A constructive proof of Higman's lemma in Isabelle

Stefan Berghofer*

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
http://www.in.tum.de/~berghofe/

Abstract. Higman's lemma, a specific instance of Kruskal's theorem, is an interesting result from the area of combinatorics, which has often been used as a test case for theorem provers. We present a constructive proof of Higman's lemma in the theorem prover Isabelle, based on a paper proof by Coquand and Fridlender. Making use of Isabelle's newly-introduced infrastructure for program extraction, we show how a program can automatically be extracted from this proof, and analyze its computational behaviour.

1 Introduction

Higman's lemma [8] is an interesting problem from the field of combinatorics. It can be considered as a specific instance of Kruskal's famous tree theorem, which is useful for proving the termination of term rewriting systems using so-called *simplification orders*. Higman's lemma states that every infinite sequence of words $(w_i)_{0 \leq i < \omega}$ contains two words w_i and w_j with $i < j$ such that w_i can be *embedded* into w_j. A sequence with this property is also called *good*, otherwise *bad*. Although a quite elegant *classical* proof of this statement has been given by Nash-Williams [12] using a so-called *minimal bad sequence argument*, there has been a growing interest in obtaining *constructive* proofs of Higman's lemma recently. This is due to the additional informative content inherent in constructive proofs. For example, a termination proof of a string rewrite system based on a constructive proof of Higman's lemma could be used to obtain upper bounds on the length of reduction sequences.

The first formalization of Higman's lemma using a theorem prover was done by Murthy [10] in the Nuprl system [5]. Murthy first formalized Nash-Williams' classical proof, then translated it into a constructive proof using a *double negation translation* followed by Friedman's *A-translation* and finally extracted a program from the resulting proof. Unfortunately, although correct in principle, the program obtained in this way was so huge that it was both incomprehensible and impossible to execute within a reasonable amount of time even on the fastest computing equipment available. This rather disappointing experience prompted

* Supported by DFG Graduiertenkolleg *Logic in Computer Science*, and IST project 29001 *TYPES*

several scientists to think about direct formalizations of constructive proofs of Higman's lemma, notably Murthy and Russell [11], as well as Fridlender [7], who formalized Higman's lemma using the ALF proof editor [9] based on Martin-Löf's type theory. Fridlender's paper also gives a detailed account of the history of Higman's lemma. Murthy's classical proof was also reconsidered by Herbelin, who formalized an A-translated version of it in the Coq [2] system. Seisenberger's thesis [14, 15] contains an excellent overview of various different formalizations of Higman's lemma, most of which have been carried out with the Minlog [3] proof assistant.

A particularly elegant and short constructive proof, based entirely on inductive definitions, has been suggested by Coquand and Fridlender [6]. The rest of this paper is dedicated to a formalization of this proof in the theorem prover Isabelle. To improve on previous formalizations, the central parts of the proof are formulated using the *Isar* language for human-readable proofs due to Wenzel [17]. Moreover, thanks to the design of Isabelle's program extraction framework [4], we are also able to derive a correctness statement for the extracted program *inside* the logic. This is in contrast to most other implementations of program extraction, whose correctness if often only justified by meta-theoretic arguments on paper. Finally, to make the rather abstract exposition given by Coquand and Fridlender more easily accessible, we also present an intuitive graphical description of the computational behaviour of the extracted programs.

The rest of the paper is structured as follows: In §2, we give some basic definitions concerning sequences of words. §3 is concerned with assigning computational content to proofs involving inductive datatypes and predicates, which play a central role in the formalization. §4 describes the actual formalization in Isabelle, and §5 is devoted to an analysis of the program extracted from the proof. A conclusion is given in §6.

2 Basic definitions

We start with a few basic definitions. Words are modelled as lists of letters from the two letter alphabet[1]

datatype *letter* = *A* | *B*
types *word* = *letter list*

The empty list is denoted by [], and $x \# xs$ is infix notation for *Cons x xs*. We use $[x_1, \ldots, x_n]$ to abbreviate $x_1 \# \ldots \# []$. The embedding relation on words is defined inductively as follows:

consts *emb* :: (*word* × *word*) *set*
inductive *emb*
intros
emb0: [] ⊴ *bs*

[1] It is worth noting that the extension of the proof to an arbitrary finite alphabet is not at all trivial. For details, see Seisenberger's PhD. thesis [15].

emb1: $as \trianglelefteq bs \Longrightarrow as \trianglelefteq b \# bs$
emb2: $as \trianglelefteq bs \Longrightarrow a \# as \trianglelefteq a \# bs$

Intuitively, a word *as* can be embedded into a word *bs*, if we can obtain *as* by deleting letters from *bs*. For example, $[A,\ A] \trianglelefteq [B,\ A,\ B,\ A]$. In order to formalize the notion of a good sequence, it is useful to define the set $L\ v$ of all lists of words containing a word which can be embedded into v:

consts $L :: word \Rightarrow word\ list\ set$
inductive $L\ v$
intros
L0: $w \trianglelefteq v \Longrightarrow w \# ws \in L\ v$
L1: $ws \in L\ v \Longrightarrow w \# ws \in L\ v$

A list of words is *good* if its tail is either *good* or contains a word which can be embedded into the word occurring at the head position of the list:

consts $good :: word\ list\ set$
inductive *good*
intros
good0: $ws \in L\ w \Longrightarrow w \# ws \in good$
good1: $ws \in good \Longrightarrow w \# ws \in good$

In contrast to Coquand [6], who defines *Cons* such that it appends elements to the right of the list, we use the usual definition of *Cons*, which appends elements to the left. Therefore, the predicates on lists of words defined in this section, such as the *good* predicate introduced above work "in the opposite direction", e.g. $[[A,\ A],\ [A,\ B],\ [B]] \in good$, since $[B] \trianglelefteq [A,\ B]$. In order to express the fact that every infinite sequence is good, we define a predicate *bar* as follows:

consts $bar :: word\ list\ set$
inductive *bar*
intros
bar1: $ws \in good \Longrightarrow ws \in bar$
bar2: $(\bigwedge w.\ w \# ws \in bar) \Longrightarrow ws \in bar$

Intuitively, $ws \in bar$ means that either the list of words *ws* is already *good*, or successively adding words will turn it into a good list. Consequently, $[] \in bar$ means that every infinite sequence $(w_i)_{0 \leq i < \omega}$ must be good, i.e. have a prefix $w_0\ \ldots\ w_n$ with $[w_n,\ \ldots,\ w_0] \in good$, since by successively adding words $w_0,\ w_1,\ \ldots$ to the empty list, we must eventually arrive at a list which is good. Note that the above definition of *bar* is closely related to Brouwer's more general principle of *bar induction* [16, Chapter 4, §8]. Like the accessible part of a relation, the definition of *bar* embodies a kind of well-foundedness principle.

3 Computational content of inductive datatypes and predicates

The main proof principles used in the proof of Higman's lemma are induction on datatypes and induction on the derivation of inductive predicates (or sets). In

order to extract a program from this proof, it is therefore important to investigate which programs correspond to these proof principles.

3.1 Motivation

For *inductive datatypes*, things are rather straightforward: A proof by *induction* on a datatype gives rise to a program defined by *recursion* on the very same datatype. The concept of an *inductive predicate* is quite similar to that of an inductive datatype[2]: While a datatype is characterized by a list of *constructors* together with their types, an inductive predicate is characterized by a list of *introduction rules*. Consequently, the program extracted from a proof by induction on the derivation of an inductive predicate should be a recursive function, too, where the recursion runs over a datatype which encodes the derivation. This datatype can be derived from the introduction rules in a canonical way. Each introduction rule φ corresponds to a constructor of type $\mathsf{tyof}\ \varphi$, where tyof is a *type extraction* function mapping a logical formula to the type of the program extracted from its proof. For the fragment of Isabelle comprising implication ($\Longrightarrow$) and universal quantification ($\bigwedge$), which is used to express the introduction rules, tyof is defined by

$$\begin{array}{lcl} \mathsf{tyof}\ (\bigwedge x :: \alpha.\ \varphi) & \equiv & \alpha \Rightarrow \mathsf{tyof}\ \varphi \\ \mathsf{tyof}\ (\psi \Longrightarrow \varphi) & \equiv & \mathsf{tyof}\ \psi \Rightarrow \mathsf{tyof}\ \varphi \\ \mathsf{tyof}\ (P\ \overline{t}) & \equiv & \alpha_P \end{array}$$

where φ and ψ are *computationally relevant* formulae, and P is a computationally relevant predicate variable. Every such predicate variable P is uniquely associated with a type variable α_P. For a full definition of tyof, the interested reader is referred to [4]. For an inductive predicate such as *bar*, we set $\mathsf{tyof}\ (ws \in bar) \equiv barT$, where $barT$ is a new datatype to be defined inductively. The correspondence between proof rules for inductive predicates and programs, which we have sketched above, is illustrated in Fig. 1. Intuitively, the datatype $barT$ representing the computational content of $ws \in bar$ is an infinitely branching tree from which one can read off words that, when appended to the sequence of words *ws*, turn it into a *good* sequence. The branches of this tree are labelled with words. For each appended word *w*, one moves one step closer to the leaves of the tree, following the branch labelled with *w*. When a leaf, i.e. the constructor *bar1* is reached, the resulting sequence of words must be *good*. An example for such a tree is shown in Fig. 2.

In order to reason about the correctness of programs extracted from proofs involving the *bar* predicate, we need to describe under what conditions an element of $barT$ properly represents a derivation of a formula $ws \in bar$. This connection

[2] It should be noted that this insight is the essence of expressive *type theories* based on inductive types such as the *Calculus of Inductive Constructions* [13], where these two concepts actually coincide due to the identification of propositions and types. However, this is not the case for Isabelle/HOL, which treats propositions and types as different concepts.

Proof	**Program**
induction on datatype *list-induct* : $P\ [] \Longrightarrow$ $(\bigwedge x\ xs.\ P\ xs \Longrightarrow P\ (x\ \#\ xs)) \Longrightarrow P\ ys$	recursion on datatype *list-rec* : $\alpha\ list \Rightarrow \alpha_P \Rightarrow$ $(\alpha \Rightarrow \alpha\ list \Rightarrow \alpha_P \Rightarrow \alpha_P) \Rightarrow \alpha_P$
inductive predicate introduction rules **inductive** *bar* *bar1* : $\bigwedge ws.\ ws \in good \Longrightarrow ws \in bar$ *bar2* : $\bigwedge ws.\ (\bigwedge w.\ w\ \#\ ws \in bar) \Longrightarrow$ $ws \in bar$	inductive datatype constructors **datatype** *barT* = *bar1* (*word list*) | *bar2* (*word list*) (*word* $\Rightarrow$ *barT*)
induction on derivation *bar-induct* : $vs \in bar \Longrightarrow$ $(\bigwedge ws.\ ws \in good \Longrightarrow P\ ws) \Longrightarrow$ $(\bigwedge ws.\ (\bigwedge w.\ w\ \#\ ws \in bar) \Longrightarrow$ $(\bigwedge w.\ P\ (w\ \#\ ws)) \Longrightarrow P\ ws) \Longrightarrow$ $P\ vs$	recursion on datatype *barT-rec* : *barT* $\Rightarrow$ (*word list* $\Rightarrow \alpha_P$) $\Rightarrow$ (*word list* $\Rightarrow$ (*word* $\Rightarrow$ *barT*) $\Rightarrow$ (*word* $\Rightarrow \alpha_P$) $\Rightarrow \alpha_P$) $\Rightarrow$ α_P

Fig. 1. Computational content of inductive datatypes and predicates

between datatypes (or programs) and formulae can be captured by the concept of *modified realizability* due to Kleene and Kreisel. We write $\mathsf{realizes}\ p\ \varphi$ to mean that "program p *realizes* formula φ" or, more intuitively, "p *satisfies* specification φ". For the $\Longrightarrow/\bigwedge$-fragment of Isabelle, realizes is defined by the equations

$$\begin{aligned}
&\mathsf{realizes}\ p\ (\textstyle\bigwedge x.\ \varphi) &&\equiv \textstyle\bigwedge x.\ \mathsf{realizes}\ (p\ x)\ \varphi\\
&\mathsf{realizes}\ p\ (\psi \Longrightarrow \varphi) &&\equiv \textstyle\bigwedge q.\ \mathsf{realizes}\ q\ \psi \Longrightarrow \mathsf{realizes}\ (p\ q)\ \varphi\\
&\mathsf{realizes}\ p\ (P\ \bar{t}) &&\equiv P^R\ p\ \bar{t}
\end{aligned}$$

where again φ and ψ are computationally relevant formulae, and P is a computationally relevant predicate variable. Each predicate variable P with n arguments is uniquely associated with a predicate variable P^R with $n+1$ arguments. For the inductive predicate *bar*, we set $\mathsf{realizes}\ b\ (ws \in bar) \equiv (b,\ ws) \in barR$, where *barR* is a new inductive predicate characterized by the introduction rules

$$\begin{aligned}
&ws \in good \Longrightarrow (bar1\ ws,\ ws) \in barR\\
&(\textstyle\bigwedge w.\ (f\ w,\ w\ \#\ ws) \in barR) \Longrightarrow (bar2\ ws\ f,\ ws) \in barR
\end{aligned}$$

which express that the constructors of the datatype *barT* realize the introduction rules of the predicate *bar* in the above sense. As a consequence, this means that if $(bar2\ ws\ f_0,\ ws) \in barR$ and

$$\begin{aligned}
&f_0\ w_0 = bar2\ (w_0\ \#\ ws)\ f_1, \quad f_1\ w_1 = bar2\ (w_1\ \#\ w_0\ \#\ ws)\ f_2, \quad \ldots,\\
&f_{n-1}\ w_{n-1} = bar2\ (w_{n-1}\ \#\ \cdots\ \#\ w_0\ \#\ ws)\ f_n,\\
&f_n\ w_n = bar1\ (w_n\ \#\ \cdots\ \#\ w_0\ \#\ ws)
\end{aligned}$$

then $(w_n\ \#\ \cdots\ \#\ w_0\ \#\ ws) \in good$. Note that this need not necessarily be the shortest possible *good* sequence. The induction principle for the *bar* predicate, which is shown in Fig. 1, is realized by the recursion combinator

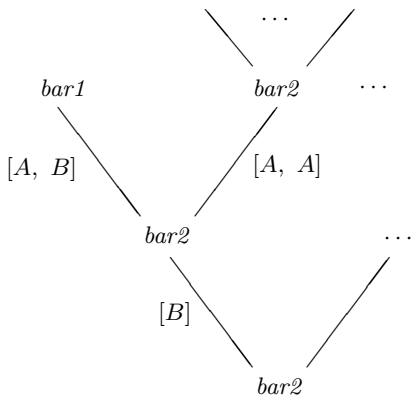


Fig. 2. Computational content of *bar*

$$barT\text{-}rec\ f\ g\ (bar1\ list) = f\ list$$
$$barT\text{-}rec\ f\ g\ (bar2\ list\ fun) = g\ list\ fun\ (\lambda x.\ barT\text{-}rec\ f\ g\ (fun\ x))$$

The corresponding correctness theorem for this realizer is

$$\begin{array}{l}(b,\ vs) \in barR \Longrightarrow\\ (\bigwedge ws.\ ws \in good \Longrightarrow P^R\ (f\ ws)\ ws) \Longrightarrow\\ (\bigwedge ws\ x.\ (\bigwedge w.\ (x\ w,\ w\ \#\ ws) \in barR) \Longrightarrow\\ \qquad (\bigwedge xa.\ (\bigwedge w.\ P^R\ (xa\ w)\ (w\ \#\ ws)) \Longrightarrow P^R\ (g\ ws\ x\ xa)\ ws)) \Longrightarrow\\ P^R\ (barT\text{-}rec\ f\ g\ b)\ vs\end{array}$$

which is easily proved by induction on the derivation of *barR*.

3.2 General scheme

We will now generalize what we have just explained by an example. Consider the general definition of an inductive set (or predicate) S

inductive S

$$\begin{array}{l}I_1 : \bigwedge \overline{x_1} :: \overline{\tau_1}.\ \varphi_1^1 \Longrightarrow \cdots \Longrightarrow \varphi_1^{s_1} \Longrightarrow (\bigwedge \overline{z_1^1} :: \overline{\sigma_1^1}.\ \overline{\psi_1^1} \Longrightarrow \overline{t_1^1} \in S) \Longrightarrow \cdots \Longrightarrow\\ \quad (\bigwedge \overline{z_1^{r_1}} :: \overline{\sigma_1^{r_1}}.\ \overline{\psi_1^{r_1}} \Longrightarrow \overline{t_1^{r_1}} \in S) \Longrightarrow \overline{u_1} \in S\\ \quad \vdots\\ I_n : \bigwedge \overline{x_n} :: \overline{\tau_n}.\ \varphi_n^1 \Longrightarrow \cdots \Longrightarrow \varphi_n^{s_n} \Longrightarrow (\bigwedge \overline{z_n^1} :: \overline{\sigma_n^1}.\ \overline{\psi_n^1} \Longrightarrow \overline{t_n^1} \in S) \Longrightarrow \cdots \Longrightarrow\\ \quad (\bigwedge \overline{z_n^{r_n}} :: \overline{\sigma_n^{r_n}}.\ \overline{\psi_n^{r_n}} \Longrightarrow \overline{t_n^{r_n}} \in S) \Longrightarrow \overline{u_n} \in S\end{array}$$

where φ_i^j are *non-recursive* premises (also called *side conditions*), i.e. do not contain S. The *recursive* premises have the form

$$\bigwedge \overline{z_i^j} :: \overline{\sigma_i^j}.\ \overline{\psi_i^j} \Longrightarrow \overline{u_i} \in S$$

where S does not occur in $\overline{\psi_i^j}$, i.e. the occurrence of the recursive set is only *strictly positive*.

Induction The rule for induction on the derivation of S has the form

$$\mathcal{I}_1 \Longrightarrow \cdots \Longrightarrow \mathcal{I}_n \Longrightarrow \overline{x} \in S \Longrightarrow P\ \overline{x}$$

where

$$\begin{array}{rcl}\mathcal{I}_i & = & \bigwedge \overline{x_i}.\ \varphi_i^1 \Longrightarrow \cdots \Longrightarrow \varphi_i^{s_i} \Longrightarrow \\ & & (\bigwedge \overline{z_i^1} :: \overline{\sigma_i^1}.\ \overline{\psi_i^1} \Longrightarrow \overline{t_i^1} \in S) \Longrightarrow \cdots \Longrightarrow (\bigwedge \overline{z_i^{r_i}} :: \sigma_i^{r_i}.\ \overline{\psi_i^{r_i}} \Longrightarrow \overline{t_i^{r_i}} \in S) \Longrightarrow \\ & & (\bigwedge \overline{z_i^1} :: \overline{\sigma_i^1}.\ \overline{\psi_i^1} \Longrightarrow P\ \overline{t_i^1}) \Longrightarrow \cdots \Longrightarrow (\bigwedge \overline{z_i^{r_i}} :: \sigma_i^{r_i}.\ \overline{\psi_i^{r_i}} \Longrightarrow P\ \overline{t_i^{r_i}}) \Longrightarrow P\ \overline{u_i}\end{array}$$

Computational content of derivations The datatype S^T representing the computational content of the derivation of S is defined by

$$\begin{array}{l}\textbf{datatype}\ S^T\ = \\ \quad I_1\ \overline{\tau_1}\ (\mathsf{tyof}\ \varphi_n^1) \cdots (\mathsf{tyof}\ \varphi_1^{s_1})\ (\overline{\sigma_1^1} \Rightarrow \mathsf{tyof}\ \overline{\psi_1^1} \Rightarrow S^T) \cdots (\overline{\sigma_1^{r_1}} \Rightarrow \mathsf{tyof}\ \overline{\psi_1^{r_1}} \Rightarrow S^T) \\ \mid \cdots \\ \mid I_n\ \overline{\tau_n}\ (\mathsf{tyof}\ \varphi_n^1) \cdots (\mathsf{tyof}\ \varphi_n^{s_n})\ (\overline{\sigma_n^1} \Rightarrow \mathsf{tyof}\ \overline{\psi_n^1} \Rightarrow S^T) \cdots (\overline{\sigma_n^{r_n}} \Rightarrow \mathsf{tyof}\ \overline{\psi_n^{r_n}} \Rightarrow S^T)\end{array}$$

Realizability predicate The predicate S^R, which establishes a connection between elements of the datatype S^T and propositions of the form $\overline{x} \in S$ is defined inductively by the introduction rules

$$\begin{array}{l}I_i^R = \bigwedge \overline{x_i}\ p_i^1 \ldots p_i^{s_i}\ f_i^1 \ldots f_i^{r_i}.\ (\mathsf{realizes}\ p_i^1\ \varphi_i^1) \Longrightarrow \cdots \Longrightarrow (\mathsf{realizes}\ p_i^{s_i}\ \varphi_i^{s_i}) \Longrightarrow \\ \quad (\bigwedge \overline{z_i^1}\ \overline{q_i^1}.\ \mathsf{realizes}\ \overline{q_i^1}\ \overline{\psi_i^1} \Longrightarrow (f_i^1\ \overline{z_i^1}\ \overline{q_i^1},\ \overline{t_i^1}) \in S^R) \Longrightarrow \cdots \Longrightarrow \\ \quad (\bigwedge \overline{z_i^{r_i}}\ \overline{q_i^{r_i}}.\ \mathsf{realizes}\ \overline{q_i^{r_i}}\ \overline{\psi_i^{r_i}} \Longrightarrow (f_i^{r_i}\ \overline{z_i^{r_i}}\ \overline{q_i^{r_i}},\ \overline{t_i^{r_i}}) \in S^R) \Longrightarrow \\ \quad\ (I_i\ \overline{x_i}\ p_i^1 \ldots p_i^{s_i}\ f_i^1 \ldots f_i^{r_i},\ \overline{u_i}) \in S^R\end{array}$$

Computational content of induction principle The above rule for induction on the derivation of the set S is realized by the recursion combinator S^T-*rec* for the datatype S^T, which is characterized by the equations

$$\begin{array}{l}S^T\text{-}rec\ g_1 \cdots g_n\ (I_i\ p_i^1 \cdots p_i^{s_i}\ f_i^1 \cdots f_i^{r_i}) = g_i\ p_i^1 \cdots p_i^{s_i}\ f_i^1 \cdots f_i^{r_i} \\ \quad (\lambda \overline{z_i^1}\ \overline{q_i^1}.\ S^T\text{-}rec\ g_1 \cdots g_n\ (f_i^1\ \overline{z_i^1}\ \overline{q_i^1})) \ \cdots\ (\lambda \overline{z_i^{r_i}}\ \overline{q_i^{r_i}}.\ S^T\text{-}rec\ g_1 \cdots g_n\ (f_i^{r_i}\ \overline{z_i^{r_i}}\ \overline{q_i^{r_i}}))\end{array}$$

The fact that this recursion combinator correctly realizes the principle of induction on the derivation d can be expressed by

$$(d,\ \overline{x}) \in S^R \Longrightarrow \mathcal{R}_1 \Longrightarrow \cdots \Longrightarrow \mathcal{R}_n \Longrightarrow P^R\ (S^T\text{-}rec\ g_1 \cdots g_n\ d)\ \overline{x}$$

where

$$\begin{array}{l}\mathcal{R}_i = \bigwedge \overline{x_i}\ p_i^1 \ldots p_i^{s_i}\ f_i^1 \ldots f_i^{r_i}.\ (\mathsf{realizes}\ p_i^1\ \varphi_i^1) \Longrightarrow \cdots \Longrightarrow (\mathsf{realizes}\ p_i^{s_i}\ \varphi_i^{s_i}) \Longrightarrow \\ \quad (\bigwedge \overline{z_i^1}\ \overline{q_i^1}.\ \mathsf{realizes}\ \overline{q_i^1}\ \overline{\psi_i^1} \Longrightarrow (f_i^1\ \overline{z_i^1}\ \overline{q_i^1},\ \overline{t_i^1}) \in S^R) \Longrightarrow \cdots \Longrightarrow \\ \quad (\bigwedge \overline{z_i^{r_i}}\ \overline{q_i^{r_i}}.\ \mathsf{realizes}\ \overline{q_i^{r_i}}\ \overline{\psi_i^{r_i}} \Longrightarrow (f_i^{r_i}\ \overline{z_i^{r_i}}\ \overline{q_i^{r_i}},\ \overline{t_i^{r_i}}) \in S^R) \Longrightarrow \\ \quad (\bigwedge \overline{z_i^1}\ \overline{q_i^1}.\ \mathsf{realizes}\ \overline{q_i^1}\ \overline{\psi_i^1} \Longrightarrow P^R\ (f_i^1\ \overline{z_i^1}\ \overline{q_i^1})\ \overline{t_i^1}) \Longrightarrow \cdots \Longrightarrow \\ \quad (\bigwedge \overline{z_i^{r_i}}\ \overline{q_i^{r_i}}.\ \mathsf{realizes}\ \overline{q_i^{r_i}}\ \overline{\psi_i^{r_i}} \Longrightarrow P^R\ (f_i^{r_i}\ \overline{z_i^{r_i}}\ \overline{q_i^{r_i}})\ \overline{t_i^{r_i}}) \Longrightarrow \\ \quad\ P^R\ (g_i\ \overline{x_i}\ p_i^1 \ldots p_i^{s_i}\ f_i^1 \ldots f_i^{r_i})\ \overline{u_i}\end{array}$$

and P^R is a new predicate variable uniquely associated with the predicate variable P in the above induction rule for S. This correctness statement for S^T-*rec* can be proved by induction on the derivation of $(d,\ \overline{x}) \in S^R$.

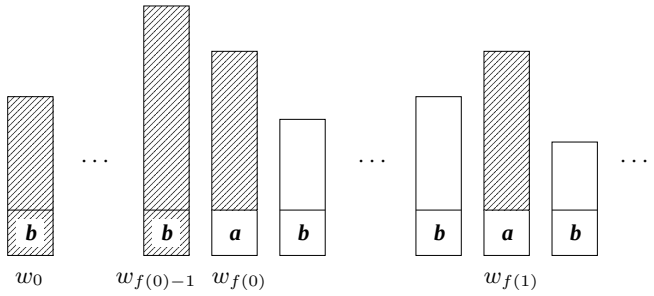


Fig. 3. Minimal bad sequence argument

4 Formalizing Higman's lemma

Before explaining the actual proof, we will briefly sketch the main idea of Nash-Williams' classical proof[3], since Coquand's proof can be considered as a constructive version of it. In order to show that every infinite sequence is good, we assume there is a bad sequence and use this to derive a contradiction. If there is a bad sequence, we may also construct a bad sequence $(w_i)_{0\leq i<\omega}$ which is minimal wrt. word length[4]. Since any infinite sequence containing the empty word is necessarily good, each w_i must have the form $a_i \# v_i$. We can find a strictly monotone function f and a letter $a \in \{A, B\}$ such that $a_{f(i)} = a$ for all i. Now consider the sequence $(v_{f(i)})_{0\leq i<\omega}$. If this sequence was bad, we could construct the sequence

$$s \;=\; w_0 \;\ldots\; w_{f(0)-1}\; v_{f(0)}\; v_{f(1)} \;\ldots$$

Because the length of $v_{f(0)}$ is smaller than the length of $w_{f(0)}$, and $(w_i)_{0\leq i<\omega}$ is minimal, this sequence must be good. For this to be possible, there must be i and j with $i < f(0)$ and $w_i \trianglelefteq v_{f(j)}$, because both $(w_i)_{0\leq i<\omega}$ and $(v_{f(i)})_{0\leq i<\omega}$ are bad. However, since $v_{f(j)} \trianglelefteq w_{f(j)}$, this implies that $w_i \trianglelefteq w_{f(j)}$, which contradicts the assumption that $(w_i)_{0\leq i<\omega}$ is bad. Hence $(v_{f(i)})_{0\leq i<\omega}$ must be good, which means that there are i and j with $i < j$ and $v_{f(i)} \trianglelefteq v_{f(j)}$, which implies that $a \# v_{f(i)} \trianglelefteq a \# v_{f(j)}$ and therefore $w_{f(i)} \trianglelefteq w_{f(j)}$, which again contradicts the assumption that $(w_i)_{0\leq i<\omega}$ is bad.

To capture the idea underlying the construction of the sequence s shown above, we introduce a relation T, where $(vs,\ ws) \in T\ a$ means that vs is obtained from ws by first copying the prefix of words starting with the letter b, where $a \neq b$, and then appending the tails of words starting with a. This construction principle is illustrated in Fig. 3, where the shaded parts correspond to the sequence s above. In order to define T, we also introduce an auxiliary relation R, where $(vs,\ ws) \in R\ a$ means that ws can be obtained from vs by

[3] A more general version of this proof for Kruskal's theorem can e.g. be found in the textbook by Baader and Nipkow [1].

[4] A sequence $(w_i)_{0\leq i<\omega}$ is *smaller* than a sequence $(v_i)_{0\leq i<\omega}$ wrt. word length, iff there is a k such that $w_j = v_j$ for all $j < k$ and $\mathit{length}(w_k) < \mathit{length}(v_k)$.

prefixing each word with the letter a. It should be noted that we could as well have defined $T\ a$ as a function which, given a list ws, yields a list vs. However, we found the relational formulation more convenient to work with.

consts $R :: letter \Rightarrow (word\ list \times word\ list)\ set$
inductive $R\ a$
intros
$R0$: $([], []) \in R\ a$
$R1$: $(vs, ws) \in R\ a \Longrightarrow (w \# vs, (a \# w) \# ws) \in R\ a$

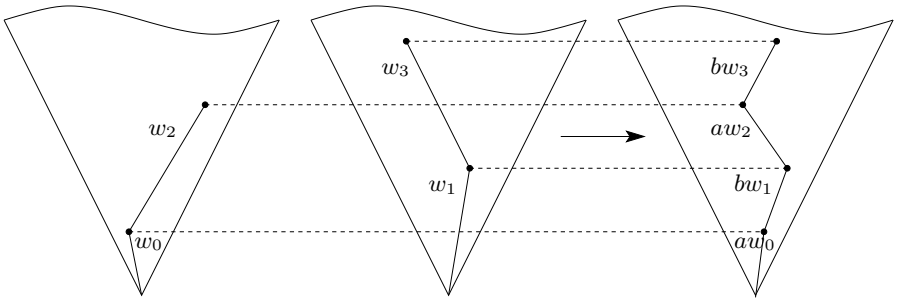
consts $T :: letter \Rightarrow (word\ list \times word\ list)\ set$
inductive $T\ a$
intros
$T0$: $a \neq b \Longrightarrow (vs, ws) \in R\ b \Longrightarrow (w \# ws, (a \# w) \# ws) \in T\ a$
$T1$: $(vs, ws) \in T\ a \Longrightarrow (v \# vs, (a \# v) \# ws) \in T\ a$
$T2$: $a \neq b \Longrightarrow (vs, ws) \in T\ a \Longrightarrow (vs, (b \# w) \# ws) \in T\ a$

The proof of Higman's lemma is divided into several parts, namely *prop1*, *prop2* and *prop3*. From the computational point of view, these theorems can be thought of as functions transforming trees. Theorem *prop1* states that each sequence ending with the empty word satisfies predicate *bar*, since it can trivially be extended to a *good* sequence by appending any word. This easily follows from the introduction rules for *bar*:

theorem *prop1*: $([] \# ws) \in bar$ **by** *rules*

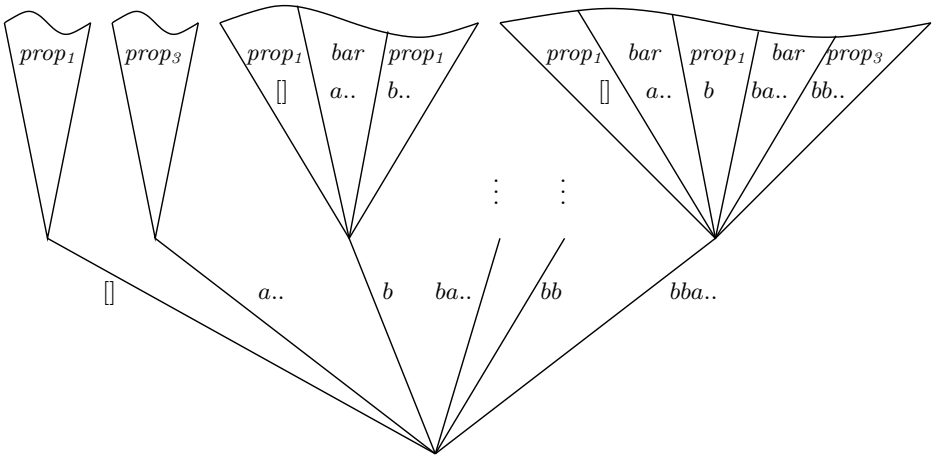
The intuition behind *prop2*, which is shown in Fig. 4, is a bit harder to grasp. Given two trees encoding proofs of $xs \in bar$ and $ys \in bar$, we produce a new tree encoding a proof of $zs \in bar$ by *interleaving* the two input trees. In order to demonstrate that $zs \in bar$, we need to show that, given a sequence of words, we can detect if appending this sequence to zs yields a *good* sequence. This is done by inspecting each word in the sequence to be appended. If the word has the form $a \# w$, we move one step ahead in the tree witnessing $xs \in bar$, whereas we move one step ahead in the tree witnessing $ys \in bar$ if it has the form $b \# w$. Whenever we reach a leaf in one of these trees, we can be sure that, due to the additional constraints on xs, ys and zs, we have turned zs into a good sequence. If the word to be appended is just the empty word $[]$, we know by *prop1* that any following word will make the sequence good. The proof of *prop2* is by double induction on the derivation of $xs \in bar$ and $ys \in bar$ (yielding the induction hypotheses I and I'), followed by a case analysis on the word w to be appended to the sequence zs.

Theorem *prop3* states that we can turn a proof of $xs \in bar$ into a proof of $zs \in bar$, where zs is the list obtained by prefixing each word in the (nonempty) list xs with the letter a. The proof together with its corresponding tree is shown in Fig. 5. Note that the subtrees of this tree (reachable via edges labelled with words w) are interleavings of other trees formed using *prop2*. In order to prove $zs \in bar$, we again consider all possible words w to be appended to zs. There are essentially two different cases which may occur:



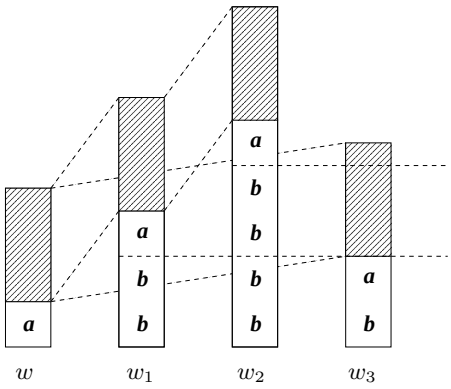
theorem *prop2*:
assumes *ab*: $a \neq b$ **and** *bar*: $xs \in bar$
shows $\bigwedge ys\ zs.\ ys \in bar \Longrightarrow (xs,\ zs) \in T\ a \Longrightarrow (ys,\ zs) \in T\ b \Longrightarrow zs \in bar$
using *bar*
proof *induct*
fix *xs zs* **assume** $xs \in good$ **and** $(xs,\ zs) \in T\ a$
show $zs \in bar$ **by** (*rule bar1*) (*rule lemma3*)
next
fix *xs ys* **assume** *I*: $\bigwedge w\ ys\ zs.\ ys \in bar \Longrightarrow (w\ \#\ xs,\ zs) \in T\ a \Longrightarrow$
$(ys,\ zs) \in T\ b \Longrightarrow zs \in bar$
assume $ys \in bar$ **thus** $\bigwedge zs.\ (xs,\ zs) \in T\ a \Longrightarrow (ys,\ zs) \in T\ b \Longrightarrow zs \in bar$
proof *induct*
fix *ys zs* **assume** $ys \in good$ **and** $(ys,\ zs) \in T\ b$
show $zs \in bar$ **by** (*rule bar1*) (*rule lemma3*)
next
fix *ys zs* **assume** *I'*: $\bigwedge w\ zs.\ (xs,\ zs) \in T\ a \Longrightarrow (w\ \#\ ys,\ zs) \in T\ b \Longrightarrow zs \in bar$
and *ys*: $\bigwedge w.\ w\ \#\ ys \in bar$ **and** *Ta*: $(xs,\ zs) \in T\ a$ **and** *Tb*: $(ys,\ zs) \in T\ b$
show $zs \in bar$
proof (*rule bar2*)
fix *w* **show** $w\ \#\ zs \in bar$
proof (*cases w*)
case *Nil* **thus** *?thesis* **by** *simp* (*rule prop1*)
next
case (*Cons c cs*) **from** *letter-eq-dec* **show** *?thesis*
proof
assume *ca*: $c = a$
from *ab* **have** $(a\ \#\ cs)\ \#\ zs \in bar$ **by** (*rules intro*: *I ys Ta Tb*)
thus *?thesis* **by** (*simp add*: *Cons ca*)
next
assume $c \neq a$ **with** *ab* **have** *cb*: $c = b$ **by** (*rule letter-neq*)
from *ab* **have** $(b\ \#\ cs)\ \#\ zs \in bar$ **by** (*rules intro*: *I' Ta Tb*)
thus *?thesis* **by** (*simp add*: *Cons cb*)
qed
qed
qed
qed
qed

Fig. 4. Proposition 2




```
theorem prop3:
  assumes bar: xs ∈ bar
  shows ⋀zs. xs ≠ [] ⟹ (xs, zs) ∈ R a ⟹ zs ∈ bar using bar
proof induct
  fix xs zs assume xs ∈ good and (xs, zs) ∈ R a
  show zs ∈ bar by (rule bar1) (rule lemma2)
next
  fix xs zs
  assume I: ⋀w zs. w # xs ≠ [] ⟹ (w # xs, zs) ∈ R a ⟹ zs ∈ bar
  and xsb: ⋀w. w # xs ∈ bar and xsn: xs ≠ [] and R: (xs, zs) ∈ R a
  show zs ∈ bar
  proof (rule bar2)
    fix w
    show w # zs ∈ bar
    proof (induct w)
      case Nil
      show ?case by (rule prop1)
    next
      case (Cons c cs)
      from letter-eq-dec show ?case
      proof
        assume c = a
        thus ?thesis by (rules intro: I [simplified] R)
      next
        from R xsn have T: (xs, zs) ∈ T a by (rule lemma4)
        assume c ≠ a
        thus ?thesis by (rules intro: prop2 Cons xsb xsn R T)
      qed
    qed
  qed
qed
```

Fig. 5. Proposition 3




```
theorem higman: [] ∈ bar
proof (rule bar2)
  fix w
  show [w] ∈ bar
  proof (induct w)
    show [[]] ∈ bar by (rule prop1)
  next
    fix c cs assume [cs] ∈ bar
    thus [c # cs] ∈ bar
      by (rule prop3) (simp, rules)
  qed
qed
```

Fig. 6. Main theorem

1. If w consists only of b's, i.e. $w = b^n$ for $0 \leq n$, appending words of the form $b^n..$ or b^m with $m < n$ to the sequence $w \# zs$ will lead to a *good* sequence due to *prop1*, whereas appending words of the form $b^m a..$ with $m < n$ will lead to a *good* sequence due to the fact that $xs \in bar$. The subtrees named *bar* in Fig. 5 correspond to witnesses of this fact.
2. Similarly, if w contains the letter a, i.e. $w = b^n a..$ with $0 \leq n$, appending words of the form $b^n..$ to the sequence $w \# zs$ can be shown to lead to a *good* sequence by appealing to the induction hypothesis. Computationally, this corresponds to a recursive call in the function producing the tree, which is why the corresponding subtrees in Fig. 5 are named *prop3*. Appending words of the form b^m or $b^m a..$ with $m < n$ can be shown to lead to a good sequence by exactly the same argument as in case 1.

The proof of *prop3* is by induction on the derivation of $xs \in bar$, followed by an induction on the word w combined with a case analysis on letters.

We can now put together the pieces and prove the main theorem. In order to prove that $[] \in bar$, it suffices to show that $[w] \in bar$ for any word w. This can be proved by induction on w. If w is empty, the claim follows by *prop1*. Otherwise, if $w = c \# cs$, we have $[cs] \in bar$ by induction hypothesis, which we can turn into a proof of $[c \# cs] \in bar$ using *prop3*. It should be noted that structural induction on lists can be viewed as the constructive counterpart of the minimality argument used in Nash-Williams' classical proof.

The proof, together with a diagram illustrating the intuition behind it, is shown in Fig. 6. The shaded parts of the drawing correspond to sequences for which we already know that they are *good* due to the induction hypothesis $[cs] \in bar$. Processing the word w_1 in Fig. 6 corresponds to following the branch labelled with $bba..$ in Fig. 5. Processing the word w_2 in Fig. 6, which starts with at least as many b's as the preceeding word w_1, corresponds to a step in the part of the rightmost subtree in Fig. 5, which was produced by a recursive call to *prop3*. In contrast, processing the word w_3, which starts with fewer b's than w_1, corresponds to a step in the part of the rightmost subtree labelled with *bar*.

5 Analyzing the computational content

The computational content of the theorem shown in Fig. 6 is an infinitely branching tree, which is a bit difficult to inspect. Using this theorem, we therefore prove an additional statement yielding a program that, given an infinite sequence of words, returns a finite prefix of this sequence which is *good*. Infinite sequences are encoded as functions of type $nat \Rightarrow \alpha$. The fact that a list is a prefix of an infinite sequence can be characterized recursively as follows:

```
consts is-prefix :: 'a list ⇒ (nat ⇒ 'a) ⇒ bool
primrec
  is-prefix [] f = True
  is-prefix (x # xs) f = (x = f (length xs) ∧ is-prefix xs f)
```

We now prove that an infinite sequence f of words has a good prefix *vs*, provided there is a prefix *ws* with $ws \in bar$. The proof is by induction on the derivation of $ws \in bar$. If the derivation tree is a leaf, this means that the current prefix is already good and we simply return it, otherwise we move ahead one step in the tree and continue the search recursively, i.e. apply the induction hypothesis.

```
theorem good-prefix-lemma:
  assumes bar: ws ∈ bar
  shows is-prefix ws f ⟹ ∃ vs. is-prefix vs f ∧ vs ∈ good using bar
proof induct
  case bar1
  thus ?case by rules
next
  case (bar2 ws)
  have is-prefix (f (length ws) # ws) f by simp
  thus ?case by (rules intro: bar2)
qed
```

The fact that any infinite sequence has a good prefix can now be obtained as a corollary of this theorem using *higman*:

```
theorem good-prefix: ∃ vs. is-prefix vs f ∧ vs ∈ good
  using higman
  by (rule good-prefix-lemma) simp+
```

As has already been noted, the function extracted from theorem *good-prefix* need not necessarily find the *shortest* good prefix. As an example, consider the following three functions representing sequences of words:

i	0	1	2	3	4	$\cdots$
$f_1(i)$	$[A,\ A]$	$[B]$	$[A,\ B]$	$[]$	$[]$	$\cdots$
$f_2(i)$	$[A,\ A]$	$[B]$	$[B,\ A]$	$[]$	$[]$	$\cdots$
$f_3(i)$	$[A,\ A]$	$[B]$	$[A,\ B,\ A]$	$[]$	$[]$	$\cdots$

When applied to f_1, *good-prefix* returns the good prefix $[[], [], [A, B], [B], [A, A]]$, which is certainly not the shortest one. The reason for this should become clear

```
letter-eq-dec ≡
λx xa.
  case x of A ⇒ case xa of A ⇒ Left | B ⇒ Right
  | B ⇒ case xa of A ⇒ Right | B ⇒ Left

prop1 ≡ λx. bar2 ([] # x) (λw. bar1 (w # [] # x))

prop2 ≡
λx xa xb xc H Ha.
  barT-rec (λws x xa H. bar1 xa)
   (λws xb r xc xd H.
      barT-rec (λws x. bar1 x)
       (λws xb ra xc.
          bar2 xc
           (λw. case w of [] ⇒ prop1 xc
                 | a # list ⇒
                    case letter-eq-dec a x of
                    Left ⇒ r list ws ((x # list) # xc) (bar2 ws xb)
                    | Right ⇒ ra list ((xa # list) # xc)))
       H xd)
   H xb xc Ha

prop3 ≡
λx xa H.
  barT-rec (λws. bar1)
   (λws x r xb.
      bar2 xb
       (list-rec (prop1 xb)
         (λa list H.
            case letter-eq-dec a xa of Left ⇒ r list ((xa # list) # xb)
            | Right ⇒ prop2 a xa ws ((a # list) # xb) H (bar2 ws x))))
   H x

higman ≡ bar2 [] (list-rec (prop1 []) (λa list. prop3 [a # list] a))

good-prefix-lemma ≡ λx. barT-rec (λws. ws) (λws xa r. r (x (length ws)))

good-prefix ≡ λx. good-prefix-lemma x higman
```

Fig. 7. Program extracted from the proof of Higman's lemma

when looking at Fig. 6: In order for the algorithm to recognize that the word $[B]$ can be embedded into some subsequent word, this word has to start with at least one B. However, since the following word starts with an A, the algorithm does not recognize that $[B]$ can be embedded into it. In contrast, when applied to f_2 and f_3, *good-prefix* returns the shortest good prefixes $[[B, A], [B], [A, A]]$ and $[[A, B, A], [B], [A, A]]$, as expected. In the case of f_2, the algorithm recognizes that $[B]$ can be embedded into $[B, A]$, since the latter starts with as many B's as the former. In the case of f_3, the algorithm recognizes that $[A]$ can be embedded into $[B, A]$, and hence, due to lemma *prop3*, also recognizes that $[A, A]$ can be embedded into $[A, B, A]$.

The Isabelle/HOL functions extracted from the proof of theorem *good-prefix* are shown in Fig. 7. The corresponding ML code, together with auxiliary functions, is given in Appendix A. The correctness theorem for *good-prefix* is

$$\mathit{is\text{-}prefix}\ (\mathit{good\text{-}prefix}\ f)\ f \wedge \mathit{good\text{-}prefix}\ f \in \mathit{good}$$

whereas for *higman*, it is simply $(\mathit{higman}, []) \in \mathit{barR}$. The correctness theorems for *prop2* and *prop3* are

$$\begin{array}{l} a \neq b \Longrightarrow \\ (\bigwedge x.\ (x, xs) \in \mathit{barR} \Longrightarrow \\ \quad (\bigwedge xa.\ (xa, ys) \in \mathit{barR} \Longrightarrow \\ \quad\quad (xs, zs) \in T\ a \Longrightarrow (ys, zs) \in T\ b \Longrightarrow (\mathit{prop2}\ a\ b\ ys\ zs\ x\ xa, zs) \in \mathit{barR})) \end{array}$$

and

$$\bigwedge x.\ (x, xs) \in \mathit{barR} \Longrightarrow xs \neq [] \Longrightarrow (xs, zs) \in R\ a \Longrightarrow (\mathit{prop3}\ zs\ a\ x, zs) \in \mathit{barR}$$

Note that of the inductive predicates defined in this section, only *bar* has a computational content. If we were not just interested in a *good* prefix, but also in the exact positions of the two words which can be embedded into each other, we would also have to assign the predicate *good* a computational content.

6 Conclusion

By formalizing Higman's lemma, we have demonstrated that Isabelle's program extraction module is capable of handling realistic examples. The formalization is rather compact and consists of only 280 lines of Isabelle definitions and proof scripts. The automatically extracted program turns out to be quite readable. Its ML version is about 70 lines in length (including auxiliary functions), and performs reasonably well on medium-size sequences. For example, a sequence of 350 words with an average length of 20 letters can be processed in 1.44 seconds on a Pentium III with 1 GHz.

Acknowledgement

I would like to sincerely thank Monika Seisenberger for suggesting this case study and for her help in doing the formalization. Thanks are also due to Tobias Nipkow and Tjark Weber for reading draft versions of this paper and suggesting numerous improvements.

References

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

[2] B. Barras et al. The Coq proof assistant reference manual – version 7.2. Technical Report 0255, INRIA, February 2002.

[3] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series*, pages 41–71. Kluwer Academic Publishers, Dordrecht, 1998.

[4] S. Berghofer. Program Extraction in simply-typed Higher Order Logic. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[5] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[6] T. Coquand and D. Fridlender. A proof of Higman's lemma by structural induction. Unpublished draft, available at http://www.math.chalmers.se/~frito/Papers/open.ps.gz, November 1993.

[7] D. Fridlender. Higman's lemma in type theory. In E. Giménez and C. Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 112–133. Springer-Verlag, 1998.

[8] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(2):326–336, 1952.

[9] L. Magnusson. *The Implementation of ALF—a Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. Phd thesis, Dept. of Computing Science, Chalmers Univ. of Technology and Univ. of Göteborg, 1994.

[10] C. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Cornell University, 1990.

[11] C. R. Murthy and J. R. Russell. A constructive proof of Higman's lemma. In J. C. Mitchell, editor, *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 257–269, Philadelphia, PA, June 1990. IEEE Computer Society Press.

[12] C. Nash-Williams. On well-quasi-ordering finite trees. *Proceedings of the Cambridge Philosophical Society*, 59(4):833–835, 1963.

[13] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[14] M. Seisenberger. Konstruktive Aspekte von Higmans Lemma. Master's thesis, Fakultät für Mathematik, Ludwig-Maximilians-Universität München, 1998.

[15] M. Seisenberger. *On the Constructive Content of Proofs*. PhD thesis, Fakultät für Mathematik, Ludwig-Maximilians-Universität München, 2003.

[16] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics, Volume 1*, volume 121 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1988.

[17] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, TU München, 2002. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.

A ML code generated from proof of Higman's lemma

```
datatype letter = A | B;

datatype nat = id0 | Suc of nat;

datatype barT = bar1 of letter list list
  | bar2 of letter list list * (letter list -> barT);

fun barT_rec f1 f2 (bar1 list) = f1 list
  | barT_rec f1 f2 (bar2 (list, funa)) = f2 list funa (fn x => barT_rec f1 f2 (funa x));

fun op__43_def0 id0 n = n
  | op__43_def0 (Suc m) n = op__43_def0 m (Suc n);

fun size_def3 [] = id0
  | size_def3 (a :: list) = op__43_def0 (size_def3 list) (Suc id0);

fun good_prefix_lemma x =
  (fn H => barT_rec (fn ws => ws) (fn ws => fn xa => fn r => r (x (size_def3 ws))) H);

fun list_rec f1 f2 [] = f1
  | list_rec f1 f2 (a :: list) = f2 a list (list_rec f1 f2 list);

datatype sumbool = Left | Right;

fun letter_eq_dec x =
  (fn xa =>
    (case x of A => (case xa of A => Left | B => Right)
      | B => (case xa of A => Right | B => Left)));

fun prop1 x = bar2 (([] :: x), (fn w => bar1 (w :: ([] :: x))));

fun prop2 x =
  (fn xa => fn xb => fn xc => fn H => fn Ha =>
    barT_rec (fn ws => fn x => fn xa => fn H => bar1 xa)
      (fn ws => fn xb => fn r => fn xc => fn xd => fn H =>
        barT_rec (fn ws => fn x => bar1 x)
          (fn ws => fn xb => fn ra => fn xc =>
            bar2 (xc, (fn w =>
                        (case w of [] => prop1 xc
                          | (xd :: xe) =>
                              (case letter_eq_dec xd x of
                                Left => r xe ws ((x :: xe) :: xc) (bar2 (ws, xb))
                                | Right => ra xe ((xa :: xe) :: xc))))))
          H xd)
      H xb xc Ha);

fun prop3 x =
  (fn xa => fn H =>
    barT_rec (fn ws => fn x => bar1 x)
      (fn ws => fn x => fn r => fn xb =>
        bar2 (xb, (fn w =>
                    list_rec (prop1 xb)
                      (fn a => fn list => fn H =>
                        (case letter_eq_dec a xa of Left => r list ((xa :: list) :: xb)
                          | Right => prop2 a xa ws ((a :: list) :: xb) H (bar2 (ws, x))))
                      w)))
      H x);

val higman : barT =
  bar2 ([], (fn w =>
              list_rec (prop1 [])
                (fn a => fn list => fn H => prop3 ((a :: list) :: []) a H) w));

fun good_prefix x = good_prefix_lemma x higman;
```