Formal Proof—Getting Started

Freek Wiedijk

A List of 100 Theorems

Today highly nontrivial mathematics is routinely being encoded in the computer, ensuring a reliability that is orders of a magnitude larger than if one had just used human minds. Such an encoding is called a *formalization*, and a program that checks such a formalization for correctness is called a *proof assistant*.

Suppose you have proved a theorem and you want to make certain that there are no mistakes in the proof. Maybe already a couple of times a mistake has been found and you want to make sure that that will not happen again. Maybe you fear that your intuition is misleading you and want to make sure that this is not the case. Or maybe you just want to bring your proof into the most pure and complete form possible. We will explain in this article how to go about this.

Although formalization has become a routine activity, it still is labor intensive. Using current technology, a formalization will be roughly four times the size of a corresponding informal LaTeX proof (this ratio is called the *de Bruijn factor*), and it will take almost a full week to formalize a single page from an undergraduate mathematics textbook.

The first step towards a formalization of a proof consists of deciding which proof assistant to use. For this it is useful to know which proof assistants have been shown to be practical for formalization. On the webpage [1] there is a list that keeps track of the formalization status of a hundred well-known theorems. The first few entries on that list appear in Table 1.

Freek Wiedijk is lecturer in computer science at the Radboud University Nijmegen, The Netherlands. His email address is freek@cs.ru.nl.

On the webpage [1] only eight entries are listed for the first theorem, but in [2] seventeen formalizations of the irrationality of $\sqrt{2}$ have been collected, each with a short description of the proof assistant.

When we analyze this list of theorems to see what systems occur most, it turns out that there are five proof assistants that have been significantly used for formalization of mathematics. These are:

proof assistant	*number of theorems formalized*
HOL Light	69
Mizar	45
ProofPower	42
Isabelle	40
Coq	39
all together	80

Currently in all systems together 80 theorems from this list have been formalized. We expect to get to 99 formalized theorems in the next few years, but Fermat's Last Theorem is the 33rd entry of the list and therefore it will be some time until we get to 100.

If we do not look for quantity but for quality, the most impressive formalizations up to now are:

Gödel's First Incompleteness Theorem: by Natarajan Shankar using the proof assistant nqthm in 1986, by Russell O'Connor using Coq in 2003, and by John Harrison using HOL Light in 2005.

Jordan Curve Theorem: by Tom Hales using HOL Light in 2005, and by Artur Korniłowicz using Mizar in 2005.

Prime Number Theorem: by Jeremy Avigad using Isabelle in 2004 (an elementary proof by Atle Selberg and Paul Erdös), and by

theorem		*number of systems in which the theorem has been formalized*
1.	The Irrationality of $\sqrt{2}$	≥ 17
2.	Fundamental Theorem of Algebra	4
3.	The Denumerability of the Rational Numbers	6
4.	Pythagorean Theorem	6
5.	Prime Number Theorem	2
6.	Gödel's Incompleteness Theorem	3
7.	Law of Quadratic Reciprocity	4
8.	The Impossibility of Trisecting the Angle and Doubling the Cube	1
9.	The Area of a Circle	1
10.	Euler's Generalization of Fermat's Little Theorem	4
11.	The Infinitude of Primes	6
12.	The Independence of the Parallel Postulate	0
13.	Polyhedron Formula	1
…		…

Table 1. The start of the list of 100 theorems [1].

John Harrison using HOL Light in 2008 (a proof using the Riemann zeta function).

Four-Color Theorem: by Georges Gonthier using Coq in 2004.

All but one of the systems used for these four theorems are among the five systems that we listed. This again shows that currently these are the most interesting for formalization of mathematics.

Here are the *proof styles* that one finds in these systems:

proof assistant	*proof style of the system*
HOL Light	procedural
Mizar	declarative
ProofPower	procedural
Isabelle	both possible
Coq	procedural

A *declarative* system is one in which one writes a proof in the normal way, although in a highly stylized language and with very small steps. For this reason a declarative formalization resembles program source code more than ordinary mathematics. In a *procedural* system one does not write proofs at all. Instead one holds a dialogue with the computer. In that dialogue the computer presents the user with *proof obligations* or *goals*, and the user then executes *tactics*, which reduce a goal to zero or more new, and hopefully simpler, *subgoals*. Proof in a procedural system is an interactive game.

In this paper we will show HOL Light as the example of a procedural system, and Mizar as the example of a declarative system.

The main advantage of HOL Light is its elegant architecture, which makes it a very powerful and reliable system. A proof of the correctness of the 394 line HOL Light "logical core" even has been formalized. On the other hand HOL has the disadvantage that it sometimes cannot express abstract mathematics—mostly when it involves algebraic structures—in an attractive way. It *can* essentially express all abstract mathematics though. Another disadvantage of HOL is that the proof parts of the HOL scripts are unreadable. They can only be understood by executing them on the computer.

Mizar on the other hand allows one to write abstract mathematics very elegantly, and its scripts are almost readable like ordinary mathematics. Also Mizar has *by far* the largest library of already formalized mathematics (currently it is over 2 million lines). However, Mizar has the disadvantage that it is not possible for a user to automate recurring proof patterns, and the proof automation provided by the system itself is rather basic. Also, in Mizar it is difficult to express the formulas of calculus in a recognizable style. It is not possible to "bind" variables, which causes expressions for constructions like sums, limits, derivatives, and integrals to look unnatural.

The Example: Quadratic Reciprocity

In this article we will look at two formalizations of a specific theorem. For this we will take the *Law of Quadratic Reciprocity*, the seventh theorem from the list of a hundred theorems. This theorem has thus far been formalized in four systems: by David Russinoff using nqthm in 1990, by Jeremy Avigad using Isabelle in 2004, by John Harrison using HOL Light in 2006, and by Li Yan, Xiquan Liang, and Junjie Zhao using Mizar in 2007.

When I was a student, my algebra professor Hendrik Lenstra always used to say that the Law of Quadratic Reciprocity is the first nontrivial theorem that a student encounters in the mathematics curriculum. Before this theorem, most proofs can be found without too much trouble by expanding the definitions and thinking hard. In contrast the Law of Quadratic Reciprocity is the first theorem that is totally unexpected. It was already conjectured by Euler and Legendre, but was proved only by the "Prince of Mathematicians", Gauss, who called it

the *Golden Theorem* and during his lifetime gave eight different proofs of it.

The Law of Quadratic Reciprocity relates whether an odd prime p is a square modulo an odd prime q, to whether q is a square modulo p. The theorem says that these are equivalent unless both p and q are 3 modulo 4, in which case they have opposite truth values. There also are two *supplements* to the Law of Quadratic Reciprocity, which say that -1 is a square modulo an odd prime p if and only if $p \equiv 1 \pmod 4$, and that 2 is a square modulo an odd prime p if and only if $p \equiv \pm 1 \pmod 8$.

The Law of Quadratic Reciprocity is usually phrased using the *Legendre symbol*. A number a is called a *quadratic residue* modulo p if there exists an x such that $x^2 \equiv a \pmod p$. The Legendre symbol $\left(\frac{a}{p}\right)$ for a coprime to p then is defined by

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue modulo } p \\ -1 & \text{otherwise.} \end{cases}$$

Using the Legendre symbol, the Law of Quadratic Reciprocity can be written as:

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}\frac{q-1}{2}}$$

The right hand side will be -1 if and only if both p and q are $3 \pmod 4$.

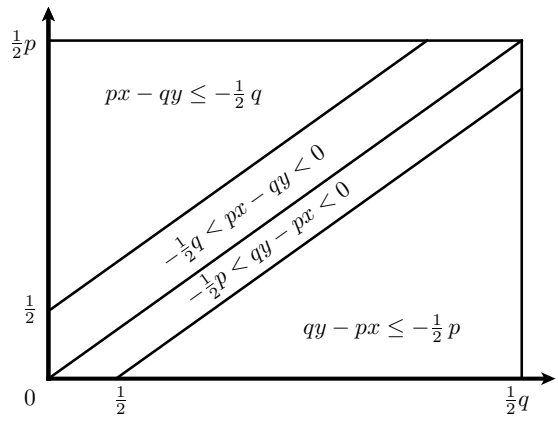
There are many proofs of the Law of Quadratic Reciprocity [3]. The formalizations shown here both formalize elementary counting proofs that go back to Gauss' third proof. We will not give details of these proofs here, but will just outline the main steps, following [4]. First by using that only half of the residues modulo p can be a square, one proves *Euler's criterion*:

$$\left(\frac{a}{p}\right) \equiv a^{\frac{1}{2}(p-1)} \pmod p$$

Using this criterion, by calculating the product of $a, 2a, \ldots, \frac{1}{2}(p-1)a$ in two different ways, one then proves *Gauss' lemma*, which says that

$$\left(\frac{a}{p}\right) = (-1)^l,$$

in which l is the number of $1 \le j \le \frac{1}{2}(p-1)$ for which there is an $-\frac{1}{2}p < a' < 0$ such that $aj \equiv a' \pmod p$. Finally one uses Gauss' lemma to derive the Law of Quadratic Reciprocity by counting lattice points in the four regions of the following figure:



HOL Light

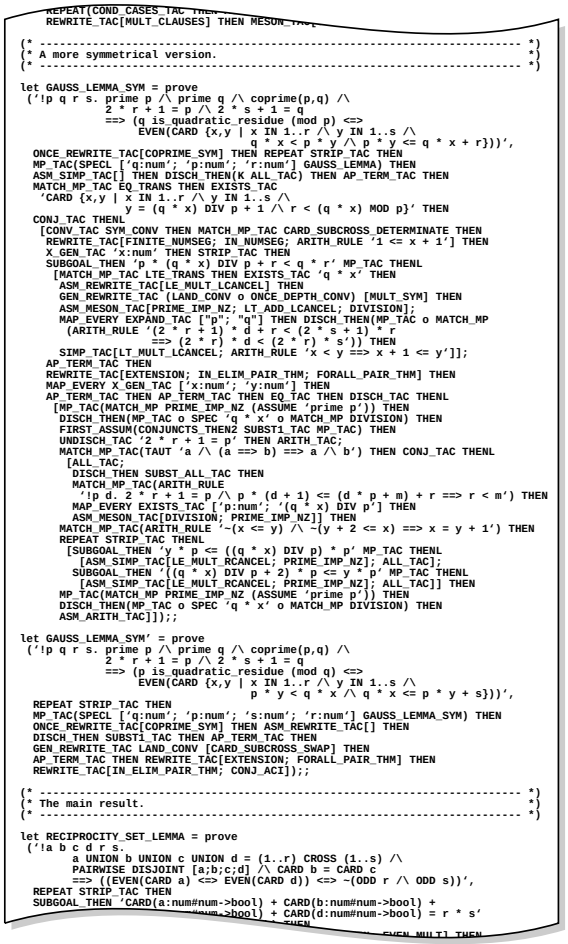
Suppose that we select HOL Light as our proof assistant. The second step will be to download and install the system. This does not take long. First download the ocaml compiler from [5] and install it. Next download the `tar.gz` file with the current version of the HOL Light sources from [6] and unpack it. Then follow the installation instructions in the `README` file. If you use Linux or Mac OS X, all you will need to do is type "`make`". Under Windows, installation is a bit more involved: you will have to copy the "`pa_j_....ml`" file that corresponds to your version of ocaml as given by "`ocamlc -version`" to a file called "`pa_j.ml`", and then compile that copy using one of the two "`ocamlc -c`" commands that are in the `Makefile`.

When you have installed the system, start the ocaml interpreter by typing "`ocaml`", and then enter the command "`#use "hol.ml";;`" This checks and loads the basic library of HOL Light, which takes a few minutes. After that you can load HOL files by typing for example "`loadt "100/reciprocity.ml";;`".

The third step will be to write a formalization of the proof. For this you will have to learn the HOL proof language. To do this it is best to study two documents: the HOL Light manual [7] and the HOL Light tutorial [8].

Instead of taking this third step and describing how one writes a formalization, here we will just look at a formalization that already exists. It can be found in the file "`100/reciprocity.ml`" (see Figure 1) and formalizes the proof from [4]. This file can also be found on the Web by itself as [9]. It consists of 753 lines of HOL Light code and proves 41 lemmas on top of the already existing HOL Light mathematical libraries. The statement of the final lemma is:

```
!p q. prime p /\ prime q /\
  ODD p /\ ODD q /\ ~(p = q)
  ==> legendre(p,q) * legendre(q,p) =
      --(&1) pow ((p - 1) DIV 2 * (q - 1) DIV 2)
```



```
    REPEAT(COND_CASES_TAC THEN
    REWRITE_TAC[MULT_CLAUSES] THEN MESON_TAC);;

(* ------------------------------------------------------------------------- *)
(* A more symmetrical version.                                               *)
(* ------------------------------------------------------------------------- *)

let GAUSS_LEMMA_SYM = prove
 (`!p q r s. prime p /\ prime q /\ coprime(p,q) /\
             2 * r + 1 = p /\ 2 * s + 1 = q
             ==> (q is_quadratic_residue (mod p) <=>
                  EVEN(CARD {x,y | x IN 1..r /\ y IN 1..s /\
                                   q * x < p * y /\ p * y <= q * x + r}))`,
  ONCE_REWRITE_TAC[COPRIME_SYM] THEN REPEAT STRIP_TAC THEN
  MP_TAC(SPECL [`q:num`; `p:num`; `r:num`] GAUSS_LEMMA) THEN
  ASM_SIMP_TAC[] THEN DISCH_THEN(K ALL_TAC) THEN AP_TERM_TAC THEN
  MATCH_MP_TAC EQ_TRANS THEN EXISTS_TAC
   `CARD {x,y | x IN 1..r /\ y IN 1..s /\
                y = (q * x) DIV p + 1 /\ r < (q * x) MOD p}` THEN
  CONJ_TAC THENL
   [CONV_TAC SYM_CONV THEN MATCH_MP_TAC CARD_SUBCROSS_DETERMINATE THEN
    REWRITE_TAC[FINITE_NUMSEG; IN_NUMSEG; ARITH_RULE `1 <= x + 1`] THEN
    X_GEN_TAC `x:num` THEN STRIP_TAC THEN
    SUBGOAL_THEN `p * (q * x) DIV p + r < q * r` MP_TAC THENL
     [MATCH_MP_TAC LTE_TRANS THEN EXISTS_TAC `q * x` THEN
      ASM_REWRITE_TAC[LE_MULT_LCANCEL] THEN
      GEN_REWRITE_TAC (LAND_CONV o ONCE_DEPTH_CONV) [MULT_SYM] THEN
      ASM_MESON_TAC[PRIME_IMP_NZ; LT_ADD_LCANCEL; DIVISION];
      MAP_EVERY EXPAND_TAC ["p"; "q"] THEN DISCH_THEN(MP_TAC o MATCH_MP
       (ARITH_RULE `(2 * r + 1) * d + r < (2 * s + 1) * r
                    ==> (2 * r) * d < (2 * r) * s`)) THEN
      SIMP_TAC[LT_MULT_LCANCEL; ARITH_RULE `x < y ==> x + 1 <= y`]];
    AP_TERM_TAC THEN
    REWRITE_TAC[EXTENSION; IN_ELIM_PAIR_THM; FORALL_PAIR_THM] THEN
    MAP_EVERY X_GEN_TAC [`x:num`; `y:num`] THEN
    AP_TERM_TAC THEN AP_TERM_TAC THEN EQ_TAC THEN DISCH_TAC THENL
     [MP_TAC(MATCH_MP PRIME_IMP_NZ (ASSUME `prime p`)) THEN
      DISCH_THEN(MP_TAC o SPEC `q * x` o MATCH_MP DIVISION) THEN
      FIRST_ASSUM(CONJUNCTS_THEN2 SUBST1_TAC MP_TAC) THEN
      UNDISCH_TAC `2 * r + 1 = p` THEN ARITH_TAC;
      MATCH_MP_TAC(TAUT `a /\ (a ==> b) ==> a /\ b`) THEN CONJ_TAC THENL
       [ALL_TAC;
        DISCH_THEN SUBST_ALL_TAC THEN
        MATCH_MP_TAC(ARITH_RULE
         `!p d. 2 * r + 1 = p /\ p * (d + 1) <= (d * p + m) + r ==> r < m`) THEN
        MAP_EVERY EXISTS_TAC [`p:num`; `(q * x) DIV p`] THEN
        ASM_MESON_TAC[DIVISION; PRIME_IMP_NZ]] THEN
      MATCH_MP_TAC(ARITH_RULE `~(x <= y) /\ ~(y + 2 <= x) ==> x = y + 1`) THEN
      REPEAT STRIP_TAC THENL
       [SUBGOAL_THEN `y * p <= ((q * x) DIV p) * p` MP_TAC THENL
         [ASM_SIMP_TAC[LE_MULT_RCANCEL; PRIME_IMP_NZ]; ALL_TAC];
        SUBGOAL_THEN `((q * x) DIV p + 2) * p <= y * p` MP_TAC THENL
         [ASM_SIMP_TAC[LE_MULT_RCANCEL; PRIME_IMP_NZ]; ALL_TAC]] THEN
      MP_TAC(MATCH_MP PRIME_IMP_NZ (ASSUME `prime p`)) THEN
      DISCH_THEN(MP_TAC o SPEC `q * x` o MATCH_MP DIVISION) THEN
      ASM_ARITH_TAC]);;

let GAUSS_LEMMA_SYM' = prove
 (`!p q r s. prime p /\ prime q /\ coprime(p,q) /\
             2 * r + 1 = p /\ 2 * s + 1 = q
             ==> (p is_quadratic_residue (mod q) <=>
                  EVEN(CARD {x,y | x IN 1..r /\ y IN 1..s /\
                                   p * y < q * x /\ q * x <= p * y + s}))`,
  REPEAT STRIP_TAC THEN
  MP_TAC(SPECL [`q:num`; `p:num`; `s:num`; `r:num`] GAUSS_LEMMA_SYM) THEN
  ONCE_REWRITE_TAC[COPRIME_SYM] THEN ASM_REWRITE_TAC[] THEN
  DISCH_THEN SUBST1_TAC THEN AP_TERM_TAC THEN
  GEN_REWRITE_TAC LAND_CONV [CARD_SUBCROSS_SWAP] THEN
  AP_TERM_TAC THEN REWRITE_TAC[EXTENSION; FORALL_PAIR_THM] THEN
  REWRITE_TAC[IN_ELIM_PAIR_THM; CONJ_ACI]);;

(* ------------------------------------------------------------------------- *)
(* The main result.                                                          *)
(* ------------------------------------------------------------------------- *)

let RECIPROCITY_SET_LEMMA = prove
 (`!a b c d r s.
        a UNION b UNION c UNION d = (1..r) CROSS (1..s) /\
        PAIRWISE DISJOINT [a;b;c;d] /\ CARD b = CARD c
        ==> ((EVEN(CARD a) <=> EVEN(CARD d)) <=> ~(ODD r /\ ODD s))`,
  REPEAT STRIP_TAC THEN
  SUBGOAL_THEN `CARD(a:num#num->bool) + CARD(b:num#num->bool) +
                CARD(c:num#num->bool) + CARD(d:num#num->bool) = r * s`
```

Figure 1. Fragment of the HOL Light formalization of the Law of Quadratic Reciprocity.

This is the Law of Quadratic Reciprocity in HOL syntax. In this expression the exclamation mark is ASCII syntax for the universal quantifier, the combination of slash and backslash is supposed to resemble the $\wedge$ sign and represents conjunction, the tilde is logical negation, and the ampersand operator maps the natural numbers into the real numbers. The functions `pow` and `DIV` represent exponentiation and division, and the term `legendre(p,q)` represents the Legendre symbol $\left(\frac{p}{q}\right)$. This last function is defined in the formalization by the HOL syntax:

```
let legendre = new_definition
 `(legendre:num#num->int)(a,p) =
     if ~(coprime(a,p)) then &0
     else if a is_quadratic_residue (mod p)
          then &1 else --(&1)`;;
```

In this the expression `num#num->int` corresponds to functions of type $\mathbb{N} \times \mathbb{N} \to \mathbb{Z}$. That is, `num` and `int` are the natural numbers and the integers, the hash symbol represents the Cartesian product, and the combination of a minus and a greater than sign is supposed to look like an arrow.

A formalization, in any proof assistant, mainly consists of a long chain of lemmas, where each lemma consists of a label, a statement, and a proof. In between these lemmas there occasionally are a few definitions. In `reciprocity.ml` there are three: a definition of the notion of quadratic residue, a definition of the Legendre symbol, and a definition of the notion of iterated product.

The formalization of the Law of Quadratic Reciprocity is too large to explain in full here. Therefore we will now zoom in on one of its smallest lemmas, the third lemma in the file (see Figure 2 below).

```
let CONG_MINUS1_SQUARE = prove
 (`2 <= p ==> ((p - 1) * (p - 1) == 1) (mod p)`,
  SIMP_TAC[LE_EXISTS; LEFT_IMP_EXISTS_THM] THEN
  REPEAT STRIP_TAC THEN
  REWRITE_TAC[cong; nat_mod;
    ARITH_RULE `(2 + x) - 1 = x + 1`] THEN
  MAP_EVERY EXISTS_TAC [`0`; `d:num`] THEN
  ARITH_TAC);;
```

Figure 2. Small lemma from the HOL Light formalization of the Law of Quadratic Reciprocity.

This ASCII text consists of the three parts mentioned above: the first line gives the label under which the result will be referred to later, the second line states the statement of the lemma, and the last three lines encode the proof.

In this proof, there are references to four earlier lemmas from the HOL Light library:

```
LE_EXISTS              !m n. m <= n <=>
                             (?d. n = m + d)
LEFT_IMP_EXISTS_THM    !P Q. (?x. P x) ==> Q <=>
                             (!x. P x ==> Q)
cong                   !rel x y. (x == y) rel <=>
                                 rel x y
nat_mod                !x y n. mod n x y <=>
                                 (?q1 q2. x + n * q1 =
                                          y + n * q2)
```

The proof of this lemma encodes a dialog with the system. We can execute the proof all at once (this happens when we load the file as a whole), but we can also process the proof step by step, in an interactive fashion. This is the way in which an HOL Light proof is developed. To do this, we enter the following command (where the # character is the prompt of the system):

```
# g `2 <= p ==> ((p - 1) * (p - 1) == 1) (mod p)`;;
```

The g command asks the system to set the "goal" to the statement between the backquotes. The system then replies with:

```
Warning: Free variables in goal: p
val it : goalstack = 1 subgoal (1 total)

`2 <= p ==> ((p - 1) * (p - 1) == 1) (mod p)`
```

indicating that it understood us and that this statement now is the current goal. Next we execute the first "tactic" (a command to the system to reduce the goal) by using the e command:

```
# e (SIMP_TAC[LE_EXISTS; LEFT_IMP_EXISTS_THM]);;
```

Note that this corresponds to the initial part of the third line of the lemma in the way that it is written in the file. The tactic SIMP_TAC uses the theorems given in its argument to simplify the goal. It transforms the goal to:

```
`!d. p = 2 + d ==> (((2 + d) - 1) * ((2 + d)
 - 1) == 1) (mod (2 + d))`
```

As already noted, the "!" symbol is the universal quantifier, which means that the statement that now is the goal is universally quantified over all natural numbers d. The existential quantifier, which will occur below, is written as "?".

We now display the rest of the interactive session without further explanations between the commands. This is a dialogue between the human user executing tactics and the computer presenting the resulting proof obligations ("goals"). In square brackets are the *assumptions* that may be used when proving the goal.

```
# e (REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

0 [`p = 2 + d`]

`(((2 + d) - 1) * ((2 + d) - 1) == 1) (mod (2
 + d))`

# e (REWRITE_TAC[cong; nat_mod;
 ARITH_RULE `(2 + x) - 1 = x + 1`]);;
val it : goalstack = 1 subgoal (1 total)

  0 [`p = 2 + d`]

`?q1 q2. (d + 1) * (d + 1) + (2 + d) * q1 = 1
 + (2 + d) * q2`

# e (MAP_EVERY EXISTS_TAC [`0`; `d:num`]);;
val it : goalstack = 1 subgoal (1 total)

 0 [`p = 2 + d`]

`(d + 1) * (d + 1) + (2 + d) * 0 = 1 + (2 + d
) * d`

# e ARITH_TAC;;
val it : goalstack = No subgoals
```

At this point the proof is finished, as there are no unproved subgoals left.

To effectively use HOL Light you will need to learn the dozens of tactics available in the system. This example uses the following tactics:

SIMP_TAC	Simplify the goal by theorems
REPEAT	Apply a tactic repeatedly until it fails
STRIP_TAC	Break down the goal
REWRITE_TAC	Rewrite conclusion of goal with equational theorems
ARITH_RULE	Linear arithmetic prover over $\mathbb{N}$
MAP_EVERY	Map tactic over a list of arguments
EXISTS_TAC	Provide a witness to an existential statement
ARITH_TAC	Tactic to solve linear arithmetic over $\mathbb{N}$

A final note about this example. The lemma talks about natural numbers, which means that for $p = 0$ the difference $p - 1$ is defined to be 0. This is called "truncated" subtraction. This complicates the proof, and also explains the need for the condition 2 <= p in the statement of the lemma. If the lemma had been stated using integers, it would have been provable without human input by the automated prover INTEGER_RULE:

```
# INTEGER_RULE `!p:int.
  ((p - &1) * (p - &1) == &1) (mod p)`;;
1 basis elements and 0 critical pairs
val it : thm = |- !p. ((p - &1) * (p - &1) ==
&1) (mod p)
```

In that case no explicit proof script would have been necessary.

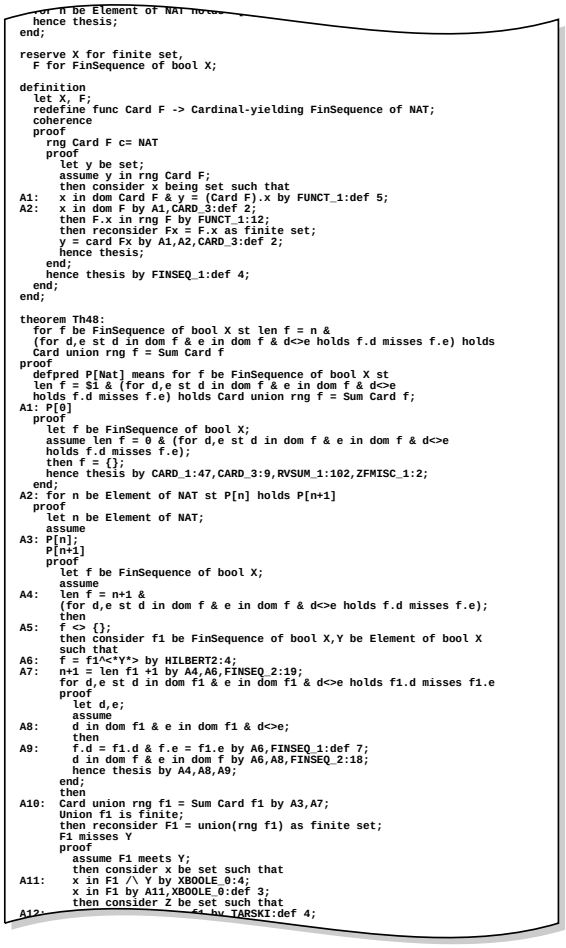
Mizar

If instead of HOL Light we choose Mizar as our proof assistant, again the second step consists of downloading and installing the system. Download the system from the Mizar website [10], unpack the tar or exe file, and follow the instructions in the README. Mizar is distributed as compiled binaries, which means that we do not need to install anything else first.

The third step then again is to write a formalization of the proof. The best way to learn the Mizar language is to work through the Mizar tutorial [11].

The Law of Quadratic Reciprocity is formalized in the file "mml/int_5.miz" (part of this file is shown in Figure 2; it also is on the Web by itself as [12]). This file again primarily consists of a long chain of lemmas. It consists of 4701 lines proving 51 lemmas. It also has 3 definitions: a definition of integer polynomials, a definition of quadratic residues, and a definition of the Legendre symbol.

To have Mizar check this file for correctness, copy the file "mml/int_5.miz" inside a fresh directory called "text", and *outside* this directory type



```
    for n be Element of NAT holds
    hence thesis;
  end;

  reserve X for finite set,
    F for FinSequence of bool X;

  definition
    let X, F;
    redefine func Card F -> Cardinal-yielding FinSequence of NAT;
    coherence
    proof
      rng Card F c= NAT
      proof
        let y be set;
        assume y in rng Card F;
        then consider x being set such that
A1:     x in dom Card F & y = (Card F).x by FUNCT_1:def 5;
A2:     x in dom F by A1,CARD_3:def 2;
        then F.x in rng F by FUNCT_1:i2;
        then reconsider Fx = F.x as finite set;
        y = card Fx by A1,A2,CARD_3:def 2;
        hence thesis;
      end;
      hence thesis by FINSEQ_1:def 4;
    end;
  end;

  theorem Th48:
    for f be FinSequence of bool X st len f = n &
    (for d,e st d in dom f & e in dom f & d<>e holds f.d misses f.e) holds
    Card union rng f = Sum Card f
  proof
    defpred P[Nat] means for f be FinSequence of bool X st
    len f = $1 & (for d,e st d in dom f & e in dom f & d<>e
    holds f.d misses f.e) holds Card union rng f = Sum Card f;
A1: P[0]
    proof
      let f be FinSequence of bool X;
      assume len f = 0 & (for d,e st d in dom f & e in dom f & d<>e
      holds f.d misses f.e);
      then f = {};
      hence thesis by CARD_1:47,CARD_3:9,RVSUM_1:102,ZFMISC_1:2;
    end;
A2: for n be Element of NAT st P[n] holds P[n+1]
    proof
      let n be Element of NAT;
      assume
A3:   P[n];
      P[n+1]
      proof
        let f be FinSequence of bool X;
        assume
A4:     len f = n+1 &
        (for d,e st d in dom f & e in dom f & d<>e holds f.d misses f.e);
        then
A5:     f <> {};
        then consider f1 be FinSequence of bool X,Y be Element of bool X
        such that
A6:     f = f1^<*Y*> by HILBERT2:4;
A7:     n+1 = len f1 +1 by A4,A6,FINSEQ_2:19;
        for d,e st d in dom f1 & e in dom f1 & d<>e holds f1.d misses f1.e
        proof
          let d,e;
          assume
A8:       d in dom f1 & e in dom f1 & d<>e;
          then
A9:       f.d = f1.d & f.e = f1.e by A6,FINSEQ_1:def 7;
          d in dom f & e in dom f by A6,A8,FINSEQ_2:18;
          hence thesis by A4,A8,A9;
        end;
        then
A10:    Card union rng f1 = Sum Card f1 by A3,A7;
        Union f1 is finite;
        then reconsider F1 = union(rng f1) as finite set;
        F1 misses Y
        proof
          assume F1 meets Y;
          then consider x be set such that
A11:      x in F1 /\ Y by XBOOLE_0:4;
          x in F1 by A11,XBOOLE_0:def 3;
          then consider Z be set such that
A12:                                    by TARSKI:def 4;
```

Figure 3. Fragment of the Mizar formalization of the Law of Quadratic Reciprocity.

```
mizf text/int_5.miz
```

This will print something like:

```
Processing: text/int_5.miz

Parser   [4701]   0:02
Analyzer [4700]   0:13
Checker  [4700]   1:14
Time of mizaring: 1:29
```

which means that the file was checked without errors. Try modifying `int_5.miz` and see whether the checker will notice that the file now no longer is correct.

The Law of Quadratic Reciprocity is the 49th lemma from the file. It reads:

```
p>2 & q>2 & p<>q
implies Lege(p,q)*Lege(q,p)=
  (-1)|^(((p-'1) div 2)*((q-'1) div 2))
```

The proof of this statement takes 1268 lines of Mizar code! Here is a smaller example of a Mizar proof. This is the 11th lemma from the file. See Figure 4 below.

```
theorem Th11:
  i gcd m = 1 & i is_quadratic_residue_mod m &
    i,j are_congruent_mod m
  implies j is_quadratic_residue_mod m
proof
  assume
A1: i gcd m = 1 &
    i is_quadratic_residue_mod m &
    i,j are_congruent_mod m;
  then consider x being Integer such that
A2: (x^2 - i) mod m = 0 by Def2;
  m divides (i - j) by A1,INT_2:19;
  then
A3: (i - j) mod m = 0 by Lm1;
  (x^2 - j) mod m
     = ((x^2 - i) + (i - j)) mod m
    .= (((x^2 - i) mod m) + ((i - j) mod m))
              mod m by INT_3:14
    .= 0 by A2,A3,INT_3:13;
  hence thesis by Def2;
end;
```

Figure 4. Small lemma from the Mizar formalization of the Law of Quadratic Reciprocity.

The lemmas from the Mizar library to which this proof refers are:

```
INT_2:19   a,b are_congruent_mod c iff
             c divides (a-b)
INT_3:13   (a mod n) mod n = a mod n
INT_3:14   (a + b) mod n =
             ((a mod n) + (b mod n)) mod n
Lm1        (x divides y implies y mod x = 0) &
           (x<>0 & y mod x = 0 implies
             x divides y)
Def2       a is_quadratic_residue_mod m iff
             ex x st (x^2 - a) mod m = 0
```

If you think that the condition "`i gcd m = 1`" is not used in this proof, you can try removing it, both from the statement and from the "`assume`" step, and see what happens when you check the file again.

The Future of Formal Mathematics

In mathematics there have been three main revolutions:

- The introduction of *proof* by the Greeks in the fourth century BC, culminating in Euclid's *Elements*.
- The introduction of *rigor* in mathematics in the nineteenth century. During this time

the nonrigorous calculus was made rigorous by Cauchy and others. This time also saw the development of mathematical logic by Frege and the development of set theory by Cantor.

- The introduction of *formal mathematics* in the late twentieth and early twenty-first centuries.

Most mathematicians are not aware that this third revolution already has happened, and many probably will disagree that this revolution even is needed. However, in a few centuries mathematicians will look back at our time as the time of this revolution. In that future most mathematicians will not consider mathematics to be definitive unless it has been fully formalized.

Although the revolution of formal mathematics already has happened and formalization of mathematics has become a routine activity, it is not yet ready for widespread use by all mathematicians. For this it will have to be improved in two ways:

- First of all, formalization is *not close enough to existing mathematical practice* yet to be attractive to most mathematicians. For instance, both HOL Light and Mizar define

$$\frac{1}{0} = 0$$

because they do not have the possibility to have functions be undefined for some values of the arguments. This is just a trivial example, but in many other places the statements of formalized mathematics are not close to their counterpart in everyday mathematics. Here there exists room for significant progress.

However, it is *not* important to have proof assistants be able to process existing mathematical texts. Writing text in a stylized formal language is easy. The fact that proof assistants are not able to understand natural language will not be a barrier to having formalization be adopted by the working mathematician.

- The second improvement that will be needed is on the side of *automation*. With this I do not mean that the computer should take steps that a mathematician would need to think about. Formalization of mathematics is about checking, and not about discovery.

However, currently steps in a proof that even a high school student can easily take without much thought often take many minutes to formalize. This lack of automation of "high school mathematics" is the most important reason why formalization currently still is a subject for a small group of computer scientists, instead of it having been discovered by all mathematicians.

Still, there are no fundamental problems that block these improvements from happening. It is just a matter of good engineering. In a few decades it will no longer take one week to formalize a page from an undergraduate textbook. Then that time will have dropped to a few hours. Also then the formalization will be quite close to what one finds in such a textbook.

When this happens we will see a quantum leap, and suddenly all mathematicians will start using formalization for their proofs. When the part of refereeing a mathematical article that consists of checking its correctness takes more time than formalizing the contents of the paper would take, referees will insist on getting a formalized version before they want to look at a paper.

However, having mathematics become utterly reliable might not be the primary reason that eventually formal mathematics will be used by most mathematicians. Formalization of mathematics can be a very rewarding activity in its own right. It combines the pleasure of computer programming (craftsmanship, and the computer doing things for you), with that of mathematics (pure mind, and absolute certainty.) People who do not like programming or who do not like mathematics probably will not like formalization. However, for people who like both, formalization is the best thing there is.

References

[1] http://www.cs.ru.nl/~freek/100/

[2] Freek Wiedijk (ed.), *The Seventeen Provers of the World*, with a foreword by Dana S. Scott, *Lecture Notes in Artificial Intelligence 3600*, Springer, 2006.

[3] G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers*, 1938. Fifth edition: Oxford University Press, 1980.

[4] Alan Baker, *A Concise Introduction to the Theory of Numbers*, Cambridge University Press, 1984.

[5] http://caml.inria.fr/ocaml/

[6] http://www.cl.cam.ac.uk/~jrh13/hol-light/

[7] http://www.cl.cam.ac.uk/~jrh13/hol-light/manual-1.1.pdf, John Harrison, *The HOL Light manual* (1.1), 2000.

[8] http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial_220.pdf, John Harrison, *HOL Light Tutorial* (for version 2.20), 2006.

[9] http://www.cs.ru.nl/~freek/notices/reciprocity.ml

[10] http://www.mizar.org/

[11] http://www.cs.ru.nl/~freek/mizar/mizman.pdf, Freek Wiedijk, *Writing a Mizar article in nine easy steps*, 2007.

[12] http://www.cs.ru.nl/~freek/notices/int_5.miz