Proof Pearl: a Formal Proof of Higman's Lemma in ACL2

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

Abstract Higman's lemma is an important result in infinitary combinatorics, which has been formalized in several theorem provers. In this paper we present a formalization and proof of Higman's Lemma in the ACL2 theorem prover. Our formalization is based on a proof by Murthy and Russell, where the key termination argument is justified by the multiset relation induced by a well-founded relation. To our knowledge, this is the first mechanization of this proof.

Keywords Higman's lemma · Formal proofs · ACL2

1 Introduction

Higman's lemma [7] is a result in the field of combinatorics, stating well-quasi-orderness of a certain embedding relation on finite strings over a well-quasi-ordered alphabet. It provides a criterion for proving termination of string rewrite systems and it is a particular case of Kruskal's tree theorem, which plays a fundamental role in the proof of well-foundedness of certain orderings used to show termination of term rewriting systems [1].

F. J. Martín-Mateos · J. L. Ruiz-Reina (✉) · J. A. Alonso · M. J. Hidalgo
~~Computational Logic Group, D~~epartment of Computer Science and Artificial Intelligence, University of Seville, E.T.S.I. Informática, Avda. Reina Mercedes, s/n. 41012 Sevilla, Spain

The best known classical proof of Higman's lemma is due to Nash-Williams [15], using a so-called *minimal bad sequence* argument. After that, efforts were devoted to obtain constructive proofs of the lemma and to interpret their computational contents, also formalizing some of the proofs using theorem provers. Murthy [13] obtained the first formal proof of the result, using the Nuprl system to apply Friedman's *A-translation* to Nash-Williams' classical proof; nevertheless, the proof obtained was so huge that its computational content was very difficult to analyze. This raised interest in obtaining direct constructive proofs of the result [3, 14, 16, 19, 20], with a clearer computational content. Some of these proofs have been formalized using proof assistants and theorem provers: ALF [5], MINLOG [18], COQ [6] or Isabelle [2].

One of these direct constructive proofs, discovered by Murthy and Russell [14], is based on defining a well-founded ordering on sets of finite descriptions of strings, and this ordering is obtained using multiset extensions of well-founded orderings. This proof had not yet been formalized in any theorem prover, but the techniques used were similar to the ones we used in a previous formalization of Dickson's lemma [11], using the ACL2 theorem prover. This addressed our attention to get a formalization of Murthy and Russell's proof of Higman's lemma in ACL2.

To our knowledge, the work we present here is the first formalization of Murthy and Russell's proof of Higman's lemma. This proof has an obvious computational content and this is reflected in our ACL2 proof because we verify a naive search algorithm (written in Common Lisp) providing the elements whose existence is stated by Higman's lemma; this algorithm has very simple properties but a non-trivial termination proof. It is also worth mentioning that a first-order logic like the ACL2 logic is expressive enough to state and prove Higman's lemma.

The rest of this paper is structured as follows. Section 2 presents a short description of the ACL2 logic and its theorem prover. In Section 3, we precisely state Higman's lemma and its formulation in the ACL2 logic. In Section 4, we introduce the formalism of patterns, a tool for defining the well-founded measure that will be essential in the proof. Section 5 presents the proof itself. Finally, in Section 6 some conclusions are given. This paper is a substantially revised version of a preliminary paper presented in [12].

2 A Brief Introduction to ACL2

ACL2 [8, 9] is a programming language, a logic, and a theorem prover supporting reasoning in the logic. The ACL2 programming language is an extension of an applicative subset of Common Lisp. The ACL2 logic is a first-order logic with equality, used for specifying properties and reasoning about the functions defined in the programming language. The formulas allowed by the ACL2 system do not have quantifiers and all the variables in them are implicitly universally quantified. The syntax of its terms and formulas is that of Common Lisp and it includes axioms for propositional logic, equality and for a number of predefined Common 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 logic has a constructive definition

of the ordinals up to ε_0, in terms of lists and natural numbers. The predicate `o-p` recognizes those ACL2 objects that represent ordinals and the relation `o<` defines the usual order between them. Although this is the only built-in well-founded relation, the user may define new well-founded relations on domains of ACL2 objects other than the one characterized by `o-p`, provided that an ordinal mapping is explicitly given and proved to be monotone.

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

An additional way to introduce new function symbols in the logic is by means of the `encapsulate` mechanism [10]. Instead of giving their definitional body, only certain properties are assumed about them; to ensure consistency, witness local functions having the same properties have to be exhibited. Inside an `encapsulate`, the properties stated need to be proved for the local witnesses, and outside, they work as assumed axioms.

A derived rule of inference, called *functional instantiation*, provides a limited higher-order-like reasoning mechanism allowing to instantiate the function symbols of a previously proved theorem, replacing them with other function symbols, provided it can be proved that the new functions satisfy the constraints or the definitional axioms of the replaced functions (depending on whether they were introduced by an `encapsulate` or by the principle of definition, respectively). See [8] for details.

The ACL2 logic is usually referred to as "quantifier-free" due to the absence of quantifiers in its formulas. Nevertheless, ACL2 provides limited support for first-order quantification, via the macro `defun-sk`, which allows (by means of a choice axiom) to define functions whose body has an outermost quantifier.

The ACL2 theorem prover mechanizes the ACL2 logic, being particularly well suited for obtaining mechanical proofs mainly based on simplification and induction. The role of the user in this mechanization is important: usually a non-trivial result is not proved in a first attempt, and the user has to lead the prover to a successful proof by means of a set of lemmas that the prover uses mainly as rewrite rules.

For the sake of readability, in this paper the ACL2 expressions will be presented using a notation closer to the usual mathematical notation than its original Common Lisp syntax, and some of the functions will also be used in infix notation. When needed, we will show the correspondence between the ACL2 functions in the source code and the mathematical notation used instead. We will use well-known Lisp functions (such as `cons`, `car` and `cdr`) to denote operations on lists. In particular, the function `endp` checks if its argument is the empty list (which is denoted as **nil**), while `consp` is its negation. The function `list` returns the list of its arguments and `caar` is an standard abbreviation for the composition of two `car`'s.

We will necessarily skip many details and some of the function definitions will be omitted. We urge the interested reader to consult the original and complete source code at http://www.glc.us.es/fmartin/acl2/higman

3 Higman's Lemma and Its ACL2 Formulation

In the following, let Σ be a set (also called an *alphabet*), and let Σ^* denote the set of finite strings over Σ.

Definition 1 Let $\preceq$ be a binary relation on Σ. Given two strings $v = v_1 \cdots v_m$ and $w = w_1 \cdots w_n$ in Σ^*, we say that v *is embedded into* w (or that w *embeds* v), denoted as $v \preceq^* w$, if there exist indices $0 < j_1 < j_2 < \ldots < j_m \leq n$ such that $v_i \preceq w_{j_i}, \forall i$. We also say that $\preceq^*$ is the *embedding relation induced* by $\preceq$ on Σ^*.

If $s \preceq t$ we usually say that s is less than t or that t is greater than s. The same terminology may be applied to the ordering $\preceq^*$ (the relation with respect to which an element is less or greater than other will be clear from the context).

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 $\{s_k : k \in \mathbb{N}\}$. An infinite sequence $\{s_k : k \in \mathbb{N}\}$ of elements of Σ is called *good* (with respect to a relation $\preceq$) if there exist indices $i < j$ such that $s_i \preceq s_j$; otherwise it is called *bad*.

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 every infinite sequence of elements of Σ is good.

Example 1 Let Σ be the set of natural numbers, $\mathbb{N}$, and $\preceq$ the least reflexive and transitive relation on $\mathbb{N}$ such that $n \preceq n + 2$ 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 check that this relation is a well-quasi-order.

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^*.*

We now describe how Higman's lemma can be formulated in the ACL2 logic. Since it is very easy to prove that $\preceq^*$ is a quasi-order on Σ^* when $\preceq$ is a quasi-order on Σ, we will focus our attention on the well-quasi-order property: every infinite sequence of strings is good, with respect to $\preceq^*$.

First of all, we represent Σ with a unary predicate `sigma-p` ($s \in \Sigma \equiv$ `sigma-p`(s))[1] and $\preceq$ with a binary predicate `sigma-<=` ($s \preceq t \equiv$ `sigma-<=`(s,t)). These functions are introduced in the logic by means of the `encapsulate`

[1]We will use the symbol $\equiv$ to indicate the correspondence between an ACL2 function and the notation used instead, in the formulas of this paper.

mechanism, only assuming about them the properties stating that $\preceq$ is a well-quasi-order on Σ.[2] The first two assumptions state that $\preceq$ is a quasi-order on Σ:

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

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

Let us now deal with the formalization of the assumption of $\preceq$ being a well-quasi-order on Σ. In principle, it should be stated by a formula expressing that every infinite sequence of elements of Σ is good, with respect to $\preceq$. But this cannot be directly stated in a first-order logic like ACL2, because it would require a second-order universal quantifier over the infinite sequences of elements of Σ. Following [14], we overcome this drawback considering an alternative definition for well-quasi-orders.

In the following, we will use $\overline{\Sigma}$ to denote the set of finite sequences of elements of Σ and the overline notation to identify the elements of $\overline{\Sigma}$. We characterize the well-quasi-order property of $\preceq$ in the following way: there exists a well-founded measure on $\overline{\Sigma}$ such that the measure of any sequence in $\overline{\Sigma}$ is greater than the measure of any extension of this sequence with an element s such that there is no element already in the sequence that is less than s. As pointed out in [14]: "*Classically, this is easily gotten from the well-quasi-orderness 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.*" In some way, this characterization resembles the principle of *bar induction* used in [3], since extensions of finite sequences play the role of "induction steps"; but unlike bar induction, this characterization can be naturally formalized in the ACL2 logic.

Example 2 For the relation defined in Example 1, a mapping from $\overline{\mathbb{N}}$ to ordinals, characterizing the well-quasi-orderness of $\preceq$, could be the following: given a finite sequence of natural numbers $\bar{s}$, the ordinal associated is $\omega \cdot 2$ if $\bar{s}$ is empty; $\omega + n$ if n is the last even (odd) number in $\bar{s}$ and there is no odd (even) numbers in $\bar{s}$; and $n + m$ if n and m are respectively the last even number and the last odd number in $\bar{s}$. It is easy to show that whenever we extend a finite sequence with a natural number not greater (w.r.t. $\preceq$) than any of the elements already in the sequence, this measure strictly decreases.

In our ACL2 formalization, the measure characterizing well-quasi-orderness of $\preceq$ is abstractly introduced in the previous `encapsulate` by means of a function `sigma-seq-measure`. To state the properties assumed about the measure we also need two functions whose definitions are based on `sigma-p` and `sigma-<=`: the membership to $\overline{\Sigma}$, defined by the function `sigma-seq-p` ($\bar{s} \in \overline{\Sigma} \equiv$ `sigma-seq-p`$(\bar{s})$) and the existence of an element in a finite sequence $\bar{s}$ less than an

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

element t, defined by the function `exists-sigma-<=`. Then the assumed properties about `sigma-seq-measure` are the following:[3]

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

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

The first property ensures that the function `sigma-seq-measure` returns an ACL2 ordinal when its argument is a finite sequence of strings, and the second property is the constructive characterization of the well-quasi-orderness of $\preceq$.[4] Notice that we represent finite sequences by lists of elements in reverse order: that is, the ACL2 representation of the finite sequence $\langle s_1, \ldots, s_n \rangle$ is the list $(s_n \ \ldots \ s_1)$. The reason for this is that an element is more easily added to a finite sequence using `cons` than using `append` (at least from the point of view of reasoning).

The four previous assumptions state that $\preceq$ is a well-quasi-order on Σ and they are the only assumed properties about Σ and $\preceq$.

The next step is to define Σ^* and the $\preceq^*$ relation. We represent the elements of Σ^* by lists, but in this case in the usual order. Thus, the representation of a string $s_1 s_2 \ldots s_m$ is the list $(s_1 \ s_2 \ \ldots \ s_m)$. Membership to Σ^* is checked by the function `sigma-*-p` ($w \in \Sigma^* \equiv$ `sigma-*-p`(w)). For the definition of $\preceq^*$, we need the following auxiliary function `sigma-*-<=-indices`, such that given two strings w_1 and w_2 and a list of indices l, checks that w_1 is pointwise less than the corresponding sequence of elements of w_2 indicated by the indices of l:

DEFINITION:
`sigma-*-<=-indices`$(w_1,w_2,l) \Leftrightarrow$
if `endp`(l) **then t**
else `car`$(w_1) \preceq$`nth`$(\mathtt{car}(l),w_2) \wedge$`sigma-*-<=-indices`$(\mathtt{cdr}(w_1),w_2,\mathtt{cdr}(l))$

Now the relation $\preceq^*$ in Σ^* is easily defined by the following function `sigma-*-<=`, stating that there exists a strictly increasing list of indices witnessing the embedding of w_1 into w_2:

DEFINITION:
`sigma-*-<=`$(w_1,w_2) \Leftrightarrow$
$\exists\, l$ (`nat-listp`$(l) \wedge$
`bounded-indicesp`$(l,$`length`$(w_2)) \wedge$
`length`$(l) =$`length`$(w_1) \wedge$
`orderedp`$(l) \wedge$
`sigma-*-<=-indices`(w_1,w_2,l))

Here the predicate `nat-listp` checks that its argument is a list of natural numbers, `bounded-indicesp` is defined such that `bounded-indicesp`(l,x) if and

[3]For ACL2 ordinals we use the notation: $o \in Ord \equiv$ `o-p`(o) and $o_1 <_{\varepsilon_0} o_2 \equiv$ `o<`(o_1,o_2).

[4]Note that we are only considering well-quasi-orderings of order type at most ε_0, since that is the maximal ordinal type for a well-founded relation in ACL2. Nevertheless, no explicit use of this restriction is done in the proof.

only if all the elements of l are less than x, and `orderedp` checks that its argument is a strictly increasing list. In the actual ACL2 code, the existential quantification is introduced via `defun-sk`.

Having defined $\preceq^*$, let us state that every infinite sequence of elements in Σ^* is good, with respect to $\preceq^*$. For that, we use again the `encapsulate` mechanism, introducing an arbitrary infinite sequence of strings, named `f`. The only assumed property about `f` is the following:

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

Note that since the infinite sequence of strings is abstractly introduced via `encapsulate`, the property that we will prove about it (that is, that the sequence is good) will be valid for any infinite sequence of strings. To state that `f` is good, we have to show the existence of two indices $i < j$ such that $\mathtt{f}(i) \preceq^* \mathtt{f}(j)$. For that, we simply define them as the result of a naive search algorithm. The auxiliary function `get-sigma-*-<=-f` has two arguments, a natural number j and a string w, and it returns the largest index i such that $i < j$ and $\mathtt{f}(i) \preceq^* w$ whenever such an index exists (`nil` otherwise):

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

DEFINITION:
`higman-indices`$(k) =$
if $k \in \mathbb{N}$ **then let** i **be** `get-sigma-*-<=-f`$(k,\mathtt{f}(k))$
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 `get-sigma-*-<=-f` and `higman-indices`:

THEOREM: `higman-lemma`
$k \in \mathbb{N} \rightarrow$ **let** i **be** `first`(`higman-indices`(k))
j **be** `second`(`higman-indices`(k))
in $[i, j \in \mathbb{N} \wedge i < j \wedge \mathtt{f}(i) \preceq^* \mathtt{f}(j)]$

This theorem ensures that for any infinite sequence of strings $\{\mathtt{f}(k) : k \in \mathbb{N}\}$, there exist $i < j$ such that $\mathtt{f}(i) \preceq^* \mathtt{f}(j)$ (and the function `higman-indices` explicitly provides these 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 it decreases with respect to a given well-founded relation in every recursive call. We describe this measure and the corresponding well-founded relation in the next sections.

Similarly to what is pointed out in [14], there exists an asymmetry in our ACL2 formulation of Higman's Lemma. Namely, the assumption of $\preceq$ being a well-quasi-order on Σ is formalized assuming the existence of a well-founded measure on finite sequences; nevertheless, we do not explicitly use this constructive characterization to state that $\preceq^*$ is a well-quasi-order on Σ^*. Instead, we establish that every infinite sequence of strings is good, the classical formulation of well-quasi-orderness. Actually, this asymmetry is only apparent: in our proof, we will show a well-founded measure on finite sequences of elements of Σ^* that constructively characterizes the well-quasi-orderness of $\preceq^*$. This measure will be an essential component of the measure used for the termination proof of `higman-indices`.

4 Patterns

Given a finite sequence of strings, we will call a string not embedding any of the strings in the sequence a *legal extension* of that sequence. The idea behind the termination proof of `higman-indices` is the following: as we consider the successive strings of an infinite sequence, the "size" of the set of legal extensions strictly decreases, and this successive decrease cannot be infinite. To formalize this intuitive idea, we will define a well-founded measure on finite sequences of strings, measuring the set of its legal extensions. Although these sets of legal extensions may be infinite, we will see how it is possible to have finite descriptions of them, using expressions that we call *patterns*.[5] In the next section, we will use these patterns as a basis for the definition of the measure.

Given a set Σ and a relation $\preceq$ on Σ, a *simple pattern* is an expression of one of the following two types:

- $\langle -, s_1, \ldots, s_n \rangle$ (with $s_i \in \Sigma$), representing the set of strings of the form $t_1 \ldots t_m$, where $t_j \in \Sigma$ and $s_i \not\preceq t_j$, $\forall i, j$. For example, in the well-quasi-order of Example 1, $\langle -, 5, 3 \rangle$ is a simple pattern and $8 \cdot 1 \cdot 6$ is one of the strings that it represents.
- $\langle s, s_1, \ldots, s_n \rangle$ (with $s, s_i \in \Sigma$), representing the set of one-length strings t, where $t \in \Sigma$, $s \preceq t$ and $s_i \not\preceq t$, $\forall i$. Again considering Example 1, $\langle 2, 8 \rangle$ is a simple pattern and the one-length string 6 is one of the strings that it represents.

A *sequential pattern* (or just a *pattern*) is an expression of the form $\pi_1 \ldots \pi_n$, where each π_i is a simple pattern; it represents the set of strings of the form $w_1 \ldots w_n$, where each w_i is in the set of strings represented by π_i. In the following, 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 $\mathcal{S}(\Pi)$ or $\mathcal{S}(\pi)$ respectively.

[5] Patterns are analogue to *sequential regular expressions* in [14].

A *set of patterns* represents the union of the sets of strings represented by each of its patterns. In the following, we will use the letter $\mathcal{P}$ to denote a set of patterns and $\mathcal{S}(\mathcal{P})$ to denote the set of strings represented by $\mathcal{P}$.

In our ACL2 formalization, we represent simple patterns of the form $\langle -, s_1, \ldots, s_n \rangle$ by the list `(nil` s_n `...` s_1`)`, and simple patterns of the form $\langle s, s_1, \ldots, s_n \rangle$ by the list `((`s`)` s_n `...` s_1`)`. 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. Note also that in order to know the type of a simple pattern π we simply check `(consp (car` π`))`. Patterns are represented by the list of their simple patterns, and finite sets of patterns by the list of their patterns. The predicates `simple-pattern-p`, `pattern-p` and `pattern-list-p` are defined to recognize, respectively, simple patterns, patterns and finite sets of patterns.

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

DEFINITION:

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

Given a pattern $\Pi = \pi_1 \ldots \pi_n$, we have just defined that a string $w \in \mathcal{S}(\Pi)$ if there exist strings $w_1, \ldots, w_n$, such that $w = w_1 \ldots w_n$ and $w_i \in \mathcal{S}(\pi_i), \forall i$. We call such $w_1 \ldots w_n$ a *decomposition* of w with respect to Π. The function `member-pattern`(w,Π) below returns a pair $\langle$`res`, `val`$\rangle$ where `res` is a boolean indicating if $w \in \mathcal{S}(\Pi)$ and, if that is the case, `val` is a decomposition `(`w_1 `...` w_n`)` witnessing it:

DEFINITION:

`member-pattern`(w,Π) =
  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))
       then 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**$\rangle$

This function needs some explanation. To check if a string w is in $\mathcal{S}(\Pi)$, we consider several cases. If w is empty, then Π has to be also empty or a concatenation of simple patterns of the form $\langle -, s_1, \ldots, s_n \rangle$ (the only ones that can represent the empty string). If w is not empty, we consider two cases depending on the first simple pattern in Π: if it has the form $\pi = \langle s, s_1, \ldots, s_n \rangle$, then the first element of w has to be in $\mathcal{S}(\pi)$ and the rest of w in the set of strings represented by the rest of Π. Otherwise, if the first element of Π has the form $\pi = \langle -, s_1, \ldots, s_n \rangle$, then we also have two cases: either w is in the set of strings represented by the rest of Π, or the first element of w is in $\mathcal{S}(\pi)$ and the rest of w is in $\mathcal{S}(\Pi)$.

As we will see, finite set of patterns are expressive enough to represent the set of legal extensions of any finite sequence of strings. But before illustrating this fact with an example, it will be helpful, for a better understanding, to show an equivalent definition of the embedding relation:

DEFINITION:
`sigma-*-<=-alt`$(w_1,w_2) \Leftrightarrow$
  if `endp`(w_1) **then t**
  elseif `endp`(w_2) **then nil**
  elseif `car`$(w_1) \preceq$ `car`(w_2)
    then `sigma-*-<=-alt`(`cdr`(w_1),`cdr`(w_2))
  else `sigma-*-<=-alt`$(w_1$,`cdr`(w_2))

This function `sigma-*-<=-alt` implements an algorithm for deciding if one string w_1 is embedded in another string w_2. Note that the idea is to search an element in w_2 greater (with respect to $\preceq$) than the first element of w_1. If such an element is not found, we return `nil`; otherwise, we proceed recursively with the rest of w_1 and with the remaining elements of w_2. We proved in ACL2 the following non-trivial result, stating that this algorithmic definition is equivalent to the declarative definition of `sigma-*-<=`:

THEOREM: `sigma-*-<=-alt-sigma-*-<=-equivalence`
  `sigma-*-<=`$(w_1,w_2) \leftrightarrow$ `sigma-*-<=-alt`(w_1,w_2)

Due to its algorithmic nature, it turns out that `sigma-*-<=-alt` is more convenient than `sigma-*-<=` for reasoning in ACL2 about the embedding relation. Indeed, our proof of Higman's lemma is done using `sigma-*-<=-alt`, although it is finally stated using `sigma-*-<=`, thanks to the above equivalence theorem. The algorithmic definition will also give us a more intuitive idea on how we can obtain patterns representing the set of legal extensions of a finite sequence of strings.

Let us see this with an example. Considering again the well-quasi-order of Example 1, let $\{f_k : k \in \mathbb{N}\}$ be an infinite sequence of strings over $\mathbb{N}$ such that $f_0 = 3 \cdot 2$, $f_1 = 1 \cdot 2$ and $f_2 = 1 \cdot 5 \cdot 3$. We are going to incrementally compute sets of patterns representing the legal extensions of the finite sequences $\langle f_0 \rangle$, $\langle f_0, f_1 \rangle$ and $\langle f_0, f_1, f_2 \rangle$, respectively. Note that initially we have the empty sequence of strings, whose set of legal extensions is Σ^*, that can be represented by the pattern $\langle - \rangle$.

Starting with $f_0 = 3 \cdot 2$, and taking into account the definition of `sigma-*-<=-alt`, it is not difficult to see that there are two possible reasons for a given string w not to embed f_0:

– There are no elements in w greater than 3. This case is described by the pattern $\Pi_1 = \langle -, 3 \rangle$.

– There is an element in w greater than 3, but after it, there are no elements greater than 2. That is, w is a string composed of three parts: an initial substring with no elements greater than 3, described by the simple pattern $\pi_1 = \langle -, 3\rangle$; a one-length string greater than 3, described by $\pi_2 = \langle 3\rangle$; and a final substring with no elements greater than 2, described by the simple pattern $\pi_3 = \langle -, 2\rangle$. Thus, the representation of this case is the pattern $\Pi_2 = \pi_1\pi_2\pi_3 = \langle -, 3\rangle\langle 3\rangle\langle -, 2\rangle$.

Let us consider now $f_1 = 1 \cdot 2$. Note that $f_1 \in \mathcal{S}(\Pi_1)$, since every element of f_1 is not greater than 3. Let us see now how we can obtain from Π_1 a set of patterns describing the set of strings $w \in \mathcal{S}(\Pi_1)$ such that $f_1 \not\preceq^* w$. We will call this operation the *reduction* of a pattern with respect to a string, and any of the patterns obtained by reducing a pattern will also be called a *reduction*. In this case, we modify Π_1 in such way that any string greater than f_1 is excluded. Again, there are two possible reasons why a string $w \in \mathcal{S}(\Pi_1)$ may fail to embed f_1:

– There are no elements in w greater than 1. Since the string is in $\mathcal{S}(\Pi_1)$ we also know that there are no elements in w greater than 3. That is, we can describe this case by the pattern $\langle -, 3, 1\rangle$.
– There is an element in w greater than 1, but after it, there are no elements greater than 2; and since the string is in $\mathcal{S}(\Pi_1)$, no element in w is greater than 3. This means that w is a string composed of three parts: an initial substring with no elements greater than 3 and no elements greater than 1, described by the simple pattern $\langle -, 3, 1\rangle$; a one-length string greater than 1, but not greater than 3, described by $\langle 1, 3\rangle$; and a final substring with no elements greater than 3 and no elements greater than 2, described by the simple pattern $\langle -, 3, 2\rangle$. Thus, the representation of this case is the pattern $\langle -, 3, 1\rangle\langle 1, 3\rangle\langle -, 3, 2\rangle$.

Now, let us consider $f_2 = 1 \cdot 5 \cdot 3$, which is a more complicated case. First, note that since $\Pi_2 = \pi_1\pi_2\pi_3 = \langle -, 3\rangle\langle 3\rangle\langle -, 2\rangle$, then $f_2 \in \mathcal{S}(\Pi_2)$. In fact, $1 \in \mathcal{S}(\pi_1)$, $5 \in \mathcal{S}(\pi_2)$ and $3 \in \mathcal{S}(\pi_3)$ is a decomposition of f_2 with respect to Π_2. In this case, for every π_i, $(i = 1, 2, 3)$, and for every reduction of π_i with respect to its corresponding substring of the decomposition, we obtain a reduction of Π_2, just replacing π_i by that reduction, and leaving the rest unchanged. That is, we obtain the following reductions:

– Pattern $\langle -, 3, 1\rangle\langle 3\rangle\langle -, 2\rangle$, obtained by replacing in Π_2 the simple pattern $\pi_1 = \langle -, 3\rangle$ by its reduction with respect to 1.
– Pattern $\langle -, 3\rangle\langle 3, 5\rangle\langle -, 2\rangle$, obtained by replacing in Π_2 the simple pattern $\pi_2 = \langle 3\rangle$ by its reduction with respect to 5.
– Pattern $\langle -, 3\rangle\langle 3\rangle\langle -, 2, 3\rangle$, obtained by replacing in Π_2 the simple pattern $\pi_3 = \langle -, 2\rangle$ by its reduction with respect to 3.

In this example, there is only one reduction for each simple pattern, because each π_i has only one reduction by its corresponding substring. But in general each simple pattern produces as many reductions as patterns obtained reducing the simple pattern by its corresponding substring in the decomposition. In particular, if the corresponding substring is the empty string, no reduction is produced by that simple pattern.

Let $w \in \mathcal{S}(\Pi_2)$ and let $w_i \in \mathcal{S}(\pi_i)$, $(i = 1, 2, 3)$, be a decomposition of w with respect to Π_2. If in addition, $f_2 \not\preceq^* w$, then $1 \not\preceq^* w_1$, or $5 \not\preceq^* w_2$ or $3 \not\preceq^* w_3$ (since

otherwise $f_2 = 1 \cdot 5 \cdot 3 \preceq^* w_1 w_2 w_3 = w$). Therefore, it is clear that w is described by some of the three reductions above.

After all the reductions carried out, we have obtained the set of patterns $\mathcal{P} = \{\langle -, 3, 1\rangle, \langle -, 3, 1\rangle\langle 1, 3\rangle\langle -, 3, 2\rangle, \langle -, 3, 1\rangle\langle 3\rangle\langle -, 2\rangle, \langle -, 3\rangle\langle 3, 5\rangle\langle -, 2\rangle, \langle -, 3\rangle\langle 3\rangle\langle -, 2, 3\rangle\}$. As we have informally justified, every legal extension w of the sequence $\langle f_0, f_1, f_2\rangle$ is in the set of strings represented by at least one of the patterns in $\mathcal{P}$.

It is worth pointing out that in general the sets of patterns obtained by these reductions may contain non-disjoint patterns or even patterns that also describe strings that are not legal extensions.[6] As we will see, this is not a problem: the only properties that we need about these sets of patterns obtained in every reduction step are that they describe every legal extension and that they strictly decrease in some well-founded sense.

Having illustrated with examples the different types of reductions of a pattern with respect to a string, let us now define the reduction operation in a precise and general way. First, we define the *reduction of a simple pattern* π with respect to a string $w \in \mathcal{S}(\pi)$:

- If $\pi = \langle s, s_1, \ldots, s_n\rangle$ then its reduction with respect to w is $\langle s, s_1, \ldots, s_n, w\rangle$. Note that in this case, necessarily $w \in \Sigma$ (it is a one-length string).
- If $\pi = \langle -, s_1, \ldots, s_n\rangle$ and $w = t_1 \ldots t_m$ (with $t_i \in \Sigma$), the reduction of π with respect to w results in the following patterns:
  - $\langle -, s_1, \ldots, s_n, t_1\rangle$
  - $\langle -, s_1, \ldots, s_n, t_1\rangle\langle t_1, s_1, \ldots, s_n\rangle\langle -, s_1, \ldots, s_n, t_2\rangle$
  - $\ldots$
  - $\langle -, s_1, \ldots, s_n, t_1\rangle\langle t_1, s_1, \ldots, s_n\rangle\langle -, s_1, \ldots, s_n, t_2\rangle \ldots \langle t_{m-1}, s_1, \ldots, s_n\rangle\langle -, s_1, \ldots, s_n, t_m\rangle$

  From the definition of `sigma-*-<=-alt`, we can see that these patterns represent the different reasons why `sigma-*-<=-alt(w,v)` may return `nil`, for a string $v \in \mathcal{S}(\pi)$. Therefore, any strings v such that $w \not\preceq v$ is described by one of these patterns. It is worth pointing out that the reductions we define in this case are a little bit simpler than the ones defined in [14].

In our ACL2 formalization, 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 to easily add 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)))
```

[6] For instance, in the example, the string $1 \cdot 3$ is in $\mathcal{S}(\langle -, 3\rangle\langle 3, 5\rangle\langle -, 2\rangle)$ and in $\mathcal{S}(\langle -, 3\rangle\langle 3\rangle\langle -, 2, 3\rangle)$; and the string $3 \cdot 5 \cdot 7$, which is greater than f_2, is described by $\mathcal{S}(\langle -, 3, 1\rangle\langle 3\rangle\langle -, 2\rangle)$.

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

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

As we have discussed in the example, the reduction of a sequential pattern depends on its components. Given a pattern $\Pi = \pi_1 \ldots \pi_n$ and a string $w \in \mathcal{S}(\Pi)$, let $w_1 \ldots w_n$ be a decomposition of w with respect to Π such that $w_i \in \mathcal{S}(\pi_i)$, $\forall i$. The *reduction of a pattern* Π with respect to w (more precisely, with respect to a given decomposition $w_1 \ldots w_n$ of w) is the set of patterns $\pi_1 \ldots \pi_{i-1} \pi'_1 \ldots \pi'_m \pi_{i+1} \ldots \pi_n$, obtained for every $1 \leq i \leq n$ and every pattern $\pi'_1 \ldots \pi'_m$ reduction of 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 (a decomposition of w):

DEFINITION:
`reduce-simple-pattern-list(`ws`,`Π`) =`
  if `endp(`Π`)` **then nil**
  else `append-list-car(reduce-simple-pattern(car(`ws`),car(`Π`)),`
                `cdr(`Π`)) @`
      `cons-list-cdr(car(`Π`),`
                `reduce-simple-pattern-list(cdr(`ws`),cdr(`Π`)))`

where the symbol @ is the "append" operation between lists, and the functions `append-list-car` and `cons-list-cdr` behaves, 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 that is not the case, the function returns the list with Π as the only element.

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

Note that since we are defining this reduction operation as an ACL2 function, we reduce with respect to one specific decomposition of w (the one given by `member--pattern`). But any other decomposition would be valid.

Finally, to reduce a set of patterns $\mathcal{P}$ with respect to a string $w \in \mathcal{S}(\mathcal{P})$, we simply replace one pattern $\Pi \in \mathcal{P}$ such that $w \in \mathcal{S}(\Pi)$, by its reductions with respect to w. The function `reduce-pattern-set` implements this operation:

DEFINITION:
`reduce-pattern-set(`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-set(`w`,cdr(`$\mathcal{P}$`)))`

Note that we reduce with respect to only one pattern, although in principle the string w may match with several patterns in the set. And again we use a specific pattern: the first pattern matched by `member-pattern`. As shown in [14], it is sufficient to reduce with respect to one of these patterns, and any of them would be valid for that purpose.

Once defined how we reduce a set of patterns, we can compute a set of patterns for every finite sequence of strings: we simply iterate the reduction over the successive strings of the sequence. The auxiliary function `reduce-pattern-sequence` implements this iteration receiving as input a finite sequence of strings $\overline{w}$ (in reverse order) and an initial set of patterns $\mathcal{P}$.

DEFINITION:

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

Using the above function, the function `pattern-sequence` computes a set of patterns for the finite sequence of strings that it receives as input. Note that it starts with the set $\{\langle - \rangle\}$ (representing Σ^*), where $\langle - \rangle$ is built by the 0-ary function `initial-pattern`:

DEFINITION:

`pattern-sequence`$(\overline{w}) =$
  `reduce-pattern-sequence`$(\overline{w}$,`list`(`initial-pattern`())$)$

As we will prove in the next section, the set of patterns computed by this function is guaranteed to represent, at least, the legal extensions of the sequence that it receives as input.

5 A Well-founded Measure and the Termination Proof

In the previous section, we have assigned a set of patterns to every finite sequence of strings. In this section, that set of patterns will be the basis to define a measure on finite sequences of strings. Once defined, we will show that there is a well-founded relation on this measure and then we will establish that this well-founded measure characterizes the well-quasi-orderness of $\preceq^*$. Finally, as a corollary, we will show termination of the function `higman-indices` presented in Section 3.

5.1 A Well-founded Measure on Finite Sequences of Strings

The main idea underlying the definition of the measure is that simple patterns can be measured by an ordinal, and that when we reduce a pattern, although the number of patterns and simple patterns may increase, nevertheless every new simple pattern that appears in a reduction has a strictly smaller measure. This is a typical situation that can be modeled using multiset relations.

A *multiset* M over a set A is a function from A to the set of natural numbers. If $M(x) > 0$ for only finitely many $x \in A$, then we say that the multiset is finite. The set of all finite multisets over A is denoted as $\mathcal{M}(A)$. Multisets are a formalization

of the intuitive idea of "sets with repeated elements", $M(x)$ being the number of occurrences of x in the multiset M. In ACL2, finite multisets will be represented simply by lists.

The measure assigned to a finite sequence will be a multiset of multisets of ordinals. First, we assign an ordinal measure to simple patterns, taking into account that $\preceq$ is a well-quasi-order on Σ and that our constructive characterization assumes the existence of an ordinal measure `sigma-seq-measure` on finite sequences of elements of Σ: if o is the measure of the sequence $\overline{s}$, then the measure of a simple pattern of the form $\langle -, \overline{s}\rangle$ is defined as $o + 1$, and the measure of a simple pattern of the form $\langle t, \overline{s}\rangle$ is o. Second, the measure of a pattern is the multiset (a list in ACL2) of the measures of the simple patterns in it. Finally, the measure of a set of patterns is the multiset (a list in ACL2) of the measures of the patterns in it. The ACL2 functions `simple-pattern-measure`, `pattern-measure` and `pattern-list-measure` (whose straightforward definitions we omit) compute respectively the measure of simple patterns, patterns and sets of patterns. The function `sigma-*-seq-measure` uses this last function and the function `pattern-sequence` presented at the end of the previous section, to define the measure of a finite sequence of strings:

DEFINITION:
`sigma-*-seq-measure`$(\overline{w}) =$
`pattern-list-measure`(`pattern-sequence`$(\overline{w})$)

For example, the following table summarizes the measures of the initial subsequences in the example discussed in the previous section. We denote multisets using double brace notation.

Initial subsequence	Set of patterns	Measure
{}	$\{\langle -\rangle\}$	$\{\{\omega \cdot 2 + 1\}\}$
$\{3 \cdot 2\}$	$\{\langle -, 3\rangle,$	$\{\{\omega + 4\}\},$
	$\langle -, 3\rangle\langle 3\rangle\langle -, 2\rangle\}$	$\{\{\omega + 4, \omega \cdot 2, \omega + 3\}\}$
$\{3 \cdot 2,$	$\{\langle -, 3, 1\rangle,$	$\{\{\omega + 2\}\},$
$1 \cdot 2\}$	$\langle -, 3, 1\rangle\langle 1, 3\rangle\langle -, 3, 2\rangle,$	$\{\{\omega + 2, \omega + 3, 6\}\},$
	$\langle -, 3\rangle\langle 3\rangle\langle -, 2\rangle\}$	$\{\{\omega + 4, \omega \cdot 2, \omega + 3\}\}$
$\{3 \cdot 2,$	$\{\langle -, 3, 1\rangle,$	$\{\{\omega + 2\}\},$
$1 \cdot 2,$	$\langle -, 3, 1\rangle\langle 1, 3\rangle\langle -, 3, 2\rangle,$	$\{\{\omega + 2, \omega + 3, 6\}\},$
$1 \cdot 5 \cdot 3\}$	$\langle -, 3, 1\rangle\langle 3\rangle\langle -, 2\rangle,$	$\{\{\omega + 2, \omega \cdot 2, \omega + 3\}\},$
	$\langle -, 3\rangle\langle 3, 5\rangle\langle -, 2\rangle,$	$\{\{\omega + 4, \omega + 5, \omega + 3\}\},$
	$\langle -, 3\rangle\langle 3\rangle\langle -, 2, 3\rangle\}$	$\{\{\omega + 4, \omega \cdot 2, 6\}\}$

Let us show a well-founded relation on the measures just defined. For that, multiset relations will be essential:

Definition 3 Given a relation $<$ on a set A, the *multiset relation* induced by $<$ on $\mathcal{M}(A)$, denoted as $<_{\mathcal{M}}$, is defined as: $N <_{\mathcal{M}} M$ 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 $y < x$.

In this definition, the subset relation and the union and difference operations have to be understood in the multiset sense, taking into account different occurrences of the same element. Intuitively, $N <_{\mathcal{M}} M$ if we can obtain N from M removing some

elements and replacing them with finitely many elements, whenever they are smaller, with respect to $<$, than some of the elements removed. The main property of multiset relations on finite multisets is that they preserve well-foundedness:

Theorem 2 (Dershowitz & Manna, [4]) *Let $<$ be a well-founded relation on a set A, and $<_{\mathcal{M}}$ the multiset relation induced by $<$ on $\mathcal{M}(A)$. Then $<_{\mathcal{M}}$ is well-founded.*

It is then clear that the multiset relation induced by the multiset relation induced by the usual order between ordinals, will be enough for our purposes. Let us see how we formalize this in ACL2.

As we said in Section 2, the only predefined well-founded relation in ACL2 is `o<`, implementing the usual order between ordinals less than ε_0. If we want to define a new well-founded relation, we have to explicitly provide a monotone ordinal function, and prove the corresponding order-preserving theorem. In our case, we should prove that the multiset relation induced by `o<` is a well-founded relation and then prove that the multiset relation induced by that relation is also well-founded. And in both cases we would have to define ordinal mappings and prove the corresponding theorems about their monotonicity.

Fortunately, we do not have to do this: we use the `defmul` tool. This macro, previously implemented and used by the authors for other formalizations (see [17]), automatically generates the definitions and the proofs of the theorems needed to introduce as a well-founded relation in ACL2 the multiset relation induced by a given well-founded relation.

In our case, we need two `defmul` calls. The first automatically generates the definition of a 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 automatically generates the definition of `mul-mul-o<`, implementing the multiset relation on finite multisets of finite multisets of ordinals (the measure of a set of patterns) 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 [17]. For simplicity, in the following we denote `mul-o<` as $<_{\varepsilon_0,\mathcal{M}}$ and `mul-mul-o<` as $<_{\varepsilon_0,\mathcal{M}_{\mathcal{M}}}$.

5.2 Well-Quasi-orderness of $\preceq^*$

We now show that `sigma-*-seq-measure` decreases with respect to $<_{\varepsilon_0,\mathcal{M}_{\mathcal{M}}}$ when we extend a finite sequence of strings with a legal extension. This will characterize the well-quasi-orderness of $\preceq^*$.

The first step is to show that the simple patterns that appear when reducing a simple pattern π with respect to $w \in \mathcal{S}(\pi)$, have a strictly smaller ordinal measure. We have two cases:

- If π is of the form $\langle s, s_1, \ldots, s_n\rangle$, then $w \in \Sigma, s \preceq w$ and $s_i \not\preceq w, \forall i$. In this case, the reduction of π with respect to w is the simple pattern $\pi' = \langle s, s_1, \ldots, s_n, w\rangle$. Note that $\neg$`exists-sigma-<=`$(\bar{s},w)$ where $\bar{s} = \langle s_1, \ldots, s_n\rangle$, and thus the well-quasi-order characterization of $\preceq$ ensures that `sigma-seq-measure(cons(`w`,`$\bar{s}$`))` is less than `sigma-seq-measure(`$\bar{s}$`)`. Therefore, the measure of π' is less than the measure of π.

– If π is of the form $\langle -, s_1, \ldots, s_n \rangle$, then $w = t_1 \ldots t_m$, with $t_j \in \Sigma$ and 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 these two types:
  - Simple patterns of the form $\pi' = \langle -, s_1, \ldots, s_n, t_j \rangle$. In this case, again the measure of π' is less than the measure of π, for reasons analogue to the previous case.
  - Simple patterns of the form $\pi' = \langle t_j, s_1, \ldots, s_n \rangle$. Then, if $\bar{s} = \langle s_1, \ldots, s_n \rangle$ and $o =$`sigma-seq-measure`$(\bar{s})$, the measure of π' is o, less than the measure of π, which is $o + 1$.

The following ACL2 lemma establishes this property about reductions of simple patterns (here $\pi' \in \Pi$ indicates that the simple pattern π' is a component of the pattern Π):

LEMMA: `reduce-simple-pattern-property`

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

To prove this lemma, we explicitly provide the following induction scheme to ACL2 by means of an induction hint (here Θ represents a generic property about its arguments):

$$\frac{\begin{array}{l}\texttt{endp}(w) \rightarrow \Theta(\Pi, \pi, \pi', w) \\ \neg\texttt{endp}(w) \wedge \texttt{consp}(\texttt{car}(\pi)) \rightarrow \Theta(\Pi, \pi, \pi', w) \\ \neg\texttt{endp}(w) \wedge \neg\texttt{consp}(\texttt{car}(\pi)) \wedge \Theta(\texttt{cdr}(\texttt{cdr}(\Pi)), \pi, \pi', \texttt{cdr}(\texttt{cdr}(w))) \rightarrow \Theta(\Pi, \pi, \pi', w)\end{array}}{\Theta(\Pi, \pi, \pi', w)}$$

As for (sequential) patterns, every reduction is obtained by removing one simple pattern and replacing it with its reductions. Therefore, from the property above and the definition of the multiset relation, it is easy to prove that the reductions of a pattern have a strictly smaller measure, as established by the following lemma:

LEMMA: `reduce-pattern-property`

$$\texttt{pattern-p}(\Pi_2) \wedge w \in \Sigma^* \wedge w \in \mathcal{S}(\Pi_2) \wedge \Pi_1 \in \texttt{reduce-pattern}(w,\Pi_2) \rightarrow \texttt{measure-pattern}(\Pi_1) <_{\varepsilon_0,\mathcal{M}} \texttt{measure-pattern}(\Pi_2)$$

Since the reduction of a set of patterns consists of removing one of the patterns of the set and replacing it by its reductions, then again by the definition of the multiset relation and as a consequence of the previous lemma we have that the measure of a set of patterns strictly decreases after reducing it:

LEMMA: `reduce-pattern-set-property`

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

To prove this lemma, the system automatically figures out the following suitable induction scheme:

$$\frac{\begin{array}{l}\texttt{endp}(\mathcal{P}) \rightarrow \Theta(\mathcal{P}, w)\\ \neg\texttt{endp}(\mathcal{P}) \wedge w \in \mathcal{S}(\texttt{car}(\mathcal{P})) \rightarrow \Theta(\mathcal{P}, w)\\ \neg\texttt{endp}(\mathcal{P}) \wedge w \notin \mathcal{S}(\texttt{car}(\mathcal{P})) \wedge \Theta(\texttt{cdr}(\mathcal{P}), w) \rightarrow \Theta(\mathcal{P}, w)\end{array}}{\Theta(\mathcal{P}, w)}$$

The heuristics of ACL2 for automatically guessing the right induction scheme for a conjecture are based on the definitions of the recursive functions in the formula. In this case, the above induction scheme is exactly the one suggested by the recursive definition of the function `reduce-pattern-set` introduced in Section 4: two base cases corresponding to the base cases of the recursion and one inductive case (with one induction hypothesis) corresponding to the only recursive call. See [8] for a more detailed explanation of the duality between recursion and the suggested induction.

Note that in all the three lemmas above, we have a condition requiring that the string with respect to which we reduce has to be in the set of strings that the pattern (or the set of patterns) describes. Thus, in our case we have to guarantee that the successive patterns we obtain from a finite sequence describe every legal extension. This is a consequence of the following lemmas. First, every string is in the initial set of patterns:

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

Second, the reduction process only "removes" strings that are not legal extensions. To stablish this property we start proving it for the function `reduce-simple-pattern`:

LEMMA: `member-pattern-list-reduce-simple-pattern`
$w \in \Sigma^* \wedge u \in \Sigma^* \wedge \texttt{simple-pattern-p}(\pi) \wedge w \in \mathcal{S}(\pi) \wedge u \in \mathcal{S}(\pi) \wedge w \not\preceq^* u$
$\rightarrow u \in \mathcal{S}(\texttt{reduce-simple-pattern}(w,\pi))$

To prove this lemma, the system automatically uses the induction scheme based on the recursive definition of `sigma-*-<=-alt` (introduced in Section 4).

Next, we state a similar property about `reduce-pattern`, automatically proved by simplification:

LEMMA: `member-pattern-list-reduce-pattern`
$w \in \Sigma^* \wedge u \in \Sigma^* \wedge \texttt{pattern-p}(\Pi) \wedge u \in \mathcal{S}(\Pi) \wedge w \not\preceq^* u$
$\rightarrow u \in \mathcal{S}(\texttt{reduce-pattern}(w,\Pi))$

Then, we stablish and prove the property about `reduce-pattern-set` (again, its recursive definition suggests a suitable induction scheme for this conjecture):

LEMMA: `member-pattern-list-reduce-pattern-set`
$w \in \Sigma^* \wedge u \in \Sigma^* \wedge \texttt{pattern-list-p}(\mathcal{P}) \wedge u \in \mathcal{S}(\mathcal{P}) \wedge w \not\preceq^* u$
$\rightarrow u \in \mathcal{S}(\texttt{reduce-pattern-set}(w,\mathcal{P}))$

Now we prove the property about the whole reduction process. More precisely, if $u \in \mathcal{S}(\mathcal{P})$ then u is still described by the set of patterns obtained after iteratively reducing $\mathcal{P}$ with respect to a given finite sequence of strings w ($w \in \Sigma^* \equiv$

sigma-⋆-seq-p($\overline{w}$)), provided that u is a legal extension of $\overline{w}$ (checked by the function exists-sigma-⋆-<=).

LEMMA: member-pattern-list-reduce-pattern-sequence
$u \in \Sigma^* \wedge \overline{w} \in \overline{\Sigma^*} \wedge$ 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($\overline{w}$,$\mathcal{P}$))

This property is proved by the system using the induction scheme based on the recursive definition of reduce-pattern-sequence.

Finally, as an easy consequence of all the above lemmas, we can establish that the measure assigned to a finite sequence of strings, strictly decreases whenever we extend the sequence with a legal extension, a property that characterizes the well-quasi-orderness of $\preceq^*$:

THEOREM: sigma-⋆-<=-well-quasi-order-characterization
$u \in \Sigma^* \wedge \overline{w} \in \overline{\Sigma^*} \wedge \neg$exists-sigma-⋆-<=($\overline{w}$,$u$)
$\rightarrow$ sigma-⋆-seq-measure(cons(u,$\overline{w}$))
$<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$ sigma-⋆-seq-measure($\overline{w}$)

5.3 Termination of Higman-Indices

From the well-quasi-orderness characterization of $\preceq^*$, we can prove the termination of the function higman-indices, our final goal in our ACL2 formalization of Higman's lemma.

The idea is simple: just assign to its input argument k the measure of the initial sequence of the first k strings, $\langle f_0, \ldots, f_{k-1} \rangle$. The following function computes the measure, where initial-segment-f(k) builds that initial subsequence:

DEFINITION:
higman-indices-measure(k) =
sigma-⋆-seq-measure(initial-segment-f(k))

As a consequence of sigma-⋆-<=-well-quasi-order-characterization, if f_k is not greater than any of $f_0 \ldots f_{k-1}$ (that is, the recursive case in the definition of higman-indices), then the measure of the argument in the recursive call of higman-indices strictly decreases with respect to the well-founded relation $<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$. That is, we have the following theorem, proved automatically by simplification:

THEOREM: higman-indices-termination-property
$k \in \mathbb{N} \wedge \neg$get-sigma-⋆-<=-f($k$,f($k$))
$\rightarrow$ higman-indices-measure($k+1$)
$<_{\varepsilon_0,\mathcal{M}_\mathcal{M}}$ 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 Section 3 is easily proved.

6 Concluding Remarks

We have presented a formalization in ACL2 of Murthy and Russell's constructive proof of Higman's lemma. That is, we show that for every infinite sequence of strings in a well-quasi-ordered alphabet we can find two strings in the sequence such that the first one is embedded in the second one. For that, we introduce the infinite sequence using the encapsulation principle and prove that a naive recursive algorithm searching for such two strings always terminates. Termination is proved by showing that there exists a well-founded measure that decreases in every recursive step of that search. This well-founded measure is mainly based on a well-founded relation on finite sequences of strings from the alphabet, defined using patterns that finitely describe the set of legal extensions. We also needed to formalize the well-founded multiset relation induced by a given well-founded relation, but that was done in a completely automated way using the `defmul` tool.

As for the proof development, we followed a standard interaction with the ACL2 theorem prover. That is, we had in mind the main lemmas and definitions needed, as suggested by the paper proof. But to obtain successful proof attempts, we needed to prove a number of additional lemmas that the prover mainly used as rewrite rules (although sometimes they are used because of an explicit user hint). Usually these additional lemmas are suggested by inspecting failed proofs. The complete development can be consulted at http://www.glc.us.es/fmartin/acl2/higman.

To quantify the proof effort, the complete formalization contains 58 definitions and 120 lemmas (with 28 non trivial proof hints explicitly given), which also 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 the 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: although it was originally developed to prove Newman's Lemma about abstract reductions [17], it was designed in a very general way such that it has turned out to be useful in other formalization tasks.

As a particular case, we also obtained an ACL2 proof of Dickson's lemma. Dickson's lemma is a particular case of Higman's lemma, where Σ is the set of natural numbers and the infinite sequences of strings considered are of the same fixed length. A proof (different from the one reported in [11]) has easily been obtained from our formalization, via functional instantiation.

Although mainly based on [14], our proof is slightly different. For example, the way we define reductions of simple patterns is not the same. Of course, 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 [14]. For example, to prove the lemma `member-pattern-list-reduce-pattern-sequence` 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 was not trivial in our formalization, because we must explicitly compute the indices from the embedding of w_3 into the substring of $w_2 w_4$, after the last position indicated by the indices from the embedding of w_1 into w_2.

As we have said in the introduction, there are several constructive mechanizations of Higman's lemma in the literature. The most recent formalizations are given in [2], using the Isabelle system, and [18], using MINLOG. These proofs are based on

Coquand and Fridlender's constructivization [3] of the Nash-Williams classical proof, restricted to the case of a two-letter alphabet Σ and using equality as the well-quasi-ordering on the alphabet (although [18] extends the proof for a finite alphabet).

Those formalizations were based on inductively defined predicates. For example, a predicate *Bar* on finite sequences of strings is defined in the following way: a finite sequence verifies *Bar* if it is good, or if for every string added to the finite sequence, the resulting finite sequence verifies *Bar*. Intuitively, $Bar(\overline{ws})$ if $\overline{ws}$ is already good or if successively adding strings will turn it into a good sequence. Therefore, Higman's lemma is formulated as $Bar(\langle\rangle)$. The proof is mainly based on inductions on derivations of inductively defined predicates. Thus, a comparison with the formalization we present here is difficult, because the source proofs are very different. Anyway, it would be an interesting challenge to formalize in the ACL2 logic the proof in [3].

We should also say that all these previous formalizations are concerned with program extraction from proofs, which it is not our case. Our approach is just the opposite: we start with a program solving the problem and then we prove its properties. This results in a more concise code: for example, our program has 18 lines of Common 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 $w_i \preceq^* w_j$, this is not the case in [2, 18].

Acknowledgements We thank the anonymous reviewers for their constructive comments and suggestions that helped to improve the paper. This work has been supported by Spanish Ministry of Science and Innovation through the project MTM2009-13842-C02-02.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Berghofer, S.: A constructive proof of Higman's lemma in Isabelle. In: Types for Proofs and Programs, TYPES'04, vol. 3085, pp. 66–82. LNCS, Springer, Berlin (2004)
3. Coquand, T., Fridlender, D.: A proof of Higman's lemma by structural induction. Unpublished draft, available at http://www.brics.dk/~daniel/texts/open.ps (1993)
4. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM **22**(8), 465–476 (1979)
5. Fridlender, D.: Higman's Lemma in Type Theory. Ph.D. thesis, University of Göteborg (1997)
6. Herbelin, H.: A program from an A-translated impredicative proof of Higman's Lemma. Available at http://coq.inria.fr/contribs/HigmanNW.html (1994)
7. Higman, G.: Ordering by divisibility in abstract algebras. Proc. Lond. Math. Soc. **3**(2), 326–336 (1952)
8. Kaufmann, M., Manolios, P., Moore, J S.: Computer-Aided Reasoning: An Approach. Kluwer Academic, Boston (2000)
9. Kaufmann, M., Moore, J S.: ACL2 Version 3.5, Homepage: http://www.cs.utexas.edu/users/moore/acl2/ (2009)
10. Kaufmann, M., Moore, J S.: Structured theory development for a mechanized logic. J. Autom. Reason. **26**(2), 161–203 (2001)
11. Martín-Mateos, F.J., Alonso, J.A., Hidalgo, M.J., Ruiz-Reina, J.L.: A formal proof of Dickson's lemma in ACL2. In: Proceedings of LPAR'03, vol. 2850, pp. 49–58. LNAI, Springer, Berlin (2003)
12. Martín-Mateos, F.J., Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J.: Proof pearl: a formal proof of Higman's lemma in ACL2. In: Proceedings of TPHOL'05, vol. 3603, pp. 358–372. LNCS, Springer, Berlin (2005)

13. Murthy, C.: Extracting Constructive Content from Classical Proofs. Ph.D. thesis, Cornell University (1990)
14. Murthy, C., Russell, J.R.: A constructive proof of Higman's lemma. In: Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 257–269 (1990)
15. Nash-Williams, C.: On well-quasi-ordering finite trees. Proc. Camb. Philos. Soc. **59**(4), 833–835 (1963)
16. Richman, F., Stolzenberg, G.: Well quasi-ordered sets. Adv. Math. **97**, 145–153 (1993)
17. Ruiz-Reina, J.L., Alonso, J.A., Hidalgo, M.J., Martín-Mateos, F.J.: Termination in ACL2 Using Multiset Relations. In: Thirty Five Years of Automating Mathematics, Kluwer Academic, Boston (2003)
18. Seisenberger, M.: On the Constructive Content of Proofs, Ph.D. thesis, Fakultät für Mathematik, Ludwig-Maximilians-Universität München (2003)
19. Seisenberger, M.: An Inductive Version of Nash-Williams' Minimal-Bad-Sequence Argument for Higman's Lemma. In: Types for Proofs and Programs, TYPES'00, vol. 2277, pp. 233–242. LNCS, Springer, Berlin (2002)
20. Simpson, S.G.: Ordinal numbers and the Hilbert basis theorem. J. Symb. Log. **53**(3), 961–974 (1988)