of Π) with respect to a list of strings (the components of w).

DEFINITION:
`reduce-simple-pattern-list`(w-lst,Π) =
  if `endp`(Π) **then nil**

else append-list-car(reduce-simple-pattern(car(w-lst),car(Π)),
cdr(Π)) @
cons-list-cdr(car(Π),
reduce-simple-pattern-list(cdr(w-lst),cdr(Π)))

where the symbol @ is the "append" operation between lists, and the functions append-list-car and cons-list-cdr behave schematically as follows:

(append-list-car '(l_1 ... l_n) 'l = '(l_1 @ l ... l_n @ l)
(cons-list-cdr 'x '(l_1 ... l_n)) = '((x . l_1) ... (x . l_n))

The function reduce-pattern(w,Π) computes the reduction of the pattern Π with respect to the string w, whenever $w \in \mathcal{S}(\Pi)$. If this is not the case, the function returns the initial pattern Π:

Definition:
reduce-pattern(w,Π) =
let ⟨res, val⟩ **be** member-pattern(w,Π)
in if res **then** reduce-simple-pattern-list(val,Π)
else Π

Now we deal with set of patterns. In the following let be denote $\mathcal{P}$ a set of patterns and $\mathcal{S}(\mathcal{P})$ the set of strings represented by the patterns in $\mathcal{P}$ ($\mathcal{P}$ is a set of patterns ≡ pattern-list-p($\mathcal{P}$)). The function reduce-pattern-list describes how the set of patterns $\mathcal{P}$ is reduced with respect to a string w:[4]

Definition:
reduce-pattern-list(w,$\mathcal{P}$) =
if endp($\mathcal{P}$) **then** $\mathcal{P}$
else let ⟨res, val⟩ **be** member-pattern(w,car($\mathcal{P}$))
in if res **then** reduce-pattern(w,car($\mathcal{P}$)) @ cdr($\mathcal{P}$)
else cons(car($\mathcal{P}$),reduce-pattern-list(w,cdr($\mathcal{P}$)))

The function reduce-pattern-sequence-list iterates the reduction process over a finite sequence of strings. It must be noticed that the sequence of strings is provided in reverse order:

Definition:
reduce-pattern-sequence-list($\overline{w}$,$\mathcal{P}$) =
if endp($\overline{w}$) **then** $\mathcal{P}$
else reduce-pattern-list(
car($\overline{w}$),reduce-pattern-sequence-list(cdr($\overline{w}$),$\mathcal{P}$))

[4] Since the reduction process produces patterns representing disjoint sets of strings, it is enough to make the reduction with respect to the first pattern matched by member-pattern. It must be noticed that it would be sufficient to make one reduction even in the case in which $\mathcal{P}$ contains patterns representing non-disjoint sets of strings.

Recall that our goal is to assign a well-founded measure to every initial subsequence of `f`, in such a way that every time a new element appears in the sequence such that it is not bigger than any of the previous elements, then the corresponding measure of the extended subsequence is strictly smaller. Intuitively, we will measure the "size" of the set of elements that can be the next in the sequence without affecting the well-quasi-order-ness property. Since we have seen that this set of strings can be represented by a set of patterns, we will measure sets of patterns.

First, we assign an ordinal measure to simple patterns, based on the ordinal measure provided by `sigma-seq-measure`: if o is the measure of the sequence $\overline{s}$, then the measure of the simple pattern $\langle -, \overline{s} \rangle$ is $\omega^o + 1$, and the measure of the simple pattern $\langle t, \overline{s} \rangle$ is ω^o. The measure of a pattern is the multiset of the measures of the simple patterns in it. Finally, the measure of a set of patterns is the multiset of measures of the patterns in it. The ACL2 functions `simple-pattern-measure`(π), `pattern-measure`(Π) and `pattern-list-measure`($\mathcal{P}$) compute respectively the measure of the simple pattern π, of the pattern Π and of the set of patterns $\mathcal{P}$. We use this last function to associate a measure to every index k:[5]

DEFINITION:

```
higman-indices-measure(k) =
  pattern-list-measure(
    reduce-pattern-sequence-list(initial-segment-f(k − 1),
                                 list(initial-pattern())))
```

where the function `initial-pattern`() builds the initial pattern $\langle - \rangle$ and the function `initial-segment-f`(k) builds the list of strings `(`f_k ... f_1 f_0`)`. Note that this measure is a finite multiset of finite multisets of ordinals.

The following table summarizes the measures in the given example:

Initial subsequence	Set of patterns	Measure
$\{\}$	$\{\langle - \rangle\}$	$\{\!\{\{\!\{\omega^{\omega \cdot 2} + 1\}\!\}\}\!\}$
$\{3 \cdot 2\}$	$\{\langle -, 3 \rangle,$ $\langle -, 3 \rangle\langle 3 \rangle\langle -, 2 \rangle\}$	$\{\!\{\{\!\{\omega^{\omega+3} + 1\}\!\},$ $\{\!\{\omega^{\omega+3} + 1, \omega^{\omega \cdot 2}, \omega^{\omega+2} + 1\}\!\}\}\!\}$
$\{3 \cdot 2,$ $1 \cdot 2\}$	$\{\langle -, 3, 1 \rangle,$ $\langle -, 3, 1 \rangle\langle 1, 3 \rangle\langle -, 3, 2 \rangle,$ $\langle -, 3 \rangle\langle 3 \rangle\langle -, 2 \rangle\}$	$\{\!\{\{\!\{\omega^{\omega+1} + 1\}\!\},$ $\{\!\{\omega^{\omega+1} + 1, \omega^{\omega+3}, \omega^{3+2} + 1\}\!\},$ $\{\!\{\omega^{\omega+3} + 1, \omega^{\omega \cdot 2}, \omega^{\omega+2} + 1\}\!\}\}\!\}$
$\{3 \cdot 2,$ $1 \cdot 2,$ $1 \cdot 5 \cdot 3\}$	$\{\langle -, 3, 1 \rangle,$ $\langle -, 3, 1 \rangle\langle 1, 3 \rangle\langle -, 3, 2 \rangle,$ $\langle -, 3, 1 \rangle\langle 3 \rangle\langle -, 2 \rangle,$ $\langle -, 3 \rangle\langle 3, 5 \rangle\langle -, 2 \rangle,$ $\langle -, 3 \rangle\langle 3 \rangle\langle -, 2, 3 \rangle\}$	$\{\!\{\{\!\{\omega^{\omega+1} + 1\}\!\},$ $\{\!\{\omega^{\omega+1} + 1, \omega^{\omega+3}, \omega^{3+2} + 1\}\!\},$ $\{\!\{\omega^{\omega+1} + 1, \omega^{\omega \cdot 2}, \omega^{\omega+2} + 1\}\!\},$ $\{\!\{\omega^{\omega+3} + 1, \omega^{\omega+5}, \omega^{\omega+2} + 1\}\!\},$ $\{\!\{\omega^{\omega+3} + 1, \omega^{\omega \cdot 2}, \omega^{3+2} + 1\}\!\}\}\!\}$

[5] Note that since `f` is fixed, then k is sufficient to represent the initial subsequence $\{f_0 \ldots f_{k-1}\}$.

2.3 Termination Proof of `higman-indices`

The last step in this formal proof is to define a well-founded relation and prove that the given measure decreases with respect to it in every recursive call of the function `higman-indices`. We will define it as the relation on finite multisets induced by a well-founded relation. Intuitively, this relation is defined in such a way that a smaller multiset can be obtained by removing a non-empty subset of elements, and adding elements which are smaller than some of the removed elements. In [4], Dershowitz and Manna show that if the base relation is well-founded, then the relation induced on finite multisets is also well-founded.

As we said above, the only predefined well-founded relation in ACL2 is `o<`, implementing the usual order between ordinals less than ε_0. The function `o-p` recognizes those ACL2 objects representing such ordinals. If we want to define a new well-founded relation in ACL2, we have to explicitly provide a monotone ordinal function, and prove the corresponding order-preserving theorem (see [5] for details). Fortunately, we do not have to do this: we use the `defmul` tool. This tool, previously implemented and used by the authors in [13], automatically generates the definitions and proves the theorems needed to introduce in ACL2 the multiset relation induced by a given well-founded relation.

In our case, we need two `defmul` calls. The first one automatically generates the definition of the function `mul-o<`, implementing the multiset relation on finite multisets of ordinals (the measure of a pattern) induced by the relation `o<`; and the second one automatically generates the definition of `mul-mul-o<`, implementing the multiset relation of finite multisets of finite multisets of ordinals (the measure of a pattern list) induced by the relation `mul-o<`. These calls also automatically prove the theorems needed to introduce these relations as well-founded relations in ACL2. See details about the `defmul` syntax in [13]. For simplicity, in the following we denote `mul-o<` as $<_{\varepsilon_0,\mathcal{M}}$ and `mul-mul-o<` as $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$.

We finally prove that the measure decreases with respect to $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$ in the recursive call of the function `higman-indices`, hence justifying its termination. We now explain the main lemmas needed to prove this result. We start with the ones related to simple patterns. We have two cases:

- If the simple pattern is $\pi = \langle s, s_1, \ldots, s_n \rangle$ and a string $w \in \mathcal{S}(\pi)$, then $w \in \Sigma$, $s \preceq w$ and $s_i \not\preceq w, \forall i$. In this case the reduction of π with respect to w is the simple pattern $\pi' = \langle s, s_1, \ldots, s_n, w \rangle$. Then $\neg$`exists-sigma-<=`$(\overline{s},w)$ where $\overline{s} = \langle s_1, \ldots, s_n \rangle$. Thus the well-quasi-order characterization of $\preceq$ ensures that `sigma-seq-measure`(`cons`$(w,\overline{s})$) is less than `sigma-seq-measure`$(\overline{s})$. Therefore, the measure of π' is less than the measure of π.
- If the simple pattern is $\pi = \langle -, s_1, \ldots, s_n \rangle$ and a string $w \in \mathcal{S}(\pi)$, then $w = t_1 \ldots t_m$ such that $s_i \not\preceq t_j, \forall i, j$. In this case the reduction of π with respect to w is a set of patterns whose components are of one of two kinds:
  - Simple patterns as $\pi' = \langle -, s_1, \ldots, s_n, t_j \rangle$. Then, if $\overline{s} = \langle s_1, \ldots, s_n \rangle$, we have $\neg$`exists-sigma-<=`$(\overline{s},t_j)$. Thus the well-quasi-order characterization of $\preceq$ ensures that `sigma-seq-measure`(`cons`$(t_j,\overline{s})$) is less than

`sigma-seq-measure`($\overline{s}$). Therefore, the measure of π' is less than the measure of π.

- Simple patterns as $\pi' = \langle t_j, s_1, \ldots, s_n \rangle$. Then, if $\overline{s} = \langle s_1, \ldots, s_n \rangle$ and $o =$`sigma-seq-measure`($\overline{s}$), the measure of π' is ω^o and the measure of π is $\omega^o + 1$. Therefore, the measure of π' is less than the measure of π.

The following ACL2 lemma summarizes these considerations:

LEMMA: `reduce-simple-pattern-property`
(`simple-pattern-p`(π) $\wedge$ $w \in \Sigma^*$ $\wedge$ $w \in \mathcal{S}(\pi)$
$\wedge$ $\Pi \in$ `reduce-simple-pattern`(w,π) $\wedge$ $\pi' \in \Pi$)
$\rightarrow$ `measure-simple-pattern`(π') $<_{\varepsilon_0}$ `measure-simple-pattern`(π)

where $\pi' \in \Pi$ indicates that the simple pattern π' is a component of the pattern Π.

Given a pattern $\Pi = \pi_1 \ldots \pi_n$ and $w \in \mathcal{S}(\Pi)$, to compute the patterns in the reduction of Π with respect to w, we consider $w = w_1 \ldots w_n$ such that $w_i \in \mathcal{S}(\pi_i), \forall i$, and $\pi'_1 \ldots \pi'_m$ obtained by reducing the simple pattern π_i with respect to w_i. Then, the pattern $\Pi' = \pi_1 \ldots \pi_{i-1} \pi'_1 \ldots \pi'_m \pi_{i+1} \ldots \pi_n$ is in the reduction of Π with respect to w. The previous lemma ensures that the measure of π'_j is less than the measure of $\pi_i, \forall i, j$, therefore, the measure of Π' is obtained from the measure of Π, replacing the measure of π_i with the smaller measures of $\pi'_1, \ldots, \pi'_m$. Then, the measure of Π' is a multiset less than the measure of Π:

LEMMA: `reduce-pattern-property`
`pattern-p`(Π_2) $\wedge$ $w \in \Sigma^*$ $\wedge$ $w \in \mathcal{S}(\Pi_2)$ $\wedge$ $\Pi_1 \in$ `reduce-pattern`(w,Π_2)
$\rightarrow$ `measure-pattern`(Π_1) $<_{\varepsilon_0,\mathcal{M}}$ `measure-pattern`(Π_2)

Finally, as a direct consequence of the previous lemma, if $w \in \mathcal{S}(\mathcal{P})$, then the measure of `reduce-pattern-list`(w,$\mathcal{P}$) is smaller than the measure of $\mathcal{P}$ with respect to $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$:

LEMMA: `reduce-pattern-list-property`
$w \in \Sigma^* \wedge$ `pattern-list-p`($\mathcal{P}$) $\wedge$ $w \in \mathcal{S}(\mathcal{P})$
$\rightarrow$ `pattern-list-measure`(`reduce-pattern-list`(w,$\mathcal{P}$))
$<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$ `pattern-list-measure`($\mathcal{P}$)

Now, we prove that the reduction process only removes strings bigger than some string in the sequence with respect to which the reduction is made. The following lemma establishes this property. If $u \in \mathcal{S}(\mathcal{P})$ then u is still in the set of strings represented by the set of patterns obtained after reducing $\mathcal{P}$ with respect to a given finite sequence of strings $\overline{w}$ ($\overline{w} \in \overline{\Sigma^*} \equiv$ `sigma-*-seq-p`($\overline{w}$)), provided that u is not bigger than any of the strings of $\overline{w}$ (this condition is checked by the function `exists-sigma-*-<=`, whose definition we omit here):

LEMMA: `exists-pattern-reduce-pattern-sequence-list`
($u \in \Sigma^*$ $\wedge$ `sigma-*-seq-p`($\overline{w}$) $\wedge$ `pattern-list-p`($\mathcal{P}$)
$\wedge$ $u \in \mathcal{S}(\mathcal{P})$ $\wedge$ $\neg$`exists-sigma-*-<=`($\overline{w}$,u))
$\rightarrow$ $u \in \mathcal{S}$(`reduce-pattern-sequence-list`($\overline{w}$,$\mathcal{P}$))

In addition, every string is in the initial set of patterns $\{\langle - \rangle\}$:

LEMMA: `initial-pattern-exists-pattern`
$w \in \Sigma^* \rightarrow w \in \mathcal{S}(\texttt{list}(\texttt{initial-pattern}()))$

As a consequence of the above two lemmas, if f_k is not bigger than any of $f_0 \ldots f_{k-1}$ (that is, the recursive case in the definition of `higman-indices`), then if $\mathcal{P}$ is the set of patterns generated in the k-th step, $w \in \mathcal{S}(\mathcal{P})$. So we can use the lemma `reduce-pattern-list-property` to conclude that the measure of the argument in the recursive call in `higman-indices` decreases with respect to $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$. That is, we have the following theorem:

THEOREM: `higman-indices-termination-property`
$k \in \mathbb{N} \wedge \neg\texttt{get-sigma-*-<=-f}(k,\texttt{f}(k))$
$\rightarrow \texttt{higman-indices-measure}(k+1) <_{\varepsilon_0,\mathcal{M}_\mathcal{M}} \texttt{higman-indices-measure}(k)$

This is exactly the proof obligation generated to show the termination of the function `higman-indices`. Thus, its definition is admitted in ACL2 and then the theorem `higman-lemma` presented in subsection 2 is easily proved.

3 Concluding Remarks

The complete ACL2 files with definitions and theorems are available on the Web at `http://www.cs.us.es/~fmartin/acl2/higman/`. To quantify the proof effort, it was done in about 3 weeks of partial time work and only 43 definitions and 76 lemmas (with 17 non trivial proof hints explicitly given) were needed, which gives an idea of the degree of automation of the proof. The development benefits from the previously developed `multiset` book, which provides a proof of well-foundedness of the multiset relation induced by a well-founded relation. It is worth emphasizing the reuse of the `defmul` tool for generating multiset well-founded relations in ACL2 (see [13] for more uses of this tool).

Our proof is slightly different from the one presented in [10]. For example, our construction of the order used to prove the termination of `higman-indices` is more concise. Another important point is the level of detail that we must have in the formalization; this reveals important properties needed in the development of the proof that are not mentioned in [10]. For example, to prove the lemma `exists-pattern-reduce-pattern-sequence-list` we need a stability property about $\preceq^*$: $w_1 \preceq^* w_2 \wedge w_3 \preceq^* w_4 \rightarrow w_1 w_3 \preceq^* w_2 w_4$. The proof of this property is not trivial in our formalization.

There are several constructive mechanizations of Higman's lemma in the literature, for example [2] in the Isabelle system (the author also has the same work done in the COQ system), based on Coquand's constructivization [3] of Nash-William classical proof [11]; and [14] in the MINLOG system. These works are related to program extraction from proofs, whereas our work starts with a program solving the problem and then we prove its properties. This different approach results in a more concise code: our program has 18 lines of LISP code

whereas in [2] the program has 70 lines of ML code; and a more simple result: our program returns the first elements in the sequence such that $w_i \preceq^* w_j$, but this is not the case in [2,14]. On the other hand these works are more restricted than the presented here: in [2] the set Σ has only two values and the well-quasi-order relation is equality; this is not the case in [14] where the proof developed is more general, but the MINLOG formalization is restricted to a finite alphabet.

Acknowledgment

We are grateful to Matt Kauffman and the referees for their helpful comments and suggestions on this paper. Thanks to Matt we have a better understanding of the ACL2 behavior.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. S. Berghofer. *A constructive proof of Higman's lemma in Isabelle.* In *Types for Proofs and Programs, TYPES'04.* LNCS, 3085: 66-82. Springer Verlag, 2004.
3. T. Coquand and D. Fridlender. *A proof of Higman's lemma by structural induction.* Unpublished draft, available at http://www.brics.dk/ daniel/texts/open.ps, 1993.
4. N. Dershowitz and Z. Manna. Proving Termination with Multiset Orderings. *Communications of the ACM* 22(8):465–476, 1979.
5. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
6. M. Kaufmann and J S. Moore. *ACL2 Version 2.9*, 2005.
Homepage: `http://www.cs.utexas.edu/users/moore/acl2/`
7. M. Kaufmann and J S. Moore. *Structured Theory Development for a Mechanized Logic.* Journal of Automated Reasoning, 26(2): 161–203, 2001.
8. F.J. Martín–Mateos, J.A. Alonso, M.J. Hidalgo, and J.L. Ruiz–Reina. A Formal Proof of Dickson's Lemma in ACL2. In *Proceedings of LPAR'03.* LNAI, 2850: 49-58. Springer Verlag, 2003.
9. I. Medina–Bulo, F. Palomo, J.A. Alonso, J.L Ruiz–Reina. Verified Computer Algebra in ACL2 (Gröbner basis Computation) In *Proceedings of AISC'04.* LNAI, 3249: 171-184. Springer Verlag, 2004.
10. C. Murthy, J.R. Russell. A Constructive Proof of Higman's Lemma. In *Fifth annual IEEE Symposium on Logic in Computer Science*, pp 257-267, 1990.
11. C. Nash–Williams. *On well-quasi-ordering finite trees.* Proceedings of the Cambridge Philosophical Society, 59:833–835, 1963.
12. H. Perdry. Strong noetherianity: a new constructive proof of Hilbert's basis theorem. Available at `http://perdry.free.fr/StrongNoetherianity.ps`
13. J.L. Ruiz–Reina, J.A. Alonso, M.J. Hidalgo, and F.J. Martín–Mateos. Termination in ACL2 Using Multiset Relations. In *Thirty Five Years of Automating Mathematics.* Kluwer Academic Publisher, 2003.
14. M. Seisenberger. *An Inductive Version of Nash-Williams' Minimal-Bad-Sequence Argument for Higman's Lemma.* In *Types for Proofs and Programs, TYPES'00.* LNCS, 2277: 233-242. Springer Verlag, 2001.
15. S.G. Simpson. Ordinal numbers and the Hilbert basis theorem. *Journal of Symbolic Logic* 53(3): 961–974, 1988.