The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/72455

Please be advised that this information was generated on 2017-07-13 and may be subject to change.
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.

<table>
<thead>
<tr>
<th>proof assistant</th>
<th>number of theorems formalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOL Light</td>
<td>69</td>
</tr>
<tr>
<td>Mizar</td>
<td>45</td>
</tr>
<tr>
<td>ProofPower</td>
<td>42</td>
</tr>
<tr>
<td>Isabelle</td>
<td>40</td>
</tr>
<tr>
<td>Coq</td>
<td>39</td>
</tr>
<tr>
<td>all together</td>
<td>80</td>
</tr>
</tbody>
</table>

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:


**Prime Number Theorem:** by Jeremy Avigad using Isabelle in 2004 (an elementary proof by Atle Selberg and Paul Erdös), and by...
<table>
<thead>
<tr>
<th>theorem</th>
<th>number of systems in which the theorem has been formalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The Irrationality of $\sqrt{2}$</td>
<td>$\geq 17$</td>
</tr>
<tr>
<td>2. Fundamental Theorem of Algebra</td>
<td>4</td>
</tr>
<tr>
<td>3. The Denumerability of the Rational Numbers</td>
<td>6</td>
</tr>
<tr>
<td>4. Pythagorean Theorem</td>
<td>6</td>
</tr>
<tr>
<td>5. Prime Number Theorem</td>
<td>2</td>
</tr>
<tr>
<td>6. Gödel’s Incompleteness Theorem</td>
<td>3</td>
</tr>
<tr>
<td>7. Law of Quadratic Reciprocity</td>
<td>4</td>
</tr>
<tr>
<td>8. The Impossibility of Trisecting the Angle and Doubling the Cube</td>
<td>1</td>
</tr>
<tr>
<td>9. The Area of a Circle</td>
<td>1</td>
</tr>
<tr>
<td>10. Euler’s Generalization of Fermat’s Little Theorem</td>
<td>4</td>
</tr>
<tr>
<td>11. The Infinitude of Primes</td>
<td>6</td>
</tr>
<tr>
<td>12. The Independence of the Parallel Postulate</td>
<td>0</td>
</tr>
<tr>
<td>13. Polyhedron Formula</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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).


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:

<table>
<thead>
<tr>
<th>proof assistant</th>
<th>proof style of the system</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOL Light</td>
<td>procedural</td>
</tr>
<tr>
<td>Mizar</td>
<td>declarative</td>
</tr>
<tr>
<td>ProofPower</td>
<td>procedural</td>
</tr>
<tr>
<td>Isabelle</td>
<td>both possible</td>
</tr>
<tr>
<td>Coq</td>
<td>procedural</td>
</tr>
</tbody>
</table>

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:

\[
\left( \frac{p}{q} \right) \left( \frac{q}{p} \right) = (-1)^{\frac{p-1}{2} \cdot \frac{q-1}{2}}
\]

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:

\[
\left( \frac{a}{p} \right) = a^{\frac{1}{2}(p-1)} \pmod{p}
\]

Using this criterion, by calculating the product of \( a, 2a, ..., \frac{1}{2}(p-1)a \) 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 \( 1 \leq j \leq \frac{1}{2}(p-1) \) 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 “load “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:

\[
\text{if } q. \text{ prime } p \land \text{ prime } q \land \\
\text{odd } p \land \text{odd } q \land \text{ } (p = q) \Rightarrow \text{legendre}(p,q) \ast \text{legendre}(q,p) = \\
\text{--(\&1) pow ((p - 1) div 2 \ast (q - 1) div 2)}
\]
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).

\[
\text{let CONG\_MINUS1\_SQUARE = prove}
\]
\[
\text{('2 <= p ==> ((p - 1) \times (p - 1) == 1) (mod p),)}
\]
\[
\text{SIMP_TAC[LEFT\_IMPL\_THM; LEFT\_IMPL\_THM; LE_EXISTS_THM] THEN}
\]
\[
\text{REPEAT STRIP_TAC THEN}
\]
\[
\text{CONG_MINUS1_SQUARE} :: 
\]
\[
\text{FIRST\_ASSUM(MP\_TAC o MATCH\_MP GAUSS\_LEMMA_4) THEN}
\]
\[
\text{REPEAT(COND\_CASES\_TAC THEN}
\]
\[
\text{ASM\_SIMP\_TAC[CONG\_SQUARE; LEFT\_IMPL\_THM]};
\]

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:

\[
\text{LET\_IMPL\_THM}
\]
\[
\text{!P Q, (?x. P x) ==> Q <=}
\]
\[
\text{(?x. P x)}
\]
\[
\text{cong}
\]
\[
\text{!x y. (x == y) rel <=}
\]
\[
\text{rel x y}
\]
\[
\text{nat\_mod}
\]
\[
\text{!x y n. \text{mod} n x y <=}
\]
\[
\text{(\&1 \text{mod} n x y)}
\]

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):

\[
\text{# g '2 <= p ==> ((p - 1) * (p - 1) == 1) (mod p);};
\]
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)';;
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...
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