rijksuniversiteit
groningen

faculteit Wiskunde en
Natuurwetenschappen

Evaluation of Isabelle
with a proof of the
Perfect Number Theorem

Bacheloronderzoek Wiskunde en Informatica

November 2009

Student: Mark IJbema

Eerste Begeleider: Jaap Top

Tweede Begeleider: Wim H. Hesselink

Contents

Chapter 1

Introduction

In this bachelor's thesis I will discuss my research on proof assistent Isabelle. Using the Perfect Number Theorem as test case, I tried to find out how usable Isabelle is. In chapters 4,7 and the appendix I discuss what exactly I looked at and the results. As context I offer chapter 1 on the general context of proofs and automated theorem proving and chapter 2 on Isabelle.

In chapter 5 I discuss my proof of the Perfect Number Theorem and in chapter 6 I discuss how to prove in Isabelle using my proof as example. If the reader is already familiar with the proof and/or Isabelle he can choose to skip one or both of these chapters.

In chapter 8 I draw conclusions of my research. Finally, in the appendix are some concrete bugs and problems which might be of interest to the developers of Isabelle and Proof General.

Chapter 2

Context

2.1 Formal proofs

The field of Mathematics has a long history of formalisation. While intuition proves to be a powerful tool in mathematics, it simply does not suffice when trying to convince others of your theories. Therefore the concept of the proof has been central in mathematics since the days of Euclid.

Euclid tried to formalise mathematics by starting with a only few assumptions, which he called axioms. The assumptions were chosen in such a way that they are in agreement with our intuition. From these axioms he derived other mathematical facts, by using only logical steps. If one does this, and the axioms are true, then the derived mathematical facts, called theorems and lemmas, have to be true as well. Since Euclid did only geometry, he took a few basic geometric axioms, and proved a lot of geometric and arithmetic theorems from it, but got stuck when he tried to prove more complex theorems (the famous three Greek problems).

After Euclid, mathematics evolved, and many new theories were introduced, but the first real attempt to create one unifying theory was not undertaken until the start of the twentieth century. In 1911-1913 Whitehead and Russell published the Principia Mathematica which contained what they considered to be the basis of all mathematics. This was done in an effort to fulfill Hilbert's Program to formalize all mathematics. This program was effectively nullified by Gödel when he proved that every sufficiently complicated mathematical system had to be incomplete (in some sense).

Even though it might not be possible to prove everything conclusively it is still interesting to prove almost everything, and in a non-refutable way. In 1969 De Bruijn started working on the Automath program, the first automated theorem prover, and had one of his Ph.D. students encode and check the entire Landau book on the foundations on analysis. The field of computer verification kept on developing and in 1994 the QED manifesto was published. The goal of the QED manifesto was to compile all mathematical knowledge into one library, which could then be checked by a computer for correctness.

2.2 Formal proofs with the computer

The major pitfall in proofs is of course that the steps have to be "logical", and this is ill-defined. When is a step logical, and when does it merely seem logical? For instance, it seems logical that $(a - b) + b$ is equal to a, however, this is not necesarily true in the natural numbers[1]. Therefore it was important for the mathematical community to have some way to make sure proofs are correct.

The standard method of making sure a proof is correct is by having it checked by peers. Before the advent of journals this was done by sending letters to famous colleagues, who would then give feedback. Nowadays, after some reviewing by local colleagues, a proof is submitted in the form of a paper to a peer-reviewed journal. The journal then sends it to other mathematicians who supposedly know enough of the subject to find errors in the proof. The comments of the reviewers get sent back, and the researcher then uses these to fix errors, or, if they are insurmountable, the paper is rejected. After publication, there is of course still the possibility that a subscriber to the journal finds an error, and calls attention to this. In this way, so many people look at a proof, that if it has not been contested after a few years it is assumed to be correct.

Of course this system is not foolproof. If one does research in a very specific field, there might not be enough peers who are able to point out errors. It is also possible that the results are not really interesting, so no-one looks into it, etcetera. So another solution is desirable.

This is where the computer comes in. With the computer it is easier to define what a logical step is. As with regular proofs, you still have to assume some axioms, both axioms on the theory itself and axioms on the logic used for the reasoning steps. From these axioms you can derive new theorems by combining them. You have to make sure that each step has to be true according to an axiom or a previously proven theorem. In this way you can build mathematics from the ground up, and since the computer can check whether each step is true, the resulting theory must be true.

Of course there is again a caveat: whereas the old method depended on enough eyeballs to make sure the proof is correct, the automated method requires the computer to be correct. This means there must be no errors in the hardware, no errors in the proving algorithms, etcetera. However, since by running the prover on different hardware architectures makes errors by hardware improbable, the most important thing is the software. This is why most provers have a very small core of methods, which then have to be proven correct, and more advanced methods are expressed within the provers themselves, and must therefore be correct if the basic methods are correct.

In this way verifying the correctness of a proof is reduced to two easier tasks: express the proof in a formal computer-checkable language, and verify the prover. Actually the last part is seldom done, and instead for this task one relies on the community again, in the same way we did for "old" proofs.

[1]For instance, take $a = 3$ and $b = 5$ then $(a - b) + b = (3 - 5) + 5 = 0 + 5 = 5 \neq 3 = a$ (depending on your definition of course, since the natural numbers have no additive inverse; this however is the way it is defined in Isabelle)

2.3 Proving with the computer

Proving with the computer is actually quite different from proving on paper. Since the computer requires each step to be completely logical, one spends a lot of time proving the obvious. In many ways proving with the computer can be compared to proving something to a toddler ("why? why? why?").

However, it is important to recall that in other aspects a computer is a lot smarter and faster than a human. For instance, a logical formula can be proven by rewriting it to a basic tautology. However, with the computer it might be faster to just try all options by brute force. Even though this also takes longer for the computer to verify, it is less work for the human. A lot of provers also offer other automated methods, which search for theorems and try them out, to see if a theorem proves the current statement. So in some ways a computer proof is more difficult, in others less.

2.3.1 Usefulness of formalisation

To have a meaningful discussion about the usefulness of formalisation we first need to consider what uses it could have. The most important use is making a proof more reliable. This might seem useless, since for most proofs (like the one which I prove in my case study), it is not necessary to prove them using the computer. The proof is clear, and each mathematician who knows the field will have no problems following the reasoning, and verifying its correctness. However, some proofs are so complicated (the proof of Fermat's Last Theorem is a good example) or have so many cases (the proof of the four colour theorem) that this is nearly impossible. Such proofs benefit in credibility by a formal proof.

Another use is improving upon existing proofs. Since you have to work very detailed when proving on the computer, you might discover ways to do a proof smarter. This happened when Werner and Gonthier tried to formalise the proof by Appel and Haken. They started out by formalising the proof by Appel and Haken, but stumbled upon a much cleaner proof during this proces.

Then there are some other areas where formalised proofs might be useful at some point, but where the current systems probably are not sophisticated enough to suffice. The first is in education. When you give students a proof assistent like Isabelle they will be forced to think very rigorously about their proofs. It might be possible that they will learn to think in a more formal and structured way compared to learning to prove on paper.

A last use of theorem provers is probably still very much speculative, but might prove feasible in due time, is using it instead of peer reviewing. The peer reviewing has several distinct disadvantages. Again, the four colour theorem is a good example. There were so many false proofs and counterexamples that it was a lot of work to decide whether proofs were false or correct, and especially for amateurs it is hard to be taken seriously in such a situation. We see much of the same happening to the milennium problems. However, when people would use a theorem prover for their proofs this would not be necessary, because the proof would be valid if the theorem prover accepts it.

Chapter 3

Isabelle

For my research I chose to use the tool Isabelle. Isabelle provides a nice readable syntax, which should be verifiable by humans as easy as by the computer. Because using Isabelle also meant using some other tools I will use this chapter to explain the tools used.

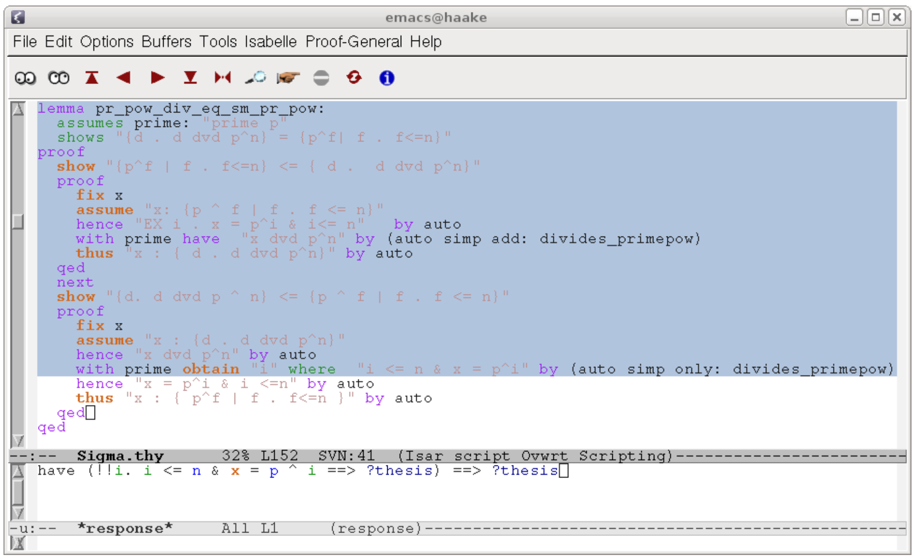
3.1 Isabelle/Isar

Isabelle is a generic proof assistant. What this basically means is that Isabelle offers a language to write proofs in, and a program to check those proofs. Isabelle supports multiple logic systems, but when referring to Isabelle in this document we will be referring to Isabelle used with Higher Order Logic (which is also the default).

Isabelle (without extensions) supports only tactic-style (backward) proofs. A backward proof is a proof where you start with the end-goal, and rewrite the end-goal using rules until you have rewritten your goal to a known tautology, at which point you are done. This is rather counterintuitive, when you are used to regular mathematical proofs. An important disadvantage of this type of proof is that only the rules used are saved into the proof itself. The intermediate results are only visible when stepping through the proof with Isabelle.

In contrast, a regular mathematical proof goes forward: you assume something, rewrite your assumptions until they are rewritten to your goal. This type of forward proof is made possible in Isabelle by the extension called Isar. The most important advantage of using Isar is that it is obvious to a human reader of the proof how the proof works. In a backward Isabelle proof a human reader would have to figure out the current proof state (which is very hard or even impossible). Another advantage is that forward proofs make it easier to see where the proof could be genereralised or improved, so this might result in better proofs as well.

A second, maybe a bit less obvious advantage of forward proofs, is that they make the proofs less brittle. If for instance an update of the Isabelle system changes a theorem slightly, the proof will fail from the point on where the theorem is used. In Isar, one can simply replace the failing step (temporarily) with sorry (see section 6.3), and step through the proof to verify if all other steps still hold, and then reprove all the failing steps.



```
lemma pr_pow_div_eq_sm_pr_pow:
  assumes prime: "prime p"
  shows "{d . d dvd p^n} = {p^f| f . f<=n}"
proof
  show "{p^f | f . f<=n} <= { d .  d dvd p^n}"
  proof
    fix x
    assume "x: {p ^ f | f . f <= n}"
    hence "EX i . x = p^i & i<= n"   by auto
    with prime have  "x dvd p^n" by (auto simp add: divides_primepow)
    thus "x : { d . d dvd p^n}" by auto
  qed
  next
  show "{d. d dvd p ^ n} <= {p ^ f | f . f <= n}"
  proof
    fix x
    assume "x : {d . d dvd p^n}"
    hence "x dvd p^n" by auto
    with prime obtain "i" where  "i <= n & x = p^i" by (auto simp only: divides_primepow)
    hence "x = p^i & i <=n" by auto
    thus "x : { p^f | f . f<=n }" by auto
  qed
qed
--:--  Sigma.thy      32% L152   SVN:41   (Isar script Ovwrt Scripting)------------------------
have (!!i. i <= n & x = p ^ i ==> ?thesis) ==> ?thesis

-u:--  *response*      All L1      (response)--------------------------------------------------
```

Figure 3.1: Proof General

3.2 Proof General

Isabelle has a command line interface to create proofs, but using that is akin to using edlin to edit text-documents. Luckily there is a GUI for Isabelle as well, in the form of Proof General. Proof General is a framework for different Proof Systems, but I will restrict my description to how it is used for Isabelle.

Proof General is a program build upon Emacs, which provides a way to both edit and verify proofs, in an integrated context. In Proof General you type out a proof, and then check it line by line using Proof General[1]. The Proof General advances a line if it syntactically and semantically accepts the proof, and then it locks the proved region. In this way you cannot edit the proved region, so you know everything in the proved region is a correct proof.

3.2.1 Sledgehammer

Sometimes when you have a trivial step in your proof, but you do not know which theorems are needed to prove the step, you can use the Sledgehammer tool. The Sledgehammer tool invokes external provers like "e" and "vampire" to see if they can construct a so-called metis proof. Here metis is a proof method, which is given a list of theorems as argument. The provers search for theorems metis needs, and then return the appropriate command. This gives the human prover time to get a cup of coffee, and saves him the time to search for the needed theorems.

After one retrieved the proof this way, one can either choose to keep it, or to base another command upon it (for instance by taking the primary theorem from the list, and hoping auto will do the rest).

[1]For PVS users: This means that what you see in your proof document is the full proof. There is no Lisp-code somewhere in another file which is also needed for the proof.

Chapter 4

Case Study

For this research I decided to evaluate the Isabelle system by proving a theorem which to our knowledge has not been proven in Isabelle before. However, not wanting to do just a toy problem, I decided to choose a theorem which was at least of interest. To this end I chose a problem from the top 100 theorems list[1]. I chose nr 70 (the Perfect Number Theorem), because it looked feasible, and was not done yet in Isabelle.

4.1 Research goals

During the proving I wanted to find out how hard it is to get started using Isabelle. Specifically I wanted to know how discoverable Isabelle is and how easy it is to find required documentation.

Furthermore I wanted to find out what the best way is to get started with Isabelle, and document that. Moreover I wanted to know which things are important to know beforehand, and what is not important until later.

Lastly, I wanted to see what could be better about Isabelle and related tools, document bugs and shortcomings and submit those to the respective projects.

[1]This is a list of 100 important/beautiful theorems, and Freek Wiedijk maintains a record[8] of which theorems have been proven in which theorem prover. Thus, by proving a theorem from the list, I would increase the score for Isabelle.

Chapter 5

The Perfect Number Theorem

To prove the Perfect Number Theorem we first need to introduce several new concepts. To make this report readable for anyone who did high school courses in Mathematics we start by providing some basic concepts regarding numbers. Then we introduce the σ-function and finally we prove the Perfect Number Theorem.

5.1 Basics

To be able to reason about numbers it is important to consider the concept of primes and coprimality:

Definition 5.1. *A prime is a number equal to or larger than two which is divisible by only itself and one.*

Definition 5.2. *The greatest common divisor (gcd) of two numbers is the largest number which divides both numbers.*

Definition 5.3. *Two numbers are coprime if ther greatest common divisor is one.*

Lastly, we will introduce a lemma regarding powers of primes:

Lemma 5.4.

$$d|p^e \Leftrightarrow \exists f, d = p^f, f \leq e$$

Here $a|b$ stands for a divides b without a remainder. We omit the proof of, since the lemma already is available in Isabelle.

5.2 σ-function

The Perfect Number Theorem is defined in terms of the sigma function, therefore we start by introducing the function and proving some basic facts about it.

Definition 5.5. *The function $\sigma(m)$ is defined as the sum of all the divisors of m (including 1 and m).*

Lemma 5.6. *$\sigma(n) = n + 1$ if and only if n is prime.*

Proof. ($\Leftarrow$) If n is prime, its only divisors are 1 and n. Since $\sigma(n)$ is the sum of those two, it is equal to $n + 1$.

($\Rightarrow$) If $\sigma(n) = n+1$, and we also know every number is divided by itself and 1, we know the set of divisors to be at least $\{1, n\}$. We now prove that there cannot be another divisor of n, say a. If there were such a divisor, then $1, a, n$ would be a subset of the set of divisors, and thus $\sigma(n) \geq 1 + a + n > 1 + n$, which is in contradiction with the assumption, so there cannot be such an a. Therefore the set of divisors is exactly $\{1, n\}$ and thus n is prime. $\square$

Lemma 5.7. *When p is prime: $\sigma(p^n) = \frac{p^{n+1}-1}{p-1}$*

Proof. We will prove this in two parts:

1. When p is prime: $\sigma(p^n) = \sum_{i=0}^{n} p^i$
2. When $x > 1$: $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$

The first part follows trivially from lemma 4.4. The second part is easily proved by some rewriting:

$$\begin{aligned}(x-1)\sum_{i=0}^{n} x^i &= \sum_{i=0}^{n} x * x^i - \sum_{i=0}^{n} x^i \\ &= \sum_{i=1}^{n+1} x^i - \sum_{i=0}^{n} x^i \\ &= x^{n+1} - 1\end{aligned}$$

$\square$

Lemma 5.8. *When n and m are coprime: $\sigma(n * m) = \sigma(n) * \sigma(m)$*

Proof. We first rewrite the sum, so that we only need to prove something about sets:

$$\begin{array}{rrcl} & \sigma(n*m) &=& \sigma(n)*\sigma(n) \\ \Leftrightarrow & \sum \text{divisors(n * m)} &=& (\sum \text{divisors}(n)) * (\sum \text{divisors}(n)) \\ \Leftrightarrow & \sum_{x|n*m} x &=& \left(\sum_{x|n} x\right) * \left(\sum_{y|n} y\right) \\ \Leftrightarrow & \sum_{x|n*m} x &=& \left(\sum_{x|n}\left(\sum_{y|n} x*y\right)\right) \\ \Leftrightarrow & \sum\{x : x|n*m\} &=& \sum\{x*y : x|n \wedge y|m\} \end{array}$$

Now we only need to show that the two sets we sum are equal. We do this by showing that each set is a subset of the other. The first proof is trivial:

$$\begin{array}{l} x \in \{x * y : x|n \wedge y|m\} \\ \Rightarrow \exists a, b : a * b = x \wedge a|m \wedge b|n \\ \Rightarrow x|m*n \\ \Rightarrow x \in \{x : x|n*m\} \end{array}$$

The other half is actually the same proof, but the other way around:

$$\begin{aligned}
&x \in \{x : x|n * m\}\\
&\Rightarrow x|m * n\\
&\Rightarrow \exists a, b : a * b = x \wedge a|m \wedge b|n\\
&\Rightarrow x \in \{x * y : x|n \wedge y|m\}
\end{aligned}$$

The only thing which might not be obvious immediately is that such a a and b exist. However, we can actually calculate them: $a = \gcd(x, m)$ and $b = \gcd(x, n)$. They obviously divide m and n respectively, but we also know $a * b = x$: Each prime power factor of x is either a factor of m or n, never of both (since they are coprime), never of none (since x divides their product).

□

5.3 Perfect Number Theorem

We now have all the required basics and concepts to define perfect numbers and formulate the Perfect Number Theorem.

Definition 5.9. *A perfect number m is a number for which $\sigma(m) = 2 * m$*

Theorem 5.10. *Let m be an even perfect number, then:*

$$\exists p : (\text{prime } 2^p - 1) \quad \wedge \quad \big(m = 2^{p-1} * (2^p - 1)\big)$$

Proof. Given m even and m perfect, write: $m = 2^{n-1} * A$ with 2 and A coprime. Since m even we know $n \geq 2$.

Now, since m is perfect, and by using the beforementioned lemma's on values of $\sigma(n)$:

$$\begin{array}{rrcl}
 & 2 * m & = & \sigma(m)\\
\Rightarrow & 2 * (2^{n-1} * A) & = & \sigma(2^{n-1} * A)\\
\Rightarrow & 2^n * A & = & \sigma(2^{n-1})\sigma(A)\\
\Rightarrow & 2^n * A & = & (2^n - 1)\sigma(A)
\end{array} \tag{5.1}$$

Then:

$$\begin{array}{rrcl}
 & 2^n - 1 & | & 2^n A\\
\Rightarrow & 2^n - 1 & | & A
\end{array}$$

Thus: $\exists B : (2^n - 1)B = A$. And since $n \geq 2$ we know $B < A$.

We will now prove that this means that $B = 1$, by contradiction. Let us assume that $B \neq 1$, then $B > 1$. But then $B, 1, A$ are alle distinct, and they are all divisors of A, so:

$$\begin{array}{rrcl}
 & \sigma(A) & \geq & 1 + B + A\\
\Rightarrow & (2^n - 1)\sigma(A) & \geq & (2^n - 1)(1 + B + A)\\
\Rightarrow & (2^n - 1)\sigma(A) & \geq & (2^n - 1) + (2^n - 1)B + (2^n - 1)A\\
\Rightarrow & (2^n - 1)\sigma(A) & \geq & (2^n - 1) + A + (2^n - 1)A\\
\Rightarrow & (2^n - 1)\sigma(A) & \geq & (2^n - 1) + 2^n A
\end{array}$$

But this is in contradiction with equation 5.1, so B must be equal to 1. This also means that $A = 2^n - 1$. Then, according to equation 5.1:

$$\begin{array}{rrcl} & 2^n A & = & (2^n - 1)\sigma(A) \\ \Rightarrow & 2^n(2^n - 1) & = & (2^n - 1)\sigma(A) \\ \Rightarrow & 2^n & = & \sigma(A) \end{array}$$

Therefore A is prime. □

The reverse of the Perfect Number Theorem is also true, and was already proved by Euclid (Book 9, proposition 36):

Theorem 5.11. *Let $2^n - 1$ be prime, then $2^{n-1}(2^n - 1)$ is perfect*

Proof. Since $2^n - 1$ is odd, 2^{n-1} and $2^n - 1$ are coprime. Therefore, again using previous lemma's on values of $\sigma(n)$:

$$\begin{array}{rcl} \sigma(2^{n-1}(2^n - 1)) & = & \sigma(2^{n-1}\sigma(2^n - 1)) \\ & = & (2^n - 1)(2^n) \\ & = & 2 * 2^{n-1}(2^n - 1) \end{array}$$

□

Chapter 6

Proving using Isabelle

Proving a theorem using a theorem prover is quite different from proving a theorem by hand. Therefore, in this chapter I will explain how to prove a theorem in Isabelle. I will first describe the general process[1], and then go into specifics for Isabelle.

The intention of this chapter is not to be a full-blown tutorial. It is meant to be an overview of the (most important) ways of proving theorems in Isabelle, and to provide a quick way to find what kind of technique you might want to use. After that, you probably still need to look up the technique in a tutorial. I will mention some useful tutorials at the end of this chapter.

When proving with Isabelle you need to be very precise. Usually, in a proof on paper you skip steps which are evidently true, and omit proofs of trivial lemmas, as readers can easily verify these themselves. However, when you are proving with Isabelle you really need to make each step and each proof explicit. How does one go through this process from a proof on paper to a proof in Isabelle? I will explain this in this chapter.

6.1 Preparing the proof on paper

First you need to have a proof, so prove the theorem the usual way. When you have your proof, you could in principle implement it directly in Isabelle. However, some steps might turn out to be quite involved, so your proof quickly becomes unwieldy.

An important lesson I learned from Software Engineering is that you should keep your functions small, and the same is true for (computer) proofs. Make sure that each theorem does exactly one (small) thing; if it does several, split it up by extracting sub-lemmas.

So the next step is taking your proof, and trying to identify the “big facts” you used, which are not available in Isabelle and thus require you to prove them yourself. Furthermore you need to find the parts of your proofs which could be extracted to lemmas. Then rewrite the proof on paper after splitting the proof up into smaller steps.

For each lemma used, try to be precise: what preconditions do you really need? Is this the most general conclusion? It might be easier to prove a lemma

[1]Although I will talk about Isabelle, you can read “theorem prover” for each occurence.

for a general prime p than to prove it for 2. On the other hand, make sure that you do not make it too difficult, maybe a certain lemma is provable for any number $n > 1$, but the proof is much easier if you know it to be prime.

For a good example of this, see theorem "sigmasemimultiplicative" in section B.3. In the original proof in section 5.2, this was a lemma about two coprime integers m and n. However, I choose to prove it only for p^n and m, with p and m coprime, and p prime. But I only needed it for 2^n and m odd. So I did not choose the most general option, but not the most specific either.

6.2 Entering the proof into Isabelle

Typically a proof will start by the statement of a few facts (the assumptions of the lemma). From these you derive new facts (using the **from** and **have** keywords), and you iterate this process until you have proven the fact which is the conclusion (where you use the **show** keyword). However, most of the times you would have a proof where each new fact follows (almost) directly from the previous. Isabelle provides a shorthand for this with the **hence** keyword (and **thus** for the conclusion). For instance:

```
lemma sigma_imp_prime: "sigma(n)=n+1 ==> prime n"
proof -
  assume ass:"sigma(n)=n+1"
  hence "n>1 & divisors(n)={1,n}" by (metis insert_commute
      sigma_imp_divisors)
  thus "prime n" by (simp add: primedivisors)
qed
```

Sometimes just using the previous fact does not suffice, and you need to incorporate other facts. You can do this with the **with** keyword.

```
lemma exp_is_max_div:
  assumes m0:"m>0" and p: "prime p"
  shows "~ p dvd (m div (p^(exponent p m)))"
proof (rule ccontr)
  assume "~ ~ p dvd (m div (p^(exponent p m)))"
  hence a:"p dvd (m div (p^(exponent p m)))" by auto
  from m0 have "p^(exponent p m) dvd m" by (auto simp add:
      power_exponent_dvd)
  with a have "p*(p^exponent p m) dvd m" by (metis divides_mul_l
      dvd_mult_div_cancel local.a nat_mult_commute)
  with p have "m=0" by (auto simp add: power_Suc_exponent_Not_dvd)
  with m0 show "False" by auto
qed
```

In my opinion **with** should only be used when using some general fact (like an assumption). Otherwise using the **moreover** construction might be more appropriate.

```
lemma sigma_finiteset2[simp]:
assumes m0: "m>0"
shows "m>0 ==> finite {(x::nat) * b |b. b dvd m}"
proof -
  from m0 have "finite (divisors m)" by simp
  hence "finite ((op * x)`(divisors m))" by auto
  moreover have "{x * b |b. b dvd m} = (op * x)`(divisors m)" by
      (auto simp add: divisors_def)
  ultimately show "?thesis" by auto
qed
```

A somewhat different type of proof is when we want to prove $A = B$ and prove this by rewriting A until it is equal to B. In Isabelle we can use the **also** keyword with the ... operator (which recapitulates the right hand side of the previous expression), which allows us to write such rewriting proofs:

```
theorem simplify_sum_of_powers: "(x - 1::nat) * (\<Sum>i=0 .. n .
    x^i)  = x^(n + 1) - 1" (is "?l = ?r")
proof (cases)
  assume "n = 0"
  thus "?l = x^(n+1) - 1" by auto
  next
  assume "n~=0"
  hence n0: "n>0" by auto
  have "?l  = (x::nat)*(\<Sum>i=0 .. n . x^i) - (\<Sum>i=0 .. n .
      x^i)" by (simp only: distribute_min_mult)
  also have "... = (\<Sum>i=0 .. n . x^(Suc i))     - (\<Sum>i=0 ..
      n . x^i)" by (simp add: setsum_right_distrib)
  also have "... = (\<Sum>i=Suc 0 .. Suc n . x^i)  - (\<Sum>i=0 ..
      n . x^i)" by (metis One_nat_def
      setsum_shift_bounds_cl_Suc_ivl)
  also with n0 have "... = ((\<Sum>i=Suc 0 .. n . x^i)+x^(Suc n)) -
       (x^0 + (\<Sum>i=Suc 0 .. n . x^i))"  by (auto simp add:
      setsum_Un_disjoint nat_interval_minus_zero2)
  finally show "?thesis" by auto
qed
```

Of course, all previous methods can also be mixed. When you get the hang of which method is best suited to which proofs, your proofs become much more readable.

6.3 Proving the steps

Now you know how to enter your proof into Isabelle. However, you still need to prove its correctness. In Isabelle you do this by proving that each step is correct. You can prove this in several ways (actually, these are exactly the same ways you can prove a complete lemma). These are the alternatives:

1. write a complete (sub)proof
2. describe which list of proof methods prove the step
3. give a method by which it is proven
4. omit the proof

I will start by describing the last, since I find it to be one of the nicest features of Isabelle. You can just "prove" a step by saying **sorry**. In this way you can skip difficult steps, prove the rest of the theorem, and come back to this step to prove it later, or maybe split it off to a separate lemma. I cannot emphasize enough how useful this is in top-down proving.

The second way is by using **apply** to execute proof methods. You do this by applying different methods in a backwards proof until the step is proven. Often, just one method will suffice, and you can just use this step with the **by** keyword (option number 3).

The first option is to write a complete subproof. You can either do this in-place, or create a separate lemma for this step, and possibly generalize it. If you encounter many steps where you need to do this, you might want to consider going back to your paper proof, and making it more detailed first.

Even though you might be able to prove a lot of the steps by hand, because you know which theorem to use or are able to find it quickly enough in the library, sometimes you know a step to be true, but have no clue as to how to prove it. In these cases you can use the "Sledgehammer" tool. By pressing Ctrl-C,A,S you invoke the Sledgehammer, which invokes external provers. After a minute or so they might return a command which proves the step[2].

6.4 Proof methods

There are several methods to prove a certain proposition. I will not treat them all in detail, but I will try to highlight the most important ones, and suggest when to use them.

6.4.1 metis

The metis method is used by proofs delivered by sledgehammer. The metis method tends to use a lot of very small, unimportant theorems, and because of this it is not always clear which theorems are the most essential for proving the proposition. I prefer to select the most important theorems and then use them in combination with auto when possible.

6.4.2 simp

The simp method invokes the simplifier. It applies all defined simplification rules, in order to unify the premise and the conclusion. If you want to use some theorem along with the simplifier you can add the theorem with add (like (simp add: othertheorem)). If you have rules which are actual simplifications you can also mark them as such.

6.4.3 auto

This is one of the most powerful methods of Isabelle. First the simplifier by auto and then some more methods are applied. You can again add methods to the simplifier by using add (like (auto simp add: othertheorem)).

6.4.4 blast

The power tool when it comes to solving purely logical propositions is the blast method. If a proposition can be solved by pure logic (which means:no arithmetic or knowledge of the problem domain), blast will solve it most of the times.

6.4.5 subst

Use the subst method in combination with a theorem which states an equality, to replace something which matches the left-hand side of the equality by the right-hand side in your own proposition.

[2]Unfortunately, the output of Sledgehammer is not perfect, and needs some tweaking. Also see my comments in section A.2

6.4.6 rule

The rule method is one of the simplest methods. If the preconditions and conclusion of your proof are a match for the (more general) preconditions and conclusion of a certain theorem, you can use that theorem with rule (like this: (rule sometheorem)).

6.5 Finishing up

When you have a working proof, inspect your proof, and for each lemma ask yourself the following question:

1. Should this be a lemma or a theorem?
   (this is merely for the human reader, to Isabelle this makes no difference)
2. Can I weaken the preconditions?
3. Can I generalise the conclusion?
4. Should this be a simplification rule?

Then you could, depending on what exactly you produced, consider adding additional theorems regarding the new concepts you introduced, and make a useful library out of it for submission to the Archive of Formal Proof. You also may want to touch up the layout the proof generates in LaTeX.

6.6 Teaching yourself to prove in Isabelle

Learning Isabelle is not something you can do by only reading a tutorial. You need to dive into it, and code out some proofs yourself. I did this by proving something, reading something, and then repeating. An order I would recommend would be something like this:

1. Read this chapter
2. Try to create a little proof (like "there exist numbers > 37"); this is not as easy as it sounds.
3. Read the Isar tutorial [6]
4. Explore the library online [2]
5. Try another proof
6. Read the book on Isabelle [7]
7. Prove theorems; keep this chapter, the Isar tutorial, the library and the Isabelle / Proof General Cheat Sheet[3] at hand

If you have any questions which are not answered in aforementioned resources you can try to find additional tutorials (there are quite a few, try the Isabelle site[5] as starting point) or ask a question on the Isabelle mailinglist[4]. But most important is to get a feel for what you can and cannot do with Isabelle, and the only way to obtain that is by just working with it.

Chapter 7

Evaluation

After proving a decently sized proof in Isabelle, I have gotten a reasonable impression of Isabelle. I will first discuss Isabelle itself, and after that the development environment (mainly Proof General). After that I will return to the possible uses, and whether I think Isabelle is suitable for them at this time.

7.1 Isabelle

Isabelle is a powerful theorem prover, with a sizable library. One notable shortcoming in the library is however arithmetic regarding natural numbers and the minus operator. Partly, this shortcoming can be avoided by replacing $n-1$ with n and n with $n+1$. However, I still felt that sometimes I needed to prove very trivial steps. Another example is finite sets. There were a lot of sets which I needed to prove finite, where in my opinion, this was rather trivial. I think you want a library so large, that at least all trivial steps are already proven. Of course, what is trivial is a matter of opinion.

All in all I am rather satisfied with Isabelle. Even though the language is not very discoverable, it is not very hard. After mastering the language you are able to write very nice, human-readable proofs. This means that the proofs are not only useful for convincing people of the truth of theorems, but also for showing how a certain problem is solved and providing new insights.

7.2 Proof General

I think Proof General provides a usable IDE[1] for Isabelle. However, coming from a Computer Science background, it is rather disappointing. Whereas several decades ago programmers needed to write code in plain-text editors, they now have fullblown IDE's with syntax highlighting, autocompletion and refactoring capacities[2]. In this respect Proof General feels a bit outdated, because it offers only highlighting and very basic IDE options.

One of the other problems of missing a good IDE is that you need a good browser for the documentation, and the best I could find was searching the

[1]Integrated Development Environment

[2]For instance: you can rename a function, and all references to that function get renamed as well

library documentation using Google. However, for me this was far from ideal, and learning where to find needed lemmas took me a lot of time.

There is an effort to port Proof General from Emacs to Eclipse, but since that was still unstable I decided not to test that version. However, I sincerely hope that the port wil bring some more userfriendliness to Proof General. I think that at the moment Proof General is not very well suited to the needs of the general mathematical population[3].

I think if more of the actions needed to do manually at the moment could be automated, it would really improve adaptation of Isabelle.

7.3 Isabelle vs. PVS

This was the second time I used a theorem prover. The first time was during a course on automated reasoning, in which we used PVS. I prefer Isabelle, because you have the whole proof in one file, and the proof is very well suited to humans as well. However, I do not have as much experience with PVS as I have with Isabelle, so I cannot state anything conclusive about this.

7.4 Implications for applicability

To return to the uses of theorem provers I think Isabelle is very well suited to proving complex proofs (and it has been used for that), and for finding new ways to prove theorems. However, I do not think that it can be used as a tool to teach students to think formally or, as expected, as a replacement for the peer-review system. At the moment Isabelle simply is not userfriendly enough, and the learning curve is too high, and it would annul all the benefits theorem provers have to offer. I do believe this will improve in the future, and think it will at least be usable for education in a few years.

[3]Who, in my experience, tend to be rather reluctant to using computers for programming-like activities.

Chapter 8

Conclusion

For this research I proved the Perfect Number Theorem in Isabelle. I chose the Perfect Number Theorem as case study for Isabelle because it is in a list of 100 theorems which is used to showcase theorem provers. I submitted my final proof to the Archive of Formal Proofs[1], and it will be added shortly. It has already been announced on the top 100 list[8]

From proving the theorem I have obtained a good impression of Isabelle. The language is powerful, and the library is good, however the interface could be better. I also saw some concrete points which could be improved upon, which I mention in the appendix, and which will be submitted to the respective developers.

Lastly, I provided some pointers in this thesis for students/mathematicians who want to start using Isabelle.

Bibliography

[1] Archive of formal proofs, 2009. http://afp.sourceforge.net/.

[2] Hol (higher-order logic) theory library, 2009. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/index.html.

[3] Isabelle / proof general cheat sheet, 2009. http://www.phil.cmu.edu/~avigad/formal/FormalCheatSheet.pdf.

[4] Isabelle mailinglist, 2009. http://lists.cam.ac.uk/mailman/listinfo/cl-isabelle-users.

[5] Isabelle website, 2009. http://isabelle.in.tum.de/index.html.

[6] Tobias Nipkow. A tutorial introduction to structured isar proofs, 2009. http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-overview.pdf.

[7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[8] Freek Wiedijk. Formalizing 100 theorems, 2009. http://www.cs.ru.nl/~freek/100/.

Appendix A

Bugs and shortcomings in the used programs

As I already mentioned in my general evaluation, I ran into several bugs and shortcomings regarding the interface part of Isabelle and related tools. In this chapter I will name a few in more detail. To any developers reading those sections, please don't take them the wrong way, I appreciate all the tools I used, and I noticed a lot of progress in usability when switching to Isabelle2009. However, I really believe the usability might be the biggest hurdle at the moment when starting to prove with a theorem prover, as a mathematician.

A.1 Isabelle: Minus one

In Isabelle "-" can be part of an identifier. This has the annoying side-effect that "n-1" gets parsed as one identifier. I do not see why one would want the minus sign to be part of an identifier. Most programming languages do just fine without the minus in identifiers, since you can use the underscore for the same functionality. Therefore, I would advise disabling the possibility to use minus in identifiers.

A.2 Isabelle: Problems with Sledgehammer

When you invoke Sledgehammer, and it succeeds in delivering a metis command with a list of theorems to prove the current step, it delivers a (clickable) **apply** command. When you click on this command, it inserts the command into the proof as a proved step. There are several problems with this. The first is that it delivers an **apply** command, where in a forward proof a **by** would be desirable. And because it already adds the **apply** as part of the locked section, you first need to undo it, and replace it by **by**. After this you need to reprove the command, which (as is usual with metis) takes quite some time.

A second, more annoying, problem is that the list of theorems is overcomplete. This shows itself in two ways. The first is that the list of theorems contains theorems which are not used at all. The second is that the list contains facts from the proof itself, which are already included by **with** or using.

To add insult to injury, when you apply the proof, the list of unused theorems is mentioned, but this disappears when you retract the proof. This means that to remove the unused theorems you first need to copy the list of unused theorems to a scratchpad somewhere, then retract the proof, and then manually search for the theorems in the list of used theorems, and remove them.

Thus, my advice for the interface:

1. Detect whether we are in a forward proof or a backward proof, and offer **apply** or **by** depending on which one, or alternatively, always offer both.
2. After receiving a proof command from a prover, remove the theorems already in **with** or using, and actually **apply** the proof step by Isabelle, and remove unused theorems. Alternatively, offer an option to remove unused theorems by just clicking on the list, preferably without unlocking the command.

A.3 Isabelle: Error messages

The error messages tend to be a bit unclear. I think a bit of brainstorming about common errors, and displaying useful error messages for these might help a lot. As an example one might look at how compilers developed in this aspect over the last decades.

A.4 Isabelle: find solutions to toplevel goals

The new Isabelle has the (very useful) feature of automatically finding solutions to top-level goals. However, when it presents such a solution, it only returns a message like:

```
The current goal could be solved directly with:
  Groebner_Basis.class_semiring.mul_d: ?x * (?y + ?z) = ?x * ?y +
      ?x * ?z
  Groebner_Basis.class_semiring.semiring_rules(34): ?x * (?y + ?z)
      = ?x * ?y + ?x * ?z
  Nat.add_mult_distrib2: ?k * (?m + ?n) = ?k * ?m + ?k * ?n
  Nat.nat_distrib(2): ?k * (?m + ?n) = ?k * ?m + ?k * ?n
  Ring_and_Field.field_class.field_eq_simps(21): ?ab * (?bb + ?ca)
      = ?ab * ?bb + ?ab * ?ca  [field, pordered_ab_group_add]
```

Why does it not provide a way to use the solution, such as offer a clickable button as used in the result of Sledgehammer?

A.5 Proof General: Undo in locked regions

In Proof General, if you prove a part, the proved region gets locked. This makes it impossible to edit this section. However, if you undo an edit, you can also undo edits in the locked region. Though this does not affect the "in-memory" proof, it affects what is seen on the screen, and therefore makes it possible to have a fully checked proof on screen, which is false. Either undo should be blocked in locked regions, or when an undo is done in a locked region, the locking should roll back to before the undo.

Appendix B

Isabelle Proof

The following proof is the LaTeXversion of the Isabelle proof, autogenerated from the Isabelle source of the proof.

B.1 Basics needed

theory *PerfectBasics*
imports *Main Divides Primes NatBin* ~~/src/HOL/Algebra/Exponent*
begin

lemma *setsum-mono4*:
  assumes *finite* (*A*::*nat set*) **and** *finite* (*B*::*nat set*) **and** *A* <= *B*
  shows $\sum$ *A* <= $\sum$ *B*
by (*auto simp add*: *setsum-mono2 assms*)

lemma *emptysetsumstozero*: *A*={} ==> $\sum$ (*A*::*nat set*) = *0* **by** *simp*
lemma *smallerorequal*: (*x*::*nat*) <= *Suc n* <−> *x* <= *n* ∨ *x* = *Suc n* **by** *auto*
lemma *seteq-imp-setsumeq*: *A*=*B* ==> $\sum$ *A* = $\sum$ *B* **by** *auto*

lemma *exp-is-max-div*:
  assumes *m0*:*m*>*0* **and** *p*: *prime p*
  shows ~ *p dvd* (*m div* (*p*ˆ(*exponent p m*)))
proof (*rule ccontr*)
 assume ~ ~ *p dvd* (*m div* (*p*ˆ(*exponent p m*)))
 hence *a*:*p dvd* (*m div* (*p*ˆ(*exponent p m*))) **by** *auto*
 from *m0* **have** *p*ˆ(*exponent p m*) *dvd m* **by** (*auto simp add*: *power-exponent-dvd*)
 with *a* **have** *p*∗(*p*ˆ*exponent p m*) *dvd m* **by** (*metis divides-mul-l dvd-mult-div-cancel local.a nat-mult-commute*)
 with *p* **have** *m*=*0* **by** (*auto simp add*: *power-Suc-exponent-Not-dvd*)
 with *m0* **show** *False* **by** *auto*
qed

lemma *coprime-exponent*:
 assumes *p*:*prime p* **and** *m*:*m*>*0*
 shows *coprime p* (*m div* (*p*ˆ(*exponent p m*)))
proof (*rule ccontr*)
 assume ~ *coprime p* (*m div p* ˆ *exponent p m*)
 hence *EX q*. *prime q* & *q dvd p* & *q dvd* (*m div* (*p*ˆ(*exponent p m*))) **by** (*auto simp*

```
add: coprime-prime-dvd-ex)
  hence EX q. q = p & q dvd (m div (p^(exponent p m)))
apply (metis p prime-1 prime-def) done
  hence  EX q. p dvd (m div (p^(exponent p m))) by auto
  hence p dvd (m div (p^(exponent p m))) by auto
  with p m show False by (auto simp add: exp-is-max-div)
qed

lemma add-mult-distrib-three: (x::nat)*(a+b+c)=x*a+x*b+x*c
proof –
  have (x::nat)*(a+b+c) = x*((a+b)+c) by auto
  hence x*(a+b+c) = x*(a+b)+x*c by (metis add-mult-distrib2 nat-add-commute
nat-add-left-commute)
  thus x*(a+b+c) = x*a+x*b+x*c by (metis add-mult-distrib2 nat-add-commute
nat-add-left-commute)
qed

lemma nat-interval-minus-zero: {0..Suc n} = {0} Un {Suc 0..Suc n} by auto
lemma nat-interval-minus-zero2:
 assumes n>0
 shows {0..n} = {0} Un {Suc 0..n} by (auto simp add: nat-interval-minus-zero)

lemma distribute-min-mult: ((a::nat) – 1)*c=a*c – c by (metis diff-mult-distrib2
nat-mult-1-right nat-mult-commute)
theorem simplify-sum-of-powers: (x – 1::nat) * (∑ i=0 .. n . x^i)  = x^(n + 1) –
1 (is ?l = ?r)
proof (cases)
  assume n = 0
  thus ?l = x^(n+1) – 1 by auto
  next
  assume n~=0
  hence n0: n>0 by auto
  have ?l  = (x::nat)*(∑ i=0 .. n . x^i) – (∑ i=0 .. n . x^i) by (simp only:
distribute-min-mult)
  also have ... = (∑ i=0 .. n . x^(Suc i))    – (∑ i=0 .. n . x^i) by (simp add:
setsum-right-distrib)
  also have ... = (∑ i=Suc 0 .. Suc n . x^i)  – (∑ i=0 .. n . x^i) by (metis
One-nat-def setsum-shift-bounds-cl-Suc-ivl)
  also with n0 have ... = ((∑ i=Suc 0 .. n . x^i)+x^(Suc n)) – (x^0 + (∑ i=Suc
0 .. n . x^i))  by (auto simp add: setsum-Un-disjoint nat-interval-minus-zero2)
  finally show ?thesis by auto
qed

end
```

B.2 Sum of divisors function

```
theory Sigma
imports Main Divides Primes NatBin PerfectBasics Infinite-Set
begin

constdefs
  divisors :: nat => nat set
  divisors (m::nat) == {(n::nat) . (n::nat) dvd m}
  sigma :: nat => nat
  sigma m == ∑ n |n dvd m . n

lemma sigmadivisors: sigma(n) = ∑ (divisors(n)) by (auto simp: sigma-def divisors-def)

lemma divisors-eq-dvd[iff]: (a:divisors(n)) = (a dvd n)
apply (subst divisors-def)
apply (subst mem-def)
apply (subst Collect-def)
apply blast
done

lemma mult-divisors: (a::nat)*b=c==>a: divisors c by (unfold divisors-def dvd-def,blast)
lemma mult-divisors2: (a::nat)*b=c==>b: divisors c by (unfold divisors-def dvd-def,auto)

lemma divisorsfinite[simp]:
   assumes n>0
   shows finite (divisors n)
proof −
   from assms have  divisors n = {m . m dvd n & m <= n} by (auto simp
only:divisors-def dvd-imp-le)
  hence divisors n <= {m . m<=n} by auto
  thus finite (divisors n) by (metis Collect-def finite-Collect-le-nat finite-subset)
qed

lemma everything-div-of-zero[simp]: m:divisors(0) by(auto simp add: divisors-def)
lemma zerodivisors-infinite[simp]: infinite(divisors 0)
proof −
  have ALL (m::nat). EX n. m<n & n:divisors(0) by auto
  thus infinite(divisors 0) by (auto simp add: infinite-nat-iff-unbounded)
qed

lemma sigma0[simp]: sigma(0) = 0 by (simp add: sigma-def)
lemma sigma1[simp]: sigma(1) = 1 by (simp add: sigma-def)

lemma primedivisors: prime (p::nat) <−> divisors p = {1,p} & p>1 by (auto simp
add: divisors-def prime-def)

lemma prime-imp-sigma: prime (p::nat) ==> sigma(p) = p+1
proof −
  assume prime (p::nat)
  hence p>1 ∧ divisors(p) = {1,p} by (simp add: primedivisors)
  hence p>1 ∧ sigma(p) = ∑ {1,p} by (auto simp only: sigmadivisors divisors-def)
  thus sigma(p) = p+1 by simp
qed

lemma sigma-third-divisor:
  assumes 1 < a a < n a : divisors n
  shows 1+a+n <= sigma(n)
proof −
  from assms have finite {1,a,n} & finite (divisors n) & {1,a,n} <= divisors n by
auto
  hence ∑ {1,a,n} <= ∑ (divisors n) by (simp only: setsum-mono2)
  hence ∑ {1,a,n} <= sigma n by (simp add: sigmadivisors)
  with assms show ?thesis by auto
qed

lemma divisor-imp-smeq: a : divisors (Suc n) ==> a <= Suc n
apply (auto simp add: divisors-def)
apply (metis Suc-neq-Zero Suc-plus1 divides-ge)
done

lemma sigma-imp-divisors: sigma(n)=n+1 ==> n>1 & divisors n = {n,1}
proof
  assume ass:sigma(n)=n+1
  hence n~=0 & n~=1
    apply auto
    apply (metis gr0I n-not-Suc-n sigma0)
    apply (metis One-nat-def n-not-Suc-n sigma1)
  done
  thus conc1: n>1 by simp

  show divisors n = {n,1}
  proof (rule ccontr)
    assume divisors n ~= {n,1}
    with conc1 have divisors n ~= {n,1} & 1<n by auto
    moreover from ass conc1 have 1 : divisors(n) & n : divisors n & ~0 : divisors
n by (simp add: dvd-def divisors-def)
     ultimately have (EX a . a~=n & a~=1 & 1<n & a : divisors n) & ~ 0 :
divisors n by auto
     hence         (EX a . a~=n & a~=1 & 1<n & a~=0 & a : divisors n) by
metis
    hence         EX a . a~=n & a~=1 & 1~=n & a~=0 & finite {1,a,n} & finite
(divisors n) & {1,a,n} <= divisors n by auto
    hence         EX a . a~=n & a~=1 & 1~=n & a~=0 & ∑ {1,a,n} <= sigma
nby (metis setsum-mono4 sigmadivisors)
    hence         EX a . a~=0 & (1+a+n) <= sigma n by auto
    hence         1+n<sigma n by auto
    with ass show False by auto
  qed
qed


lemma sigma-imp-prime: sigma(n)=n+1 ==> prime n
proof -
  assume ass:sigma(n)=n+1
  hence n>1 & divisors(n)={1,n} by (metis insert-commute sigma-imp-divisors)
  thus prime n by (simp add: primedivisors)
qed


lemma dvd-imp-divisor:
  assumes x dvd y
  shows x : (divisors y)
proof
  from assms show x dvd y by auto
qed
```

```
lemma pr-pow-div-eq-sm-pr-pow:
  assumes prime: prime p
  shows {d . d dvd p^n} = {p^f| f . f<=n}
proof
  show {p^f | f . f<=n} <= { d .  d dvd p^n}
  proof
    fix x
    assume x: {p  ^ f | f . f <= n}
    hence EX i . x = p^i & i<= n   by auto
    with prime have  x dvd p^n by (auto simp add: divides-primepow)
    thus x : { d . d dvd p^n} by auto
  qed
  next
  show {d. d dvd p  ^ n} <= {p  ^ f | f . f <= n}
  proof
    fix x
    assume x : {d . d dvd p^n}
    hence x dvd p^n by auto
   with prime obtain i where  i <= n & x = p^i by (auto simp only: divides-primepow)
    hence x = p^i & i <=n by auto
    thus x : { p^f | f . f<=n } by auto
  qed
qed

lemma rewsop-help: {f m | (m::nat) . m<=n} = f`{m . m<=n} by (subst image-def,blast)
lemma rewrite-sum-of-powers:
assumes p: (p::nat)>1
shows (∑ {p^m | m . m<=(n::nat)}) = (∑ i = 0 .. n . p^i) (is ?l = ?r)
proof −
  have ?l = setsum id {(op  ^ p) m |m . m<= n} by auto
  also have ... = setsum id ((op  ^ p)`{m . m<= n}) by (simp only: rewsop-help)
 moreover with p have inj-on (op ^p) {m . m<=n} by (auto simp add: inj-on-def)
  ultimately have      ?l = setsum (op  ^ p) {m . m<=n} by (auto simp only:
setsum-reindex-id)
  moreover have {m::nat . m<=n} = {0..n} by auto
  ultimately show ?l = (∑ i = 0 .. n . p^i) by auto
qed

theorem sigmaprimepower: prime p ==> (p − 1)*sigma(p^(e::nat)) =  (p^(e+1)
− 1)
proof −
  assume prime p
 hence sigma(p^(e::nat)) = (∑ i=0 .. e . p^i) by (auto simp add: pr-pow-div-eq-sm-pr-pow
sigma-def rewrite-sum-of-powers prime-def)
 thus (p − 1)*sigma(p^e)=p^(e+1) − 1 by (auto simp only: simplify-sum-of-powers)

qed

lemma sigmaprimepowertwo: sigma(2^(n::nat))=2^(n+1) − 1
proof −
  have (2 − 1)*sigma(2^(n::nat))=2^(n+1) − 1 by (auto simp only: sigmaprime-
power two-is-prime)
  thus sigma(2^(n::nat))=2^(n+1) − 1 by auto
```

```
qed

lemma sigma-finiteset1[simp]:
assumes m0: (m::nat)>0 and p1: p>1
shows finite {p ^ f * b | f b . f <= n & b dvd m}
proof −
  have {p^f * b | f b . f <= n & b dvd m} <= {0 .. p^n*m}
  proof
    fix x
    assume x : {p ^ f * b |f b. f <= n & b dvd m}
    then obtain f b where x=p^f*b & f<=n & b dvd m by auto
     with m0 p1 have x=p^f*b & f<=n & b <= m & p>1 by (auto simp add:
dvd-imp-le)
    hence           x<=p^n*b & b<=m by auto
    hence x<=p^n*m by (metis le-trans mult-le-cancel2 nat-mult-commute)
    thus x : {0 .. p^n*m} by auto
  qed
  thus finite {p^f * b | f b . f <= n & b dvd m} by (simp add: finite-subset)
qed


lemma sigma-finiteset2[simp]:
assumes m0: m>0
shows m>0 ==> finite {(x::nat) * b |b. b dvd m}
proof −
  from m0 have finite (divisors m) by simp
  hence finite ((op * x)`(divisors m)) by auto
  moreover have {x * b |b. b dvd m} = (op * x)`(divisors m) by (auto simp add:
divisors-def)
  ultimately show ?thesis by auto
qed


lemma prodsums-eq-sumprods-help2:
  assumes ndvd: ~ p dvd m and p1: p>(1::nat)
  shows ∑ ({p ^ f * b | f b . f <= n & b dvd m} Int {p^f*b | f b . f=Suc n & b
dvd m}) = 0
proof −
  have !!b f ba. [| f <= n; p * p ^ n * b = p ^ f * ba; b dvd m; ba dvd m |] ==>
False
  proof −
    fix b ba f
    assume ass: f <= n p * p ^ n * b = p ^ f * ba b dvd m ba dvd m
    then obtain e where edef: f+e = Suc n by (metis Suc-plus1 le-add-diff-inverse
that trans-le-add1)
    hencep^(Suc n) = p^(f+e) by auto
    hence p^(Suc n) = p^f*p^e by (simp only: power-add)
    with ass have p^f*p^e * b = p ^ f * ba  by auto
    with p1 have p^e*b=ba by auto
    moreover from edef ass have e>0 by auto
    ultimately have p dvd ba by auto
    with ass have p dvd m by (metis dvd.order-trans )
    with ndvd show False by auto
  qed
```

thus ?thesis
apply (simp only: Int-def)
apply auto
apply (rule emptysetsumstozero)
apply auto
done
qed

lemma sum-pow-plus-suc-eq-sum-upto-suc:
assumes p:(p::nat)>1
shows $\sum$ {p ^ f |f. f <= n} + p^(Suc n)= $\sum$ {p ^ f |f. f <= Suc n}
(**is** ?lhs = ?rhs)
proof −
from p **have** ?lhs = ($\sum$ i = 0 .. n . p^i) + p^(Suc n) **by** (simp only: rewrite-sum-of-powers)

hence ?lhs = ($\sum$ i = 0 .. Suc n . p^i) **by** simp
with p **show** ?lhs = ?rhs **by** (subst rewrite-sum-of-powers)
qed

lemma rewrite-power-times-sum:
assumes p1:(p::nat)>1
shows p^x*($\sum$ {b . b dvd m}) = $\sum$ {p^f*b | f b . f=x & b dvd m} (**is** ?l = ?r)
proof −
have ?l = setsum (op * (p^x)) {b . b dvd m} **by** (auto simp add: setsum-right-distrib)
moreover from p1 **have** inj-on (op * (p^x)) {b . b dvd m} **by** (simp add: inj-on-def)
ultimately also have ?l = (setsum id ((op * (p^x))' {b . b dvd m})) **by** (auto simp only: setsum-reindex-id)
thus ?thesis **by** (auto simp add:conj-commute image-def)
qed

lemma prodsums-eq-sumprods-help-help: (A&B&C)|(A&D&C)<−>A&(B|D)&C **by** blast
lemma prodsums-eq-sumprods-help:
assumes m0: (m::nat)>0 **and** p1:(p::nat)>1 **and** cop: coprime p m
shows $\sum$ {p^f*b | f b . f <= n & b dvd m} + p^(Suc n)*($\sum$ {b . b dvd m}) = $\sum$ {p^f*b | f b . f <= Suc n & b dvd m}
(**is** $\sum$?lhsa + ?lhsb = $\sum$?rhs)
proof −
from p1 **have** ?lhsb = ($\sum$ {p^f*b | f b . f=Suc n & b dvd m}) (**is** ?lhsb = $\sum$?lhsbn) **by** (auto simp only: rewrite-power-times-sum)
moreover from m0 p1 **have** finite ?lhsa **by** simp
moreover from m0 **have** finite ?lhsbn **by** simp
ultimately have $\sum$?lhsa + ?lhsb = $\sum$ (?lhsa Un ?lhsbn) + $\sum$ (?lhsa Int ?lhsbn) **by** (auto, auto simp only: setsum-Un-Int)
moreover {
have $\sum$ (?lhsa Un ?lhsbn) = $\sum$ ({x . (EX f b. (x = p ^ f * b & f <= n & b dvd m) $\vee$ (x = p^f*b & f=Suc n & b dvd m))}) **by**(rule seteq-imp-setsumeq, auto simp del:power-Suc)
also have ... = $\sum$ {p ^ f * b | f b . (f <= n $\vee$ f=Suc n) & b dvd m} **by** (subst

prodsums-eq-sumprods-help-help, *auto*)
  finally have $\sum$ (*?lhsa Un ?lhsbn*) = $\sum$ {*p* ˆ *f* ∗ *b* | *f b* . *f* <= *Suc n* & *b dvd m*} **by** (*simp only*: *smallerorequal*)}
 moreover{
  from *cop p1* **have** ~ *p dvd m* **by** (*metis coprime-def dvd-div-mult-self gcd-mult′ nat-less-le*)
  with *p1* **have** ~ *p dvd m p*>*1* **by** *auto*
  hence $\sum$ (*?lhsa Int ?lhsbn*) = *0* **by** (*rule prodsums-eq-sumprods-help2*)}
 ultimately show *?thesis* **by** *auto*
qed

lemma *prodsums-eq-sumprods*:
 assumes *p1*: *p*>*Suc 0* **and** *cop*: *coprime p m* **and** *m0*: *m*>*0*
 shows ($\sum$ {*p*ˆ*f*| *f* . *f*<=*n*})∗($\sum$ {*b* . *b dvd m*}) = ($\sum$ {*p*ˆ*f*∗*b*| *f b* . *f*<=*n* & *b dvd m*})
proof (*induct n*)
 show $\sum${(*p*::*nat*) ˆ *f* |*f*. *f* <= *0*} ∗ $\sum${*b*. *b dvd m*} = $\sum${*x*. *EX f b*. *x* = *p* ˆ *f* ∗ *b* & *f* <= *0* & *b dvd m*} **by** *auto*
 next
 fix *n*
 show $\sum$ {(*p*::*nat*) ˆ *f* |*f*. *f* <= *n*} ∗ $\sum${*b*. *b dvd m*} = $\sum${*p* ˆ *f* ∗ *b* | *f b* . *f* <= *n* & *b dvd m*}
    ==> $\sum${*p* ˆ *f* |*f*. *f* <= *Suc n*} ∗ $\sum${*b*. *b dvd m*} = $\sum${*p* ˆ *f* ∗ *b* | *f b* . *f* <= *Suc n* & *b dvd m*} (**is** *?lhs* = *?rhs* ==> *?lhsn* = *?rhsn*)
 proof −
  assume *?lhs* = *?rhs*
  hence *?lhs* + *p*ˆ(*Suc n*)∗($\sum${*b* . *b dvd m*})= *?rhs*+ *p*ˆ(*Suc n*)∗($\sum${*b* . *b dvd m*}) **by** *auto*
  moreover{**have** $\sum${*p* ˆ *f* |*f*. *f* <= *n*} ∗ $\sum${*b*. *b dvd m*} + *p*ˆ(*Suc n*)∗($\sum${*b* . *b dvd m*})=($\sum${*p* ˆ *f* |*f*. *f* <= *n*} + *p*ˆ(*Suc n*))∗($\sum${*b* . *b dvd m*}) **by** (*simp add*: *add-mult-distrib*)
   with *p1* **have** $\sum${*p* ˆ *f* |*f*. *f* <= *n*} ∗ $\sum${*b*. *b dvd m*} + *p*ˆ(*Suc n*)∗($\sum${*b* . *b dvd m*})=*?lhsn* **by** (*simp only*: *sum-pow-plus-suc-eq-sum-upto-suc prime-def*)}
  moreover from *m0 p1 cop* **have** $\sum${*p* ˆ*f*∗*b* | *f b* . *f* <= *n* & *b dvd m*}+ *p*ˆ(*Suc n*)∗($\sum${*b* . *b dvd m*}) = *?rhsn* **by** (*subst prodsums-eq-sumprods-help*,*auto*)
  ultimately show *?lhsn* = *?rhsn* **by** *simp*
 qed
qed

lemma *rewrite-for-sigmasemimultiplicative*:
 assumes *p*: *prime p*
 shows {*p*ˆ*f*∗*b*| *f b* . *f*<=*n* & *b dvd m*} = {*a*∗*b*| *a b* . *a dvd* (*p*ˆ*n*) & *b dvd m*}
proof
 show {*p* ˆ *f* ∗ *b* | *f b*. *f* <= *n* & *b dvd m*} <= {*a* ∗ *b* |*a b*. *a dvd p* ˆ *n* & *b dvd m*}
 proof
  fix *x*
  assume *x* : {*p* ˆ *f* ∗ *b* | *f b*. *f* <= *n* & *b dvd m*}
  then obtain *b f* **where** *x* = *p*ˆ*f*∗*b* & *f* <= *n* & *b dvd m* **by** *auto*
  with *p* **show** *x* : {*a* ∗ *b* |*a b*. *a dvd p* ˆ *n* & *b dvd m*} **by** (*auto simp add*: *divides-primepow*)
 qed

```
  show {a * b |a b. a dvd p ^ n & b dvd m} <= {p ^ f * b |f b. f <= n & b dvd m}
  proof
    fix x
    assume x:{a * b |a b. a dvd p ^ n & b dvd m}
    then obtain a b where abdef: x = a*b & a dvd p^n & b dvd m by auto
    moreover with p obtain i where a = p^i & i<=n & b dvd m by (auto simp
add: divides-primepow)
    ultimately show x : {p ^ f * b |f b. f <= n & b dvd m} by auto
  qed
qed


lemma div-decomp-comp:
  assumes cop:coprime m n
  shows a dvd m*n <-> (EX b c . a = b * c & b dvd m & c dvd n) by (auto simp
only: division-decomp mult-dvd-mono)


theorem sigmasemimultiplicative:
  assumes p: prime p and cop: coprime p m and m0:m>0
  shows sigma (p^n) * sigma m = sigma (p^n * m) (is ?l = ?r)
proof -
 from cop have cop2: coprime (p^n) m by (auto simp add: coprime-exp coprime-commute)
  have              ?l=(∑ {a . a dvd p^n})*(∑ {b . b dvd m}) by (simp add:
sigma-def)
  also from p  have     ...=(∑ {p^f| f . f<=n})*(∑ {b . b dvd m}) by (simp
add: pr-pow-div-eq-sm-pr-pow)
  also from m0 p cop  have ... = (∑ {p^f*b| f b . f<=n & b dvd m}) by (auto
simp add: prodsums-eq-sumprods prime-def)
   also have           ... = (∑ {a*b| a b . a dvd (p^n) & b dvd m}) by (rule
seteq-imp-setsumeq,rule rewrite-for-sigmasemimultiplicative[OF p])
  finally have       ?l = (∑ {c . c dvd (p^n*m)}) by (subst div-decomp-comp[OF
cop2])
  thus               ?l = sigma (p^n*m) by (auto simp add: sigma-def)
qed

end
```

B.3 Perfect Number Theorem

```
theory Perfect
imports Main Divides Primes NatBin Sigma
begin

constdefs
  perfect :: nat => bool
  perfect m == m>0 & 2*m = sigma m

theorem perfect-number-theorem:
  assumes even: even m and perfect: perfect m
  shows ∃ n . m = 2^n*(2^(n+1) − 1) ∧ prime (2^(n+1) − 1)
proof
  from perfect have m0: m>0 by (auto simp add: perfect-def)
```

```
let ?n = exponent 2 m
let ?A = m div 2^?n
let ?np = (2::nat)^(?n+1) − 1

from even have 2 dvd m by (simp add: nat-even-iff-2-dvd)
with m0 have n1: ?n >= 1 by (simp add: exponent-ge two-is-prime)

from m0 have 2^?n dvd m by (rule power-exponent-dvd)
hence m = 2^?n∗?A by (simp only: dvd-mult-div-cancel)
with m0 have mdef: m=2^?n∗?A & coprime 2 ?A by (simp add: two-is-prime coprime-exponent)
moreover with m0 have a0: ?A>0 by (metis gr0I less-not-refl nat-0-less-mult-iff)

moreover{from perfect have 2∗m=sigma(m) by (simp add: perfect-def)
        with mdef have 2^(?n+1)∗?A=sigma(2^?n∗?A)by auto}
ultimately have 2^(?n+1)∗?A=sigma(2^?n)∗sigma(?A) by (simp add: sigmasemimultiplicative two-is-prime)
hence formula: 2^(?n+1)∗?A=(?np)∗sigma(?A) by (simp only: sigmaprime-powertwo)

from n1 have (2::nat)^(?n+1) >= 2^2 by (simp only: power-increasing)
hence nplarger:?np>= 3 by auto


let ?B = ?A div ?np

from formula have ?np dvd 2^(?n+1)∗?A by auto
hence ?np dvd ?A by (metis Suc-plus1 coprime-divprod coprime-minus1 nat.simps(2) nat-1-add-1 nat-mult-commute nat-power-eq-0-iff)
hence bdef: ?np∗?B = ?A by (simp add: dvd-mult-div-cancel)
with a0 have b0: ?B>0 by (metis Suc-eq-add-numeral-1-left Suc-plus1 div-mult2-eq nat-0-less-mult-iff nat-1-add-1)

from nplarger a0 have bsmallera: ?B < ?A by auto

have ?B = 1
proof (rule ccontr)
  assume ~?B = 1
  with b0 bsmallera have 1<?B ?B<?A by auto
  moreover from bdef have ?B : divisors ?A by (rule mult-divisors2)
  ultimately have 1+?B+?A <= sigma ?A by (rule sigma-third-divisor)
  with nplarger have ?np∗(1+?A+?B) <= ?np∗(sigma ?A) by (auto simp only: nat-mult-le-cancel1)
  with bdef have ?np+?A∗?np + ?A∗1 <= ?np∗(sigma ?A) by (simp only: add-mult-distrib-three mult-commute)
  hence ?np+?A∗(?np + 1) <= ?np∗(sigma ?A) by (simp only: add-mult-distrib2)
  with nplarger have (2^(?n+1))∗?A < ?np∗(sigma ?A) by (simp add: mult-commute)
  with formula show False by auto
qed

with bdef have adef: ?A=?np by auto
```

```
  with formula have   ?np*2^(?n+1) =(?np)*sigma(?A) by auto
  with nplarger adef have          ?A + 1=sigma(?A) by auto
  with a0 have prime ?A by (simp add: sigma-imp-prime)
  with mdef adef show m = 2^?n*(?np) & prime ?np by simp
qed

lemma even-minus-one-odd:
  assumes even (n::nat) n>0
  shows odd (n − 1)
proof
  assume even (n − 1)
  with assms have  odd n by (simp add: even-num-iff)
  with assms show False by auto
qed

theorem euclb9prop36:
  assumes p: prime (2^(n+1) − (1::nat))
  shows perfect ((2^n)*(2^(n+1) − 1))
proof (unfold perfect-def, auto)
  from assms show (2::nat)*2^n > Suc 0 by (auto simp add: prime-def)
  next
  have p2: prime 2 by (rule two-is-prime)
  moreover from p have prime (2^(n+1) − 1) by simp
  moreover {
    have even ((2::nat)^(n+1)) ((2::nat)^(n+1))>0 by auto
    hence odd ((2::nat)^(n+1) − 1) by (rule even-minus-one-odd)
    hence 2 ~= ((2::nat)^(n+1) − 1) by auto}
  ultimately have coprime 2 (2^(n+1) − 1) by (rule distinct-prime-coprime)
  with p p2 have prime 2 coprime 2 (2^(n+1) − 1) 2^(n+1) − 1 > (0::nat) by
(auto simp add: prime-def)
  hence  sigma (2^n*(2^(n+1) − 1)) = (sigma(2^n))*(sigma(2^(n+1) − 1)) by
(simp only: sigmasemimultiplicative)
 also from assms have ... = (sigma(2^(n)))*(2^(n+1)) by (auto simp add: prime-imp-sigma)
  also have ... = (2^(n+1) − 1)*(2^(n+1)) by (simp add: sigmaprimepowertwo)
  finally show 2 * (2 ^ n * (2 * 2 ^ n − Suc 0)) = sigma (2 ^ n * (2 * 2 ^ n −
Suc 0)) by auto
qed

end
```