Proof pearl: elementary inductive and integer-based proofs about balanced words

Judicaël Courant[1] and Jean-François Monin[1]

VERIMAG - Centre Équation, 2 avenue de Vignate, F-38610 Gières, France
{judicael.courant|jean-francois.monin}@imag.fr
http://www-verimag.imag.fr/~{courant|monin}/

Abstract. Inductive characterizations of words containing the same number of 'a' and 'b' can easily be given. However, formally proving the completeness of some of them turns out to be trickier than one may expect. We discuss and compare two Coq developments relating such balanced words to their inductive characterizations. One is based on auxiliary inductive structures and elementary arguments. The other one is based on a discrete variant of the intermediate value theorem. The extraction of the computational contents of these solutions leads to parsing algorithms of dramatically different efficiencies.

1 Introduction

It is a well-known fact that, in order to prove a property P about an inductively defined set S, one sometimes needs to prove a stronger property using the basic induction principle associated to S (such as basic Peano's induction principle) or to use a stronger induction principle (such as well-founded induction on a suitable relation), or even a combination of both.

It should be noticed that, in a constructive setting, any solution ultimately reduces to the basic induction principle, simply because the latter exhibits *the* canonical way of constructing objects in S. For instance, although in the well-known example of euclidian division on natural numbers, one can use well-founded induction on $<$ (finding q and $r < b$ such that $a = bq + r$ with $b > 0$ reduces to finding q' and r such that $a - b = bq' + r$ when $b < a$, since *in that case* we have $a - b < a$), well-foundedness of $<$ does not come for free. It is itself proven by induction on natural numbers and a close look at the reduction steps involved in the computation of q and r reveals that all numbers a, $a-1$, $\ldots 0$ are considered (not only a, $a - b$, $a - 2b$ and so on). We don't pretend that a direct induction on a would provide an interesting and really new solution to euclidian division. However in some situations, sticking to basic induction principles has an interest: for instance, [McB03] shows how one can give a structurally recursive unification algorithm; hence seeing the actual program suffices for being convinced it terminates.

That said, the most relevant criterium in formal proof engineering is the same as the one in software engineering: human effort measured as development time and size of the code written (hence ease of maintenance). Then, when faced to a new problem, to what extent should one spend effort to formalize an appropriate piece of mathematics, when it is still not available? Is it worth to stick to elementary arguments, at the price of thinking more at the inductive structure of the problem at hand?

We do not intend to provide a general answer to these questions here. Our ambition is limited to presenting and comparing the two approaches in the context of a problem on words. Both solutions, although non-trivial, can be presented in full detail in little amount of space.

Consider the the set Σ^* of finite words over the alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$. We say that a word u in Σ^* is *balanced* if the number of occurrences of the letter a in u is equal to the number of occurrences of the letter b in u. An obvious characterization of the set of balanced words B_0 is inductively defined as follows:

- the empty word is balanced: $\varepsilon \in B_0$;
- if a word is balanced, inserting an a and a b anywhere in it yields a balanced word; that is, if u, v and w are in Σ^* and if $uvw \in B_0$, then $u\mathsf{a}v\mathsf{b}w \in B_0$ and $u\mathsf{b}v\mathsf{a}w \in B_0$.

If w is non-empty in an inductive step of B_0, it ends either with a or b, and then a corresponding b or a should respectively occur in the prefix of the word. This leads us to consider the (seemingly strictly) weaker inductive definition B_1:

- $\varepsilon \in B_1$;
- if u and v are in Σ^*, and $uv \in B_1$, then $u\mathsf{a}v\mathsf{b} \in B_1$ and $u\mathsf{b}v\mathsf{a} \in B_1$.

It is clear that B_0 contains B_1 and that all words in B_0 are balanced. What about the converse—completeness wrt the set of balanced words? Loosely, we can argue as follows about B_0. If a balanced word t is non-empty, it contains at least one a (because otherwise, t would only be made of b, which is impossible because it is balanced). Similarly, t contains at least one b. Depending on whether a occurs before b in t or conversely, we have $t = u\mathsf{a}v\mathsf{b}w$ or $t = u\mathsf{b}v\mathsf{a}w$, where uvw is balanced. Then we see that an induction on the length of balanced words will do.

As for the completeness of B_1, consider the rightmost letter x in a non-empty balanced word tx. Then t contains at least one $\overline{x}$, where $\overline{\mathsf{a}} \overset{\text{def}}{=\!=} \mathsf{b}$ and $\overline{\mathsf{b}} \overset{\text{def}}{=\!=} \mathsf{a}$ (because otherwise tx would not be balanced), which means that we have $tx = u\overline{x}vx$, with uv a balanced word.

Now consider the still weaker inductive definition B_2:

- $\varepsilon \in B_2$;
- if $u \in B_2$ and $v \in B_2$ then $u\mathsf{a}v\mathsf{b} \in B_2$ and $u\mathsf{b}v\mathsf{a} \in B_2$.

The previous line of reasoning does not work any longer. However we will see that B_2 is still a complete inductive characterization of balanced words.

Even for B_0 and B_1, a complete formalisation requires more work than may be expected at first sight.

We present two opposite approaches to this problem. The first one (section 3), sticks to the inductive definition of words (we don't even consider induction on the length of words) and gets elementary proofs. The second one (section 4) is based on the theorem of intermediate values. Thanks to a massive use of automated tactics and tacticals, the proof script remains quite short even for the latter solution. Both versions share a common set of definitions given in section 2. Finally, measures, conclusions and perspectives are given in section 5.

All proofs were checked within the Coq proof assistant [The05,BC04].

2 Problem statement and notation

2.1 CIC and Coq in a nutshell

Our formalization is carried out in the framework of the calculus of inductive constructions (CIC), with the help of the Coq proof assistant. The reader is referred to [The05,BC04] for detailed explanations, but in a few words, a formal development consists of a sequence of definitions and theorems. The main difference with usual mathematics is that basic objects are not (untyped) sets, but inductive types and functions (typed λ-terms). The type system includes polymorphic types and dependent types. Following Curry-Howard isomorphism, propositions can be seen as types inhabited by the proofs of those propositions; similarly, predicates are dependent types—propositions depending on values. Hence $A \rightarrow B$ is interpreted as a functional type when A and B are datatypes and as an implication when A and B are logical formulæ. Similarly, $\forall x : A, Px$ may be interpreted as a universally quantified formula or as the product of a type family ($\Pi_{x \in A} Px$).

A definition binds an identifier to a λ-term. General course-of-values recursion is not allowed, because (strong) normalization of the calculus is required in order to preserve the consistency of the underlying logic. However, structural recursion is allowed on all inductive types, which is enough for encoding any provably terminating recursive function.

Types are themselves values in higher types. For instance, propositions have the type *Prop* and datatypes have the type *Set*. Hence we can construct families of propositions and datatypes by structural induction on an inductive type.

In Coq, proofs are not directly expressed as typed λ-terms by the end user. Instead, she or he interactively specifies how to construct a proof term by means of commands called *tactics* or combination of tactics called *tacticals*. Coq includes a language for developing user-defined tactics. We return to this feature in section 4.1, where we describe useful tactics (typically *splitsolve*), which we systematically use for discharging routine subgoals.

The proofs presented below stick to the formal Coq proofs. They are automatically derived from the actual Coq proof scripts using the *coqdoc* tool of Jean-Chritophe Filliâtre [The05] and a small number of cosmetic transformations.

2.2 Words

Let us come back to words. Words are inductively defined with the "Snoc" convention: letters are added to the right.

Inductive Σ^*: $Set := \varepsilon$: Σ^* | $Snoc : \Sigma^* \to \Sigma \to \Sigma^*$.

Such a statement defines:

- a new inductive type: Σ^*;
- its contructors: ε and *Snoc*.

Moreover, Coq automatically constructs the expected induction principle on Σ^*. In the formal development, we use the infix notation $u :: x$ for $Snoc\ u\ x$. In this paper, we simply write ux.

The following notions are easily defined by structural recursion on words:

- the catenation of two words u and v, denoted by $u\,v$
- the length of a word u, denoted by $|u|$;
- for x in Σ, the *x-length* of u, denoted by $|u|_x$, which is the number of occurrences of x in u.

In this paper, we use the convention that x, y and z range over letters and u, v and w range over words. This convention or/and the typing context always make clear whether juxtaposition denotes functional application or word construction.

Definition 1. *A word u is* balanced *if* $|u|_{\mathsf{a}} = |u|_{\mathsf{b}}$.

A useful induction principle on words is well-founded induction on their length. The following commented proof script claims and proves it (using the automated tactic *splitsolve0* described section 4.1):

Lemma *length_rect* :
  $\forall$ (P : $\Sigma^* \to$ *Type*) u,
  ($\forall$ u_0, ($\forall$ v, $|v| < |u_0| \to P\ v$) $\to P\ u_0$) $\to P\ u$.
We first introduce variables and the inductive step:
intros P u H_Step.
We claim the following properties over natural numbers:
assert (*Gen* : ($\forall$ n u_1, *length* $u_1 < n \to P\ u_1$)).
  To prove it, we can use the induction principle for natural numbers:
  induction n.
    The case 0 is trivial thanks to *H_Step*:
    splitsolve0 1...
    The case ($S\ n$) holds thanks to *H_Step* and the induction hypothesis:
    intros u_1 *l_u1_Sn*; *case* (*le_lt_eq_dec* ($|u_1|$) n).
      auto with *arith*...
      exact (*IHn* u_1)...
      intro e; *apply H_Step. rewrite e. exact IHn...*

  The main claim now clearly holds (just apply *Gen* to (S $|u|$) and u):
  apply (*Gen* (S ($|u|$)) u). *splitsolve0* 1.
Qed.

2.3 Languages

The languages B_0, B_1 and B_2 are defined as inductive properties[1] on Σ^*.

Inductive $B_0 : \Sigma^* \to \mathit{Set}$:=
| $B_0\_e : B_0\ \varepsilon$
| $B_0\_ab : \forall\ u\ v\ w,\ B_0\ (uvw) \to B_0\ (u\mathsf{a}\,v\mathsf{b}\,w)$
| $B_0\_ba : \forall\ u\ v\ w,\ B_0\ (uvw) \to B_0\ (u\mathsf{b}\,v\mathsf{a}\,w)$

Inductive $B_1 : \Sigma^* \to \mathit{Set}$:=
| $B_1\_e : B_1\ \varepsilon$
| $B_1\_ab : \forall\ u\ v,\ B_1\ (uv) \to B_1\ (u\mathsf{a}\,v\mathsf{b})$
| $B_1\_ba : \forall\ u\ v,\ B_1\ (uv) \to B_1\ (u\mathsf{b}\,v\mathsf{a})$

Inductive $B_2 : \Sigma^* \to \mathit{Set}$:=
| $B_2\_e : B_2\ \varepsilon$
| $B_2\_ab : \forall\ u\ v,\ B_2\ u \to B_2\ v \to B_2\ (u\mathsf{a}\,v\mathsf{b})$
| $B_2\_ba : \forall\ u\ v,\ B_2\ u \to B_2\ v \to B_2\ (u\mathsf{b}\,v\mathsf{a})$

For $i \in \{0,1,2\}$, we can use B_i_other instead of clauses B_i_ab and B_i_ba.

Lemma $B_0\_other : \forall\ u\ v\ w\ x,\ B_0\ (uvw) \to B_0\ (uxv\overline{x}w)$.
Lemma $B_1\_other : \forall\ u\ v\ x,\ B_1\ (uv) \to B_1\ (uxv\overline{x})$.
Lemma $B_2\_other : \forall\ u\ v\ x,\ B_2\ u \to B_2\ v \to B_2\ (u\overline{x}vx)$.

We want to prove, for $i \in \{0,1,2\}$: $\forall u,\ |u|_{\mathsf{a}} = |u|_{\mathsf{b}} \leftrightarrow B_i\,u$. Correctness, that is $\forall u, B_i\,u \to |u|_{\mathsf{a}} = |u|_{\mathsf{b}}$, is easy: just apply the induction principle for B_i. As for completeness ($\forall u, |u|_{\mathsf{a}} = |u|_{\mathsf{b}} \to B_i\,u$), in the following we consider only B_1 and B_2, since B_1 trivially entails B_0.

3 Elementary inductive proofs

It is clear that a direct attempt to prove $\forall u, |u|_{\mathsf{a}} = |u|_{\mathsf{b}} \to B_i\,u$ by induction on the structure of u will fail. The idea is then to find a suitable generalization P_i of B_i such that $\forall u, |u|_{\mathsf{a}} = |u|_{\mathsf{b}} \to P_i\,u$ is provable by induction on u.

3.1 Completeness of B_1

In the course of an induction step, we analyze a word ux. If we want ux to be balanced, we have to consider the case where u is not balanced. For instance if $x = \mathsf{a}$, we expect that $|u|_b = 1 + |u|_a$. In general, we may have an arbitrary excess number n of y (where y is a or b) in a word u. An idea is then to encode this number in a word by considering uy^n. Here is the key lemma.

Lemma $B_1\_completeness\_aux : \forall\ u\ n\ y,\ |u|_{\overline{y}} = n + |u|_y \to B_1\ (uy^n)$.
The proof is by induction on u.
induction u as [|*u IndHyp x*]; *intros n y*.

[1] In the first version of this development, the type of B_0, B_1 and B_2 was $\Sigma^* \to \mathit{Prop}$ instead of $\Sigma^* \to \mathit{Set}$. The difference is not significant here, this change was made for extraction purposes only (see Section 5.3).

The case where $u = \varepsilon$ reduces to $0 = n \rightarrow B_1\ (\varepsilon\ y^n)$ because $|\varepsilon|_y = 0$

The following tactic amounts to applying clause B_1_*e*.

splitsolve0 1...

In the inductive step, we consider the word ux, which yields the goal: $|ux|_{\overline{y}} = n + |ux|_y \rightarrow B_1\ (uxy^n)$. Let us compare y with $\overline{x}$.

case (*eq_or_other y x*); *intro e*; *rewrite e*; *clear e y*; *splitsolve0* 0.

When $y{=}x$, we get: $|u|_{\overline{x}} = n + S\ (|u|_x) \ \vdash\ B_1\ (u\,x^{n+1})$, hence we can use the induction hypothesis on $n+1$ and x:

apply (*IndHyp* (*S n*) *x*); *splitsolve0* 0...

When $y{=}\overline{x}$, we get: $|u|_x + 1 = n + |u|_{\overline{x}} \ \vdash\ B_1\ (ux\overline{x}^n)$. Then we decide to compare n with 0.

destruct n; *simpl* in * ⊢ * .

When $n{=}0$, the goal reduces to $B_1\ (ux)$. Then we apply the induction hypothesis on 1 and the *opposite* letter.

apply (*IndHyp* 1 *x*); *splitsolve0* 0...

When the previous n is a successor (the successor of the new n provided by the tactic *destruct*), the new goal is $B_1\ (ux\overline{x}^n\overline{x})$. We can apply B_1_*other* with the induction hypothesis on n and $\overline{x}$.

apply B_1_*other*. *apply* (*IndHyp* n $\overline{x}$); *splitsolve0* 0...

Qed.

The desired theorem is a simple corollary of the previous lemma, taking $n = 0$ and any letter for y.

3.2 Completeness of B_2

The inductive definition of B_1 allowed us to count the deficit of a given letter wrt. the other in a word. In the last case we consider, we use the fact that a y we get from the stock we have on the right of uy^n may fit any $\overline{y}$ in u.

We don't have this freedom any longer for B_2. In order to delete the rightmost letter of a balanced word wy, we have to decompose w into $u\overline{y}v$, such that v is itself balanced.

So we have to record the structure of a word with a finer grain than before. To this effect, we introduce a family C of predicates as follows. First, we say that a word w is *split* by a predicate P and a letter x if there exist two words u and v such that $P\,u$, $B_2\,v$ and $w = uxv$. The family C is then defined by the following equations:

$$C_{x,0} \stackrel{\text{def}}{=} B_2 \tag{1}$$

$$C_{x,n+1} \stackrel{\text{def}}{=} \mathit{split}\, C_{x,n}\, x \tag{2}$$

Intuitively, we have $C_{x,n}\,u$ if u can be decomposed as $u_0xu_1x\ldots xu_n$, with $\forall i \in [0,n], B_2\,u_i$. The formula $C_{y,n}\,u$ will play here the role played by $B_1\,u$ in section 3.1. We first need three easy lemmas.

Lemma $B_2\_app : \forall\ u,\ B_2\ u \to \forall\ v,\ B_2\ v \to B_2\ (uv).$

Proof: by induction on the construction of $B_2\,v$.

Lemma $C\_app : \forall\ x\ u\ n,\ C_{x,n}\ u \to \forall\ v,\ B_2\ v \to C_{x,n}\ (uv).$

Proof: by case analysis on n, using lemma *B2_app*.

Lemma $C\_other : \forall\ x\ u\ v\ n,\ C_{\overline{x},n}\ u \to B_2\ v \to C_{\overline{x},n}\ (u\overline{x}vx).$

Proof: easy corollary of lemma *C_app*.

Now we have the main lemma.

Lemma $B_2\_completeness\_aux : \forall\ u\ n\ y,\ |u|_y = n + |u|_{\overline{y}} \to C_{y,n}\ u.$
induction u as [|*u IndHyp x*]; *intros n y*.

The case where $u = \varepsilon$ reduces to $0 = n \to C_{y,n}\ \varepsilon$ because $|\varepsilon|_y = 0$

splitsolve0 1...

In the inductive step, we consider the word ux, which yields the goal: $|ux|_y = n + |ux|_{\overline{y}} \to C_{y,n}\ (ux)$. Let us compare y with $\overline{x}$.

case (*eq_or_other y x*); *intro e*; *rewrite e*; *clear e y*; *splitsolve0* 0.

When $y{=}x$, we get: $S(|u|_x) = n + |u|_{\overline{x}} \ \vdash\ C_{x,n}\ (ux)$. Then we decide to compare n with 0.

destruct n; *simpl* in * ⊢ * .

When $n{=}0$, the goal reduces to $B_2\ (ux)$. Then we apply the induction hypothesis with 1 and the *opposite* letter.

elim (*IndHyp* 1 $\overline{x}$); *splitsolveB2o* 1...

When the previous n is a successor (the successor of the new n provided by the tactic *destruct*), the new goal is *split* $(C_{x,n})\ x\ (ux)$. The witnesses are u and ε, using the induction hypothesis on n and x.

generalize (*IndHyp n x*) B_2*_e*; *intros. refine* (*split_ex* _ _ u ε _ _); *auto...*

When $y{=}\overline{x}$, we get: $|u|_{\overline{x}} = n + (|u|_x + 1) \ \vdash\ C_{\overline{x},n}\ (ux)$. By induction hypothesis on $n{+}1$ and $\overline{x}$, we get w and v such that $u{=}\ wxv)$, $C_{\overline{x},n}\ w$ and $B_2\ v$, so that we apply lemma *C_other*.

elim (*IndHyp* (*S n*) $\overline{x}$); [*intros w v cnw bv* | *splitsolve0* 0].
apply C_other; *assumption...*

Qed.

As for B_1, the completeness of B_2 reduces to a special case of the previous lemma, with $n = 0$ and $y = \mathsf{b}$ ($y = \mathsf{a}$ works as well).

4 Solution based on the intermediate value theorem

4.1 Automatizing proofs

Compared to the elementary proof, the solution based on the intermediate value theorem needs much more automation. In order to achieve the proof, we used a few tactics which we designed in the course of another development [CM06].

The simplest one, *progress0*, carries out a single trivial invertible reasoning step (that is, a step preserving the provability of the goal) such as introduction of variables, elimination of conjunction in hypothesis, systematic application in hypothesis and conclusion of the goal of certain rewriting database, ... The most elaborated one, *splitsolve0* features a (naive) iterative deepening proof-search, combining some non-invertible tactics together with application of *progress0*). More precisely *splitsolve0* actually is an instance of a generic tactic *splitsolve*, parameterized by a tactic doing invertible steps and a tactic doing non-invertible steps. Thus, *splitsolve0* could easily be extended to some tactics *splitsolve1* and *splitsolve2* dealing with domain-specific reasonning:

```
Ltac $progress1\ g$ := match $goal$ with
  | [ $\vdash$ $step\_up$ ?$f$ ?$z$ (?$u$?$x$) ] $\Rightarrow$ $apply\ step\_up\_charact$
  | [ $H$ : $\mathcal{C}^0$?$f$ $\vdash$ _ ] $\Rightarrow$ $unfold$ $\mathcal{C}^0$in $H$
  end
  || $progress0\ g$.

Ltac $splitsolve1\ n$ := $splitsolve\ progress1\ split0\ noni0\ n$ 0.
```

A preliminary version of these tactics has been posted on the Coq Wiki (`http://cocorico.cs.ru.nl/coqwiki/GenericTactics`).

4.2 The discrete intermediate value theorem

As a general result, we first prove a discrete version of Weierstrass intermediate value theorem over words. We define *continuous* functions over words as follows:

```
Definition $\mathcal{C}^0 f$ := $\forall\ u\ x$,
  {$f$ ($ux$) - $f$ $u$ = 1 }+{ $f$ ($ux$) = $f$ $u$ }+{ $f$ ($ux$) - $f$ $u$ = -1}.
```

Given such a function and a word u, we would like to show that for any intermediate value z between $f(\varepsilon)$ and $f(u)$, there exists some prefix v of u such that $f(v)$ takes this value. Moreover, if $z < f(v)$, we can choose v such that the next prefix vx is such that $f(ux) = z + 1$ (take for instance the longest prefix v of u such that $f(v) = z$). We say that f steps up from z to $z + 1$ at v and x. More precisely, we define stepping up as follows:

```
Inductive $step\_up$ ($f$ : $\Sigma^* \to Z$) ($z$ : $Z$) : $\Sigma^* \to Set$ :=
| $Cstep\_up$ : $\forall\ u\ x\ v$,
  $f\ u = z \to f\ (ux) = z + 1 \to step\_up\ f\ z\ (ux\,v)$.
```

Stepping up can be characterized as follows:

The following commented proof script then proves the intermediate value theorem:

Theorem *int_val*: $\forall\ f\ z\ u$, $\mathcal{C}^0 f \to f\ \varepsilon \leq z < f\ u \to step\_up\ f\ z\ u$.
induction u as [| *v IHu x*];*intros.*

The case $u = \varepsilon$ is trivial since the hypothesis $f(\varepsilon) \leq z < f(u)$ is absurd then:
splitsolve1 0%*nat...*

We now consider the case $u = vx$:

assert ({ $z = f\ v$ }+{ $z < f\ v$ }).

The above claim automatically resolves by continuity of f and since $z < f(vx)$:

splitsolve1 2%*nat*...

The main case is now trivial: either $f(v) = z$ and $f(vx) = z{+}1$ and we are done, or $z < f(v)$ and we conclude thanks to the induction hypothesis:

splitsolve1 3%*nat*...

Qed.

4.3 Main proof

In order to apply the intermediate value theorem, we first define a *balance* function telling us how much a word is unbalanced,

Definition *balance* u := *Z_of_nat* ($|u|_{\mathsf{a}}$) - *Z_of_nat* ($|u|_{\mathsf{b}}$).

Definition *balanced* u := *balance* $u = 0$.

Then we show *balance* is continuous:

Lemma *continuous_balance* : $\mathcal{C}^0$*balance*.

Once this is done, we can prove that one can split all slightly unbalanced u into balanced v and w such that $u = vxw$ for some letter x. Formally:

Lemma *unbalanced_insert_a* :
  ∀ u, *balance* $u = 1 \rightarrow$
    { v : Σ^* & { w : Σ^* | *balanced* v ∧ *balanced* w ∧ $u = v\mathsf{a}w$ }}.

Lemma *unbalanced_insert_b* :
  ∀ u, - *balance* $u = 1 \rightarrow$
    { v : Σ^* & { w : Σ^* | *balanced* v ∧ *balanced* w ∧ $u = v\mathsf{b}w$ }}.

Equivalence of B_1, B_2 and B_0 then proceed by induction over the length of the considered words. For instance, here is the script for B_1:

Theorem *B_1_compl* : ∀ u, $|u|_{\mathsf{a}} = |u|_{\mathsf{b}} \rightarrow B_1\ u$.

The proof proceeds by induction over the length of u:

intro u; *pattern u*; *apply length_rect*; *clear u*.

intros u HypInd Hbal_u.

We do a case analysis over u:

destruct u as [| *v x*].

The case $u = \varepsilon$ is trivial:

splitsolve2 1%*nat*...

Otherwise, $u = vx$. Then we have:

assert ({ - *balance* $v = 1$ }+{ *balance* $v = 1$ }).

which can be proved by a simple case analysis over x:

destruct x; *splitsolve2* 1%*nat*...

Using the previous results on unbalanced words, we can conclude with a simple case analysis over x:

generalize (*unbalanced_insert_b v*) (*unbalanced_insert_a v*).

destruct x; *splitsolve2* 2%*nat*...

Qed.

5 Conclusions

5.1 Figures

We now draw a comparative table of both developments. The points we compare are the following:

1. Length of the development, measured as the number of lines of specifications and of proof scripts.
2. Number of lemmas and theorems.
3. Average length of proof per lemma/theorem.
4. Complexity of the proof.
5. Automatization level of the proof measured as the number of lines of proof script per use of the splitsolve tactic.
6. Control of the proof.
7. Compilation speed of the proof script alone (measured on the first author's laptop, in seconds).
8. Compilation speed of the dependencies of the proof script.
9. Size of the generated compiled proof development (`.vo` file, in bytes).

These points are measured in the table below for:

- The inductive proof (IP).
- The proof using the intermediate value theorem (PUIVT), alone without the proof of the intermediate value theorem itself.
- The development of the intermediate value theorem (IVT) itself.

	IP	PUIVT	IVT
Proof length	63	64	44
Results proved	8	6	5
Average proof length	4.1	7.0	4.0
Complexity	high	low	low
Lines of proof / splitsolve	1.9	1.5	2.2
Control	high	low	low
Compilation time (s)	1.9	31.4	5.3
Compilation time of deps (s)	1.4	7.1	1.1
Size of `.vo` file	28K	140K	28K

Although the automated tactics were really needed for the development of the proof using the intermediate value theorem, the inductive proof also takes advantage of them.

As for compilation times and size of the generated `.vo` files, there is clearly a big advantage to inductive proofs here: although the support for equationnal reasonning improves, Coq is still much more efficient on inductive proofs.

The proof based on the intermediate value theorem development and the inductive proof are on a par (63 lines versus 64), which is somewhat surprising: the higher-level results you use, the shorter your proofs shoud be. On the other hand, the proof of the IVT itself turns out to be quite short (44 lines), and this result can be reused in other contexts. At least, it is certainly much more reusable than the family $C_{x,n}$ of predicates introduced in section 3.2.

5.2 Symmetry of the inductive proof

In the approach based on the IVT, we introduce the notion of a balance of a word u, namely $|u|_a - |u|_b$ where the considered numbers are *integers*. This is very natural and we may wonder why this notion is not needed in the inductive proofs, where only natural numbers are used: since these proofs proceed by induction on a word, it is intuitively clear that while traversing this word, one considers all its prefixes, whose balances are sometimes positive, sometimes null and sometimes negative.

The places where the signs of balances change can actually be identified: they are precisely the places where the induction hypothesis is used with 1 and the *opposite* letter (this letter is named $\overline{x}$ in both lemmas B_i*_completeness_aux*: for B_1, this case occurs when $y = \overline{x}$ while for B_2, this case occurs when $y = x$). In this way, the management of the case where the balance is positive and increases (respectively decreases) and of the case where the balance is negative and decreases (respectively increases) are shared in the inductive proof. The proofs succeed because the predicate to be inductively proved is quantified over y.

A closer look at the inductive proofs shows more: thanks to this quantification, letters a and b pleasantly play perfectly identical roles. Using the balance function actually introduces a bias: we may as well consider the opposite function, yielding a symmetrical but different proof. It is still unclear for us whether considering x versus $\overline{x}$ instead of a and b in the IVT-based development could lead to shorter and more elegant proofs as well.

5.3 Program extraction

All the proofs described in this paper are constructive. Their computational contents are simply parsing algorithms for the languages B_1 and B_2. These algorithms can be made explicit using a feature of Coq called *program extraction*, which is based on a separation between informative types (in the sort *Set*) and logical types (in the sort *Prop*): program extraction essentially boils down to pruning non-informative parts of a λ-term. We initially defined our inductive properties B_1 and B_2 as members of the sort *Prop* and just had to lift them, as well as the *statements* of several lemmas from *Prop* to *Set* in order to use this facility. It is quite remarkable to note that we need really few modifications, if any, in the proofs of these lemmas. This is due partly to the fact that many tactics have similar uniform behavior both on informative and non-informative goals and partly to the use of our automated tactics. Changes were due for the one part to the fact that the Omega tactics (a decision procedure for Pressburger arithmetics) splits some non-informative propositional connectives but not any informative one and for the other part to a bug of the version of Coq we used.[2]

Extraction yields functional pieces of codes, (in ML or Haskell) and we are able to compare code size and efficiency of the parsing algorithms pro-

[2] Less remarkable is the fact that our development works in Coq version 8.0pl3 but does not seem to work *as is* in Coq version 8.1beta. We would need further investigations to tell the extent of the needed changes.

vided by the two approaches. Such a study is motivated by problems encountered in a much larger experiment, the C-CoRN project at University of Nijmegen [GWZ00,CFGW], where the computational contents of a constructive proof of the fundamental theorem of algebra turned out to be untractable in practice. This was analyzed and progress were reported in [CFL05] and [Let04]. The difficulty is essentially that the whole development was primarily developed with a focus on mathematics rather than on algorithmics. On the other hand, [Let04] reports a less ambitious but resource-aware constructive approach for representing constructive real numbers and continuous functions, resulting in a successful computation of approximations of $\sqrt{2}$.

Although the scale of our experiment on words is much smaller, it shares a common point with C-CoRN: we developed the theorems without thinking to the underlying algorithms, and afterwards replaced *Prop* with *Set* in the relevant places. We are then able to compare the computational costs of our two approaches: is it worth looking for an elementary, more controlled, proof in order to get a more efficient computational contents?

5.4 Experimental results

We compared the five following parsers:

Ind B1 Parser for B1, extracted from the inductive proof.
Ind B2 Parser for B2, extracted from the inductive proof.
Ivt B1 Parser for B1, extracted from the proof using the intermediate value theorem.
Ind B2 Parser for B2, extracted from the proof using the intermediate value theorem.
Direct Parser for B2, hand-written.

We posted the challenge of writing parsers for B1 and B2 in our lab, and sent it as well to a few colleagues. We got only one answer for B1 (in prolog), and several solutions (or partial solutions) for B2. The parser we evaluated against our extracted programs has been adapted from a solution by Claude Marché and Jean-Christophe Filliâtre.

This hand-written parser uses a more compact representation of syntactic trees than the extracted ones: in extracted parsers, a node B_2_ab of a syntactic tree contains the corresponding words u and v and has two sons corresponding to the parsing of u and v whereas in the hand-written one, these words are omitted, which is more natural from a programmer's point of view. This probably has a non-negligible impact on performance since building these intermediate words from the compact tree requires words concatenation, which should require $O(n \log n)$ operations on average when parsing words of length n (assuming concatenation requires a time proportional to the size of its operands).

We give our experimental timing measures of the parsing time of randomly chosen balanced words on figures 1 and 2. Lengths of the words are given on abscissas while ordinates denote the mean parsing times in seconds (the smaller,

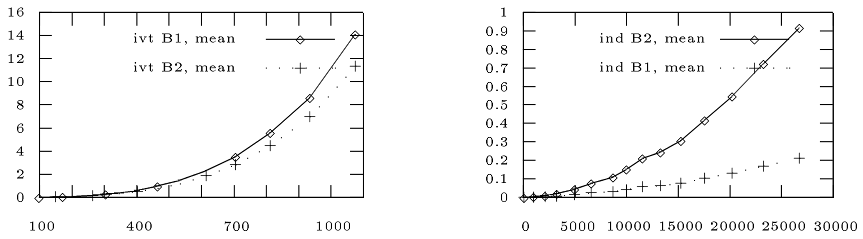

Fig. 1. Mean parsing times of ivt1, ivt2, ind1 and ind2

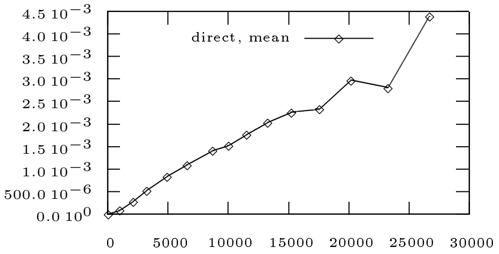

Fig. 2. Mean parsing times of direct

the better). The material needed for reproducing these results is available on the web page of the authors.

In order to help comparing these figures, we give on figures 3 and 4 the ratios of some of these execution times. It is interesting to note that the choice of the right proof method has much more impact (in terms of ratio of performance) than the choice of a direct implementation versus the extraction of the best proof. Besides ratios, a closer inspection of execution times shows that the hand-written program, the programs extracted from the inductive proof and the programs extracted from the proof based on the intermediate value theorem have time complexity of order $O(n^{\alpha})$, where α takes respectively the values 1.17, 1.7 and 3.2. However these values should be taken with care, as time consumption in strongly perturbated by automatic memory management (this is especially visible on the direct implementation, where a linear behavior is expected).

Finally, since the inductive proof was not developed with performance in mind, there might be room for improvements. Whether there is enough room for bringing the performance of extracted programs close the performance of the direct implementation is an open question.

Acknowledgments

We would like to thank Jean-Christophe Filliâtre and Claude Marché for their parser. We also thank Xavier Nicollin for pointing out that the definition of B2

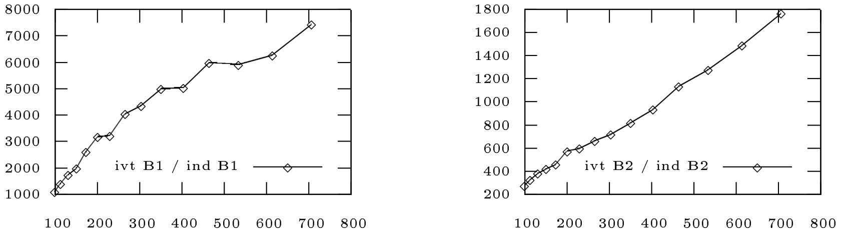

Fig. 3. Ratio ivt1/ind1 and ivt2/ind2 and for small lengths

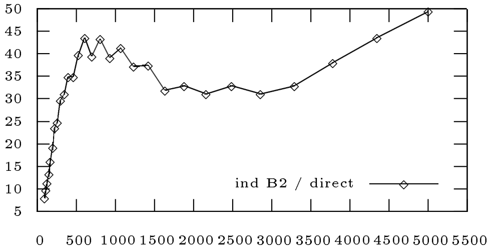

Fig. 4. Ratio ind2/direct

could be turned into an LALR(1) grammar (recognized by yacc) by a transformation preserving the syntactic structure, and Romain Janvier for proposing prolog solutions. Last but not least, we thank Yassine Lakhnech for the statement of the problem as an exercise for teaching various approaches and proof strategies on inductive definitions as well as for fruitful discussions.

References

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, volume XXV of *Texts in Theoretical Computer Science. An EATCS Series.* Springer, 2004. 469 p., Hardcover. ISBN: 3-540-20854-2.

[CFGW] Luìs Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen.

[CFL05] L. Cruz-Filipe and P. Letouzey. A Large-Scale Experiment in Executing Extracted Programs. In *12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus'2005*, 2005. To appear.

[CM06] Judicaël Courant and Jean-François Monin. Defending the bank with a proof assistant. Vienna, March 2006. To appear in WITS.

[GWZ00] Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In *TYPES*, pages 96–111, 2000.

[Let04] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[McB03] Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075, November 2003.

[The05] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.0*. Logical Project, January 2005.