A proof of Higman's lemma by structural induction

Thierry Coquand, Daniel Fridlender
Chalmers University

November 1993

Introduction

Consider words of a two letter alphabet $\{0, 1\}$. Write $x \leq y$ for x is a subword of y, that is x is of the form $i_1 \ldots i_p$ and y of the form $j_1 \ldots j_q$ and there exists $l_1 < l_2 \ldots < l_p$ such that $j_{l_k} = i_k$ for all k. Higman's lemma [7] says that for any infinite sequence $x_1, x_2, \ldots$ of such binary words, there exists $p < q$ such that $x_p \leq x_q$.

Nash-Williams has discovered a beautiful argument for proving this lemma [11].

We present here a proof by structural induction of Higman's lemma; this proof can be seen as a constructive version of Nash-Williams' proof[1].

The proof illustrates the use of inductively defined relations and proofs by rule inductions (or inductions on the structure of proofs). All objects in this proof are presented inductively, and proofs are done by the so-called "proof induction" or "rule induction", that is by induction over the structure of an hypothetical proof[2].

The power and elegance of this kind of arguments has been recognized in computing science [2, 5]. This paper gives an example of such an inductive proof for a combinatorial problem. While there exist other constructive proofs of Higman's lemma (see for instance [10, 14]), the present argument has been recorded for its extreme formal simplicity.

This simplicity allows us to give a complete description of the computational content of the proof, first in term of a functional program, which follows closely the structure of the proof, and then in term of a program with state. The second program has an intuitive algorithmic meaning. In order to show that these two programs are equivalent, we introduce an intermediary program, which is a first-order operational interpretation of the functional program. The relation between this program and the program with state is simple to establish. We can thus claim that we understand completely the computational behaviour of the proof.

It is possible to give still another description of this algorithm, in term of process computing in parallel. In this form, the connection with Nash-Williams non constructive argument is quite clear (though this algorithm was found first only as an alternative description of the computational content of the inductive proof). This inductive proof was actually found from the usual non constructive argument by using the technique described in [3]. These two facts give strong indication that this algorithm can be considered as the computational content of the Nash-Williams argument.

[1] See [10] for a general discussion on the problem of finding a constructive proof of Higman's lemma.

[2] The first systematic presentation of such an induction rule seems to be [4], under the name of "deduction induction." The importance of this proof method for computing science is stressed in [2].

1 Formalisation

The words on a two letters alphabet, denoted by $x, y, \ldots$ are introduced by:

$$\frac{}{0 : \mathsf{Bin}} \qquad \frac{x : \mathsf{Bin}}{S_0(x) : \mathsf{Bin}} \qquad \frac{x : \mathsf{Bin}}{S_1(x) : \mathsf{Bin}}$$

Finite sequences of such words, denoted by $A, B, C, \ldots$ are introduced by:

$$\frac{}{\mathsf{Nil} : \mathsf{Seq}} \qquad \frac{A : \mathsf{Seq} \quad x : \mathsf{Bin}}{Ax : \mathsf{Seq}}$$

We write $[x]$ for the finite sequence $\mathsf{Nil}x$ which consists of only one word x.

We introduce now a relation $x \leq y$ between words, a relation $\mathsf{L}(A, y)$ between finite sequences and words, and two predicates $\mathsf{Good}(A)$ and $\mathsf{Bar}(A)$ on finite sequences. They are inductively defined by the rules:

$$\frac{}{0 \leq x} \qquad \frac{x \leq y}{x \leq S_0(y)} \qquad \frac{x \leq y}{x \leq S_1(y)} \qquad \frac{x \leq y}{S_0(x) \leq S_0(y)} \qquad \frac{x \leq y}{S_1(x) \leq S_1(y)}$$

$$\frac{x \leq y}{\mathsf{L}(Ax, y)} \qquad \frac{\mathsf{L}(A, y)}{\mathsf{L}(Ax, y)}$$

$$\frac{\mathsf{L}(A, x)}{\mathsf{Good}(Ax)} \qquad \frac{\mathsf{Good}(A)}{\mathsf{Good}(Ax)}$$

$$\frac{\mathsf{Good}(A)}{\mathsf{Bar}(A)} \qquad \frac{(x : \mathsf{Bin})\mathsf{Bar}(Ax)}{\mathsf{Bar}(A)}$$

$\mathsf{Good}(A)$ means that A is a sequence $x_1 \ldots x_n$ of binary words such that there exists $p < q$ satisfying $x_p \leq x_q$.

In the last rule, we use the notation $(x : \mathsf{Bin})\phi(x)$ for expressing universal quantification over Bin. Alternatively, this rule can be thought of as having infinitely many premisses $\mathsf{Bar}(Ax)$ for each $x \in \mathsf{Bin}$. A proof of $\mathsf{Bar}(A)$ can be thought of as a well-founded tree, possibly infinitely branching.

Higman's lemma is formulated inductively as $\mathsf{Bar}(\mathsf{Nil})$. This follows the usual definition of the "eventuality" operator in modal logic (see for instance [8]): $\mathsf{Bar}(\mathsf{Nil})$ expresses that any "growing" sequence of binary words $x_1 \rightarrow x_1x_2 \rightarrow x_1x_2x_3 \ldots$ is eventually good.

2 A proof by structural induction

We introduce four binary relations $\mathsf{R}_0, \mathsf{R}_1, \mathsf{T}_0$ and T_1 between finite sequences of words.

$$\frac{}{\mathsf{R}_0(\mathsf{Nil}, \mathsf{Nil})} \qquad \frac{\mathsf{R}_0(A, C)}{\mathsf{R}_0(Ax, CS_0(x))}$$

$$\frac{}{\mathsf{R}_1(\mathsf{Nil}, \mathsf{Nil})} \qquad \frac{\mathsf{R}_1(A, C)}{\mathsf{R}_1(Ax, CS_1(x))}$$

$$\frac{\mathsf{R}_1(B,C)}{\mathsf{T}_0(Cx, CS_0(x))} \qquad \frac{\mathsf{T}_0(A,C)}{\mathsf{T}_0(Ax, CS_0(x))} \qquad \frac{\mathsf{T}_0(A,C)}{\mathsf{T}_0(A, CS_1(x))}$$

$$\frac{\mathsf{R}_0(A,C)}{\mathsf{T}_1(Cx, CS_1(x))} \qquad \frac{\mathsf{T}_1(B,C)}{\mathsf{T}_1(B, CS_0(x))} \qquad \frac{\mathsf{T}_1(B,C)}{\mathsf{T}_1(Bx, CS_1(x))}$$

Alternatively, $\mathsf{R}_0(A,C)$ holds iff C is of the form $S_0(x_1)\dots S_0(x_n)$, with $A = x_1 \dots x_n$ and $n \geq 0$. And $\mathsf{T}_0(A,C)$ holds iff $C = y_1 \dots y_n$ is of length $n > 0$, all y_p are different from 0, so that we can write $y_p = S_{i_p}(x_p)$, and $A = y_1 \dots y_{m-1} x_{l_1} \dots x_{l_k}$, where $m = l_1 < l_2 < \dots < l_k$ is the set of indexes $p \leq n$ such that $i_p = 0$. The relations R_1 and T_1 have a similar alternative description.

Proposition 1: *For all finite sequences A of words,* $\mathsf{Bar}(A0)$.

Proof: Let x be a binary word. We have directly $\mathsf{L}(A0, x)$ because $0 \leq x$, and hence $\mathsf{Good}(A0x)$. Hence, $\mathsf{Bar}(A0x)$ for all $x : \mathsf{Bin}$ and $\mathsf{Bar}(A0)$. □

The following assertions are proved by a direct induction.

Lemma: *The following implications hold*

- *If* $\mathsf{L}(A,x)$, *then* $\mathsf{L}(A, S_0(x))$ *and* $\mathsf{L}(A, S_1(x))$,
- *If* $\mathsf{R}_0(A,C)$ *and* $\mathsf{Good}(A)$, *then* $\mathsf{Good}(C)$,
- *If* $\mathsf{T}_1(A,C)$ *and* $\mathsf{Good}(A)$, *then* $\mathsf{Good}(C)$,
- *If* $\mathsf{R}_1(B,C)$ *and* $\mathsf{Good}(B)$, *then* $\mathsf{Good}(C)$,
- *If* $\mathsf{T}_0(B,C)$ *and* $\mathsf{Good}(B)$, *then* $\mathsf{Good}(C)$,
- *If* $\mathsf{R}_0(A,C)$, *then* $\mathsf{T}_0(Ax, CS_0(x))$,
- *If* $\mathsf{R}_1(B,C)$, *then* $\mathsf{T}_1(Bx, CS_1(x))$.

The last two assertions can be reformulated as follows: if $\mathsf{R}_0(A,C)$ and A is not Nil, then $\mathsf{T}_0(A,C)$, and, similarly, if $\mathsf{R}_1(B,C)$ and B is not Nil, then $\mathsf{T}_1(B,C)$.

Proposition 2: *If* $\mathsf{T}_0(A,C)$, $\mathsf{T}_1(B,C)$, $\mathsf{Bar}(A)$ *and* $\mathsf{Bar}(B)$, *then* $\mathsf{Bar}(C)$.

Proof: By double induction on the proofs of $\mathsf{Bar}(A)$ and $\mathsf{Bar}(B)$. If $\mathsf{Good}(A)$ or $\mathsf{Good}(B)$, then the proposition follows directly from the lemma. Otherwise, $\mathsf{Bar}(Ax)$ and $\mathsf{Bar}(Bx)$ for all x. In this case, we prove that $\mathsf{Bar}(Cx)$ holds for every x by case on x :

1. if x is 0, then $\mathsf{Bar}(Cx)$ follows from proposition 1,
2. if x is $S_0(y)$, then we have $\mathsf{Bar}(Ay)$, $\mathsf{Bar}(B)$, $\mathsf{T}_0(Ay, Cx)$ and $\mathsf{T}_1(B, Cx)$, hence $\mathsf{Bar}(Cx)$ by induction hypothesis,
3. if x is $S_1(y)$, then we have $\mathsf{Bar}(A)$, $\mathsf{Bar}(By)$, $\mathsf{T}_0(A, Cx)$ and $\mathsf{T}_1(By, Cx)$, hence $\mathsf{Bar}(Cx)$ by induction hypothesis. □

Proposition 3: *If* $\mathsf{R}_0(A,C)$ *and* $\mathsf{Bar}(A)$, *then* $\mathsf{Bar}(C)$.

Proof: If A is Nil, then so is C and the proposition holds directly. We can thus suppose that A is not Nil. Notice then that $\mathsf{R}_0(A,C)$ implies $\mathsf{T}_0(A,C)$ by the lemma.

The proposition is then proved by induction on the proof of $\mathsf{Bar}(A)$. If $\mathsf{Good}(A)$, then the proposition follows from the lemma. Otherwise, we have $\mathsf{Bar}(Ax)$ for all x and we prove $\mathsf{Bar}(Cx)$ by induction on x :

1. if x is 0, then $\mathsf{Bar}(Cx)$ follows from proposition 1,
2. if x is $S_0(y)$, then we have $\mathsf{Bar}(Ay)$ and $\mathsf{R}_0(Ay, Cx)$, hence $\mathsf{Bar}(Cx)$ by induction hypothesis (on the proof of $\mathsf{Bar}(A)$),
3. if x is $S_1(y)$, then we have $\mathsf{Bar}(A)$ and $\mathsf{Bar}(Cy)$ by induction hypothesis on x. Furthermore, $\mathsf{T}_0(A, Cx)$, because we have supposed that A is not Nil, and $\mathsf{T}_1(Cy, Cx)$, directly by the definition of T_1, hence $\mathsf{Bar}(Cx)$ by proposition 2. □

Proposition 4: *If* $\mathsf{R}_1(B, C)$ *and* $\mathsf{Bar}(B)$, *then* $\mathsf{Bar}(C)$.

Proof: Similar to the one of proposition 3. □

Proposition 5: *For all* x, $\mathsf{Bar}([x])$.

Proof: By induction on x, using proposition 1 for the base case, and propositions 3 and 4 in the inductive case, noticing that $\mathsf{R}_0([x], [S_0(x)])$ and $\mathsf{R}_1([x], [S_1(x)])$. □

Theorem: $\mathsf{Bar}(\mathsf{Nil})$.

Proof: Direct by proposition 5. □

Notice, as a simple corollary, that if $(n_1, m_1), (n_2, m_2), \ldots$ is a sequence of pair of integers, then there exists $i < j$ such that both $n_i \leq n_j$ and $m_i \leq m_j$. This follows from Higman's lemma by taking the ith word to be $S_1^{n_i}(S_0^{m_i}(0))$.

3 Computational analysis of the proof

We want to derive a program which expresses the computational behaviour of the proof. Our computational model will be a lazy language with *one* global exception mechanism (like in lml [1]). We call *raise* this exception. It is a polymorphic constant, and in particular is of type the empty type Void.

A functional program

First we introduce a type containing the computationally relevant information of $\mathsf{Bar}(A)$

$$\mathrm{BAR} = \mathsf{good} \ + \ \mathsf{ind}\ (\mathsf{Bin} \to \mathrm{BAR})$$

The dependency on A is removed. The type BAR has two constructors, one for each introduction in the definition of $\mathsf{Bar}(A)$. The constructor ind clearly corresponds to the second introduction when the dependency is removed. The constructor good corresponds to the first introduction. It is of arity 0 because we are interested in the behaviour of the program and not in the result[3].

$$\begin{aligned} \mathit{prop1} \ &: \ \mathrm{BAR} \\ \mathit{prop1} \ &= \ \mathsf{ind}\ (\lambda x.\mathsf{good}) \end{aligned}$$

$$\mathit{prop2} \ : \ \mathrm{BAR} \to \mathrm{BAR} \to \mathrm{BAR}$$

[3] To get a program which for an infinite sequence of words $x_1, \ldots$ gives back i and j such that x_i is a subword of x_j, one needs a more informative constructor, which we do not include in order to simplify the description.

$$
\begin{array}{llll}
\mathit{prop2} & \mathsf{good} & b & = \mathsf{good} \\
\mathit{prop2} & (\mathsf{ind}\ f) & \mathsf{good} & = \mathsf{good} \\
\mathit{prop2} & (\mathsf{ind}\ f) & (\mathsf{ind}\ g) & = \mathsf{ind}\ h \\
 & & & \quad \text{where} \quad \begin{array}{lcl} h\ \ 0 & = & \mathit{prop1} \\ h\ (S_0\ y) & = & \mathit{prop2}\ \ (f\ y)\ \ (\mathsf{ind}\ g) \\ h\ (S_1\ y) & = & \mathit{prop2}\ \ (\mathsf{ind}\ f)\ \ (g\ y) \end{array}
\end{array}
$$

$$\mathit{prop3} \ : \ \mathrm{BAR} \to \mathrm{BAR}$$

$$
\begin{array}{lll}
\mathit{prop3} & \mathsf{good} & = \mathsf{good} \\
\mathit{prop3} & (\mathsf{ind}\ f) & = \mathsf{ind}\ h \\
 & & \quad \text{where} \quad \begin{array}{lcl} h\ \ 0 & = & \mathit{prop1} \\ h\ (S_0\ y) & = & \mathit{prop3}\ \ (f\ y) \\ h\ (S_1\ y) & = & \mathit{prop2}\ \ (\mathsf{ind}\ f)\ \ (h\ y) \end{array}
\end{array}
$$

$$\mathit{prop4} \ : \ \mathrm{BAR} \to \mathrm{BAR}$$

$$
\begin{array}{lll}
\mathit{prop4} & \mathsf{good} & = \mathsf{good} \\
\mathit{prop4} & (\mathsf{ind}\ f) & = \mathsf{ind}\ h \\
 & & \quad \text{where} \quad \begin{array}{lcl} h\ \ 0 & = & \mathit{prop1} \\ h\ (S_0\ y) & = & \mathit{prop2}\ \ (h\ y)\ \ (\mathsf{ind}\ f) \\ h\ (S_1\ y) & = & \mathit{prop4}\ \ (f\ y) \end{array}
\end{array}
$$

$$\mathit{theorem} \ : \ \mathrm{BAR}$$

$$
\begin{array}{ll}
\mathit{theorem} & = \mathsf{ind}\ h \\
 & \quad \text{where} \quad \begin{array}{lcl} h\ \ 0 & = & \mathit{prop1} \\ h\ (S_0\ x) & = & \mathit{prop3}\ \ (h\ x) \\ h\ (S_1\ x) & = & \mathit{prop4}\ \ (h\ x) \end{array}
\end{array}
$$

This program has the same structure as the proof, and it seems clear that their computational behaviour are the same. To understand now this computational behaviour, we analyze how the program executes. This leads us to the operational semantics of this program.

We define a type EXPR as follows:

$$
\begin{array}{lcl}
\mathrm{EXPR} & = & \mathsf{prop1} \\
 & & +\ \mathsf{prop2}\ \mathrm{EXPR}\ \mathrm{EXPR} \\
 & & +\ \mathsf{prop3}\ \mathrm{EXPR} \\
 & & +\ \mathsf{prop4}\ \mathrm{EXPR} \\
 & & +\ \mathsf{theorem}
\end{array}
$$

We now define the function

$$\mathit{step} \ : \ \mathrm{EXPR} \to \mathsf{Bin} \to \mathrm{EXPR}$$

As can be seen in the program above, **good** is always propagated. Thus, for the sake of simplicity, we can simply raise an exception when a **good** result is generated without altering the computational behaviour.

$$
\begin{array}{llcll}
step & \mathsf{prop1} & x & = & raise \\[4pt]
step & (\mathsf{prop2}\ s\ t) & 0 & = & \mathsf{prop1} \\
step & (\mathsf{prop2}\ s\ t) & (S_0\ y) & = & \mathsf{prop2}\ (step\ s\ y)\ t \\
step & (\mathsf{prop2}\ s\ t) & (S_1\ y) & = & \mathsf{prop2}\ s\ (step\ t\ y) \\[4pt]
step & (\mathsf{prop3}\ s) & 0 & = & \mathsf{prop1} \\
step & (\mathsf{prop3}\ s) & (S_0\ y) & = & \mathsf{prop3}\ (step\ s\ y) \\
step & (\mathsf{prop3}\ s) & (S_1\ y) & = & \mathsf{prop2}\ s\ (step\ (\mathsf{prop3}\ s)\ y) \\[4pt]
step & (\mathsf{prop4}\ s) & 0 & = & \mathsf{prop1} \\
step & (\mathsf{prop4}\ s) & (S_0\ y) & = & \mathsf{prop2}\ (step\ (\mathsf{prop4}\ s)\ y)\ s \\
step & (\mathsf{prop4}\ s) & (S_1\ y) & = & \mathsf{prop4}\ (step\ s\ y) \\[4pt]
step & \mathsf{theorem} & 0 & = & \mathsf{prop1} \\
step & \mathsf{theorem} & (S_0\ x) & = & \mathsf{prop3}\ (step\ \mathsf{theorem}\ x) \\
step & \mathsf{theorem} & (S_1\ x) & = & \mathsf{prop4}\ (step\ \mathsf{theorem}\ x)
\end{array}
$$

A program with state

Now, we can code the type EXPR in the last program with binary trees.

$$
\begin{array}{rcl}
\mathrm{TREE} & = & \mathsf{end} \\
 & & +\ \mathsf{empty} \\
 & & +\ \mathsf{node}\ \mathrm{TREE}\ \mathrm{TREE}
\end{array}
$$

and we have the translation $\mathcal{T}$ from EXPR to TREE.

$$
\begin{array}{rcl}
\mathcal{T}(\mathsf{prop1}) & = & \mathsf{end} \\
\mathcal{T}(\mathsf{prop2}\ s\ t) & = & \mathsf{node}\ (\mathcal{T}(s), \mathcal{T}(t)) \\
\mathcal{T}(\mathsf{prop3}\ s) & = & \mathsf{node}\ (\mathcal{T}(s), \mathsf{empty}) \\
\mathcal{T}(\mathsf{prop4}\ s) & = & \mathsf{node}\ (\mathsf{empty}, \mathcal{T}(s)) \\
\mathcal{T}(\mathsf{theorem}) & = & \mathsf{empty}
\end{array}
$$

The program becomes[4]

$$
insert\ :\ \mathrm{TREE} \to \mathsf{Bin} \to \mathrm{TREE}
$$

$$
\begin{array}{lrrcl}
insert & \mathsf{end} & x & = & raise \\
insert & s & 0 & = & \mathsf{end} \\
insert & (\mathsf{node}\ s\ \mathsf{empty}) & (S_0\ y) & = & \mathsf{node}\ (insert\ s\ y)\ \mathsf{empty} \\
insert & (\mathsf{node}\ s\ \mathsf{empty}) & (S_1\ y) & = & \mathsf{node}\ s\ (insert\ (\mathsf{node}\ s\ \mathsf{empty})\ y) \\
insert & (\mathsf{node}\ \mathsf{empty}\ s) & (S_0\ y) & = & \mathsf{node}\ (insert\ (\mathsf{node}\ \mathsf{empty}\ s)\ y)\ s \\
insert & (\mathsf{node}\ \mathsf{empty}\ s) & (S_1\ y) & = & \mathsf{node}\ \mathsf{empty}\ (insert\ s\ y) \\
insert & (\mathsf{node}\ s\ t) & (S_0\ y) & = & \mathsf{node}\ (insert\ s\ y)\ t \\
insert & (\mathsf{node}\ s\ t) & (S_1\ y) & = & \mathsf{node}\ s\ (insert\ t\ y) \\
insert & \mathsf{empty} & (S_0\ x) & = & \mathsf{node}\ (insert\ \mathsf{empty}\ x)\ \mathsf{empty} \\
insert & \mathsf{empty} & (S_1\ x) & = & \mathsf{node}\ \mathsf{empty}\ (insert\ \mathsf{empty}\ x)
\end{array}
$$

Now, we can prove that for any s : EXPR which does not have **theorem** as a proper sub-EXPR, and for any x : Bin,

$$
\mathcal{T}(step\ s\ x) = insert\ \mathcal{T}(s)\ x
$$

[4] The order between the equations is relevant in this program.

by induction on the definition of EXPR. We can also prove that any EXPR generated from theorem by successive applications of *step* does not contain a proper sub-EXPR of the form theorem.

Notice that the insertion of a word $S_0\ x$ in a tree node $s\ t$ always consists in modifying only the subtree s. Analogously, the insertion of a word $S_1\ x$ consists in modifying only the subtree t.

Given a sequence of words $x_1, \ldots$, a session can be described as follows. We start from the empty tree and insert successively the words in the given order. We will have a sequence of trees $s_1, \ldots$ such that $s_1 = \mathsf{empty}$ and for all i, $s_{i+1} = \mathit{insert}\ s_i\ x_i$. Eventually, *insert* will raise an exception and the session finishes. If this happens while inserting the word x_n, it means that the sequence $x_1, \ldots, x_n$ is good.

The proof and program given here are symmetric in S_0 and S_1. This means that if we do another session with words $y_1, \ldots$ such that for every i, y_i is x_i with all the S_0 substituted by S_1 and all the S_1 substituted by S_0, this session will also finish when the n-th word is inserted.

The program is not optimal. If we take any sequence starting with $(S_0\ (S_0\ 0))$, $(S_1\ 0)$ and $(S_0\ (S_1\ 0))$ we may expect it to stop after inserting the third word, because the second is a subword of it. However, the session will produce the following sequence of trees:

$$
\begin{array}{lll}
\mathsf{empty} & \longrightarrow & \mathsf{node\ (node\ end\ empty)\ empty} \\
 & \longrightarrow & \mathsf{node\ (node\ end\ empty)\ end} \\
 & \longrightarrow & \mathsf{node\ (node\ end\ end)\ end}
\end{array}
$$

In view of the program with state, we may describe a session in more detail. This will be done in a later version of this paper.

4 A parallel version of this algorithm

We first recall briefly Nash-Williams argument. We consider an infinite list $\alpha = x_1, \ldots$ of elements in Bin and want to show that there exists $i < j$ such that $x_i \leq x_j$. We reason by contradiction, introducing the notion of bad streams: a stream $\alpha = x_1, x_2 \ldots$ (or sequence) is **bad** iff there is no $i < j$ such that $x_i \leq x_j$. We suppose that there exists a bad sequence $\alpha = x_1, x_2 \ldots$ We can then suppose that all x_i are distinct from 0. We introduce two programs on streams. The first one st_0 strips all the S_0 from the first place there is a S_0, and forgets from then all the words starting with S_1

$$
\begin{array}{lllll}
st_0 & 0.y.l & = & \mathit{raise} & \\
st_0 & (S_1\ x).l & = & (S_1\ x).(st_0\ l) & \\
st_0 & (S_0\ x).l & = & x.(h\ l) & \\
 & & & \text{where} & \begin{array}{llll} h & (S_0\ x).l & = & x.(h\ l) \\ h & (S_1\ x).l & = & h\ l \\ h & 0.y.l & = & \mathit{raise} \end{array}
\end{array}
$$

The second st_1 is defined symetrically. It is clear that one of the streams $(st_0\ \alpha)$ or $(st_1\ \alpha)$ is infinite. Furthermore, if one of them is infinite, it is less than α for the lexicographic ordering[5], and it is bad.

[5] We say that $x_1 x_2 \ldots$ is less than $y_1 y_2 \ldots$ iff there exists n such that $x_i = y_i$ for $i < n$ and y_n is $S_0(x_n)$ or $S_1(x_n)$.

But it is not difficult to show non constructively by a diagonalisation argument that if there is a bad sequence, there is a minimal bad sequence, and we get a contradiction if α is such a minimal bad sequence.

In this form, Nash-Williams argument describes clearly a parallel algorithm where we compute in parallel the two streams $(st_0\ l)$ and $(st_1\ l)$, calling recursively the algorithm on each of them. More precisely, if we introduce the parallel program $par\ :\ \mathsf{Void}{\rightarrow}\mathsf{Void}{\rightarrow}\mathsf{Void}$ defined by the equations

$$\begin{array}{lllll} par & raise & \_ & = & raise \\ par & \_ & raise & = & raise \end{array}$$

the program corresponding to Nash-Williams argument is

$$\phi\ l = par\ (\phi\ (st_0\ l))\ (\phi\ (st_1\ l)).$$

Notice that it is surprisingly not clear that this algorithm terminates, even using Nash-Williams' argument. This termination can be extracted however from the "open induction principle" of J.C. Raoult [12].

Indeed, the property $\Phi(\alpha)$ that expresses that this program ϕ converges on α is *open*: if ϕ converges on α, it has used only a finite amount of informations about α, and so, there exists an initial segment of α such that $\Phi(\beta)$ holds whenever β has this initial segment. Furthermore, by construction of the program, $\Phi(\alpha)$ holds if $\Phi(\beta)$ holds for all β lexicographically smaller than α. By the "open induction principle", we get that $\Phi(\alpha)$ holds, that is $\phi(\alpha)$ converges, for any stream α.

It is clear that the last algorithm we have described is doing precisely this computation, each branch of the tree corresponding to one stage of one computation done in parallel. We can make this argument completely precise. We claim that if we consider the computation of ϕ on an *incomplete* element (see [13])

$$\phi(x_1.x_2 \ldots x_n.\ \bot),$$

it is equal to *raise* iff the element

$$insert\ (\ldots(insert\ (insert\ \mathsf{empty}\ x_1)\ x_2)\ldots)\ x_n$$

is equal to *raise*.

In particular, this gives another proof that this algorithm terminates.

Conclusion

We have presented a direct constructive proof of Higman's lemma. Furthermore this proof uses only the principle of structural induction. As such, the formal structure of the proof is much simpler and transparent than the arguments in the literature [11, 10, 14], and it can be, and has been, directly mechanized. Being constructive furthermore, this proof can be seen as the description of a non trivial algorithm which, given a stream of binary words $x_1, x_2, \ldots$ computes $p < q$ such that $x_p \leq x_q$. This algorithm has been described in detail here, by a general method.

We have explained why this algorithm can be seen naturally as the computational content of Nash-Williams argument. It may be interesting to make this intuition precise, by developping a general method of extracting programs from classical proofs that gives this algorithm for Nash-Williams argument[6].

Another direction may be to "reverse" the present analysis. Given a program about streams, is it possible to look at this program as an extracted program from an inductive proof? This question may be important for an "integrated approach" [6] of programs with streams.

Acknowledgement

Thanks to Chet Murthy for pointing to us that a precise connection between the functional program and the program with states can probably be done via a first-order partial interpretation, and to John Hugues for explaining the meaning of this remark.

References

[1] L. Augustsson and T. Johnsson. "Lazy ML User's Manual." Programming Methodology Group, Department of Computer Sciences, Chalmers, Distributed with the LML compiler, 1993

[2] R.M. Burstall. "Proving properties of programs by structural induction." Computer Journal 12(1), p. 41 – 48, 1969.

[3] Th. Coquand. "Constructive Topology and Combinatorics." Constructivity in Computer Science, LNCS 613, pp. 159 - 164, 1991.

[4] H. Curry and R. Feys. *Combinatory Logic, Vol. 1.* North-Holland Publishing Company

[5] J. Despeyroux. "Proof of Translation in Natural Semantics." Proceedings of the First ACM Conference on Logic in Computer Science, pp. 193 - 205, LICS, 1986.

[6] P. Dybjer. "Comparing Integrated and External Logics of Functional Programs." Science of Computer Programming 14, pp. 59-79, 1990.

[7] G. Higman. "Ordering by divisibility in abstract algebras." Proc. London Math. Soc., volume 2, pp. 326 -366, 1952.

[8] K. Larsen. "Proof Systems for Henessy-Milner Logic with Recursion." in LNCS 299, CAAP'88, pp. 215 - 230, 1988.

[9] Murthy, C. (1990) "Extracting Constructive Content From Classical Proofs." Ph.D. Thesis, Cornell University.

[10] C. Murthy, J. R. Russell. "A Constructive Proof of Higman's Lemma." Proceedings of the Fifth ACM Conference on Logic in Computer Science, pp. 257 - 267, LICS, 1990.

[11] C. Nash-Williams. "On well-quasi-ordering finite trees." in Proc. Cambridge Phil. Soc. 59, pp. 833 - 835, 1963.

[6] It can be shown that this algorithm is different from the one we get by the A-translation method [9], which tends to prove that this method changes some intensional aspect of the proof

[12] J.C. Raoult. "Proving open properties by induction." Information Processing Letters 29, pp. 19 - 23, 1988.

[13] D. Scott. "A type-theoretical alternative to CUCH, ISWIM and OWHY." Theoretical Computer Science, 1993.

[14] F. Richman and G. Stolzenberg. "Well Quasi-Ordered Sets." Advances in Mathematics, 1993. Vol 97, pages 145-153.