The Gilbreath trick: a case study in axiomatisation and proof development in the Coq proof assistant

Gérard Huet

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1511

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

THE GILBREATH TRICK : A CASE STUDY IN AXIOMATISATION AND PROOF DEVELOPMENT IN THE COQ PROOF ASSISTANT

Gérard HUET

Septembre 1991

RR-1511

The Gilbreath Trick:
A case study in Axiomatisation and Proof Development in the Coq Proof Assistant

Gérard Huet

INRIA[1]

Abstract

We present the full axiomatisation and proof development of a non-trivial property of binary sequences, inspired from a card trick of N. Gilbreath. This case study illustrates the power and naturalness of the Calculus of Inductive Constructions as a specification language, and outlines a uniform methodology for conducting inductive proofs in the Coq proof assistant.

Le tour de cartes de Gilbreath :
Une étude de cas d'axiomatisation et de construction de preuve dans le système d'aide à la démonstration Coq

Gérard Huet

INRIA

Résumé

Nous présentons une axiomatisation et une preuve formelle complètes d'une propriété non-triviale des suites binaires, inspirée d'un tour de cartes de N. Gilbreath. Cette étude de cas illustre la puissance et la simplicité d'utilisation du Calcul des Constructions Inductives comme langage de spécification, et suggère une méthodologie uniforme pour le développement de preuves par récurrence dans le système d'aide à la démonstration Coq.

[1]This research was partially supported by ESPRIT Basic Research Action "Logical Frameworks."

Introduction

This note presents a complete formal proof, in version 5.6 of the Proof Assistant Coq, of a card trick due to Norman Gilbreath[15, 13, 14], and shown to the author by N. G. de Bruijn[5]. The trick is based on a mathematical theorem concerning the shuffles of alternated binary words. A complete formalisation of the problem is given below. Coq is an implementation of the Calculus of Inductive Constructions[11] under development at INRIA in Rocquencourt and at ENS in Lyon[12].

This note is basically a transcript of what the user should type in order to mechanically verify the proof. Coq uses a top-down proof engine which solves successive subgoals with tactics in the style of LCF[16]. In the transcript below we do not systematically show the output of the system, which lists at every step the remaining subgoals. The proof is thus fully understandable only if one runs this proof interactively with our system. Furthermore, the transcript does not faithfully represent the proof-finding activity, with its numerous erroneous attempts, and the top-down development leaving to future proofs mathematical facts which are initially assumed as axioms. Finally, this transcript does not give a feeling for the actual interactive use of our system, which may be optionally driven from a sophisticated window interface. However, we hope that this commented example will be of use to others, both in order to assess the present capabilities of our system, and in order to get some familiarity with our axiomatisation style and proof methodology.

1 Axiomatising mathematics in Coq

Every mathematical object defined or axiomatised in the system is equipped with its *type*. We also say that the object *inhabits* its type. There are two *sorts* of types: propositions and sets. Propositions are of sort *Prop*, sets are of sort *Set*. We write logical propositions for the statements of axioms and theorems. We write set specifications for the definition of mathematical notions. Sets are inhabited by their elements. Provable propositions are inhabited by proof objects, which are of no concern to the naive user.

Objects and types may be bound to a name through a definition. Names may also be declared with a type declaration, either with a global parameter declaration (for instance, axioms are names declared with a proposition as type), or with a local binding operator.

The main constructions may be briefly described as follows:

- names refer to declared variables or defined constants
- `(M N)` denotes the application of functional object M to object N.
- `[x:T]M` abstracts variable x of type T in construction M in order to construct a functional object $\lambda x \in T \cdot M$.
- `(x:T)P` as set specification, corresponds to a product type $\Pi x \in T \cdot P$, abbreviated as an arrow type $T \rightarrow P$ when x does not occur in P. As a proposition, it corresponds to universal quantification, i.e. $\forall x \in T \cdot P$. The arrow abbreviation in this case corresponds to logical implication.
- `Ind{X:S|C1; ... Cn}` defines inductively a set or a proposition, according to sort S, with n *constructors* of types C_1, ..., C_n.

Further cases refer to the constructors of an inductive type, and its elimination principles, corresponding to an induction principle for propositions, and a recursion principle for sets. These constructions are rarely used directly by the user. A higher level language, called the *mathematical vernacular*, permits to name these notions conveniently. For instance, the inductive set of natural numbers is described as:

```
Inductive Set nat = O : nat | S : nat -> nat.
```

This defines together the inductive set, bound to name `nat`, its two constructors, bound to names `O` and `S`, an induction principle of type the proposition:

```
(P:nat->Prop)(P O)->((x:nat)(P x)->(P (S x)))->(n:nat)(P n)
```

and a recursion principle, of type the specification:

```
(P:nat->Set)(P O)->((x:nat)(P x)->(P (S x)))->(n:nat)(P n)
```

A typical inductive predicate definition (here $\leq$) will look like:

```
Inductive Definition le [n:nat] : nat -> Prop =
      le_n : (le n n)
    | le_S : (m:nat)(le n m)->(le n (S m)).
```

Such a definition may be thought of as a (typed) Prolog specification. We may also write recursive definitions in a functional style:

```
Definition plus = [n,m:nat](<nat>Match n with
                                (* O *)     m
                                (* (S p) *) [p:nat]S).
```

which corresponds to the usual recursion equations for addition (the strings `(* ... *)` are just comments in this syntax).

The standard logical connectives, equality, and existential quantification, are definable as inductive notions. Indeed, the standard prelude of the system declares these basic notions, with their usual syntax. For instance, `P /\ Q` (resp. `P \/ Q`) expresses the conjunction (resp. disjunction) of propositions P and Q. The term `<nat>n=m` expresses equality of naturals n and m. It is actually shorthand for `(eq nat n m)`, where *eq* is the inductively-defined equality relation. In this case the unique constructor is reflexivity of equality, and the induction principle corresponds to the replacement of equals by equals. Such internal codings may be ignored in first approximation.

Thus, a typical mathematical axiomatisation in Coq will consist of a mixture of (higher-order) predicate calculus, non-deterministic specifications in the style of logic-programming, and recursive definitions in the style of functional-programming. Type-checking is done modulo definitional equality, which corresponds to replacing a defined constant by its definition, parameter substitution in functional applications (i.e. β-reduction in the underlying λ-calculus), and recursion unfolding (i.e. applying simplifications such as $(plus\ O\ m) \Rightarrow m$).

This blend of high-level notations leads to a smooth style of axiomatisation, relatively easy to master by anyone with a minimal training in formal methods, without requiring a complete understanding of the logical framework of the Calculus of Inductive Constructions.

2 A brief description of the theorem prover

Coq uses a top-down theorem prover which solves successive subgoals with tactics in the style of LCF[16]. The user formulates a conjecture with the command `Goal P`, where P is a well-formed proposition in the current context. The system then enters a proof-development mode, with a local top-level loop which at every step prints the remaining subgoals, and waits for a prover command in a syntax of tactics invocation. We shall briefly describe in this section the five families of tactics which are used in the proof below. It is also possible to interact with the prover through an interactive interface, displaying subgoals in windows on the screen of the user's workstation, and driven by user clicks on mouse-sensitive buttons. A full manual of the system is available in the standard distribution[12].

The first class of tactics concerns parameter introductions. The tactic `Intro H`, applied to a product goal of the form $A \rightarrow B$, enters $H : A$ in the current signature, or local context, and returns B as a subgoal. If the product is dependent, like $(x : A)(P\ x)$ (read $\forall x \in A \cdot P(x)$), we subgoal into $(P\ H)$. If the argument H is missing, the system will generate an appropriate name. If H exists already as a local hypothesis, the system complains. The tactic `Intros` iterates `Intro`, either with explicitly given names separated with blanks, or else as long as the current goal is a product.

The second class is that of resolution tactics. The tactic `Apply` tries to resolve the current goal with its argument, applied to terms, whose types become new goals to be solved. This corresponds to the refinement tactic of Nuprl[6]. This tactic may be helped by giving explicit instanciations, like in `Apply Lemma1 with (f x)`. Special cases of `Apply` are called `Left` (resp. `Right`) for the application of propositional *or* introduction to the left (resp. right). Also `Split` corresponds to *and* introduction, by splitting a goal of the form $A \wedge B$ into subgoals A and B.

The third class is the elimination tactics. They are the work horse of induction proofs. `Elim T`, when the term `T` has inductive type, does case analysis on the different constructors of this type. For instance, `Elim x`, when we have an hypothesis $x : A \wedge B$, will transform the current goal G into $A \rightarrow B \rightarrow G$. The `Induction` tactic combines introduction and elimination. Thus `Induction y` on a goal $(x : A)(y : B)(R\ x\ y)$ is equivalent to `Intros x y` followed by `Elim y`. The notation `Induction 2`, for instance, means do induction on the second un-named current hypothesis. A special tactic permits to do proofs by contradiction as follows. `Absurd P` replaces the current goal by the two goals P and $\sim P$. It is a special case of elimination of the absurd proposition, noted `{}`, combined with application of negation.

The fourth class is that of equality tactics. The `Simpl` tactics simplifies the current goal by putting it in normal form. This combines β-reduction of λ-calculus redexes, as well as primitive recursion on inductive types. The tactic `Unfold` unfolds the definition of the constant given as its argument. It applies to the first occurrence of the given constant in the current goal. If one wants to unfold the (say) third occurrence of constant C, it is possible to do it with `Unfold 3 C`. `Unfold * C` unfolds all the occurrences of constant C. Finally, `Change T` changes the current goal G by term T, after checking that G and T are interconvertible terms.

The fifth class comprises tactics which do some limited automatic search: The tactic `Trivial` tries to find the current goal either in the local hypotheses, or in a list of simplification axioms and lemmas. One enters a new such simplification with the tactic `Hint`. For instance, applications of reflexivity of equality are handled by `Trivial`. A more sophisticated theorem-proving tactic is the tactic `Auto`, which iterates such simplification to a nesting depth of 5; it implements a limited form

of automated deduction.

Tactics may be composed with the THEN tactical, concretely written as a semi-colon. Thus a command line is usually of the form: `Tactic 1; Tactic 2; ... Tactic n.` We remind the reader of the semantics of the tactical THEN: `Tactic 1; Tactic 2` applies Tactic 1 to the current goal, and then Tactic 2 to all the resulting subgoals.

The system allows selecting an arbitrary goal in the list of pending goals, by prefixing the command line by the number of the goal, followed by a colon. This facility is not used in this example, where the current goal is always the first of the list.

3 The proof

3.1 Booleans

3.1.1 $bool = \{true, false\}$, $|bool| = 2$.

The first line is the definition of booleans, as done in the prelude of the system:

```
Inductive Set bool = true : bool | false : bool.
```

Such an inductive definition defines several notions. First it defines the type corresponding to the inductively defined set `bool`, with its constructors `true` and `false`. Next it defines an induction principle `bool_ind`, of type `(P:bool->Prop)(P true)->(P false)->(b:bool)(P b)`, and a recursion principle `bool_rec`, of type `(P:bool->Set)(P true)->(P false)->(b:bool)(P b)`.

Pattern-matching by cases on the constructors of an inductive type T with n constructors, of types respectively $T_1 \rightarrow T$, ... , $T_n \rightarrow T$, may be written with the syntax `(<T'>Match e with e1 ... en)` where e is an expression of type T, and each e_i is of type respectively $T_i \rightarrow T'$.

For instance, here is the definition of a bool predicate `Is_true`, such that `(Is_true true)` is definitionally equal to the true proposition `True`, whereas `(Is_true false)` is definitionally equal to the false proposition `False`:

```
Definition Is_true = [b:bool](<Prop>Match b with
    (* true  *)  True
    (* false *)  False).
```

We show how to prove that $true = false$ is contradictory. We remind the reader of the syntax `<T>A=B` to state that elements A and B, of (set) type T, are equal.

```
Goal ~<bool>true=false.
Unfold not; Intro contr; Change (Is_true false).
```

At this point, the system prints the goal and local hypotheses as follows:

```
1 subgoal
  (Is_true false)
  ============================
    contr : <bool>true=false
```

The elimination tactic applied to hypothesis `contr` will substitute `true` for `false` in the goal. This is because equality is itself defined as an inductive predicate, and its associated induction principle amounts to Leibniz equality. By mere simplification the proof is completed:

```
Elim contr; Simpl; Trivial.
Save diff_true_false.
```

The last command `Save` enters the lemma `diff_true_false` into the global context, with type the original goal statement. The proof above is typical of proofs of "no confusion" properties. If A and B are two terms that we want to prove different, we define a caracteristic predicate P, such that $P(A)$ and $\sim P(B)$, we assume the confusion, i.e. $A = B$, convert contradiction into $P(B)$, use $A = B$ to convert it into $P(A)$, trivially true. Clearly, a general tactic could do such proofs in a uniform manner.

3.1.2 Negation

We now define the negation function, which maps `true` to `false` and conversely:

```
Definition neg = [b:bool](<bool>Match b with
    (* true *)  false
    (* false*)  true).
```

We proceed by proving a few easy properties of negation.

```
Goal (b:bool)<bool>(neg (neg b))=b.
Induction b; Simpl; Auto.
Save neg_intro.

Goal (b:bool)<bool>b=(neg (neg b)).
Induction b; Simpl; Trivial.
Save neg_elim.

Goal (b,b':bool)(<bool>b'=(neg b))->(<bool>b=(neg b')).
Induction b; Induction b'; Intros; Simpl; Trivial.
Save neg_sym.

Goal (b:bool)~<bool>(neg b)=b.
Induction b; Simpl; Unfold not; Intro; Apply diff_true_false; Auto.
Save no_fixpoint_neg.
```

3.2 Boolean words

3.2.1 Strings of bits

We now axiomatize boolean words. This set type is isomorphic to `(list bool)`.

```
Inductive Set word = empty : word
                   | bit : bool -> word -> word.
```

Next is the characteristic predicate of nonempty words:

```
Definition Nonempty = [w:word](<Prop>Match w with
    (* empty *)   False
    (* bit b w *) [b:bool][w:word][P:Prop]True).
```

Notice that here, since word is a genuinely recursive type definition, we get an extra argument P in the bit case, which corresponds to a recursive call to $Nonempty(w)$. Thus `Match` has really the full power of a recursion operateur.

3.2.2 Concatenation

We now define word concatenation. First, we give a "logical" definition, as an inductive ternary relation, in the style of Prolog:

```
Inductive Definition conc : word -> word -> word -> Prop =
  conc_empty : (v:word)(conc empty v v)
| conc_bit   : (u,v,w:word)(b:bool)(conc u v w)
                              ->(conc (bit b u) v (bit b w)).
```

Here is now word concatenation, as a functional definition, in the style of primitive recursion:

```
Definition Conc = [u,v:word](<word>Match u with
    (* empty *)   v
    (* bit b w *) [b:bool][w:word][conc_w_v:word](bit b conc_w_v)).
```

We may now relate the two definitions:

```
Goal (u,v,w:word)(conc u v w)-><word>w=(Conc u v).
Induction 1; Simpl; Trivial.
(*  (u,v,w:word)(b:bool)(conc u v w)->
       (<word>w=(Conc u v))->(<word>(bit b w)=(bit b (Conc u v))) *)
Induction 2; Trivial.
Save conc_Conc.
```

As an exercice, let us give a proof of the associativity of Conc. This property is not used below.

```
Goal (u,v,w:word)<word>(Conc u (Conc v w))=(Conc (Conc u v) w).
Induction u; Simpl; Intros; Auto.
(*  <word>(bit b (Conc y (Conc v w)))=(bit b (Conc (Conc y v) w))
  ============================
    w : word
    v : word
    H : (v:word)(w:word)(<word>(Conc y (Conc v w))=(Conc (Conc y v) w)) *)
Elim (H v w); Trivial.
Save assoc_Conc.
```

Clearly here, we see that it would be easy to increase our proof automation facilities, and get automatically such a proof from a simple indication "By induction on u." We may hope to progressively bridge the gap between our tactic control language, and a genuine mathematical vernacular expressing proof developments.

3.2.3 Singleton words

We end this section with the definition of singleton words, composed of just one bit.

```
Definition single = [b:bool](bit b empty).
```

3.3 Alternating words

A word w is *alternate* if for some bit b, w is of the form $[b\ \sim b\ b\ \sim b ...]$. We write `(alt b w)`.

3.3.1 Alt and its variants

```
Inductive Definition alt : bool -> word -> Prop =
  alt_empty   : (b:bool)(alt b empty)
| alt_bit     : (b:bool)(w:word)(alt (neg b) w)->(alt b (bit b w)).

Hint alt_empty alt_bit.
```

This `Hint` command is a pragma for the automated search tactics `Trivial` and `Auto`. It tells them to apply the clause `alt_bit` whenever possible, in order to simplify a goal of the form `(alt b (bit b w))` into `(alt (neg b) w)`. We now proceed with a more functional caracterisation of alternate.

```
Definition Alt = [b:bool][w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b':bool][w:word][P:Prop](alt (neg b) w)).
```

Same as previously, we want to "invert" our definition clauses. Here we have an extra difficulty, due to the non-linearity of the head of clause `alt_bit`: the variable b occurs twice in $(alt\ b\ (bit\ b\ w))$.

```
Goal (b,b':bool)(w:word)(alt b (bit b' w))->(alt (neg b) w).
Intros b b' w al.
Change (Alt b (bit b' w)).
Elim al; Simpl; Trivial.
Save alt_neg_intro.

Goal (b,b':bool)(w:word)(alt (neg b) (bit b' w))->(alt b w).
Intros; Elim (neg_intro b); Apply alt_neg_intro with b'; Trivial.
Save alt_neg_elim.
```

We now take care of the linearity condition. What we have done is to break the property into two: $(alt\ b\ w) \Leftrightarrow (Alt\ b\ w) \wedge (Alt'\ b\ w)$.

```
Definition Alt' = [b:bool][w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b':bool][w:word][P:Prop]<bool>b=b').

Goal (b,b':bool)(w:word)(alt b (bit b' w))-><bool>b=b'.
```

```
Intros b b' w al.
Change (Alt' b (bit b' w)).
Elim al; Simpl; Trivial.
Save alt_eq.

Goal (b,b':bool)(w:word)(alt b (bit b' w))->((<bool>b=b') /\ (alt (neg b) w)).
Intros; Split.
Apply alt_eq with w; Trivial.
Apply alt_neg_intro with b'; Trivial.
Save alt_back.
```

We end with the existence property: w is *alternate* if for some bool b we have $(alt\ b\ w)$. We use directly an inductive definition, rather than an explicit existential quantification, which is itself defined inductively anyway.

```
Inductive Definition alternate [w:word] : Prop =
  alter : (b:bool)(alt b w)->(alternate w).
```

3.4 Parities of words

We give here definitions of *odd* and *even* words. This is the corresponding property of their length, but we do this directly, without resorting to definition of length, and arithmetic facts. As previously, *Odd* (resp. *Even*) is the functional notion corresponding to the logical one *odd* (resp. *even*).

3.4.1 Odd and Even

```
Inductive Definition odd : word -> Prop =
  odd_single : (b:bool)(odd (single b))
| odd_bit    : (w:word)(odd w)->(b,b':bool)(odd (bit b (bit b' w))).

Definition Odd = [w:word](<Prop>Match w with
    (* empty *)   False
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match w with
         (* empty *)   True
         (* bit b w *) [b:bool][w:word][P:Prop](odd w))).

Inductive Definition even : word -> Prop =
  even_empty : (even empty)
| even_bit : (w:word)(even w)->(b,b':bool)(even (bit b (bit b' w))).

Hint even_empty.

Definition Even = [w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b:bool][w:word][P:Prop](<Prop>Match w with
         (* empty *)   False
```

```
        (* bit b w *) [b:bool][w:word][P:Prop](even w))).
```

3.4.2 Parity lemmas

```
Goal ~(odd empty).
Unfold not; Intro od.
Change (Nonempty empty).
Elim od; Simpl; Trivial.
Save not_odd_empty.

Hint not_odd_empty.

Goal (w:word)(b,b':bool)(odd (bit b (bit b' w)))->(odd w).
Intros w b b' od.
Change (Odd (bit b (bit b' w))).
Elim od; Simpl; Trivial.
Save odd_down.

Definition Not_single = [w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match w with
         (* empty *)   False
         (* bit b w *) [b:bool][w:word][P:Prop]True)).

Goal (b:bool)~(even (single b)).
Intro b; Unfold not; Intro ev.
Change (Not_single (single b)).
Elim ev; Simpl; Trivial.
Save not_even_single.

Goal (w:word)(b,b':bool)(even (bit b (bit b' w)))->(even w).
Intros w b b' ev.
Change (Even (bit b (bit b' w))).
Elim ev; Simpl; Trivial.
Save even_down.

Goal (w:word)(odd w)->(b:bool)(even (bit b w)).
Induction 1.
Intros b b'; Unfold single; Apply even_bit; Apply even_empty.
Intros; Apply even_bit; Auto.
Save odd_even.

Goal (w:word)(even w)->(b:bool)(odd (bit b w)).
Induction 1.
```

```
Intro b; Change (odd (single b)); Apply odd_single.
Intros; Apply odd_bit; Auto.
Save even_odd.

Hint odd_even even_odd.

Goal (w:word)(b:bool)(odd (bit b w))->(even w).
Induction w; Auto.
Intros; Apply odd_even; Apply odd_down with b0 b; Auto.
Save inv_odd.

Goal (w:word)(b:bool)(even (bit b w))->(odd w).
Induction w; Intros.
Absurd (even (single b)); Trivial.
Apply not_even_single.
Apply even_odd.
Apply even_down with b0 b; Trivial.
Save inv_even.
```

3.4.3 (odd w) ⊕ (even w)

In this section, we prove that *odd* and *even* are two disjoint complementary properties.

```
Goal (w:word)((odd w) \/ (even w)).
Induction w; Auto.
Induction 1; Intros.
Right; Auto.
Left; Auto.
Save odd_or_even.

Goal (w:word)(odd w)->(even w)->False.
Induction w; Intros.
Elim not_odd_empty; Trivial.
(*   False
  ============================
    H1 : (even (bit b y))
    H0 : (odd (bit b y))
    H : (odd y)->(even y)->False *)
Apply H.
Apply inv_even with b; Trivial.
Apply inv_odd with b; Trivial.
Save not_odd_and_even.
```

3.4.4 Parities of subwords

When we cut an even word w into subwords u and v, they are both even or both odd. But if w is odd, one is odd and the other even. This easy consequence of `odd_or_even` could be automated with a simple propositional tactic.

```
Goal (u,v,w:word)(conc u v w)->
           (((odd w) /\
                    ( ((odd u) /\ (even v))
                   \/ ((even u) /\ (odd v))))
         \/ ((even w) /\
                    ( ((odd u) /\ (odd v))
                   \/ ((even u) /\ (even v))))).
Induction 1; Intros.
Elim (odd_or_even v0); Intro.
Left; Split; Auto.
Right; Split; Auto.
Elim H1; Intros.
(* 1 (odd w0) *)
Right; Elim H2; Intros.
Split; Auto.
Elim H4; Intros.
Right; Split; Elim H5; Auto.
Left; Split; Elim H5; Auto.
(* 2 (even w0) *)
Left; Elim H2; Intros.
Split; Auto.
Elim H4; Intros.
Right; Split; Elim H5; Auto.
Left; Split; Elim H5; Auto.
Save odd_even_conc.
```

We are specially interested in the even case, which we explicitate below. This is typical: it is often simpler to prove a general disjunctive statement by induction, and then to particularise it using the fact that some cases are disjoint.

```
Goal (u,v,w:word)(conc u v w)->(even w)->
                    ( ((odd u) /\ (odd v))
                   \/ ((even u) /\ (even v))).
Intros u v w c e; Elim odd_even_conc with u v w; Intros.
Elim H; Intro o; Elim not_odd_and_even with w; Auto.
Elim H; Intros; Trivial.
Trivial.
Save even_conc.
```

3.4.5 Subwords of alternate words are alternate

```
Goal (u,v,w:word)(conc u v w)->(b:bool)(alt b w)->(alt b u).
```

```
Induction 1; Auto; Intros.
(*  (alt b0 (bit b u0)) *)
Elim alt_back with b0 b w0.
Intros eq A.
(*   (alt b0 (bit b u0))
  ============================
    A : (alt (neg b0) w0)
    eq : <bool>b0=b *)
Elim eq; Auto.
Trivial.
Save alt_conc_l.
```

This was simple enough for the prefix part. For the suffix part, we have a parity consideration:

```
Goal (u,v,w:word)(conc u v w)->(b:bool)(alt b w)->
   ( ((odd u) /\ (alt (neg b) v))
  \/ ((even u) /\ (alt b v))).
Induction 1; Intros.
Right; Split; Intros; Auto.
(*  ((odd (bit b u0)) /\ (alt (neg b0) v0))
  \/ (even (bit b u0)) /\ (alt b0 v0)
  ============================
    H2 : (alt b0 (bit b w0))
    b0 : bool
    H1 : (b:bool)(alt b w0)->
          (((odd u0) /\ (alt (neg b) v0)) \/ (even u0) /\ (alt b v0)) *)
Elim H1 with (neg b0).
Elim neg_elim with b0; Intro.
Right; Split; Elim H3; Auto.
Intro; Left; Split; Elim H3; Auto.
Apply alt_neg_intro with b; Trivial.
Save alt_conc_r.
```

Putting the two together:

```
Goal (u,v,w:word)(conc u v w)->(alternate w)->((alternate u) /\ (alternate v)).
Induction 2; Intros b ab; Split.
Apply alter with b; Apply alt_conc_l with v w; Auto.
Elim alt_conc_r with u v w b; Intros; Trivial.
Elim H1; Intros; Apply alter with (neg b); Auto.
Elim H1; Intros; Apply alter with b; Auto.
Save alt_conc.
```

This last lemma is actually not used below.

3.5 Opposite words

Two words are said to be *opposite* if they start with different bits:

```
Inductive Definition opposite : word -> word -> Prop =
  opp : (u,v:word)(b:bool)(opposite (bit b u) (bit (neg b) v)).

Definition Opp = [u:word][v:word](<Prop>Match u with
    (* empty *)   False
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match v with
         (* empty *)   False
         (* bit b w *) [b':bool][w':word][P':Prop]<bool>(neg b)=b')).

Goal (u:word)~(opposite u empty).
Unfold not; Intros u op.
Change (Nonempty empty).
Elim op; Simpl; Trivial.
Save not_opp_empty_r.

Goal (u:word)~(opposite empty u).
Unfold not; Intros u op.
Change (Nonempty empty).
Elim op; Simpl; Trivial.
Save not_opp_empty_l.

Goal (u,v:word)(b:bool)~(opposite (bit b u) (bit b v)).
Unfold not; Intros u v b op.
Apply (no_fixpoint_neg b).
Change (Opp (bit b u) (bit b v)).
Elim op; Simpl; Trivial.
Save not_opp_same.

Goal (u,v:word)(b:bool)(odd u)->(alt b u)->
                (odd v)->(alt (neg b) v)->(opposite u v).
Induction u.
Intros v b odd_empty;
Absurd (odd empty); Trivial.
Intros b u' H v; Elim v.
Intros b' H1 H2 odd_empty.
Absurd (odd empty); Trivial.
Intros b' v' H' b'' H1 H2 H3 H4.
(*  (opposite (bit b u) (bit b' v'))
  ============================
    H4 : (alt (neg b'') (bit b' v'))
    H3 : (odd (bit b' v'))
```

```
    H2 : (alt b'' (bit b u')) *)
Elim (alt_eq (neg b'') b' v'); Trivial.
Elim (alt_eq b'' b u'); Trivial.
Apply opp.
Save opposite1.

Goal (u,v:word)(b:bool)(alt b u)->(alt b v)->~(opposite u v).
Induction u.
Intros; Apply not_opp_empty_l.
Intros b u' H v; Elim v.
Intros; Apply not_opp_empty_r.
Intros b' v' H1 b'' H2 H3.
(*   ~(opposite (bit b u') (bit b' v'))
  ============================
    H3 : (alt b'' (bit b' v'))
    H2 : (alt b'' (bit b u')) *)
Elim (alt_eq b'' b' v' H3).
Elim (alt_eq b'' b u' H2).
Apply not_opp_same.
Save opposite2.
```

3.6 Paired words

A word w is said to be *paired* if it is of the form: $[b_1\ {\sim}b_1\ b_2\ {\sim}b_2\ ...\ b_n\ {\sim}b_n]$.

```
Inductive Definition paired : word -> Prop =
  paired_empty : (paired empty)
| paired_bit : (w:word)(paired w)->(b:bool)(paired (bit (neg b) (bit b w))).
```

Here again we have a non-linear clause. We shall give several variants of "paired". A paired word must be even. For odd words, we give below variants `paired_odd_l` and `paired_odd_r`.

$$(paired\_odd\_l\ b\ w) \Leftrightarrow w = [b\ b_1\ {\sim}b_1\ b_2\ {\sim}b_2\ ...\ b_n\ {\sim}b_n].$$

```
Definition paired_odd_l = [b:bool][w:word](paired (bit (neg b) w)).

Goal (b:bool)(w:word)(paired w)->(paired_odd_l b (bit b w)).
Unfold paired_odd_l; Intros.
Apply paired_bit; Trivial.
Save paired_odd_l_intro.

Goal (b:bool)(w:word)(paired_odd_l (neg b) w)->(paired (bit b w)).
Unfold paired_odd_l; Intros.
Elim (neg_intro b); Trivial.
Save paired_odd_l_elim.
```

Similarly:

$$(paired\_odd\_r\ b\ w) \Leftrightarrow w = [b_1\ \sim b_1\ b_2\ \sim b_2\ ...\ b_n\ \sim b_n\ \sim b].$$

```
Definition paired_odd_r = [b:bool][w:word](paired (Conc w (single b))).
```

An even word is *paired rotated* iff rotating it by one bit makes it paired:

$$(paired\_rot\ b\ w) \Leftrightarrow w = [b\ b2\ \sim b2\ ...\ bn\ \sim bn\ \sim b].$$

```
Inductive Definition paired_rot : bool -> word -> Prop =
  paired_rot_empty : (b:bool)(paired_rot b empty)
| paired_rot_bit : (b:bool)(w:word)(paired_odd_r b w)
                                  ->(paired_rot b (bit b w)).

Goal (w:word)(b:bool)(paired_rot b w)->(paired_odd_r b (bit (neg b) w)).
Induction 1.
Intro; Unfold paired_odd_r; Simpl.
Unfold single; Apply paired_bit.
Apply paired_empty.
Intros b0 b' w'; Unfold paired_odd_r; Intros.
Simpl; Apply paired_bit; Auto.
Save paired_odd_r_from_rot.
```

Finally, a word is said to be *paired between* if it is obtained by prefixing and suffixing a paired word with the same bit b:

$$(paired\_bet\ b\ w) \Leftrightarrow w = [b\ b_1\ \sim b_1\ b_2\ \sim b_2\ ...\ b_n\ \sim b_n\ b].$$

```
Inductive Definition paired_bet [b:bool] :  word -> Prop =
  paired_bet_bit : (w:word)(paired_odd_r (neg b) w)->(paired_bet b (bit b w)).

Goal (b:bool)(w:word)(paired_bet (neg b) w)->(paired_odd_r b (bit b w)).
Intros b w pb.
Elim (neg_intro b).
Elim pb.
Unfold * paired_odd_r. (* Unfold twice *)
Intros; Simpl.
Apply paired_bit; Trivial.
Save paired_odd_r_from_bet.
```

3.7 Shuffling two words

Here we come to our main notion: `(shuffle u v w)`, meaning word w may be obtained by shuffling words u and v. We deal here with a truly non-deterministic specification.

3.7.1 Random shuffle

```
Inductive Definition shuffle : word -> word -> word -> Prop =
  shuffle_empty :    (shuffle empty empty empty)
| shuffle_bit_left : (u,v,w:word)(shuffle u v w) ->
                              (b:bool)(shuffle (bit b u) v (bit b w))
| shuffle_bit_right : (u,v,w:word)(shuffle u v w) ->
                              (b:bool)(shuffle u (bit b v) (bit b w)).
```

3.7.2 The shuffling lemma

This lemma is the main result of this note. It gives the inductive invariant associated with the shuffling of alternated words.

```
Goal (u,v,w:word)(shuffle u v w)->(b:bool)(alt b u)->
  ( ((odd u) /\ ( ((odd v)  /\ ((alt (neg b) v) -> (paired w))
                            /\ ((alt b v)       -> (paired_bet b w)))
              \/ ((even v) /\ ((alt b v)       -> (paired_odd_l b w)))
                            /\ ((alt (neg b) v) -> (paired_odd_r (neg b) w))))
 \/ ((even u) /\ ( ((odd v) /\ ((alt (neg b) v) -> (paired_odd_r b w))
                            /\ ((alt b v)       -> (paired_odd_l b w)))
              \/ ((even v) /\ ((alt b v)       -> (paired_rot b w))
                            /\ ((alt (neg b) v) -> (paired w))))))).
Induction 1; Intros.
(* 0. empty case *)
Right.
Split; Auto.
Right.
Split; Auto.
Split; Intro.
Apply paired_rot_empty.
Apply paired_empty.
(* 1. u before v *)
Elim (alt_eq b0 b u0); Trivial.
Elim (H1 (neg b0)); Intros.
(* 1.1 *) Right.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
Elim H6; Intros.
(* 1.1.1 *) Left.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Apply paired_odd_r_from_bet; Auto.
Apply paired_odd_l_intro; Apply H9; Elim (neg_elim b0); Auto.
```

```
Elim H6;Intros.
Elim H7; Intros.
(* 1.1.2 *) Right.
Split; Auto.
Split; Intro.
Apply paired_rot_bit; Elim (neg_intro b0); Apply H8; Elim (neg_elim b0); Auto.
Apply paired_odd_1_elim; Auto.
(* 1.2 *) Left.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 1.2.1 *) Left.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Apply paired_odd_1_elim; Auto.
Apply paired_bet_bit; Apply H9; Elim (neg_elim b0); Auto.
(* 1.2.2 *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split.
Split; Auto.
Intro.
Apply paired_odd_1_intro; Apply H10; Elim (neg_elim b0); Auto.
Intro; Pattern 2 b0; Elim (neg_intro b0).
Apply paired_odd_r_from_rot; Auto.
Apply alt_neg_intro with b; Trivial.
(* 2. v before u *)
Elim (H1 b0); Intros.
(* 2.1 *) Left.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 2.1.1 *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split.
Split; Auto.
Intro.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_odd_1_intro; Apply H9; Apply alt_neg_intro with b; Auto.
Intro.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_r_from_bet; Elim (neg_elim b0); Apply H10;
```

```
 Apply alt_neg_elim with b; Auto.
(* 2.1.2 *) Left.
Elim H6; Intros.
Elim H7; Intros.
Split; Auto.
Split; Intros.
Apply paired_odd_1_elim.
Elim (alt_eq (neg b0) b v0); Trivial.
Elim (neg_elim b0).
Apply H10.
Elim (neg_intro b0).
Apply alt_neg_intro with b; Auto.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_bet_bit; Apply H8; Apply alt_neg_intro with b; Auto.
(* 2.2 *) Right.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 2.2.1 *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intros.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_rot_bit; Apply H9; Apply alt_neg_intro with b; Auto.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_1_elim.
Elim (neg_elim b0); Apply H10; Elim (neg_intro b0);
 Apply alt_neg_intro with b; Auto.
(* 2.2.2 *) Left.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intros.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_r_from_rot; Apply H9; Elim (neg_intro b0);
 Apply alt_neg_intro with b; Auto.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_odd_1_intro; Apply H10; Apply alt_neg_intro with b; Auto.
Trivial.
Save Shuffling.
```

Remark. This proof has the natural propositional structure to examine the eight cases. There are thus eight trivial cases to examine in the base case (shuffling two empty lists), and eight sub-cases in each of the two non-degenerate cases of `shuffle_bit`. We thus appeal to sixteen sub-lemmas, corresponding to "verification conditions" in Floyd-Hoare terminology, the "program" being "evi-

dence" that the result of the shuffle of subwords of an even alternated word is paired. These sixteen verification conditions reduce to the six lemmas: `paired_odd_l_intro` and `paired_odd_l_elim` (each used four times), `paired_rot_bit`, `paired_bet_bit`, `paired_odd_r_from_bet`, and `paired_odd_r_f` (each used twice). Now the reader may wonder by what stroke of luck we just developed the right lemmas above. Of course the development of the proof went top-down, not only subgoaling locally in each lemma, but globally as well. I.e., the statements of the above six lemmas were precisely generated by the proof of `Shuffling`. This is extremely easy in our system, where we may at any point of the proof declare a new axiom, which gets inserted in the global context and is thus usable instantly. A second pass changes these "axioms" into goals themselves to be proven.

3.8 The main theorem

3.8.1 Rotating a word

We express the `paired_rot` property as the composition of a rotating operation `rotate` and of the property `paired`.

$$rotate\ [b_1\ b_2\ \ldots\ b_n] = [b_2\ \ldots\ b_n\ b_1].$$

```
Definition rotate [w:word](Elim{w,word}
   (* empty *)    empty
   (* bit b u *)  [b:bool][u,v:word](Conc u (single b))).

Goal (w:word)(b:bool)(paired_rot b w)->(paired (rotate w)).
Induction 1.
Intro; Simpl; Apply paired_empty.
Intros b' w'; Simpl.
Unfold paired_odd_r; Trivial.
Save paired_rotate.
```

We now prove the main theorem of this note. We use the section mechanism in order to state our hypotheses in the global context and shorten the quantification prefixes.

3.8.2 Stating the hypotheses of the main result

```
Section Main.
Variable x:word.
Hypothesis Even_x : (even x).
Variable b:bool. (* witness for (alternate x) *)
Hypothesis A : (alt b x).
Variables u,v:word.
Hypothesis C : (conc u v x).
Variable w:word.
Hypothesis S : (shuffle u v w).

Goal (alt b u).
Apply alt_conc_l with v x.
Apply C.
```

```
Apply A.
Save Alt_u.
```

3.8.3 Case 1: u is odd

```
Section Case1.
Hypothesis Odd_u : (odd u).

Goal ~(even u).
Red; Intro.
Elim not_odd_and_even with u; Trivial.
Apply Odd_u.
Save Not_even_u.

Goal (odd v).
Elim even_conc with u v x.
Intro H; Elim H; Trivial.
Intro H; Elim H; Intro; Elim Not_even_u; Trivial.
Apply C.
Apply Even_x.
Save Odd_v.

Goal (alt (neg b) v).
Elim alt_conc_r with u v x b.
Intro H; Elim H; Trivial.
Intro H; Elim H; Intro; Elim Not_even_u; Trivial.
Apply C.
Apply A.
Save Alt_v.

Goal (opposite u v).
Apply opposite1 with b.
Apply Odd_u.
Apply Alt_u.
Apply Odd_v.
Apply Alt_v.
Save Opp_uv.

Goal (paired w).
Elim Shuffling with u v w b.
Intro H; Elim H; Intros H1 H2; Elim H2; Intros.
Elim H0; Intros.
Elim H4; Intros.
Apply H5.
Apply Alt_v.
```

```
Intros; Elim H0; Intros.
Elim H3; Intros.
Elim not_odd_and_even with v; Trivial.
Apply Odd_v.
Intro H; Elim H; Intro; Elim Not_even_u; Trivial.
Apply S.
Apply Alt_u.
Save Case1.
End Case1.
```

3.8.4 Case 2: u is even

```
Section Case2.
Hypothesis Even_u : (even u).

Goal ~(odd u).
Red; Intro; Elim not_odd_and_even with u; Trivial.
Apply Even_u.
Save Not_odd_u.

Goal (even v).
Elim even_conc with u v x.
Intro H; Elim H; Intro; Elim Not_odd_u; Trivial.
Intro H; Elim H; Trivial.
Apply C.
Apply Even_x.
Save Even_v.

Goal (alt b v).
Elim alt_conc_r with u v x b.
Intro H; Elim H; Intro; Elim Not_odd_u; Trivial.
Intro H; Elim H; Trivial.
Apply C.
Apply A.
Save Alt_v.

Goal ~(opposite u v).
Apply opposite2 with b.
Apply Alt_u.
Apply Alt_v.
Save Not_opp_uv.

Goal (paired (rotate w)).
Apply paired_rotate with b.
Elim Shuffling with u v w b.
```

```
Intro case_odd_u; Elim case_odd_u; Intro; Elim Not_odd_u; Trivial.
Intro case_even_u; Elim case_even_u; Intros even_u odd_even_v; Elim odd_even_v.
Intro odd_v; Elim odd_v; Intros; Elim not_odd_and_even with v; Trivial.
Apply Even_v.
Intro case_even_v; Elim case_even_v; Intros even_v paired_hyp.
Elim paired_hyp; Intros case_b case_neg_b.
Apply case_b.
Apply Alt_v.
Apply S.
Apply Alt_u.
Save Case2.

End Case2.
```

3.8.5 Putting it all together

We recall the conditional connective, from the prelude, with its mixfix syntax:

```
Definition IF = [P,Q,R:Prop](P /\ Q) \/ ((~P) /\ R).
Syntax IF "if _ then _ else _".

Goal if (opposite u v) then (paired w) else (paired (rotate w)).
Unfold IF; Elim odd_or_even with u; Intros.
(* (odd u) : Case 1 *)
Left; Split.
Apply Opp_uv; Trivial.
Apply Case1; Trivial.
(* (even u) : Case 2 *)
Right; Split.
Apply Not_opp_uv; Trivial.
Apply Case2; Trivial.
Save Main.

End Main.
```

At this point all the local hypotheses are discharged into a general Main theorem explicitly universally quantified.

3.9 Gilbreath's trick

All is left to do is to hide the existential introduction of hypothesis *b* above, using the `alternate` property:

```
Goal (x:word)(even x)->(alternate x)->(u,v:word)(conc u v x)->
     (w:word)(shuffle u v w)->
     if (opposite u v) then (paired w) else (paired (rotate w)).
```

```
Induction 2; Intros. (* Existential intro *)
Apply Main with x b; Trivial.
Save Gilbreath_trick.
```

This Concludes our proof of Gilbreath's card trick.

4 The trick

Why is this a card trick? Our boolean words are card decks, with true for red and false for black. Take an even deck x, arranged alternatively red, black, red, black, etc. Ask a spectator to cut the deck, into sub-decks u and v. Now shuffle u and v into a new deck w. When shuffling, note carefully whether u and v start with opposite colors or not. If they do, the resulting deck is composed of pairs red-black or black-red; otherwise, you get the property by first rotating the deck by one card. The trick is usually played by putting the deck behind your back after the shuffle, to perform "magic". The magic is either rotating or doing nothing. When showing the pairing property, say loudly "red black red black..." in order to confuse in the spectator's mind the weak *paired* property with the strong *alternate* one.

There is a variant. If the cut is favorable, that is if u and v are opposite, just go ahead showing the pairing, without the "magic part." If the spectator says that he understands the trick, show him the counter-example in the non-favorable case. Of course now you have to leave him puzzled, and refuse to redo the trick.

References

[1] R. Boyer, J Moore. "A Computational Logic." Academic Press (1979).

[2] N.G. de Bruijn. "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29–61.

[3] N.G. de Bruijn. "Automath a language for mathematics." Les Presses de l'Université de Montréal, (1973).

[4] N.G. de Bruijn. "A survey of the project Automath." (1980) in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[5] N.G. de Bruijn. "A riffle shuffle card trick and its relation to quasi crystal theory." Nieuw Archief Voor Wiskunde **5** 3 (1987) 285–301.

[6] R.L. Constable et al. "Implementing Mathematics in the Nuprl System." Prentice-Hall (1986).

[7] R.L. Constable and N.P. Mendler. "Recursive Definitions in Type Theory." In Proc. Logic of Programs, Springer-Verlag Lecture Notes in Computer Science **193** (1985).

[8] Th. Coquand. "Une théorie des constructions." Thèse de troisième cycle, Université Paris VII (Jan. 85).

[9] T. Coquand. "Metamathematical Investigations of a Calculus of Constructions." Rapport de recherche INRIA 1088, Sept. 89. In "Logic and Computer Science," ed. P. Odifreddi, Academic Press, 1990, 91--122.

[10] Th. Coquand, G. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[11] T. Coquand and C. Paulin-Mohring. "Inductively defined types." Workshop on Programming Logic, Göteborg University, Båstad, (89). International Conference on Computer Logic COLOG-88, Tallinn, Dec. 1988. LNCS 417, P. Martin-Löf and G. Mints eds., pp. 50-66.

[12] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, B. Werner. "The system Coq, V5.6, User's guide." INRIA Technical Report, to appear, Sept. 1991.

[13] M. Gardner. Mathematical Recreation column, Scientific American, Aug. 1960.

[14] M. Gardner. Chapter 9, "New Mathematical Diversions from Scientific American." George Allen and Unwin Ltd, London, 1966. Reprinted Simon and Schuster, 1971.

[15] N. Gilbreath. "Magnetic Colors." The Linking Ring, **38** 5 (1959).

[16] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF." Springer-Verlag LNCS **78** (1979).

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249 - 6399