

Diploma Thesis

Formalizing the DSA Signature Scheme in  
Isabelle/HOL



by

Sebastian Kusch

Darmstadt University of Technology

- Department of Computer Science -

Theoretical Computer Science Group

Prof. Dr. Johannes Buchmann

Supervisor: Dipl.-Math. Markus Kaiser

December 2006

---

## **Ehrenwörtliche Erklärung**

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, Dezember 2006

Sebastian Kusch

---

## Acknowledgements

I would like to thank Prof. J. Buchmann for introducing me to the captivating field of cryptography and Markus Kaiser for helping me find my way through the Isabelle system and his advices, Lawrence C. Paulson and his colleagues for creating the system and making it available to the world. I also want to thank my parents and my little sister for their constant support, Gisela for her love, cheering me up and providing me with drinks whenever I was about dry out, along with René and Hartmut who kept me from working too hard.

## CONTENTS

1. <i>Introduction</i> . . . . .	6
2. <i>Digital Signatures</i> . . . . .	9
2.1 Objectives . . . . .	9
2.2 Necessary Ingredients . . . . .	10
2.2.1 Digital Signatures based on Symmetric Cryptography .	10
2.2.2 Digital Signatures based on Asymmetric Cryptography	11
2.3 The DSA Signature Scheme . . . . .	13
2.3.1 DSA Parameters . . . . .	14
2.3.2 Signature Generation . . . . .	14
2.3.3 Signature Verification . . . . .	15
3. <i>Formal Languages</i> . . . . .	16
3.1 First Order Logic . . . . .	17
3.1.1 Syntax and Semantics . . . . .	17
3.1.2 A Sequent Calculus . . . . .	18
3.1.3 Soundness and Completeness . . . . .	19
3.2 Towards HOL . . . . .	21
3.2.1 Limitations of FOL . . . . .	21
3.2.2 Extensions within FOL . . . . .	22
3.2.3 Going beyond FOL . . . . .	23
3.3 Authentication Logic . . . . .	25
4. <i>The Isabelle/HOL System</i> . . . . .	27
4.1 The System . . . . .	27
4.2 Theorem Proving . . . . .	28
4.2.1 Theories and Axiomatic Type Classes . . . . .	28
4.2.2 An introductory proof . . . . .	29
5. <i>Prior Contributions</i> . . . . .	32
5.1 Paulson's Inductive Method . . . . .	32
5.2 Formalization of SET . . . . .	35
5.2.1 The Protocol and its Objectives . . . . .	35

5.2.2	Verification of SET in Isabelle . . . . .	37
5.3	Other Areas . . . . .	40
6.	<i>Correctness of DSA</i> . . . . .	43
6.1	A “Pen and Paper” Proof . . . . .	44
6.2	The formal proof . . . . .	46
6.2.1	A Motivation . . . . .	46
6.2.2	The Roadmap of the Proof . . . . .	48
6.2.3	What the Theorem expresses . . . . .	53
7.	<i>Conclusions</i> . . . . .	55
7.1	My practical Experience . . . . .	55
7.2	Benefits of Formalization . . . . .	56
7.3	What is left to be done . . . . .	57
	 <i>Appendix</i>	 59
A.	<i>The proofs in detail</i> . . . . .	60
A.1	The most frequent tools . . . . .	60
A.2	A closer look . . . . .	61
A.3	The file <code>dsa.thy</code> . . . . .	69

## 1. INTRODUCTION

In this thesis I am going to present a formalization of the DSA signature scheme in the Isabelle/HOL system. In contrast to high-level examinations of cryptographic protocols like the key exchange schemes suggested by Needham and Schroeder for instance, formalizations involving also the single algebraic equations are a relatively new application of theorem provers like Isabelle.

The Digital Signature Algorithm (DSA) was proposed by the United States Institute of Standards (NIST) in 1991 to be used in the Digital Signature Standard (DSS). The DSA is an efficient version of the ElGamal signature scheme based on discrete logarithms since it mainly reduces the bitlength of the exponents involved. The DSA signature scheme will be explained in detail in section 2.3.

The Isabelle system is a generic theorem proving environment developed at Cambridge University and TU Munich that can be applied to several logics. Isabelle/HOL is the specialization of Isabelle for higher order logic (HOL) whereas other logics distributed with Isabelle include the usual first order logic (FOL) or LCF which is a version of Scott's logic for computable functions. Isabelle is also often referred to as a "Proof Assistant" underlining the process of alternating automated reasoning with human intervention. However, in this thesis I will try to stick to the conventional term "Theorem Prover". A deeper introduction to the Isabelle and the Isabelle/HOL system, respectively, can be found in chapter 4.

The technical contribution of this work is a formal "one-sided" justification of the verifying scheme, i.e. showing that a legitimate signature will be actually accepted which is apparently a necessary condition for the usefulness of any cryptographic signature scheme.

One might ask why it is worth trying to formalize such a scheme (or algorithm) in order to derive properties. In my view the answer is quite simple: We apply such schemes to achieve some vital security goals like authenticity

or privacy. Algorithms involved here deserve in some sense more attention and “care” than any ingredients for word processing or algorithms in charge of sorting.

This project is an early step into the direction of formally verifying cryptographic algorithms themselves (not only protocols based on them). Even if the proof will probably turn out much longer and is likely to look a bit more complex than its “hand written” counterparts it will demonstrate that cryptographic algorithms are actually amenable to formal analysis.

Much effort has already been devoted in the field of formal program verification, aiming at proving that a program basically executes a certain algorithm. This way one tries to make sure that an implementation works as desired. However, if the algorithm itself turns out to be faulty, the verifications is not a great help any longer.

Of course, cryptographic schemes like DSA have already been examined in detail, mainly based on conventional hand-written proofs. Therefore we will probably not come across any frightening new result like a major weakness of the DSA algorithm this way. However, there are in my view at least two important advantages resulting from an examination based on a formal deduction system. On the one hand we are simply forced to create a formalized encoding that - in contrast to any “human-readable” version - is very likely to be free of any unclear or ambiguous concepts. (In the extreme case we realize only now what we are really talking about.)

On the other hand, any proof step is either created or at least *verified* by the machine (or rather the formal deduction system). Provided that our system is correct (and we will see in chapter 3 what this means and how we make sure that a “rule” our system is based on, is correct - or sound as it is usually denoted - in this sense) *everything* we prove can be regarded as correct beyond any reasonable doubt, which does not always apply to proofs written (and verified) by humans.

In the next chapter I am going to introduce the general notion of a cryptographic signature scheme as to its objectives and the technical “ingredients” we need to realize them, along with the special case of the DSA algorithm in a bit more detail. What follows in chapter 3 is an introduction to formal languages. I will briefly outline how formal proofs within a logic calculus work in general since the Isabelle system (and its colleagues) are always based on such a language. The limits of such a formal encoding - especially concern-

ing the expressive power - will also be discussed. Turning to the practical point of view, chapter 4 offers some information about the Isabelle system itself and how theorem proving works in this environment, whereas chapter 5 presents some prior contributions in the field of formalization and formal verification in Isabelle with an emphasis on cryptographic protocols.

Chapter 6 finally returns to our actual subject, namely proving the correctness of the DSA signature scheme starting with a typical “pen and paper proof” and then turning to my encoding in the Isabelle system with a “roadmap” of my formal proof comprising all relevant lemmata needed to approach the verification theorem. A discussion about the usefulness and the benefits of formal proofs along with further conclusions can be found in chapter 7, whereas the actual encoding, i.e. the complete theory file `dsa.thy` along with some explanatory annotations can be examined in the appendix.

## 2. DIGITAL SIGNATURES

### 2.1 Objectives

Digital signatures are meant to prevent (or rather to detect) unauthorized modifications to data and to authenticate the identity of the signatory. Besides, the signatory cannot repudiate a signature that was issued by him or her. The former means that - in contrast to conventional (hand written) signatures - digital signatures are tightly connected to the data they refer to. A signature for document  $m_1$  is different from that for document  $m_2$  and it is (or should be) impossible for an attacker to create a signature for  $m_2$  by knowing a valid signature for  $m_1$ ; digital signatures are in this sense even stronger than their conventional counterparts.

In detail we expect a signature scheme to meet the following requirement [7]:

- **Identity:** The signature needs to prove the signatory's identity beyond any doubt.
- **Non-Reusability:** The signature is only valid together with the original document.
- **Manipulation Prevention:** Any subsequent manipulation of the document will be detected.
- **Non-Repudiation:** The signatory cannot deny to have signed the document.

Especially the demand that the signatory is unable repudiate to have issued the signature means we cannot make use of techniques like Messages Authentication Codes (MAC's), alone. A MAC is a parameterized hash function, i.e. the hash value depends not only on the document but also on a (usually secret symmetric) key. The hash value is usually attached to the document in order to prove that it has not been altered (which means to ensure its integrity). If a secret key has been exchanged between two parties

this may allow an exchange of documents protected against manipulations by means of MAC's, but, since the key is known to *both* participants, we cannot associate a sent message (i.e. a document with a MAC value attached) to *one* of them beyond any doubt.

For the same reason any symmetric cryptographic scheme can only be applied under strong assumptions, as we will see in the next subsection.

## 2.2 Necessary Ingredients

Since I assume that the reader is already familiar with the general notion of a cryptographic system (a thorough introduction to this field can be found in [4] for example), I will confine myself to outline the difference between a symmetric and an asymmetric cryptographic scheme.

In a symmetric system, the key for encrypting a message is the same as the one that is needed to decrypt it, or - more generally - the keys might not be the same, but it is easy - from knowing one of them - to compute the other.

In contrast, an asymmetric scheme is even based on the impossibility (or rather difficulty) to compute the decryption key from knowing the one used for encryption. Usually the encryption key is called the public key that may be known to anyone who wants to send secret messages to participant  $X$ , whereas  $X$  is the only one who has access to his private key which is a prerequisite for decrypting them again.

### 2.2.1 Digital Signatures based on Symmetric Cryptography

A signature scheme based on the symmetric case necessitates a trusted third party  $T$ . Furthermore, we need to assume that every participant  $A$  has exchanged a secret symmetric key  $K_A$  with  $T$ .

If a participant  $A$  wants to sign a document for participant  $B$ , he encrypts it using his key  $K_A$  and sends it to  $T$ . Since  $T$  assumes that  $A$  is the only person (besides himself) that has access to  $K_A$ , he can confirm that the document originates from  $A$ . After decrypting it using  $K_A$  and attaching a corresponding confirmation for  $B$  expressing that the document was actually sent by  $A$ , he encrypts it (together with the confirmation) using  $K_B$  and delivers it to recipient  $B$ .

By decrypting it and reading the confirmation,  $B$  can be sure that  $A$  was the sender (and the signatory) since he knows that  $T$  confirms this claim

and  $T$  is (by definition) trustworthy.

It is obvious that the assumptions involved here are quite ambitious. First of all we just need to believe that  $T$  is not only trustworthy but also resistant to any malfunction or attack. Since it is involved in every exchange of signed documents it is on the other hand required to work relatively quickly. We also need a secure exchange of secret symmetric keys. And finally we need to assume that these keys are also stored safely (inaccessible for any other party) by the participants. The latter is basically the only difficulty arising here that also applies to asymmetric systems.

### 2.2.2 Digital Signatures based on Asymmetric Cryptography

For an asymmetric cryptosystem, let  $e$  be an arbitrary encryption key,  $d$  the corresponding decryption key, and  $E_e$  as well as  $D_d$  the associated encryption (or decryption) functions. Then  $D_d(E_e(m)) = m$  holds for every message  $m$ .

The system can be directly used for digital signatures if the encryption and decryption functions commute, that means if - under the above assumptions - the following is also valid:

$$E_e(D_d(m)) = m$$

A participant  $A$  applies his (private) decryption key  $d$  (that is only known to himself) to a message  $m$ , yielding the signature  $c := D_d(m)$ , and every recipient may use  $A$ 's (public) encryption key  $e$  to verify the signature, i.e. to check whether  $E_e(c) = m$ . The security requirements are the same as in the original application of the system in terms of a confidential exchange of messages:

- The private keys need to be stored safely. Only the legitimate users are allowed to have access to them.
- The public keys should be protected from unauthorized manipulation, stored and managed at a safe place like a trust center. If an attacker succeeds in changing the public key data, faked signatures might be mistaken for legitimate ones since a correct verification requires the correct public key.

Of course, usually one would not sign the actual messages (or documents) but their hash values, also for security reasons. In addition, it is not recommendable at all to use the same key pair for signatures *and* for encryption /

decryption operations. But this is the idea how it works.

A more general concept is the notion of *one-way functions*. These are functions  $f : X \rightarrow Y$ , such that the following holds [7]:

- **(i)** For  $x \in X$ , the function value  $f(x)$  is efficiently computable.
- **(ii)** For  $y \in Y$ , there is no efficient algorithm to find an  $x \in X$ , such that  $f(x) = y$ .

The bad news is: It is not known whether one-way functions exist even if there are a number of candidates which are believed to be of this kind. The good news: They are quite useful for electronic signatures, at least if they are equipped with a *trapdoor*. In this case we replace **(ii)** by:

- **(ii')** For  $y \in Y$ , there *is* an efficient algorithm to find an  $x \in X$ , such that  $f(x) = y$ , but this requires some additional information.

In short, one might regard the verifying operation as the one-way function. The “inverse” operation - which is signing - is impossible to perform; unless you happen to have access to the trapdoor which is the private key.

#### **Example: The RSA algorithm for digital signatures**

In order to generate an RSA key pair [4], one chooses two large prime numbers  $p$  and  $q$ <sup>1</sup> and computes their product  $n = pq$ . Besides, one chooses a number  $e$  such that  $1 < e < (p - 1)(q - 1)$  and  $\gcd(e, (p - 1)(q - 1)) = 1$ . Finally one computes  $d$  in the same range with the property  $de \equiv 1 \pmod{(p - 1)(q - 1)}$  using the extended Euclidean Algorithm. The tuple  $(n, e)$  gets the public,  $d$  the private key. Plain- as well as cipher texts are represented as natural numbers smaller than  $n$ . To generate the signature  $c$  for a document  $m$ , a participant applies his private key,

$$c := m^d \pmod n$$

whereas a recipient can verify it by means of the public one,

$$m' := c^e \pmod n$$

in order to check if  $m$  equals  $m'$ . This signature scheme works: On the one hand, it is relatively easy to see that these two exponentiations (first using the exponent  $d$ , then  $e$ ) compensate each other, i.e. that

---

<sup>1</sup> There are a few more properties helpful to increase security which are omitted here for simplicity.

$$c^e \equiv m^{de} \equiv m \pmod{n}$$

This is true since  $de \equiv 1 \pmod{(p-1)(q-1)}$ , and  $(p-1)(q-1)$  is the group order of  $(\mathbb{Z}/n\mathbb{Z})^*$ .

On the other hand, finding a signature  $c$  for a document  $m$  such that  $c^e \pmod{n} = m$  is (as far as we know) hard (presumably as hard as factorization, but this has not been shown yet), unless you have the trapdoor at your disposal, which might be the inverse exponent  $d$  (i.e. the private key) or at least the prime factors of  $n$ ,  $p$  and  $q$ , which would imply that you can also compute  $d$ . That means, whoever can produce valid signatures (and again one should mention that in reality one does not sign whole documents, but hash values) must possess the private key and is therefore (hopefully) the legitimate signatory.

From an even more general viewpoint, one might not necessarily speak about a function, but about a constraint satisfaction problem: For a given document  $m$ , find a signature  $c$  such that one or more constraints (like the verifying congruency) involving  $m$ ,  $c$  and probably some other parameters - especially the public key for instance - are satisfied. If you can use the private key (as a trapdoor) it is easy to find such a  $c$ , otherwise it is (supposed to be) impossible which means not efficiently computable, of course. Also the DSA signature algorithm can be modelled this way.

### 2.3 The DSA Signature Scheme

The Digital Signature Algorithm is a more efficient version of the ElGamal scheme [4][8], proposed by the National Institute of Standards and Technology (NIST) in August 1991 and based on discrete logarithms in the group  $(\mathbb{Z}/p\mathbb{Z})^*$  where  $p$  is a (large) prime number. In the DSA case the exponentiations concern only a subgroup of  $(\mathbb{Z}/p\mathbb{Z})^*$  which reduces the size of the exponents involved. In addition, the verification requires only two modular exponentiations instead of three. The United States Institute of Standards (NIST) recommends the Secure Hash Algorithm (SHA-1) to be applied to the data first. That means the signing / verifying does not refer to the actual data, but to its hash value using SHA-1.

## 2.3.1 DSA Parameters

In order to generate DSA signatures we need the following ingredients [9]:

- $p$ : a prime modulus, where  $2^{L-1} < p < 2^L$  for  $512 \leq L \leq 1024$  and  $L$  a multiple of 64
- $q$ : a prime divisor of  $p - 1$ , where  $2^{159} < q < 2^{160}$
- $g := h^{(p-1)/q} \bmod p$ , where  $h$  is any integer with  $1 < h < p - 1$  such that  $h^{(p-1)/q} \bmod p > 1$  (which implies that  $g$  has order  $q \bmod p$ )
- $x$ : a randomly or pseudo-randomly generated integer with  $0 < x < q$
- $y := g^x \bmod p$
- $k$ : a randomly or pseudo-randomly generated integer with  $0 < k < q$

**Note:** Usually one would first choose the prime  $q$ , then the larger prime  $p$ , as the required conditions are in this case easier to satisfy. Moreover, the parameter  $k$  is allowed to be used for *one* signature, only, and must then be replaced by a new (pseudo-)random value.

$p$ ,  $q$ ,  $g$  and  $y$  can be made public, whereas  $x$  and  $k$  need to be kept secret and  $k$  must even be regenerated for each new signature as already mentioned.

## 2.3.2 Signature Generation

Let  $h(m)$  denote the hash value of a message  $m$ , i.e. the output of the SHA-1 algorithm as recommended by the NIST. The signature of the message  $m$  is the pair of numbers  $r$  and  $s$  computed according to the equations below [9].

- $r := (g^k \bmod p) \bmod q$
- $s := (k^{-1}(h(m) + xr)) \bmod q$

In the above,  $k^{-1}$  is the multiplicative inverse of  $k$ , mod  $q$ . In the unlikely case that either  $r$  or  $s$  turn out to be 0, the signatory is (strongly) advised to choose a new (pseudo-)random value for  $k$ . The signature is transmitted along with the message to the verifier.

## 2.3.3 Signature Verification

Prior to verifying the signature in a signed message,  $p$ ,  $q$ ,  $g$  and  $y$  (together with the sender's identity, indeed) are made available to the verifier in an authenticated manner.

Let  $m'$ ,  $r'$  and  $s'$  be the received versions of the message and the signature parameters, respectively. To verify the signature, the verifier first checks to see that  $0 < r' < q$  and  $0 < s' < q$ ; if either condition is violated the signature shall be rejected. If these two conditions are satisfied the verifier computes [9]

- $w := s'^{-1} \bmod q$
- $u_1 := (h(m')w) \bmod q$
- $u_2 := (r'w) \bmod q$
- $v := ((g^{u_1}y^{u_2}) \bmod p) \bmod q$

The signature is verified if  $v = r'$ . Otherwise, the message may have been modified, the message may have been incorrectly signed by the signatory, or the message may have been signed by an impostor. The message should be considered invalid. For a proof that  $v = r'$  if  $m = m'$ ,  $r = r'$  and  $s = s'$  (both in a "human readable" style and in a formalized version in the Isabelle system) see chapter 6.

### 3. FORMAL LANGUAGES

As already mentioned, the Isabelle system is often called a Proof Assistant underlining the process of alternating automated reasoning with human intervention. On the other hand it is also often referred to as a Theorem Prover which emphasizes on its ability to prove (mathematical) theorems. From my point of view it would be fair to call it an Interactive Theorem Prover since this seems to unify both aspects.

Apart from those questions and abstracting from any implementation details, systems like Isabelle are usually based on a formal language together with (syntactic) deduction rules, often called a *sequent calculus* in logic textbooks. If the language can be associated with a formal semantics, those rules can be regarded as *sound* if and only if any syntactic deduction following such a rule is accompanied by a semantic consequence. This is especially true for First Order Logic. On the other hand one may ask if anything that follows semantically can also be deduced using a syntactic rule, which is called *completeness*. Depending on the language we are using we deal with different kinds of “expressive power”, i.e. there might be aspects we cannot express if we confine ourselves to a simple language which - on the other hand - might also lead to a less complex deduction system. If we call the things that we want to prove “theorems” we can ask how difficult it is to decide the validity of theorems within our language together with the set of deduction rules. (Or if this is at least possible.)

Even in the (relatively simple) language of propositional logic the problem is NP-complete whereas for first order logic (FOL) it turns out to be undecidable, but at least recursively enumerable: A machine equipped with a suitable set of deduction rules will eventually find *any* valid FOL formula. But if we asked it about the validity of a “wrong” formula (or - strictly speaking - a formula that cannot be proven) it would run forever.

### 3.1 First Order Logic

#### 3.1.1 Syntax and Semantics

In order to sketch a proof for the soundness and completeness of a given set of deduction rules for the first order logic I will outline what a FOL language looks like and what is meant by an interpretation. Since this is only a very short summary I apologize in advance for possible simplifications or inconsistencies. A worth reading thorough introduction to FOL can be found in [6].

In short, a FOL *language* is built from a set of symbols  $S$  consisting of a countable set of variable symbols, symbols for negation, conjunction, disjunction and implications, the equality sign, quantifier symbols, brackets, a (possibly empty) set of constant symbols and finally - for any  $n \in \mathbb{N}$  a set of function and a set of relation symbols - each of arity  $n$  - which may be empty, as well. From these ingredients terms and formulae can be built in the usual matter.

An  $S$ -*interpretation*  $I$  is a tuple  $(\alpha, \beta)$  where  $\alpha$  is an  $S$ -*structure* consisting of

- A non-empty set  $A$ , called the universe of the structure
- A mapping  $a$  defined on  $S$  s.t.
  - $a(c)$  is an element of  $A$  for each constant symbol  $c$
  - $a(f)$  is an  $n$ -ary function on  $A$  if  $f$  is an  $n$ -ary function symbol
  - $a(R)$  is an  $n$ -ary relation on  $A$  for an  $n$ -ary relation symbol  $R$

and  $\beta$  is a mapping from the set of variables into the set  $A$ .

From this, we can now define when an  $S$ -interpretation  $(\alpha, \beta)$  gets a *model* of an  $S$ -formula  $\phi$ : A structure  $\alpha$  (together with an interpretation  $\beta$  for the variables) is a model of  $\phi$ , which is expressed by

$$(\alpha, \beta) \models \phi$$

if and only if the interpretation of  $\phi$  which is itself inductively defined by the interpretation of the occurring constant, function and relation symbols (along with the interpretation of the variable symbols induced by  $\beta$ ) holds in the  $S$ -structure  $\alpha$ . (For simplicity I will leave out the  $S$  in the following lines). An interpretation is a model for a set of formulae  $\Phi$  if it is a model

for each formula  $\phi \in \Phi$ .

**Example:** Let  $S_{GR} := \{e, \circ\}$  and  $\phi$  the  $S_{GR}$ -formula  $\phi := \exists x_1 \neg x_1 \circ x_1 = e$ . Then  $\phi$  holds in any group containing at least one element of order greater than two (with the usual interpretations for the two symbols), but not in the Galois Field  $GF(2)$  when the interpretation of the  $\circ$  symbol is the addition modulo 2 and  $e$  is interpreted as 0. (In both cases the interpretation of the variable symbols is irrelevant since there are no free variables involved.)

With this intuitive “definition” of first order semantics we can now define what a (semantic) consequence between formulae (or sets of formulae) means: An  $S$ -formula  $\phi$  follows from a set of  $S$ -formulae  $\Phi$ , denoted by

$$\Phi \models \phi$$

if and only if every interpretation that is a model of  $\Phi$  is also a model of  $\phi$ :

$$(\alpha, \beta) \models \Phi \implies (\alpha, \beta) \models \phi$$

### 3.1.2 A Sequent Calculus

Apart from the undecidability of their “output”, there are quite handy sets of deduction rules which are sound and complete yielding any valid FOL theorem.

In [6] such a set (in this case called a “sequent calculus”) for FOL is presented consisting of only eleven rules. Seven of those are concerned with the purely propositional part of the language, two speak about quantifiers (in this case only about the existential quantifier since the universal quantifier can be emulated via negation and its existential counterpart) and the remaining two rules treat the equality symbol and substitution, respectively.

One of the basic propositional rules is the contradiction rule

$$\frac{\Gamma \neg \phi \quad \psi}{\Gamma \phi}$$

which should be read like this: If we can derive  $\psi$  from  $\Gamma \cup \{\neg \phi\}$  and also  $\neg \psi$  from  $\Gamma \cup \{\neg \phi\}$ , then we can derive  $\phi$  from  $\Gamma$ , in short,

$$(\Gamma \cup \{\neg\phi\} \vdash \psi \text{ and } \Gamma \cup \{\neg\phi\} \vdash \neg\psi) \implies \Gamma \vdash \phi$$

Note: For a set  $\Omega$  of formulae, we write  $\Gamma \models \Omega$  ( $\Gamma \vdash \Omega$ ) iff  $\Gamma \models \phi$  ( $\Gamma \vdash \phi$ ) for all  $\phi \in \Omega$ .

### 3.1.3 Soundness and Completeness

We say that a deduction system (here called sequent calculus) is *sound* if for any (syntactic) deduction ( $\vdash$ ) built using its rules (like the one above) there is also a corresponding semantic consequence ( $\models$ ). It is *complete* if the opposite implication is also true: For any semantic consequence it is possible to build a syntactic deduction using the rules.

Since any deduction is built using only finitely many applications of the calculus' rules - starting from the empty set with a rule without any preconditions (the space above the line is empty) or starting from a set of axioms - it suffices to show that - for each rule - soundness is preserved under its application. The soundness of the calculus follows then by induction on the number of steps needed to build up a syntactic deduction.

A soundness proof of the rule above works like this: Assume the two deductions above the line are sound. That means they correspond to the (true) semantic consequences  $\Gamma \cup \{\neg\phi\} \models \psi$  and  $\Gamma \cup \{\neg\phi\} \models \neg\psi$ . Thus, any interpretation  $I$  that is a model of  $\Gamma \cup \{\neg\phi\}$  is also a model of  $\psi$  and  $\neg\psi$  which is impossible. Consequently, there is *no* model of  $\Gamma \cup \{\neg\phi\}$ . Hence, any model for  $\Gamma$  is also a model of  $\phi$ , i.e.  $\Gamma \models \phi$ , which proves the soundness of the deduction below the line.

To get an idea of how a completeness proof for such a deduction system looks like, we first need the notions of satisfiability and consistency.

A set of formulae  $\Gamma$  is said to be *satisfiable* ( $\text{sat}(\Gamma)$ ) if it has a model. (The set  $\Gamma \cup \{\phi\}$  that we just saw is not satisfiable, for instance.) A set of formulae  $\Gamma$  is *consistent* ( $\text{cons}(\Gamma)$ ) if there is *no* formula  $\phi$  such that  $\Gamma \vdash \phi$  and  $\Gamma \vdash \neg\phi$ . It turns out that the latter definition is equivalent to the property that we *cannot* derive *everything* from a set of formulae. This can be shown using the (itself derivable) “ex falsum quodlibet”-rule, basically stating that from a wrong assumption any conclusion can be drawn. In other words, from an inconsistent set everything can be proven. Note that the notion of

consistency depends on the system of deduction rules we are using.

For completeness we need to show that for any set of formulae  $\Gamma$  and for any formula  $\phi$ , if  $\Gamma \models \phi$  then  $\Gamma \vdash \phi$  or, equivalently, if not  $\Gamma \vdash \phi$  then not  $\Gamma \models \phi$ .

First, note that not  $\Gamma \vdash \phi$  implies  $\text{cons}(\Gamma \cup \{\neg\phi\})$ : Assume not  $\text{cons}(\Gamma \cup \{\neg\phi\})$ . Then we can derive  $\phi$  from  $\Gamma \cup \{\neg\phi\}$  (since we can derive everything from an inconsistent set). On the other hand we can also derive  $\phi$  from  $\Gamma \cup \{\phi\}$  as we have in our deduction system (like in any reasonable deduction system) a rule stating that you can prove  $\phi$  from a superset of  $\{\phi\}$ . Using the “case-distinction” rule (again appearing in any reasonable deduction system like the one we are using) it follows that we can derive  $\phi$  from  $\Gamma$  alone.

On the other hand, not  $\Gamma \models \phi$  is already implied by  $\text{sat}(\Gamma \cup \{\neg\phi\})$ .

Thus, it suffices to show

$$\text{cons}(\Gamma \cup \{\neg\phi\}) \implies \text{sat}(\Gamma \cup \{\neg\phi\})$$

In other words: Any consistent set of formulae has a model. Since we have only syntactic information at our disposal (we need to find a model for our consistent set of formulae) we build a model that consists basically of terms.

The Henkin Theorem [12] states that the following Interpretation  $I^\Gamma := (\mathfrak{S}^\Gamma, \beta^\Gamma)$  gets a model of the consistent set of formulae  $\Gamma$  as long as  $\Gamma$  meets certain syntactic conditions. (It turns out that enforcing these conditions is technically the hardest part of the work, but it is always possible.)

- The universe of the structure  $\mathfrak{S}^\Gamma$  gets  $T^\Gamma := \{\bar{t} \mid t \in T^S\}$  where  $T^S$  is the set of all  $S$ -terms (i.e. the set of terms that can be built in our alphabet) and  $\bar{t}$  is the equivalence class of  $t$  regarding the equivalence relation  $\sim$  on  $T^S$ , defined by:  $t_1 \sim t_2 :\iff \Gamma \vdash t_1 = t_2$
- The interpretations of our constant, function and relation symbols are simply defined by
  - $c^{\mathfrak{S}^\Gamma} := \bar{c}$  which is the equivalence class (regarding  $\sim$ ) of the constant symbol  $c$
  - $f^{\mathfrak{S}^\Gamma}(\bar{t}_1, \dots, \bar{t}_n) := \overline{ft_1 \dots t_n}$ , i.e. the image of the equivalence classes of the terms  $t_i$  are the equivalence class of the term originating from applying the function symbol  $f$  to the terms  $t_i$

- $R^{\exists\Gamma} \bar{t}_1 \dots \bar{t}_n : \iff \Gamma \vdash R t_1 \dots t_n$  That means the equivalence classes of the terms  $t_i$  are related under  $R$  iff  $\Gamma$  implies that the  $t_i$  terms themselves are related

- Variable symbols are simply mapped to their equivalence classes:  
 $\beta^\Gamma(x) := \bar{x}$

The proof of this theorem involves several steps and relies (which cannot be surprising) on the “strength” of the deduction system whose completeness is to be shown. (As mentioned above, the notion of consistency itself depends on the system. In short, if it is too weak, virtually *everything* might be consistent.)

At first one needs to make sure that the relation denoted by  $\sim$  is actually an equivalence relation and that the interpretations of the function and relation symbols are sound (i.e. they do not depend on the representatives of the relative classes) which needs to be shown using only the rules of our deduction system.

Then, by induction (usually on the number of quantifiers and connectives arising in an arbitrary formula  $\phi$ , i.e. the base case is an atomic formula) it can be shown - provided  $\Gamma$  meets two syntactic conditions, which can always be enforced - that

$$I^\Gamma \models \phi \iff \Gamma \vdash \phi$$

which is Henkin’s Theorem. Since especially  $\Gamma \vdash \Gamma$ , it follows

$$I^\Gamma \models \Gamma$$

Thus, the (arbitrarily chosen) consistent set  $\Gamma$  has got a model which proves - according to the above argument - the completeness of the deduction system.

## 3.2 Towards HOL

### 3.2.1 Limitations of FOL

Apart from the great advantage of sound and complete deduction systems and other helpful properties there are certain issues that we simply cannot express in first order language. For instance the statement “ $a$  and  $b$  have exactly the same properties” cannot be directly translated into FOL (where properties

are usually expressed in terms of predicates, i.e. (unary) relations) since we are not allowed to quantify over relations. However, this problem can be avoided if we (syntactically) replace the predicates by ordinary objects (e.g. we replace all unary relation symbols by constant symbols) and introduce a binary “meta-predicate” *have*. Then the above statement simply becomes [15]

$$\forall P(\text{have}(a, P) \leftrightarrow \text{have}(b, P))$$

Unfortunately, there are other weaknesses of first order logic that seem to be more serious and cannot be repaired by such an ad hoc construction.

The completeness theorem for a deduction system (provided that it is at the same time sound, of course) establishes a duality between satisfiability and consistency. Since any (syntactic) proof showing that a set of formulae is inconsistent involves only finitely many applications of the rules (and therefore only finitely many formulae within the set) it follows that a set, which is *finitely* consistent, is consistent as a whole. Thus, a set is satisfiable, as long as all finite subsets are satisfiable.

Let  $\text{Th}(\mathbb{N})$  be a set of formulae that - in short - describes all properties of the natural numbers as a discrete linear ordering as we know them. Let  $\phi_n$  be a formula stating “ $x > n$ ”. Then any *finite* subset of  $\text{Th}(\mathbb{N}) \cup_{n \in \mathbb{N}} \{\phi_n\}$  has a model (e.g. the “normal” natural numbers themselves) and is therefore satisfiable. Consequently, the *whole* set  $\text{Th}(\mathbb{N}) \cup_{n \in \mathbb{N}} \{\phi_n\}$  has a model, too, which is also a model of the (sub)set  $\text{Th}(\mathbb{N})$ . This model must include an “infinitely large” element (larger than any natural number) causing this model to be different from the usual  $\mathbb{N}$  in the sense that the two structures are not isomorphic. We call this a *non-standard* model and note that it is impossible to axiomatize the natural numbers (up to isomorphism) in first order logic <sup>1</sup>

A similar construction can be used to show that we cannot axiomatize the class of all finite fields and the like this way.

### 3.2.2 Extensions within FOL

In model theory a *partial n-type* is a consistent set of formulae  $\Omega$  (usually in the context of an underlying theory like  $\text{Th}(\mathbb{N})$ , i.e. also consistent together with this theory) such that all free variables within the formulae in  $\Omega$  are (up to renaming) among  $x_1, \dots, x_n$ . To prevent a non-standard model from

<sup>1</sup> At least as long as we are restricted to the set of symbols used to describe  $\mathbb{N}$ .

arising one might now demand that a model of  $\text{Th}(\aleph)$  does not *realize* the 1-type  $\Omega := \bigcup_{n \in \mathbb{N}} \phi_n$  above, i.e. - in short - there is no element  $a$  such that  $\beta(x) := a$  yields true statements about the structure.

Another attempt to compensate FOL weaknesses is the well-known (FOL) axiomatic set theory by Zermelo and Fraenkel (abbreviated ZF or ZFC if one also includes the so-called axiom of choice, respectively) [10] based on the first order language with no function or constant symbols and the only (binary) relation symbol denoted by  $\{\in\}$ , consisting of six axioms together with two “schemes of axioms” (i.e. no single axioms but an axiom for each formula of suitable shape). As to the above example of (unintended) non-standard models, ZFC allows us to express the *Peano Axioms* which themselves are strong enough to axiomatize the natural numbers up to isomorphism.

Moreover, since virtually *all* objects or phenomena arising in mathematics like graphs or functions can be formalized using the language of set theory, ZFC can be regarded as a basis for mathematics, and a consistency proof for ZFC would in principle prove the consistency of modern mathematics, as well. Unfortunately such a consistency proof for ZFC (like for any “strong” axiomatic system) is impossible according to Goedel’s Second Incompleteness theorem [11]. (Ironic remark: Of course, one could proof its consistency in some sense, but only if it was *inconsistent*.) Even worse, according to his First Incompleteness Theorem, there are always sentences  $\phi$  such that neither  $\phi$  nor  $\neg\phi$  can be proven from ZFC (provided that ZFC is consistent, which we assume).

### 3.2.3 Going beyond FOL

Instead of making use of the additional system of ZFC we can also directly express the Peano Axioms in the so-called *second order logic*. According to [6] an alphabet for second order logic is an extension of an FOL alphabet in the sense that it has additionally - for each  $n \in \mathbb{N}$  - an countably infinite set of  $n$ -ary *relation variables*  $V_0^n, V_1^n, V_2^n, \dots$  which we may denote by  $X, Y, \dots$  for simplicity. The rules for building formulae are extended by

- $Xt_1, \dots, t_n$  for terms  $t_i$  and an  $n$ -ary relation variable  $X$
- $\exists X\phi$  for a relation variable  $X$  and a formula  $\phi$

The semantic interpretation of an  $n$ -ary relation variable is simply an  $n$ -ary relation within the structure. Since we can now quantify over relations

and therefore over subsets (the universal quantifier can again be emulated using the  $\exists$ -quantifier together with the negation symbol) we can easily express the Peano Axioms and the like.

In contrast to FOL we can also axiomatize all *finite* structures (e.g. by formulating a sentence  $\phi$  demanding that all injective functions are surjective where we “encode” functions using relations describing their graphs). On the other hand, we can (even based on FOL) demand that - for all  $n \in \mathbb{N}$  - the structure has at least  $n$  distinct elements by a sentence  $\psi_n$ . Then, the set  $\{\phi\} \cup_{n \in \mathbb{N}} \{\psi_n\}$  is apparently not satisfiable (a structure satisfying  $\phi$  must have exactly  $n_0$  elements for a natural number  $n_0$  which would immediately contradict  $\psi_{n_0+1}$ , for instance), but every finite subset is.

It might be surprising that this already proves the non-existence of a sound *and* complete deduction system for second order logic: Assume there was such a system. Then any finitely consistent set would be consistent. Thus any finitely satisfiable set would be satisfiable as a whole, which is not the case as we have seen. Fortunately there are deduction systems which are reasonably strong, but they will never be complete in principle.

In the more general case of the so-called higher order logic (HOL) one may also introduce relation symbols that take themselves relation symbols as arguments [15]. Such a predicate (i.e. relation symbol) of order  $n + 1$  takes arguments that are order  $n$  predicates. An analogous construction can be used for function symbols but this does not enhance the expressive power since one could also “encode” the function as relations describing their graphs.

Furthermore, HOL is usually based on a *typing scheme*. One often distinguishes between individuals, sets of individuals, (ordered) tuples of individuals and so on. We may also introduce the set of booleans, such that relation membership can be expressed by a function from (tuples of) individuals into the set of booleans. Such constructs often lead to easier or more natural representations but do not increase the expressive power in principle. In the Isabelle/HOL system for instance, we have not only types but collection of types, called *records*.

What we can express in the “Isabelle interpretation” of higher order logic will be demonstrated by the contributions in the following sections. Fortunately, the mere fact that we lose the ability to recursively enumerate all valid formulae or sentences if we go beyond first order logic does not keep us from proving interesting theorems in Isabelle / HOL.

### 3.3 Authentication Logic

At the end of this chapter I want to give an example of a specialized, rather than “generic” formal language, designed for expressing and examining certain aspects of cryptographic protocols, mainly concerning the authentication of principals and/or data. The BAN logic [5] (named after its inventors Burrows, Abadi and Needham) is a relatively simple formalism to examine key exchange protocols and the like aiming at verifying certain properties or revealing redundant steps. Using the BAN logic one may try to answer questions like this: “After performing the authentication protocol in question - do all principals believe that they communicate with each other, not with an attacker?” In this sense such logics are often referred to as *believe logics*.

The BAN logic distinguishes between different sorts, namely *principals*, *keys* and *formulae*. In addition there are also *nonces*, that are - in short - random numbers exchanged together with keys or other data within a protocol step that can be used to ensure the “freshness” of a message warding off replay attacks. BAN Formulae can be build using the following ingredients (here called operators) [7]:

Operator	Meaning
$P \equiv X$	$P$ is convinced of formula $X$
$P \triangleleft X$	$P$ has received formula $X$
$P \sim X$	$P$ has sent formula $X$ which he believes in
$P \Rightarrow X$	$P$ is authorized to confirm the validity of formula $X$
$\#(X)$	formula $X$ is fresh, i.e. has not been used yet
$P \leftrightarrow^K Q$	$K$ is a common, secret key between $P$ and $Q$
$\rightarrow^K P$	$K$ is $P$ 's public key
$P \rightleftharpoons^K Q$	$K$ is a common secret between $P$ and $Q$
$\{X\}^K$	formula $X$ enciphered using $K$
$\langle X \rangle^Y$	formula $X$ , signed using $Y$

In addition there is also a system of deduction rules expressing which new formulae can be derived from others. The so-called *Jurisdiction-Rule* for example

$$\frac{P \equiv Q \Rightarrow X, P \equiv Q \equiv X}{P \equiv X}$$

states that - if  $P$  believes that  $Q$  believes in  $X$  and furthermore  $P$  believes that  $Q$  is authorized (or competent) to judge the validity of  $X$  - then  $P$  will also believe in (the validity or correctness of)  $X$ . An instance of this rule might a situation where  $Q$  is a *trusted third party* like a trust center and  $X$

is a statement associating a principal  $R$  (other than  $P$ ) with a public key. Since  $P$  regards the trust center  $Q$  as authorized to certify  $R$ 's public key, he or she will accept (and believe in) such an assertion from  $Q$ .

Of course, using the BAN logic we cannot formalize any statements regarding the confidentiality of exchanged messages and the like since this is beyond the scope of its language.

In addition, it seems to be hard (or rather impossible) to state anything about the correctness (let alone the completeness) of such a deduction system since we are lacking a formalized semantics. However, if we regard the rules as correct (in some informal sense) it can be a useful mean to verify complex protocols. Since we are here in the world of believe logic, one might say: As long as we believe in our deduction rules, we may also believe in anything that we build from it.

## 4. THE ISABELLE/HOL SYSTEM

### 4.1 *The System*

According to its native web site [27] Isabelle is a “Generic Theorem Proving Environment” and - at the same time - a ”Generic Proof Assistant”. However, it is also often just referred to as an ”Interactive Theorem prover”. I will stick to the latter designation even if the variety of terms connected with the system could be seen as a first demonstration of its strength and versatility.

Developed at Cambridge University and TU Munich the Isabelle system has been constantly improved and refined, and is now available in its 2005 edition. With a core written in Standard ML, it is *generic* in the sense that it has the capacity to accept a variety of formal calculi (like HOL in our case).

As we will see in detail in the following subsection theorem proving with Isabelle is often based on a “human-guided” manipulation of the proof state, where the system itself only executes the given commands and - of course - verifies their applicability until finally all subgoals have been proven (e.g. via reduction to already proven lemmata). On the other hand there are also strong tools that can be applied to handle (suitable) proofs (or at least considerable parts of it) automatically. The user is free to alternate these two paradigms arbitrarily.

Isabelle comes with a large theory library of formally verified mathematics, including elementary number theory (for example, Gauss’s law of quadratic reciprocity), analysis (basic properties of limits, derivatives and integrals), algebra (up to Sylow’s theorem) and set theory (the relative consistency of the Axiom of Choice). Using appropriate include-commands we may base our (new) theorems upon those in the libraries by referring to them during the proof process wherever they are applicable.

## 4.2 Theorem Proving

In this section I am going to outline how theorem proving using the Isabelle / HOL system works in general such that the application to the DSA system later on can be understood. For more specific information - also concerning supporting tools like the emacs based *Proof General* - please refer to the Isabelle manuals, for a thorough introduction, [16] is a worth-reading contribution.

### 4.2.1 Theories and Axiomatic Type Classes

Theorems along with necessary definitions and - of course - the corresponding proofs are always organized in *theory files*. A file concerned with the theory of cyclic groups for instance - i.e. a collection of their interesting properties expressed by theorems and/or lemmata - might be called `CyclicGroup.thy`.

As to the definitions appearing in such a file, a (possible, but not the only) way to get started is the notion of an *Axiomatic Type Class*. Such a class can be a specialization (or a generalization) of another one where the most general class is the *type* class. A possible definition of the axiomatic type class of semigroups may look like this [26]:

```
consts
  times:: 'a ⇒ 'a ⇒ 'a (infixl ⊙ 70)
axclass semigroup ⊆ type
  assoc : (x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)
```

At first the constant *times* is defined, in this case a function (that can be abbreviated by  $\odot$ ) of a certain priority (70) that takes two arguments of type *type* yielding another *type* element. Since the *type* type is the most general type (the top of the type hierarchy) it can be applied to arbitrary datatypes. (Syntactically, 'a is a type variable.)

The next line introduces the axiomatic type class of semigroups as a direct subclass of *type* such that the associativity law (of the *times* function) holds. From this we can define further subclasses (i.e. axiomatic type classes that are specializations) of the *semigroup* class like a monoid or group class. On the other hand we may also use the definitions we have introduced so far to derive interesting properties of our axiomatic type classes in terms of theorems or lemmata that need to be proven.

As already mentioned the proof process is either human guided or done by one (or more) of Isabelle’s built-in tools - or both, of course - but are always based on (syntactic) “rules” that may be basic (or itself derived) rules for propositional logic, first order logic or set theory (similar to the one we have seen in section 3.1.2), but also more generally “prior knowledge” in terms of already proven theorems or lemmata or axiomatic definitions like the associativity law above.

Such rules need to be applied until all subgoals have been proven (i.e. the proof has been finished). To explain the notion of a subgoal, let  $[P, Q] \implies P \wedge Q$  be a (not really interesting) “theorem” we want to prove for two propositional variables  $P$  and  $Q$ . At the beginning of the proof we face only one subgoal (which is the original goal in this case), namely  $P \wedge Q$  and have the two premises  $P$  and  $Q$  at our disposal.

Applying “rules” (basic rules for propositional logic or prior lemmata etc.) manipulates the proof state (i.e. especially the collection of pending subgoals). In this simple example the *conjunction introduction* rule (one of the basic propositional rules, present in any reasonable deduction system) suggests itself. The rule says: From  $P$  and  $Q$  we can derive  $P \wedge Q$ . Applying the rule removes the subgoal  $P \wedge Q$  replacing it by  $P$  and  $Q$ , which are now the new subgoals and can be easily proven since they are among the premises each. As even this simple proof process might sound a bit confusing at first glance, we will now examine another example in a bit more detail.

#### 4.2.2 An introductory proof

To demonstrate how to apply rules (as well as prior knowledge) manually and - at the same time - make use of Isabelle’s automatic “built-in tools” let us have a look at a typical proof example, in this case for an arithmetic lemma for natural numbers, where “dvd” denotes the divides-relation [16]:

lemma *mult\_dvd\_mono*: “[ $i$  dvd  $m$ ,  $j$  dvd  $n$ ]  $\implies$   
 $i * j$  dvd ( $m * n :: nat$ )”

To prove this lemma we will on the one hand apply Isabelle’s *simp* tool, a “simplifier” that rewrites (arithmetic and other) terms according to pre-defined (or explicitly added) rules until no further simplification is possible. On the other hand we make use of the following “rules”:

The definition of the divides relation:

*dvd\_def*: “ $m \text{ dvd } n \equiv \exists k. n = m * k$ ”

The elimination rule for the existential quantifier:

*exE*: “ $[\exists x. ?P x, \wedge x. ?P x \implies ?Q] \implies ?Q$ ”

Variables equipped with “?” or “ $\wedge$ ” occur in this shape mainly for internal reasons. In short, the former denotes an arbitrary choice corresponding to a variable that can be instantiated by an arbitrary term during the proof process whereas the latter symbol acts as a universal quantifier in principle. The rule says: “For arbitrary predicates (or formulae, respectively)  $P$  and  $Q$ , if - for all  $x$  -  $P(x)$  implies  $Q$ , then from *any* (arbitrary)  $x$  with  $P(x)$  we can infer  $Q$ ”. Finally we will need the corresponding existential introduction rule:

*exI*: “ $?P ?x \implies \exists x. ?P x$ ”

which should be self-explanatory.

Confronted with the lemma as the original subgoal we start the proof by unfolding the *dvd*-definition

**apply** (*simp add: dvd\_def*):

which asks the simplifier to process our subgoal using (besides the conventional simplifying rules) especially the definition of the divides relation. This causes Isabelle to present the new subgoal

1.  $[\exists k. m = i * k, \exists k. n = j * k] \implies \exists k. m * n = i * j * k$

The simplifier has just replaced any occurrence of the *dvd* symbol by the equivalent definition. If we now regard the premises we have, it is apparent that they look like the first premise of the *exE*-rule. Isabelle’s *erule* method for rules allows us to eliminate a subgoal immediately by unifying it with one of the premises we already have. (Applying a rule  $[P_1, \dots, P_n] \implies Q$  in Isabelle means basically that the current subgoal is unified with  $Q$  and removed. The new subgoals get  $P_1, \dots, P_n$ . Usually we would have to prove each of those in the next steps. However, using the *erule* method (if applicable), the first subgoal is unified with a premise we already have and is therefore done. We only need to prove the  $n - 1$  remaining ones.)

We want to get the  $k$ ’s on the left out of the scope of the existential quan-

tifiers. (In fact, at this point one has already the plan of building a suitable  $k$  for the right-hand side out of the  $k$ 's on the left, namely their product.) This can be done by applying the  $exE$ -rule twice(!) (once for each occurring  $k$ ), in short

```
apply (erule: exE)
apply (erule: exE)
```

Since we have applied the *erule* method, only the second subgoal (or premise) of the  $exE$  rule is left:

$$1. \wedge k \ ka[m = i * k, n = j * ka] \implies \exists k. m * n = i * j * k$$

Since the two  $k$ 's are now in the scope of the same quantifier, one of them has been automatically renamed and is now called  $ka$ . Still confronted with the task to prove an existential statement on the right, one may have the idea to come up with a suitable example, since this apparently proves such a claim. On the other hand, this is exactly what the  $exI$ -rule expresses. However, applying this rule “generically” would just cause a plain  $x$  to appear in the resulting subgoal which would be hard to process further. As we know (hopefully) that the  $k$  we are looking for is just  $k * ka$ , we can apply the rule with the additional constraint that the  $x$  in the rule is to be instantiated with  $k * ka$  which can be done using the method *rule\_tac*:

```
apply (rule_tac: x = "k * ka" in exI):
```

resulting in the subgoal

$$1. \wedge k \ ka[m = i * k, n = j * ka] \implies m * n = i * j * (k * ka)$$

Proving this is easy for the simplifier with the usual laws of arithmetic:

```
apply (simp)
```

This time Isabelle answers “no subgoals!!” indicating that the proof has been finished. Having proved our lemma we can from now on refer to it using its name *mult\_dvd\_mono*. It may turn out to be useful in further proofs. We close our proof script typing a relieved

```
done.
```

## 5. PRIOR CONTRIBUTIONS

Theorem proving with Isabelle can be useful in many fields where the domain of interest can be given a formalized concept. Of course, the whole world of (pure) mathematics is a good candidate. Even if there are fundamental limitations regarding the completeness of *any* formal (rich) axiomatic system as we have seen in section 3.2.2, Isabelle has been successfully applied to numerous areas. The formal proof of Sylow's first theorem in the library is only one example, in this case for the field of algebra.

But also “real-life” phenomena (at least if one happens to be a computer scientist) have been studied with Isabelle. The Unix theory by Markus Wenzel is a huge formalization of parts of the operation system (mainly concerned with the file system and the way users may access it) and proves - for instance - that certain invariants hold, even after the execution of allowed operations.

Cryptographic protocols is another field where the Isabelle system has been extensively deployed as we will explore in a bit more detail. In contrast to the emphasis of this work where I am mainly concerned with the (algebraic) equations arising from the DSA standard, those contributions examine complex protocols from a high-level point of view in order to derive statements on properties like the amount of knowledge a “spy” might be able to collect. One usually abstracts from the implementation details. Encryption functions are often treated as black boxes and are assumed to be resistant against attacks like crypto-analysis. The contributions and techniques presented in the following subsections are not meant as a complete survey, but as a (necessarily incomplete) collection of interesting examples.

### 5.1 *Paulson's Inductive Method*

The inductive approach to verifying cryptographic protocols by Lawrence C. Paulson [21][23] regards (or rather defines) protocols as sets of *traces*, where traces are lists of communication events. These lists are inductively defined,

whereas properties of traces can be proved by induction on them.

A communication event is the exchange of a *message* among *agents* which are the main (or rather the only) “characters” in the game. Usually, there are three types of agents, namely the server, “friendly” agents (with “names” 1,2,...) and the “spy”, whereas the datatype message is defined as follows [3]:

```

datatype msg =
  Agent agent
  | Number nat
  | Nonce nat
  | Key key
  | MPair msg msg
  | Hash msg
  | Crypt key msg

```

The difference between a number and a nonce is that the latter cannot be guessed in our model. In addition, hashing is assumed to be collision-free, the encryption function is perfect (i.e. cannot be broken).

Starting from a set of messages  $H$  we define three new sets of messages, called  $\text{parts } H$ ,  $\text{analz } H$  and  $\text{synth } H$  that are technically defined as the least sets closed under specific operations. (Fortunately, those least-fixpoint operator constructions are available in the Isabelle/HOL system.) The set  $\text{parts } H$  which describes all components of a trace that could *potentially* be recovered is defined according to the following rules [3]:

- $X \in H \Rightarrow X \in \text{parts } H$
- $\text{Crypt } KX \in \text{parts } H \Rightarrow X \in \text{parts } H$
- $\{X, Y\} \in \text{parts } H \Rightarrow X \in \text{parts } H$
- $\{X, Y\} \in \text{parts } H \Rightarrow Y \in \text{parts } H$

The set  $\text{analz } H$  - everything one can cryptographically access, especially using the keys that one already possesses - is determined by [3]

- $X \in H \Rightarrow X \in \text{analz } H$
- $\text{Crypt } KX \in \text{analz } H, K^{-1} \in \text{analz } H \Rightarrow X \in \text{analz } H$
- $\{X, Y\} \in \text{analz } H \Rightarrow X \in \text{analz } H$

## 5.1. PAULSON'S INDUCTIVE METHOD 5. PRIOR CONTRIBUTIONS

---

- $\{X, Y\} \in \text{analz } H \Rightarrow Y \in \text{analz } H$

Finally we define the set  $\text{synth } H$  that is the set a spy could build up from  $H$  by adding agents names and numbers, forming compound messages (e.g. from messages he or she intercepts) and applying hash as well as encryption functions using keys already contained in  $H$  [3]:

- Agent  $A \in \text{synth } H$
- Number  $N \in \text{synth } H$
- $X \in H \Rightarrow X \in \text{synth } H$
- $X \in \text{synth } H \Rightarrow \text{Hash } X \in \text{synth } H$
- $X \in \text{synth } H, K \in \text{synth } H \Rightarrow \text{Crypt } KX \in \text{synth } H$
- $X \in \text{synth } H, Y \in \text{synth } H \Rightarrow \{X, Y\} \in \text{synth } H$

The attacker (or the spy, respectively) observes all traffic, controls the network, sends fraudulent messages and controls a set of *compromised* agents. Since he has not the word “attacker” written on his forehead, others accept him as an honest agent. Let  $H$  be the set of messages the spy can see in a trace, namely the exchanged messages along with the data that comprised agents have stored. (In a public key scenario, for instance, the spy does not only know the public keys of all participants (that are known to everyone) but also the private keys of the compromised agents.) The set of messages the spy can derive can then be modelled as

$$\text{synth}(\text{analz}(H))$$

There are three types of possible events occurring in a trace [3]:

- Says  $A B X$ :  $A$  sends message  $X$  to  $B$
- Notes  $A X$ :  $A$  stores  $X$  internally
- Gets  $A X$ :  $A$  receives  $X$

The protocol steps are modelled as the extension of a trace by means of new events. Lost messages are modelled with the agent not responding.

**Example:** The Needham-Schroeder Public Key-Protocol (or rather its first steps) aims at authenticating two agents towards each other (e.g. in order to agree on a secret session key for a confidential exchange of data between them later on). Let  $ns\_public$  be the set of all legal traces in our protocol (definable in Isabelle as a set of lists of events). In the first step agent  $A$  sends his name along with a Nonce  $N_a$  to agent  $B$ , encrypted using  $B$ 's public key  $K_b$ , yielding the following inductive definitions [3]:

- $[] \in ns\_public$  (the empty trace is a legal trace)
- $evs1 \in ns\_public, N_a \notin used(evs1) \Rightarrow Says\ A\ B\ Crypt\ K_b\ \{N_a, A\} \# evs1 \in ns\_public$

where  $\#$  denotes the concatenation. In the next step  $B$  sends  $N_a$  back to  $A$ , which proves his or her identity as  $B$  (since he or she was able to decrypt the message encrypted with  $B$ 's public key) and attaches a new nonce  $N_b$ . Since this time the (whole) message is encrypted using  $A$ 's PK, the response (which is step 3) consisting of  $N_b$  proves now  $A$ 's identity. Thus we have analogous inductive definitions extending  $ns\_public$  according to step 2 and 3. But - in addition - we also have the following *fake* rule in our model (which is the same in every protocol) [3]:

- $evs \in ns\_public, X \in analz(spies(evs)) \Rightarrow Says\ Spy\ B\ X \# evs \in ns\_public$

where  $spies(evs)$  for a list  $evs$  basically denotes the set of all list items. The rule says that the spy may send *any* data he or she possesses (e.g. in order to cause an honest agent to expose some useful information in a response). Finally, we have in every protocol also the so-called *ups* rule modelling the loss of session keys.

Using these definitions it is possible to prove relatively easily (27 Isabelle commands according to [3]) lemmas like this: If  $A$  receives message 2 (including nonce  $N_a$ ) then  $B$  must have sent this message, i.e.  $A$  authenticates  $B$ .

## 5.2 Formalization of SET

### 5.2.1 The Protocol and its Objectives

The Secure Electronic Transaction (SET) protocol for a credit card-based, online payment system, is mainly based on works from enterprises involving

Visa and Microsoft, but also MasterCard, IBM, Netscape and CyberCash. Its main objectives are [23]:

- Confidential transmission of order- and payment data
- Securing the integrity of any transmitted data
- Authenticating merchant and customer along with protecting merchants from dishonest customers and vice versa (by means of a prior compulsory registration)

There are three parties involved in the actual purchase phase: The customer, the merchant and the payment gateway that is essentially a bank. Of general interest might be a peculiar “ingredient” involved here. In order to share information only with those parties which really need it the concept of a *dual signature* has been invented: We split a message into two parts  $m_1$  and  $m_2$ . Each part is hashed; both hash values  $h_1$  and  $h_2$  are concatenated and hashed again. Only the latter hash value is finally signed yielding signature  $s$ . Both recipients get signature  $s$ , whereas the first one receives  $m_1$  (the part of the message exclusively meant for him) and  $h_2$ , the second one is equipped with  $m_2$  and  $h_1$ . This way both parties receive only the relevant parts of the information in plain (or rather those parts they are allowed to read) but can check the integrity and authenticity of the whole message. In this scenario the customer will share the order information with the merchant but not with the payment gateway, the payment information on the other hand concerns the bank but not the merchant.

There are five main stages involved (which may be seen themselves as protocols each): Cardholder registration, merchant registration, purchase request, payment authorization and payment capture. In the registration phases (where the customer is equipped with a key pair for digital signatures by a trust center, for instance) the actual authentication takes place outside the protocol. Another interesting feature is the application of hybrid techniques (here called digital envelopes) for the exchange of confidential messages: A secret symmetric key  $K$  is encrypted using the recipient’s public key whereas the actual data is protected by means of  $K$  which reduces the amount of costly public key encryption/decryption operations but complicates the protocol analysis.

We will now have a closer look at the Cardholder Registration stage, since this is where Paulson et al. found an interesting flaw as we will see in the next section. Cardholder Registration (that may itself be seen as a complete protocol) consists of the following steps [1]:

- **Initiate request:** Cardholder requests registration
- **Initiate response:** Certificate authority returns its public key
- **Registration form request:** Cardholder transmits his Primary Account Number(PAN), i.e. usually his credit card number (encrypted using the CA's PK)
- **Registration form:** CA sends an application form suitable for the bank that issued the credit card
- **Cardholder certificate request:** Cardholder returns the completed application form along with his public signature key and a 20 byte random number, the CardSecret
- **Cardholder certificate:** CA delivers a certificate (containing the cardholder's public key) and (data for computing) the 20 byte PANSecret

In detail, the Cardholder certificate message from the CA to the cardholder looks like this:

$$6. CA \rightarrow C: \text{Crypt}_{KC2} \left( \text{Sig}_{CA} (C, NC3, CA, \text{NonceCCA}) \right. \\ \left. \text{Cert}_{CA} (\text{pubSKC}, \text{PANSecret}) \right. \\ \left. \text{Cert}_{RCA} (\text{pubSKCA}) \right)$$

where  $\text{PANSecret} := \text{CardSecret} \oplus \text{NonceCCA}$ . The message contains the CA's signature of the cardholder's name, its own name and two nonces, along with the actual certificate (issued by the CA) for the cardholder's public key and his PANSecret. In addition, the CA's own certificate (issued by a *Root CA*) is attached. The whole message is encrypted using the pre-shared key KC2. We will address the curious way the PANSecret is determined in the next subsection.

### 5.2.2 Verification of SET in Isabelle

Lawrence C. Paulson together with Giampaolo Bella, Fabio Massaci and Piero Tramontano deployed the Isabelle theorem prover to the SET protocol (family) in order to examine the protocol's properties, especially concerning the security goals that are to achieve [1][14][23].

Even if specialized verification tools might be more powerful than Isabelle, the latter turns out to be more flexible which was already an advantage concerning the first modelling tasks: Since the SET protocol involves not only

the “generic” agents presented in section 5.1 but also specialized characters like the payment gateway Paulson et al. could easily adjust the *protocol* theory, already present in the Isabelle library.

A first (or maybe even the hardest) challenge was interpreting the documentation (consisting of a *Business Description*, a *Programmer’s Guide* and a *Formal Protocol Definition*) such that a formal model could be approached. In contrast to simpler protocols (in 1978 six pages were sufficient to describe Needham/Schroeder) we are talking about several hundreds of pages. According to Paulson the documentation suffers from a lack of concise formulations. In addition, clear statements about the protocol’s precise objectives are missing. Fields declared as “optional” were another source of confusion since they may play a major role in achieving (and proving) security features. Finally, Paulson et al. had to insert reasonable assumptions about the undefined parts like the way a certificate authority handles customers’ registration in detail.

The formalization of the protocol steps (they were treated separately according to the partition presented above) followed the inductive paradigm described in 5.1 such that the actual encoding looked as presented in the Needham Schroeder example, even if here - due to more complex message formats and the like - the specification of a rule looks much more complicated and longer.

Apart from the last stage of Purchase Capture (whose investigation had not been completed according to [23]) Paulson et al. discovered - under reasonable assumptions - no major attacks against the registration or the purchase phase(s). Most of the (assumed) security objectives could be formally derived, even if the so-called digital envelopes caused problems and were treated in a more abstract way (implicitly assuming that two parties actually succeed in agreeing on the same session key). The SET documentation did not tell them what properties to prove, so they specified them themselves. Of course, we want the PAN and the PANSecret to remain secret, for instance. On the other hand, each party must be assured that all the other parties agree on all the essential details. The latter fails in one interesting respect: The identity of the intended payment gateway is not mentioned in a suitable message of the purchase phase, but this does not appear to allow any significant attack.

A bit more serious is what Paulson has to say about the PANSecret and the way it is determined in the Cardholder Registration phase, even if this

is not a result of formal analysis but of mere inspection. The cardholder must present the PANSecret when making purchases, it gets in some sense a part of his identity. One might ask what we need it for since the cardholder already identifies himself (beyond any reasonable doubt) by means of his private signature key.

Even if one assumes that the PANSecret is necessary, why does it need to be computed using (as an ingredient) NonceCCA from the CA? Without NonceCCA, message 6 - presented at the end of the last subsection - could be sent completely in plain since - apart from NonceCCA - it would only contain “unclassified” information like certificates. This would not only make the protocol simpler but also more secure: Any exchange of encrypted messages apparently involves the risk of being “broken”, i.e. especially the key used for encryption could potentially be compromised; and this would make it easier to break other keys as well, as long as they somehow depend on the first one.

Since the session key for message 6 (which as we have seen could be sent in plain if SET did without the annoying PANSecret) needed to be exchanged in message 5 (again encrypted by a key) we yield (due to the encryption of message 6) a longer *dependency chain* among keys which is - according to Paulson - relatively hard to model but obviously increases the risk for each key in the chain to be broken somehow, exposing the data encrypted by it. In addition, the computation of the PANSecret as an exclusive-OR gives the CA full control over its value, so hopefully your certificate authority is as trustworthy as one might expect.

As to the technical conclusions Paulson et al. draw, also complex protocol (families) like SET seem to be amenable to formal analysis using powerful tools like Isabelle/HOL, even if one sometimes faces huge subgoals consisting of hundreds lines and more. Diagnosing why a proof has failed is often another challenging task. Paulson supposes that formalizations could get easier and more convenient based on a more generic treatment of digital envelopes abstracting from the details, with the digital envelopes themselves formalized (or verified) separately.

As far as cryptographic protocols there are many sources of possible misunderstandings in today’s documentations. For future improvements in this field Paulson suggests three essential aspects a protocol documentation should contain [23, page 10]:

- an abstract version of the message flow, comprising the core security features, only
- the protocol’s precise objectives, expressed as guarantees to each party
- the protocol’s operation environment, including the threat model

### 5.3 Other Areas

There are numerous other examples where the Isabelle theorem prover has been deployed to cryptographic affairs. However, in almost any of those cases the device under test was a whole cryptographic protocol where the single algorithms involved were mainly treated as black boxes. That is why these examples - even if highly interesting as such - do not seem to differ too much from the formal verification of the SET protocol in the last subsection, in principle.

As far as other fields are concerned, the Isabelle system has proved successful (in the full sense of the word) in a variety of applications. The following list is merely meant as a little collection of applications from various domains.

**Contributions in the libraries:** The Isabelle/HOL library alone offers meanwhile a wide selection of interesting theories that can be easily imported in order to enrich home-made creations. One can find almost everything worth knowing about the basic algebraic operations and the divides relation for natural numbers, but also more advanced topics in number theory like the Chinese Remainder Theorem. Apart from the world of algebra there is also a theory of lists, and even examinations of more practical subjects of computer science like Java and Unix are at hand.

**Source code verification of a micro kernel in C++:** Michael Hohmuth, Hendrik Tews and Shane G. Stephens deployed the Isabelle/HOL at the Dresden University of Technology in order to derive security relevant properties of their so-called Fiasco microkernel [13] written in a subset of C++. They developed a formal semantics for this subset of (“SafeC++”) in order to express it in HOL. Even if I unfortunately could not find out too much about any actual results concerning the verification (we speak about an ongoing project), their semantic model of SafeC++ (a relatively rich subset of the whole language) seems interesting, but not too surprising at first

glance. The semantics of SafeC++ expressions are basically defined as (system) state transitions including modelling of value assignments to variables and even pointer arithmetic can be expressed, where pointers are interpreted as locations in the memory, as usually.

**A sound and complete theorem prover for FOL:** Tom Ridge (Edinburgh University) and James Margetson created a system that looks at first like a distracting curiosity [20]: A first order theorem prover, modelled within the HOL version of the Isabelle proof assistant. The core of this work is simply a deduction system for first order logic (as outlined in section 3) together with models for the language's syntax and semantics. The goal was to formally prove the correctness and completeness of the modelled deduction system.

The correctness proof works in general as described in chapter 3, namely by induction on the length of a sequent, whereas the completeness is shown in a technically different way: Any valid consequence must be derivable, which is equivalent to the condition, that anything that cannot be derived is not (semantically) valid, either. In other words, it does not hold in every model. Any (successful) derivation contains only finitely many steps, whereas any unsuccessful attempt to derive something somehow ends up in an infinite sequence. In short, Ridge et al. were able to construct a model for any such infinite sequence, such that the implication one was (unsuccessfully) trying to infer, does not hold semantically in this model, either, which already proves the completeness.

**Tiling the mutilated chessboard by dominos:** Lawrence C. Paulson has not only developed "serious" applications like the verification of cryptographic protocols like the SET protocol presented above, but also a formalization of the following problem [19]: Suppose two diagonally opposite squares are removed from a chessboard, can the remaining 62 squares be tiled by (31) dominos? No. Since each domino covers a white and a black square, the total number of white squares that are covered must always be equal to the number of (covered) black ones. On the other hand, removing two diagonally opposite squares leaves always more black squares than white ones, or vice versa. This simple argument, however, turns out to be relatively hard to formalize in order to prove the correctness of the negative answer.

According to Paulson, it was even relatively difficult to derive - from the definition of a tiling - that each domino will always cover a black and a white square. The set of (possible) dominos was modelled as a set of pairs of pairs

of natural numbers (the words “of pairs” are repeated intentionally) and the set of tilings is inductively defined as the set of unions of smaller tilings that do not overlap. Apart from a few minor difficulties like the one above, the mechanical proof as such turned out to be straightforward according to Paulson. The example is interesting since it had stood as a challenge to the automatic reasoning community after its posting dating back to 1964.

## 6. CORRECTNESS OF DSA

I will now return to the actual subject of this thesis, namely proving the correctness of the DSA signature scheme in Isabelle/HOL. When we use the term *correctness* we should first define what we mean by this. We shall regard a signature algorithm (or signature scheme) as correct if and only if the legitimate signatory obeying the algorithm will not be rejected by a verifier (as long as the verifier himself “works correctly”).

The opposite direction - a successful verification guarantees that the signature originates from the legitimate signatory - would probably be even much more interesting. Unfortunately, such a proof is impossible, since an attacker could always be lucky enough to *guess* exactly the output of the legitimate signatory even if this is apparently highly improbable. But even a proof showing that it is extremely *hard* to generate a valid signature as a principal other than the legitimate signatory (i.e. without the private signature key as a trap door) has not been found yet. Since an attacker who is able to efficiently compute discrete logarithms could relatively easily break DSA (and other cryptosystems like ElGamal, of course), such a proof would have to include that problems like determining discrete logarithms (but also the factorization problem) are hard in the sense that there exist no efficient algorithms for their solution.

For the time being we must confine ourselves to the salving fact that an efficient algorithm for the discrete logarithm or the factorization problem (or any other essential approach to break RSA, ElGamal or DSA) has not been found yet, as long as we do not have (mature) quantum computers at our disposal. If this changes one day (due to new theoretical findings or due to considerable progress in the field of quantum computers) most of today’s cryptosystem will no longer be safe. Fortunately, there already exist alternatives for the “quantum computer age” like the McEliece Cryptosystem, for instance.

### 6.1 A “Pen and Paper” Proof

Remember that for a DSA signature we need the following ingredients (for details, see section 2.3):

- prime numbers  $p$  and  $q$  with  $q \mid (p - 1)$
- an element  $g$  of order  $q$  in the group  $(\mathbb{Z}/p\mathbb{Z})^*$
- random integers  $x$  and  $k$  smaller than  $q$ , along with  $y := g^x \bmod p$

where  $(x, k)$  is kept private representing the private key.

The signature for a message  $m$  is  $(r, s)$  where

- $r := (g^k \bmod p) \bmod q$
- $s := (k^{-1}(h(m) + xr)) \bmod q$

Assume that the verifier receives message  $m'$  and signature  $(r', s')$  and that  $m' = m$ ,  $r' = r$ ,  $s' = s$  (i.e. no transmission error occurred). In addition, we assume that these values have been computed according to the above equations. We need to show that the verifier accepts the signature.

Since both  $r$  and  $s$  have been reduced modulo  $q$ , the first condition the verifier needs to check -  $r$  and  $s$  are smaller than  $q$  - is apparently satisfied.

Thus, the verifier will accept the signature iff the following holds:

$$((g^{(h(m)s^{-1}) \bmod q} y^{(rs^{-1}) \bmod q}) \bmod p) \bmod q = r$$

We will show that this is the case. In the following “ $\equiv$ ” stands for the equivalency modulo  $p$ , i.e.  $a \equiv b \iff a \bmod p = b \bmod p$ .

$$\begin{aligned} & g^{(h(m)s^{-1}) \bmod q} y^{(rs^{-1}) \bmod q} \\ & \equiv g^{(h(m)s^{-1}) \bmod q} g^{x(rs^{-1}) \bmod q} \end{aligned}$$

(by definition of  $y$ )

$$\equiv g^{h(m)s^{-1}} g^{xrs^{-1}}$$

(since  $g$  has order  $q$  (and thus  $g^x$  has an order that is a divisor of  $q$ ) and therefore reducing the exponent modulo  $q$  does not change the result of the exponentiation)

$$\equiv g^{h(m)s^{-1}+xrs^{-1}}$$

(by a well-known law of exponentiation)

$$\equiv g^{s^{-1}(h(m)+xr)}$$

(by the distributive law)

$$\equiv g^k$$

where the latter equivalency follows from the defining equation of  $s$ , namely  $s := (k^{-1}(h(m) + xr)) \bmod q$ , after multiplying both sides with  $k$  and  $s^{-1}$ . Summing up,

$$(g^{(h(m)s^{-1}) \bmod q} y^{(rs^{-1}) \bmod q} \equiv g^k) \bmod p \iff$$

$$(g^{(h(m)s^{-1}) \bmod q} y^{(rs^{-1}) \bmod q}) \bmod p = g^k \bmod p \implies$$

$$((g^{(h(m)s^{-1}) \bmod q} y^{(rs^{-1}) \bmod q}) \bmod p) \bmod q = (g^k \bmod p) \bmod q = r$$

where the latter equation shows that the verifier will accept the signature.

This proof has been intentionally written in a “annoyingly” informal style to underline the drawbacks of pen and paper proofs, like the risk of misunderstandings and misinterpretations. For example, the last three lines of equations with the implication arrows connecting them only express that the first two ones are equivalent and that the third one follows from them. What we really want to express is that these implications hold *and* that the first equation (and consequently the others as well) is a true statement, since we have just derived it.

In addition, the justification above using the defining equation for  $s$  fails to mention that for this step we need again the fact that two exponentations (with  $g$  as basis) yield the same result if the exponents are equivalent modulo  $q$ .

Consequently, when reading a proof like this, we should always remember the “meta-rule” stating that a mathematical formula (or a statement or a proof) is always to read the way it was meant, probably the strongest weapon against misunderstandings and errors of any kind.

**Note:** For another pen and paper proof that is a bit more concise, please refer to [9], for instance.

## 6.2 The formal proof

### 6.2.1 A Motivation

Proofs like the one presented above always contain references to arithmetic (and other) “rules” where the reader has to (intuitively) verify that the rule is correct and - even more difficult - that the rule is applicable to the current situation. Besides, we often find formulations like “it is easy to see that” or “the following is obvious”. Hopefully, in such situations the reader will actually comprehend the obviousness of the argument involved. But even if he does not, he (or she) will probably quickly believe in it, since statements of this kind are simply too convincing.

Of course, the DSA signature scheme is correct in the sense described at the beginning of this chapter. We do not need to argue about this. However, one might ask we could not translate the usual proofs like the one above into a formalized version, based on an explicitly defined set of preconditions and axioms, such that we could also convince a machine whose reasoning is based on a formal language together with certain (evidential correct) deduction rules.

Misunderstandings, misinterpretations and of course incorrect proof steps are likely to be prevented this way since our machine (or the program, respectively), would complain on time. In addition, we can keep track of the rules and axioms that are actually involved in our reasoning. It might turn out that certain (implicit) preconditions that we believed to be essential in our “blurred” hand written arguments are redundant and can be eliminated which may result in even more general and stronger theorems.

In addition, formal proofs often tend to be closer to the actual implementation if we talk about algorithms that are translated into programs. This can apply to inductive definitions of functions, for instance. Maybe we generate pen and paper proofs about the features of our new crypto-module,

but our modelling fails to consider the way our algorithms are implemented in sufficient detail. Of course, this argument mainly concerns the field of program verification, rather than the verification of algorithms themselves as in this case.

If we restrict ourselves to a formal framework like FOL or ZFC (which is usually necessary in order to approach formal proof styles) we are also less prone to unsound concepts and definitions. The pitiable Barber of Sevilla (often referred to as the Barber Paradox) who shaves precisely those men that do not shave themselves, is an instance of such a construction, since - if we ask if the barber shaves himself - both possible answers will lead to a contradiction. The reason for this is the unsound way the Barber's customers are "defined", namely  $\forall x(\text{shaves}(\text{barber}, x) \iff \neg \text{shaves}(x, x))$  which is not a definition but an unsatisfiable formula like  $P \wedge \neg P$ .

Based on the formal framework of ZFC we could define the set of the Barber's customers by means of a (fixed) binary *shaves*-relation as  $Customers := \{x \mid \neg \text{shaves}(x, x)\}$ . But then we could not simply *claim* that this is at the same time the set of all men who are shaved by the barber, since this would alter the *shaves*-relation. In fact, we would probably need two different relation symbols, one for the usual do-it-yourself-shaving and one for the activity of the barber. Even if we do not know if ZFC is consistent, unsound constructions of this kind are not allowed.

Another source of confusion that is likely to be prevented in formal automatic reasoning may be called the illegal application of proof schemes. If we were a (very, very..) inattentive audience, we might believe in the following argument: "If we add up a set of real numbers, the order apparently does not matter. Why? Well, we have the commutative law for addition. Thus, the claim follows by induction on the number of summands."

Of course, the claim is false. An interesting counterexample is a series that is convergent but not absolutely convergent, i.e.  $\lim_{n \rightarrow \infty} (\sum_{i=1}^n a_i) = b$  for a (finite) real number  $b$ , but  $\lim_{n \rightarrow \infty} (\sum_{i=1}^n |a_i|) \rightarrow \infty$ . In this case, the elements of the series can be rearranged in a way such that the series takes on any arbitrary real value: For *any* arbitrary real number  $b^*$ , there is a permutation  $\pi : N \rightarrow N$  such that  $\lim_{n \rightarrow \infty} (\sum_{i=1}^n a_{\pi(i)}) = b^*$  (for details, see [2], for instance).

What went wrong? In the above "proof", induction has been deployed to derive a commutative law for additions involving sets of summands of

arbitrary cardinality. However, by the usual induction starting with the conventional commutativity law as the base case we can only prove the commutativity for arbitrary *finite* sums. Of course, systems like Isabelle would never tolerate such a “transfinite abuse” of ordinary induction.

### 6.2.2 The Roadmap of the Proof

What follows is a summary of the most important steps of the formal proof in Isabelle/HOL in order to outline the general course. The actual encoding along with all necessary lemmata and - of course their proofs - can be found in the appendix.

Instead of directly proving the verification equation in the group  $(Z/pZ)^*$  I decided to consider the more general case of an arbitrary finite cyclic group. On the one hand we yield stronger (more general) statements this way. On the other hand the reasoning gets a bit simpler since we can eliminate the reductions modulo  $p$  (appearing in the hand written proof) where  $p - 1$  is the group order.

To reduce the amount of possible confusion the following listing contains the relevant parameters together with their “spelling” in the actual theory file and their role in the DSA algorithm. As far as possible I will stick to the notations used in the official DSA specification presented in section 2.3. The exception is the natural number  $n$  which is our group order replacing  $p - 1$  in the specification. Also note that if we consider an arbitrary finite cyclic group rather than  $(Z/pZ)^*$  we have to carefully distinguish between group elements and ordinary natural numbers.

The only parameter that causes a slight headache this way is the element  $r$  since it appears in the specification both as a group element of  $(Z/pZ)^*$  (reduced modulo  $q$ ) and as an exponent. I decided not to give up the general point of view of arbitrary finite cyclic groups due to this difficulty. I rather regard the element  $r$  in my model as a natural number, therefore only appearing as an exponent, not as a group element. Instead, in the verifying congruency I will show that the left-hand side equals to the (unique) group element  $g^k$ . In the special case of  $(Z/pZ)^*$  it should be reduced modulo  $p$ , written  $g^k \bmod p$ , and since we are in the subgroup generated by the element  $g$  of order  $q$ , we can further reduce its encoding modulo  $q$ , yielding  $(g^k \bmod p) \bmod q$ , which is used to define parameter  $r$ .

Of course, in practice the signatory has to transmit the group element  $g^k$  in a “sealed” way not exposing the exponent  $k$ , such that at the same time the exponent  $r$  can be easily extracted. For  $(Z/pZ)^*$  - and this is the only case allowed (or rather suggested) by the specification - these two parameters coincide.

However, the verifying congruency holds even if we regard  $r$  and  $g^k$  as independent of each other, as long as the second component of the signature, namely  $s$ , is computed as defined in the specification. Therefore, my *verifying* theorem later on is strictly speaking a generalization of the actual verification of DSA, concerning the group we are regarding as well as concerning the relationships between the parameters involved.

The following elements will appear in my encoding:

- natural numbers  $n$  and  $q$ , as defined above - such that our group order gets  $n$  - along with the quotient  $u := n/q$ , or equivalently, as it will appear in the actual encoding,  $uq = n$
- a primitive root (i.e. especially a group element)  $h$  and a group element  $g$  such that  $g := h^u$ , which is the same definition as in section 2.3 since  $u := n/q$
- natural number  $x$  and group element  $y := g^x$  playing the same role as in the specification
- natural numbers  $k$ ,  $r$  and  $s$  (according to the specification) along with natural numbers  $k^{-1}$  and  $s^{-1}$  (or  $k\_inv$  and  $s\_inv$ , as written in the theory file, respectively), denoting the inverse (modulo  $q$ ) of the  $k$  and the  $s$  values

Using the above “ingredients” the correctness proof of the DSA signature algorithm) in the theory file `dsa.thy` comprises the following steps:

- **constants definitions**
  - the natural number  $n$  such that  $n$  represents the order of our finite group
  - a primitive root  $h$  for a finite cyclic group (which is from Isabelle’s point of view of no specific type but annotated by a *type variable*)
  - the functions  $inv$  and  $pow$  where  $inv(g)$  yields the inverse of a group element  $g$  and  $pow(g,k)$  for a natural number  $k$  is the  $n$ -th exponentiation of  $g$  regarding the group operation

In our theory these steps are handled using the following encoding:

```

consts
  h::"a"
  n::"nat"
  inv::"a"  $\Rightarrow$  "a"
  pow::"[a, nat]  $\Rightarrow$  'a"

```

As already explained  $'a$  is a type variable, which means the functions *inv* for instance may be applied to arbitrary types. However, characteristic properties like the (semantic) meaning of *inv* as the inverse of a group element will probably hold only in the particular axiomatic type classes that we can define now.

- **definitions of the axiomatic typeclasses**

- axiomatic typeclass *group\_mult*, which is a subtype of *monoid\_mult* in the Isabelle library, where *mult* means basically that the group operation is denoted by “\*”
- subclass *cyclic\_group\_mult*, the class of all cyclic groups, i.e. there is a primitive root *h*, such that each element can be written as a power of *h*
- *finite\_cyclic\_group\_mult* that a cyclic group of fixed (but arbitrary) finite cardinality *n* and the “universe” of our proof, i.e. we do not consider  $(\mathbb{Z}/p\mathbb{Z})^*$  but arbitrary finite cyclic groups

Since these are definitions the whole theory is based on I also present their “spelling” in Isabelle here.

```

axclass group_mult  $\subseteq$  monoid_mult
  left_inverse: "inv a * a = 1"
  pow_def_0: "pow a 0 = 1"
  pow_def_suc: "pow a (Suc k) = (pow a k) * a"

```

```

axclass cyclic_group_mult  $\subseteq$  group_mult
  generator: " $\exists k. \text{pow } h \ k = a$ "

```

```

axclass finite_cyclic_group_mult  $\subseteq$  cyclic_group_mult
  finite: "pow h n = 1"

```

A group is just a monoid with the additional property of the *left\_inverse* rule. (From this one could also derive a corresponding *right\_inverse* rule if necessary, i.e. the latter already follows from this and does not need to be added as an axiom.) In the definition of cyclic groups the group element  $a$  is implicitly  $\forall$ -quantified like all variable symbols occurring in such formulae, unless they are equipped with another (explicit) quantifier or defined as a constant.

With these definitions we can now derive first lemmata, mainly concerned with the *pow* function:

- **basic associativity lemmata for groups**

- lemma *pow\_assoc1* stating that  $a^k * a^l = a^{k+l}$
- lemma *pow\_assoc2* revealing the hardly surprising finding  $(a^k)^l = a^{kl}$

I tried to keep these lemmata as general as possible. The first one, for instance, is valid (and defined) for arbitrary groups, not only for finite cyclic groups (even if we only need it there). In between there is from time to time a collection of relatively uninteresting lemmata (that are omitted here).

- **a lemma about the order of group elements**

lemma *subgroup\_generator*, mainly concerned with the role of the element we denoted by  $g$  in the DSA specification generating the subgroup in which all the DSA computations finally take place:  
 $uq = n \wedge g = h^u \implies g^q = 1$ , i.e. element  $g$  generating our subgroup has an order that divides  $q$  (and as a corollary  $uq = n \wedge g = h^u \implies g^{qk} = 1$ )

Based on some more auxiliary lemmata we now approach the discovery that reducing the exponent modulo  $q$  is irrelevant for exponentiations with this element  $g$  as basis. Note that  $g$  is always defined in terms of the primitive root  $h$  which has been defined as a constant.

- **lemmata for reduced exponents**

- lemma *pow\_mod1* :  $uq = n \wedge g = h^u \implies g^{r+kq} = g^r$

- lemma *pow\_mod2* :  $uq = n \wedge g = h^u \wedge \exists l(k = r + lq) \implies g^k = g^r$
- lemma *pow\_mod3* :  $uq = n \wedge g = h^u \implies g^k = g^{k \bmod q}$
- lemma *pow\_mod4* :  $uq = n \wedge g = h^u \wedge k \bmod q = l \bmod q$   
 $\implies g^k = g^l$

The lemmata we have proven so far are almost sufficient to approach the verification congruency. But there is still a little gap we have to close. We need from the defining equation of natural number  $s$ , that is

$$s := (k^{-1}(h(m) + xr)) \bmod q$$

the following congruency modulo  $q$ :

$$k \equiv s^{-1}(h(m) + xr) \bmod q$$

which looks for the “human reader” like a matter of course, since this seems to follow simply by multiplying both sides with  $s^{-1}$  and then with  $k$ . However, if we have a closer look we realize that we also need the commutativity of multiplication along with the finding that the equivalency modulo  $q$  for a natural number  $q$  is actually a *congruency* relation, both for addition and multiplication. Furthermore, we need to apply those rules several times and (in a formal proof) we cannot simply say something like “analogously it follows ...” skipping the details in the second round.

Even if it is not about a technically difficult deduction these reasons cause the formal proof in Isabelle to turn out relatively long and bulky, hard to read for a human eye. Of course, it might be possible to prove the following lemma - basically expressing the above consequence - in a slightly more elegant fashion. If I regard my proof scheme in retrospect this applies especially to this proof. However, writing *elegant* proves in Isabelle is beyond the scope of this thesis.

- **a necessary ingredient**

$$\begin{aligned}
&\text{lemma } \textit{inv\_exp}: \\
&a = b^{-1}c \bmod q \wedge \\
&aa^{-1} \bmod q = 1 \wedge \\
&bb^{-1} \bmod q = 1 \\
&\implies b \bmod q = a^{-1}c \bmod q
\end{aligned}$$

- the final step

theorem *verify*:

$$uq = n \wedge$$

$$h^u = g \wedge$$

$$y = g^x \wedge$$

$$s = k^{-1}(H + xr) \bmod q \wedge$$

$$ss^{-1} \bmod q = 1 \wedge$$

$$kk^{-1} \bmod q = 1$$

$$\implies g^{((s^{-1})H) \bmod q} y^{(r(s^{-1})) \bmod q} = g^k$$

In the above,  $H$  denotes the hash value of the message that is to sign,  $h$  is our primitive root. These two steps take both a bit longer than the previous lemmata. This is mainly due to term-rewriting operations, rather than technically interesting reasoning.

### 6.2.3 What the Theorem expresses

I have already pointed out the difference between my theorem and the verifying equation appearing in the specification. The latter is in some sense a special case of my version: The generic finite cyclic group is “instantiated” by  $(\mathbb{Z}/p\mathbb{Z})^*$  and we add - logically as a further assumption -  $r = g^k$ . Moreover, the reduction modulo  $q$  of both sides of the equation needs to be left out in my theorem since this kind of reduction is only defined for natural numbers (or integers), but not for elements of arbitrary groups.

Thus, we may ask now if the actual verifying equation in the specification follows from my formalization, or what we need to assume for this. Strictly speaking, we can transfer the theorem to the special case in the specification under the following assumptions:

- $(\mathbb{Z}/p\mathbb{Z})^*$  is a finite cyclic group.
- Reductions modulo a natural number  $q$  “preserve” equality, i.e.  
 $a = b \implies a \bmod q = b \bmod q$ .
- Adding an assumption does not reduce the set of consequences, in short,  
 $(\Phi \implies \phi) \implies (\Psi \implies \phi)$  for  $\Phi \subseteq \Psi$ .

Concerning these aspects I will not use the term “obvious” (since this would be exactly what I criticized concerning common informal proofs at the beginning of this section). Instead, I claim that the first assumption could

also proven in Isabelle/HOL relatively easily. The second one appeared in my encoding as a simple lemma (and if this statement did not hold, “mod” would not even be a function), whereas the third “meta-claim” is part of any reasonable deduction system - especially present in Isabelle.

Therefore I assert that my theorem expressing the verifying equation (or verifying congruency) actually entails the special case suggested in the DSA specification. In addition, it is more generic and can therefore be applied to a wider spectrum of instances. Summing up and eventually returning to the world of informal and vague statements:

**Corollary:** The DSA signature scheme is correct

## 7. CONCLUSIONS

### 7.1 *My practical Experience*

After a few months of working with the Isabelle / HOL system - at first in terms of reading the documentation and introductory papers, then the actual application aiming at formalizing and proving theorems in practice and finally creating and gradually adjusting what I regarded as a suitable DSA theory - I would assess several parts of the system separately.

Of course, I am far from being an expert. Therefore, my judgement should not be taken too seriously. I am lacking a sufficient amount of comparative experience with other theorem provers (or proof assistants). Moreover, proof attempts that failed may have resulted in some cases from an inadequate comprehension of the method or tool I was applying.

In short, the automatic tools (i.e. the tools designed for handling some parts of a proof automatically like *simp*, *blast* and *arith*) did not really turn out to be very helpful in my case. For instance, as far as the simplifier is concerned, I found it relatively difficult to fine-tune its behaviour (mainly by enlarging or pruning the amount of rules it may use) such that it does enough to yield some results without “messing up” any terms too strongly and without starting to loop. However, I have to admit that I applied the simplifier quite often, mainly for proving relatively easy steps.

*blast* sometimes managed to prove interesting (also harder) subgoals, but also in this case it often appeared to be necessary to break a proof down into little pieces first, until also a human would have been able to prove it in a few steps, using only atomic rules. Of course, this might be different in case of proofs that involve nested predicate logic expressions, for instance.

Due to these reasons (or maybe due to my impatience) I mainly concentrated on what I regard as the simple core features of Isabelle (which always worked without any difficulties leading to a proof process that was productive and easy to survey, but not always short), namely the rules along with

referring to (prior) lemmata and creating subgoals, using the simplifier and *blast* merely at the “leaves” of my proofs.

## 7.2 Benefits of Formalization

The following aspects have in part already been discussed in the sections above, in part however, they become more apparent as soon as one actually works in the field of formal verification in practice.

A necessary condition for this kind of treatment - independent of the verification tool we are using - is always a formalized encoding of the “device under test”. As far as the DSA signature scheme is concerned this might be a relatively easy task (since algebraic equations by their very nature are already “close” to a formal encoding suitable for systems like Isabelle), but for systems like complete cryptographic protocols a more complex framework needed to be developed as we have seen in section 5.1 in terms of Paulson’s Inductive Method, and according to Paulson the Specification that was issued by Visa and Mastercard was in part lacking precise formalizations; in a (genuinely) formal specification suitable also for system like Isabelle this would probably be prevented.

The mutilated chessboard problem in section 5.3 is an instance where coming up with a useful formalization appears difficult, whereas the argument behind the proof itself is extremely trivial from a human point of view. However, also for DSA and similar algorithms we have to carefully define the “universe” of our model along with all necessary operations and axioms involved such that we get neither too generous (by deploying too many preconditions) nor too restricted. In the following steps we try to draw important conclusions, gradually approaching what we attempt to prove. If we have already a pen and paper proof in our mind (as in this case) we need to somehow translate the arguments into our formalized language which is not always a trivial task. Sometimes one realizes that a proof step appearing easy at first glance requires a deeper examination in reality. Rewriting a congruency (modulo a prime number  $q$  as in this proof, for instance) is an “atomic step” within our informal reasoning. However, trying to convince Isabelle of these allegedly easy proof steps, one notices that in fact there are numerous algebraic findings as preconditions involved here that need to be somehow inserted into the proof.

Thus, in my view writing formal proofs with systems like Isabelle / HOL may even change the way we reason ourselves. Of course, one should not start to unfold every mathematical proof one comes across in this depth, since this would consume too much human computing time. But it might be a good idea to be always aware of the fact that our informal reasoning could (and sometimes should) be broken down into formal atomic steps of deduction.

Depending on the theorem(s) we want to prove, systems like Isabelle can sometimes come up with a derivation much faster than a human alone. There might be cases where there has not been found any pen and paper proof so far and we deploy the system in order to find one. This is at least for new lemmata (which have hardly been examined before) a realistic scenario, and as far as my work is concerned, at least two or three times tools like *blast* somehow handled the remainder of the proof within one step, although I had no clear idea on how this might work, at first glance.

But also the translation of existing human-written proofs - probably the much more frequent case - is often a good idea, since it not only deepens our understanding of the proof steps involved, as explained above. We always know exactly which rules and techniques have been deployed for our derivations. As to “pure mathematics”, this might sound like a relatively weak argument, since in this field we always claim that anything we deploy (like especially mathematics proof techniques) is correct. But even there a careless human verifier might tolerate an illegal application of a rule, as my little example of the “transfinite induction” in section 6.2.1 should have at least suggested.

Even stronger is the argument in cases where we have to make some (in part temporary) assumptions in order to prove certain properties. This might apply to the examination of cryptographic protocols, for instance. If we realize later on, that some of those assumption can no longer be regarded as correct or realistic, we know exactly - based on our neat and formal derivations - which implications or theorems should be revised.

### 7.3 What is left to be done

In contrast to high-level examinations of cryptographic protocols, there have not been too many attempts to formalize the single cryptographic algorithms like the DSA in this case. I can only assume that this is in part due to the

“less innovative” reasoning scheme involved here: Proving the correctness of DSA has already been performed in many pen and paper proofs and does therefore not appear too thrilling in the formalized version, at first glance, whereas reasoning based on Paulson’s inductive method looks more exotic from a human point of view.

Nevertheless, we need more such formal encodings and proofs of cryptographic (and other) algorithms in the future as I tried to underline in this thesis. Of course, a bit shorter proofs (compared to my encoding presented in the appendix) would not hurt in some cases. To preserve the compatibility with readers not familiar with the way Isabelle proofs work, the so-called Isar syntax expressing proofs in a more natural language might be a good compromise, too.

As far as the Isabelle project is concerned, I am sure that this powerful and (above all) highly versatile system will further develop in the future. Even if it is already today a reliable companion for creating and / or verifying formal proofs in a variety of fields, there is - of course - always something that could be further optimized one day.

From the user point of view (and this is so far the only position I can represent) there might be cases, where one would appreciate more detailed explanations for a failed proof attempt. The proof process would get even easier (almost luxurious) if there was in advance - depending on the current step - a selection of applicable rules and the like. On the other hand, one could argue that this would reduce the (desired) human part of the collaboration too strongly. Improvements concerning the automatic tools that are part of the Isabelle systems are probably always possible, but since this involves too many technical details that I cannot honestly judge, I will abstain from any inappropriate recommendations.

## APPENDIX

## A. THE PROOFS IN DETAIL

The interested reader may now look up the proofs of all lemmata and the theorem in the file `dsa.thy` presented in section A.3. What now follows is a short explanation of the rules and techniques that will appear there. In section A.2 the proof of an arbitrarily chosen lemma is presented in detail, including the pending subgoals in each step.

My theory file `dsa.thy` uses several lemmata contained in the theories of natural numbers, the divides relation and ordered groups in the library, that is `HOL/Nat.thy`, `HOL/Divides.thy` and `HOL/OrderedGroup.thy`, respectively. In addition, my axiomatic typeclasses are inheritors of the monoid `_mult` typeclass located in the theory `HOL/OrderedGroup.thy`. This theory already contains some definitions of (subclasses of) groups, but since these are always special cases like Abelian groups or groups with some notion of ordering I decided to define my own axiomatic typeclass of “plain” groups comprising only the skeleton axioms.

### A.1 *The most frequent tools*

I will briefly explain a few important Isabelle rules, commands and tools that often occur in the theory, and the way they manipulate the proof state. A short introduction to Isabelle can be found in chapter 4. Again, for more detailed explanations, please refer to [16], for instance.

- *sym* and *trans*: These rules simply express the symmetry and the transitivity of the “=”-relation.
- *subst*: Re-writes expressions in a subgoal according to a certain rule, like the commutativity law, for instance. Sometimes this is followed by the *back* command telling Isabelle to try out another substitution alternative in case we are not satisfied with the first result.
- *assumption*: Proves a subgoal that is already among the assumptions we have at our disposal. Thus, the subgoal is done.

- *rule\_tac*: As already explained in chapter 4, this command allows us to apply a rule obeying certain constraints. For instance, (?-“quantified”) variables in the rule can be instantiated by a given term.
- *subgoal\_tac*: This command inserts a further subgoal  $P$ . That means, on the one hand, we have one more subgoal to prove, namely  $P$ , on the other hand, the remaining subgoals can now be proven with  $P$  as a further assumption. In my proofs, this command is often followed by *defer* which simply changes the order of the subgoals I want to prove next.
- *simp* and *blast*. The simplifier has already been mentioned. The latter is a tool mainly designed for automatic proving in the field of predicate logic.

I have to admit that quite often I made use of the bulky *subgoal\_tac* method that inserts a subgoal into the set of premises which then needs to be proven itself. I found it useful in order to “guide” Isabelle into the right direction. For instance, one often has quite a precise idea of the forward direction of a proof. On the other hand, typical Isabelle reasoning works backwards. For a theorem  $P_0 \Rightarrow P_n$ , let  $P_0 \Rightarrow P_1 \dots \Rightarrow P_n$  be the general course of a (forward) proof, where  $P_i \Rightarrow P_{i+1}$  are “atomic” steps that can be handled by applying a single rule in Isabelle. After starting the proof process you are confronted with the (only) subgoal  $[P_0] \Rightarrow P_n$ . Inserting  $P_{n-1}$  using *subgoal\_tac* leads to the subgoals  $[P_0, P_{n-1}] \Rightarrow P_n$  and  $[P_0] \Rightarrow P_{n-1}$ . The first subgoal can be settled quickly as we have said, whereas the second one is at least a bit easier than the original subgoal, since  $P_{n-1}$  is “closer” to  $P_0$  than  $P_n$ .

Of course, in simple cases Isabelle would be able to automatically generate the subgoal  $P_{n-1}$  as soon as we try to apply a rule where the conclusion corresponds to  $P_n$  and one of its premises can be unified with  $P_{n-1}$ . However, in more complex situations such a proof attempt fails until the system is explicitly equipped with  $P_{n-1}$  as a premise in advance using the above methodology.

## A.2 A closer look

To demonstrate what a typical proof looks like “from inside” I will briefly outline how the proof state evolves in the proof of lemma *pow\_mod4*. Most

of the other proofs in my theory are not too different in principle, so I chose this average instance as a representative for the others.

- **Step 0:**

- **Command:** `lemma pow_mod4:`  

$$u * q = n \wedge g = \text{pow } h \ u \wedge k \bmod q = l \bmod q$$

$$\implies \text{pow } g \ k = \text{pow } (g::'a::\text{finite\_cyclic\_group\_mult}) \ l$$
- **Explanation:** the original lemma is declared
- **Resulting subgoal(s):**  

$$* \ u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q$$

$$\implies \text{pow } g \ k = \text{pow } g \ l$$

- **Step 1:**

- **Command:** `apply (subgoal_tac "pow g k = pow g (k mod q)")`
- **Explanation:** Inserts the above subgoal as a further assumption
- **Resulting subgoal(s):**  

$$* \ u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q,$$

$$\text{pow } g \ k = \text{pow } g \ (k \bmod q)$$

$$\implies \text{pow } g \ k = \text{pow } g \ l$$

$$* \ u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q$$

$$\implies \text{pow } g \ k = \text{pow } g \ (k \bmod q)$$

- **Step 2:**

- **Command:** `defer`
- **Explanation:** Switches the subgoals' order
- **Resulting subgoal(s):**  

$$* \ u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q$$

$$\implies \text{pow } g \ k = \text{pow } g \ (k \bmod q)$$

$$* \ u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q,$$

$$\text{pow } g \ k = \text{pow } g \ (k \bmod q)$$

$$\implies \text{pow } g \ k = \text{pow } g \ l$$

- **Step 3:**

- **Command:** `apply (blast intro: pow_mod3)`

– **Explanation:** Proves the first subgoal using blast equipped with former lemma

– **Resulting subgoal(s):**

$$\begin{aligned} & * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\ & \text{pow } g \ k = \text{pow } g \ (k \bmod q) \\ & \implies \text{pow } g \ k = \text{pow } g \ l \end{aligned}$$

• **Step 4:**

– **Command:** *apply (subgoal\_tac "pow g l = pow g (l mod q)")*

– **Explanation:** Inserts a further subgoal

– **Resulting subgoal(s):**

$$\begin{aligned} & * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\ & \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\ & \implies \text{pow } g \ k = \text{pow } g \ l \\ & * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\ & \text{pow } g \ k = \text{pow } g \ (k \bmod q) \\ & \implies \text{pow } g \ l = \text{pow } g \ (l \bmod q) \end{aligned}$$

• **Step 5:**

– **Command:** *defer*

– **Explanation:** Switches the subgoals' order

– **Resulting subgoal(s):**

$$\begin{aligned} & * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \text{pow } g \ k = \text{pow } \\ & \text{pow } g \ (k \bmod q) \\ & \implies \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\ & * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \text{pow } g \ k = \text{pow } \\ & \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\ & \implies \text{pow } g \ k = \text{pow } g \ l \end{aligned}$$

• **Step 6:**

– **Command:** *apply (blast intro: pow\_mod3)*

– **Explanation:** Proves the first subgoal using blast equipped with former lemma

– **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ k = \text{pow } g \ l
\end{aligned}$$

- **Step 7:**

- **Command:** *apply (subgoal\_tac "pow g (k mod q) = pow g (l mod q)")*

- **Explanation:** Introduces above subgoal

- **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ k = \text{pow } g \ l
\end{aligned}$$

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q)
\end{aligned}$$

- **Step 8:**

- **Command:** *defer*

- **Explanation:** Switching the subgoals..

- **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \text{pow } g \ k = \text{pow } \\
& \text{g } (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q)
\end{aligned}$$

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \text{pow } g \ k = \text{pow } \\
& \text{g } (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \text{pow } g \ (k \bmod q) = \\
& \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ k = \text{pow } g \ l
\end{aligned}$$

- **Step 9:**

- **Command:** *apply (simp)*

- **Explanation:** Simplifier solves the first subgoal easily, since  $k \bmod q = l \bmod q$  is among the assumptions

- **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q), \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ k = \text{pow } g \ l
\end{aligned}$$

- **Step 10:**

- **Command:** *apply (subgoal\_tac “pow g k = pow g (l mod q)”)*

- **Explanation:** Inserts above subgoal;  
“plan”: show finally  $\text{pow } g \ k = \text{pow } g \ l$  using transitivity / symmetry of “=” and the assumptions we already have, i.e. remaining steps are logically trivial

- **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q), \text{pow } g \ k = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ k = \text{pow } g \ l
\end{aligned}$$

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ k = \text{pow } g \ (l \bmod q)
\end{aligned}$$

- **Step 11:**

- **Command:** *defer*

- **Explanation:** No longer a secret..

- **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ k = \text{pow } g \ (l \bmod q)
\end{aligned}$$

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q), \text{pow } g \ k = \text{pow } g \ (l \bmod q) \\
& \implies \text{pow } g \ k = \text{pow } g \ l
\end{aligned}$$

- **Step 12:**

- **Command:** *apply (rule\_tac s = “pow g (k mod q)” in trans)*
- **Explanation:** Applies transitivity rule  $r = s \wedge s = t \implies r = t$  unifying  $s$  with  $\text{pow } g \text{ (k mod q)}$  in order to solve first subgoal; of course, these steps could have been handled more quickly
- **Resulting subgoal(s):**
  - \*  $u * q = n, g = \text{pow } h \text{ u, k mod q} = \text{l mod q},$   
 $\text{pow } g \text{ k} = \text{pow } g \text{ (k mod q)}, \text{pow } g \text{ l} = \text{pow } g \text{ (l mod q)}$   
 $\text{pow } g \text{ (k mod q)} = \text{pow } g \text{ (l mod q)}$   
 $\implies \text{pow } g \text{ k} = \text{pow } g \text{ (k mod q)}$
  - \*  $u * q = n, g = \text{pow } h \text{ u, k mod q} = \text{l mod q},$   
 $\text{pow } g \text{ k} = \text{pow } g \text{ (k mod q)}, \text{pow } g \text{ l} = \text{pow } g \text{ (l mod q)}$   
 $\text{pow } g \text{ (k mod q)} = \text{pow } g \text{ (l mod q)}$   
 $\implies \text{pow } g \text{ (k mod q)} = \text{pow } g \text{ (l mod q)}$
  - \*  $u * q = n, g = \text{pow } h \text{ u, k mod q} = \text{l mod q},$   
 $\text{pow } g \text{ k} = \text{pow } g \text{ (k mod q)}, \text{pow } g \text{ l} = \text{pow } g \text{ (l mod q)}$   
 $\text{pow } g \text{ (k mod q)} = \text{pow } g \text{ (l mod q)}, \text{pow } g \text{ k} = \text{pow } g \text{ (l mod q)}$   
 $\implies \text{pow } g \text{ k} = \text{pow } g \text{ l}$

- **Step 13:**

- **Command:** *apply (simp)*
- **Explanation :** Solves the first subgoal (by assumption)
- **Resulting subgoal(s):**
  - \*  $u * q = n, g = \text{pow } h \text{ u, k mod q} = \text{l mod q},$   
 $\text{pow } g \text{ k} = \text{pow } g \text{ (k mod q)}, \text{pow } g \text{ l} = \text{pow } g \text{ (l mod q)}$   
 $\text{pow } g \text{ (k mod q)} = \text{pow } g \text{ (l mod q)}$   
 $\implies \text{pow } g \text{ (k mod q)} = \text{pow } g \text{ (l mod q)}$
  - \*  $u * q = n, g = \text{pow } h \text{ u, k mod q} = \text{l mod q},$   
 $\text{pow } g \text{ k} = \text{pow } g \text{ (k mod q)}, \text{pow } g \text{ l} = \text{pow } g \text{ (l mod q)}$   
 $\text{pow } g \text{ (k mod q)} = \text{pow } g \text{ (l mod q)}, \text{pow } g \text{ k} = \text{pow } g \text{ (l mod q)}$   
 $\implies \text{pow } g \text{ k} = \text{pow } g \text{ l}$

- **Step 14:**

- **Command:** *apply (simp)*
- **Explanation:** Solves the first (remaining) subgoal as above

– **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q), \text{pow } g \ k = \text{pow } g \ (l \bmod \\
& q) \\
& \implies \text{pow } g \ k = \text{pow } g \ l
\end{aligned}$$

• **Step 15:**– **Command:** *apply (rule\_tac s = "pow g (l mod q)" in trans)*– **Explanation:** Another element in the transitivity chain..– **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q), \text{pow } g \ k = \text{pow } g \ (l \bmod \\
& q) \\
& \implies \text{pow } g \ k = \text{pow } g \ (l \bmod q) \\
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q), \text{pow } g \ k = \text{pow } g \ (l \bmod \\
& q) \\
& \implies \text{pow } g \ (l \bmod q) = \text{pow } g \ l
\end{aligned}$$

• **Step 16:**– **Command:** *apply (simp)*– **Explanation:** Solves again first subgoal by assumption– **Resulting subgoal(s):**

$$\begin{aligned}
& * u * q = n, g = \text{pow } h \ u, k \bmod q = l \bmod q, \\
& \text{pow } g \ k = \text{pow } g \ (k \bmod q), \text{pow } g \ l = \text{pow } g \ (l \bmod q) \\
& \text{pow } g \ (k \bmod q) = \text{pow } g \ (l \bmod q), \text{pow } g \ k = \text{pow } g \ (l \bmod \\
& q) \\
& \implies \text{pow } g \ (l \bmod q) = \text{pow } g \ l
\end{aligned}$$

• **Step 17:**– **Command:** *apply (simp)*– **Explanation:** The remaining one is also easily solved

- **Resulting subgoal(s):**  $\emptyset$
- **Step 18:** (not an actual proof step)
  - **Command:** *done*
  - **Explanation:** Inserts proven lemmata into knowledge base
  - **Resulting subgoal(s):**  $\emptyset$

A.3 The file *dsa.thy*

```

(* Title:  dsa.thy
   Author: Sebastian Kusch, Darmstadt University of Technology
*)

theory dsa imports Main begin

(* definitions of primitive root, group cardinality and
   functions *)

consts

h:: "'a"
n  :: "nat"
inv:: "'a => 'a"
pow:: "[ 'a, nat ] => 'a"

(* definitions of axiomatic type classes *)

axclass group_mult \<subseteq> monoid_mult
left_inverse: "inv a * a = 1"
pow_def_0: "pow a 0 = 1"
pow_def_suc: "pow a (Suc k) = (pow a k) * a"

axclass cyclic_group_mult \<subseteq> group_mult
generator: "\<exists>a. pow h a = A"

axclass finite_cyclic_group_mult \<subseteq> cyclic_group_mult
finite: "pow h n = 1"

(* first lemmata, dealing with the associativity of the pow
   function *)

lemma pow_assoc0: "pow a k * (pow a l * a) = pow a k * pow a l
* (a::'a::semigroup_mult)"
apply (simp add: OrderedGroup.mult_assoc)
done

lemma pow_assoc1: "pow a k * pow a l =

```

```

pow (a::'a::group_mult) (k+1)"
apply (induct_tac 1)
apply (simp add: pow_def_0)
apply (simp add: pow_def_suc)
apply (simp add: pow_assoc0)
done

lemma h1: "pow a (k + k*1) = pow a k * pow (a::'a::group_mult)
(k*1)"
apply (simp add: pow_assoc1)
done

lemma h2: "pow (pow a k) (Suc 1) = pow (pow a k) (Suc 0) * pow
(pow (a::'a::group_mult) k) 1"
apply (simp add: pow_assoc1 )
done

lemma h3: "pow a (Suc 0) = (a::'a::group_mult)"
apply (simp add: pow_def_suc)
apply (simp add: pow_def_0)
done

lemma h4: "pow (pow a k) (Suc 1) = pow a k * pow (pow (a::'a::
group_mult) k) 1"
apply (subgoal_tac "pow (pow a k) (Suc 1) = pow (pow a k)
(Suc 0) * pow (pow a k) 1")
defer
apply (rule h2)
apply (rule_tac s = "pow (pow a k) (Suc 0) * pow (pow a k) 1"
in trans)
apply (assumption)
apply (subst h3)
apply (simp)
done

lemma pow_assoc2: "pow (pow a k) 1 = pow (a::'a::group_mult)
(k*1)"
apply (induct_tac 1)
apply (simp add: pow_def_0)
apply (rule sym)
apply (simp add: add_mult_distrib2)

```

```

apply (simp add: h1)
apply (rule sym)
apply (simp add: h4)
done

```

```

lemma h5: "pow 1 k = (1::'a::group_mult)"
apply (induct_tac k)
apply (simp add: pow_def_0)
apply (simp add: pow_def_suc)
done

```

(\* the following lemma states that  $\text{order}(g)$  divides  $n$  \*)

```

lemma subgroup_generator: "u * q = n & g = pow h u
\<Longrightrightarrow> pow (g::'a::finite_cyclic_group_mult) q = 1"
apply (simp)
apply (simp add: pow_assoc2)
apply (simp add: finite)
done

```

```

lemma h6: "pow (g::'a::group_mult) (k*q) = pow (pow g q)k"
apply (rule sym)
apply (simp add: pow_assoc2)
apply (subst nat_mult_commute)
apply (simp)
done

```

```

lemma h7: "u * q = n & g = pow h u \<Longrightrightarrow>
pow (g::'a::finite_cyclic_group_mult) (k*q) = 1"
apply (subst h6)
apply (subgoal_tac "pow g q = 1")
defer
apply (rule subgroup_generator)
apply (blast)
apply (simp)
apply (simp add: h5)
done

```

(\* approaching the finding that for exponentiations with  $g$  as basis the exponent may be reduced mod  $q$  \*)

```

lemma pow_mod1: "u * q = n & g = pow h u \<Longrightarrow>
pow g (r + k*q) = pow (g::'a::finite_cyclic_group_mult) r"
apply (subgoal_tac "pow g (r + k*q) = pow g r * pow g (k*q)")
defer
apply (rule sym)
apply (rule pow_assoc1)
apply (rule_tac s = "pow g r * pow g (k*q)" in trans)
apply (assumption)
apply (subgoal_tac "pow g (k*q) = 1")
defer
apply (rule h7)
apply (blast)
apply (simp)
done

```

```

lemma pow_mod2: "u * q = n & g = pow h u & (\<exists>l.(k =
r + l*q))
\<Longrightarrow>
pow g k = pow (g::'a::finite_cyclic_group_mult) r"
apply (blast intro: pow_mod1)
done

```

```

lemma pow_mod3: "u * q = n & g = pow h u \<Longrightarrow>
pow g k = pow (g::'a::finite_cyclic_group_mult) (k mod q)"
apply (subgoal_tac "\<exists>l.(k = (k mod q) + l*q)")
defer
apply (rule mod_eqD)
apply (simp)
apply (rule pow_mod2)
apply (blast)
done

```

```

lemma pow_mod4: "u * q = n & g = pow h u & k mod q = 1 mod q
\<Longrightarrow>
pow g k = pow (g::'a::finite_cyclic_group_mult) 1"
apply (subgoal_tac "pow g k = pow g (k mod q)")
defer
apply (blast intro: pow_mod3)
apply (subgoal_tac "pow g 1 = pow g (1 mod q)")
defer
apply (blast intro: pow_mod3)

```

```

apply (subgoal_tac "pow g (k mod q) = pow g (l mod q)")
defer
apply (simp)
apply (subgoal_tac "pow g k = pow g (l mod q)")
defer
apply (rule_tac s = "pow g (k mod q)" in trans)
apply (simp)
apply (simp)
apply (rule_tac s = "pow g (l mod q)" in trans)
apply (simp)
apply (simp)
done

```

```

lemma h8: "a = b \<Longrightarrow> a mod q = b mod (q::nat)"
apply (simp)
done

```

```

lemma h9: "a * b mod q = b * a mod (q::nat)"
apply (simp add: mult_commute)
done

```

```

lemma h10: "a * ainv mod q = 1 \<Longrightarrow>
(a * ainv mod q) * c = (c::nat)"
apply (simp)
done

```

(\* the following lemma is needed to re-write the defining equation for s such that the desired congruency for k is obtained.. \*)

```

lemma inv_exp: "a = binv * c mod q & a * ainv mod q = 1 & b *
binv mod q = 1
\<Longrightarrow> b mod q = ainv * c mod (q::nat)"
apply (subgoal_tac "a mod q = binv * c mod q")
defer
apply (simp)
apply (subgoal_tac "b * (a mod q) = b * (binv * c mod q)")
defer
apply (simp)
apply (subgoal_tac "b * (a mod q) mod q = b * (binv * c mod q)
mod q")

```

```

defer
apply (simp)
apply (subgoal_tac "b * a mod q = b * (a mod q) mod q")
defer
apply (subst mod_mult1_eq)
apply (simp)
apply (subgoal_tac "b * (binv * c) mod q = b * (binv * c mod q)
mod q")
defer
apply (subst mod_mult1_eq)
apply (simp)
apply (subgoal_tac "b * a mod q = b * (binv * c mod q) mod q")
defer
apply (rule_tac s = "b * (a mod q) mod q" in trans)
apply (assumption)
apply (assumption)
apply (subgoal_tac "b * a mod q = b * binv * c mod q")
defer
apply (rule_tac s = "b * (binv * c mod q) mod q" in trans)
apply (assumption)
apply (rule sym)
apply (subgoal_tac "b * binv * c = b * (binv * c)")
defer
apply (simp)
defer
apply (subgoal_tac "b * binv * c mod q = b * (binv * c) mod q")
defer
apply (rule_tac a = "b * binv * c" and b = "b * (binv * c)"
in h8)
apply (assumption)
defer
apply (rule_tac s = "b * (binv * c) mod q" in trans)
apply (assumption)
apply (assumption)
apply (subgoal_tac "b * binv * c mod q = b * binv mod q * c
mod q")
defer
apply (subst mod_mult1_eq')
apply (simp)
apply (subgoal_tac "a * b mod q = b * binv * c mod q")
defer

```

```

apply (rule_tac s = "b * a mod q" in trans)
apply (rule_tac a = a and b = b in h9)
apply (assumption)
apply (subgoal_tac "b * binv * c mod q = c mod q")
defer
apply (simp)
apply (subgoal_tac "a * b mod q = c mod q")
defer
apply (rule_tac s = "b * binv * c mod q" in trans)
apply (assumption)
apply (assumption)
apply (subgoal_tac "ainv * (a * b mod q) = ainv * (c mod q)")
defer
apply (simp)
apply (subgoal_tac "ainv * (a * b mod q) mod q = ainv *
(c mod q) mod q")
defer
apply (simp)
apply (subgoal_tac "a * (ainv * b mod q) mod q = a * (ainv * b)
mod q")
defer
apply (subst mod_mult1_eq)
back
back
apply (simp)
apply (subgoal_tac "ainv * (a * b mod q) mod q = ainv *
(c mod q) mod q")
defer
apply (simp)
apply (subgoal_tac "ainv * (a * b mod q) mod q = ainv *
(a * b) mod q")
defer
apply (subst mod_mult1_eq)
back
back
apply (simp)
apply (subgoal_tac "ainv * (c mod q) mod q = ainv * c mod q")
defer
apply (subst mod_mult1_eq)
back
apply (simp)

```

```

apply (subgoal_tac "ainv * (a * b) mod q = ainv * a * b mod q")
defer
apply (simp add: nat_mult_assoc)
apply (subgoal_tac "ainv * a * b mod q = (ainv * a mod q) * b
mod q")
defer
apply (subst mod_mult1_eq')
apply (simp)
apply (subgoal_tac "(ainv * a mod q) * b mod q = (a * ainv
mod q) * b mod q")
defer
apply (simp add: mult_commute)
apply (subgoal_tac "(a * ainv mod q) * b mod q = b mod q")
defer
apply (rule h8)
apply (rule h10)
apply (blast)
apply (subgoal_tac "b mod q = ainv * a mod q * b mod q")
defer
apply (rule_tac s = "a * ainv mod q * b mod q" in trans)
apply (rule sym)
apply (assumption)
apply (rule sym)
apply (assumption)
apply (subgoal_tac "b mod q = ainv * a * b mod q")
defer
apply (rule_tac s = "ainv * a mod q * b mod q" in trans)
apply (assumption)
apply (rule sym)
apply (assumption)
apply (subgoal_tac "b mod q = ainv * (a * b) mod q")
defer
apply (rule_tac s = "ainv * a * b mod q" in trans)
apply (assumption)
apply (rule sym)
apply (assumption)
apply (subgoal_tac "b mod q = ainv * (a * b mod q) mod q")
defer
apply (rule_tac s = "ainv * (a * b) mod q" in trans)
apply (assumption)
apply (rule sym)

```

```

apply (assumption)
apply (subgoal_tac "b mod q = ainv * (c mod q) mod q")
defer
apply (rule_tac s = "ainv * (a * b mod q) mod q" in trans)
apply (assumption)
apply (assumption)
apply (rule_tac s = "ainv * (c mod q) mod q" in trans)
apply (assumption)
apply (assumption)
done

(* ..yielding finally the verification theorem
   note: H denotes the hash value of our document,
         h is a primitive root *)

theorem verify: "u * q = n & pow h u = g & y = pow g x
& s = kinv * (H + x*r) mod q & s * sinv mod q = 1 & k * kinv mod q = 1
\<Longrightarrow>
pow g (sinv * H mod q) * pow y (r * sinv mod (q::nat))
= pow (g::'a::finite_cyclic_group_mult) k"
apply (subgoal_tac "sinv * (h + x*r) mod q = k mod q")
defer
apply (rule sym)
apply (rule_tac a = s and binv = kinv and c = "h + x*r" in
inv_exp)
apply (blast)
apply (subgoal_tac "pow g (sinv * h mod q) * pow y (r * sinv
mod q)
= pow g (sinv * h) * pow y (r * sinv mod q)")
defer
apply (subgoal_tac "pow g (sinv * h mod q) = pow g
(sinv * h)")
defer
apply (rule sym)
apply (rule pow_mod3)
apply (blast)
defer
apply (simp)
apply (rule_tac s = "pow g (sinv * h) * pow y (r * sinv mod q)"
in trans)
apply (assumption)

```

```

apply (simp)
apply (simp add: pow_assoc2)
apply (subgoal_tac "pow g (x * (r * sinv mod q)) = pow g ((r *
sinv mod q) * x)")
defer
apply (simp add: nat_mult_commute)
apply (subgoal_tac "pow g ((r * sinv mod q) * x) = pow (pow g
(r * sinv mod q)) x")
defer
apply (rule sym)
apply (rule pow_assoc2)
apply (subgoal_tac "pow (pow g (r * sinv mod q)) x = pow (pow g
(r * sinv)) x")
defer
apply (rule sym)
apply (subgoal_tac "pow g (r * sinv) = pow g (r * sinv mod q)")
defer
apply (rule pow_mod3)
apply (blast)
defer
apply (simp)
apply (simp)
apply (simp add: pow_assoc2)
apply (simp add: pow_assoc1)
apply (subgoal_tac "(sinv*h + r*sinv*x) mod q = sinv *
(h + x*r) mod q")
apply (subgoal_tac "(sinv*h + r*sinv*x) mod q = k mod q")
defer
apply (rule_tac s = "sinv * (h + x*r) mod q" in trans)
apply (assumption)
apply (simp)
defer
apply (rule_tac u = u and q = q in pow_mod4)
apply (blast)
apply (rule_tac a = "sinv*h + r*sinv*x" and b = "sinv *
(h + x*r)" in h8)
apply (simp add: nat_distrib)
done

```

## BIBLIOGRAPHY

- [1] Giampaolo Bella, Fabio Massacci, Lawrence C. Paulson, and Piero Tramontano, *Formal verification of cardholder registration in SET*, In F. Cuppens, Y. Deswarte, D. Gollman, and M. Waidner, editors, *Computer Security — ESORICS 2000*, LNCS 1895, pages 159–174. Springer, 2000
- [2] Theodor Bröcker, *Analysis 1*, 2. Auflage, , pp 50-51, Spektrum Akademischer Verlag, Heidelberg
- [3] Lukas Bruegger, *Seminar on Current Topics in Information Security*, ETH Zuerich, December 5, 2005
- [4] Johannes Buchmann, *Einführung in die Kryptographie*, 2., erweiterte Auflage, Springer Verlag, Heidelberg, 2001
- [5] Burrows, M., Abadi, M., and Needham, R. M., *A Logic of Authentication*, *ACM Transactions on Computing Systems* 8(1) pp. 18-36 (Feb. 1990)
- [6] H.-D. Ebbinghaus, J. Flum, W. Thomas, *Einführung in die mathematische Logik*, Spektrum Akademischer Verlag, 1996
- [7] Claudia Eckert, *IT-Sicherheit*, 2. Auflage, Oldenburg Wissenschaftsverlag, 2003
- [8] Taher ElGamal, *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, *IEEE Transactions on Information Theory*, v. IT-31, n. 4, 1985, pp. 469-472 or *CRYPTO 84*, pp. 10-18, Springer-Verlag
- [9] FIPS-PUB 186-2, *Digital Signature Standard (DSS)*, U.S. Department of Commerce / National Institute of Standards and Technology, 2000 January 27
- [10] Fraenkel, Abraham A., Yehoshua Bar-Hillel and Azriel Levy *Foundations of Set Theory*, Amsterdam, North-Holland Publishing, 1973(1958)

- 
- [11] K. Gödel *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*, I. Monatshefte für Mathematik und Physik 38 (1931)
- [12] Henkin, Leon, *The Completeness of the First-Order Functional Calculus*, Journal of Symbolic Logic. 14: 159-166, 1949
- [13] M. Hohmuth and H. Tews and S. Stephens, *Applying source-code verification to a microkernel — the VFiasco project*, Technical Report TUD-FI02-03-Marz
- [14] Mastercard & VISA, *SET secure Electronic Transaction Specification: Formal Protocol Definition*, May 1997, available at [http://www.setco.org/set\\_specifications.html](http://www.setco.org/set_specifications.html)
- [15] Dale Miller, *A short article for the Encyclopedia of Artificial Intelligence: Second Edition “Logic, Higher-order”*, February 1991, available at <http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/AIencyclopedia.pdf>
- [16] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel, *A Proof Assistant for Higher-Order-Logic*, Springer Verlag, Heidelberg, 2005, also available at <http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html>
- [17] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel, *Isabelle’s Logics: HOL*, available at <http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/logics-HOL.pdf>
- [18] Tobias Nipkow, *Structured Proofs in Isar* available at <http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/isar-overview.pdf>
- [19] Lawrence C. Paulson, *A Simple Formalization and Proof for the Mutilated Chess Board*, Technical Report 394, Comp. Lab., Univ. Camb., 1996.
- [20] Lawrence C. Paulson, *Isabelle’s Logics*, available at <http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/logics.pdf>
- [21] L. C. Paulson, *The inductive approach to verifying cryptographic protocols*, Journal of Computer Security 6 (1998), pp. 85-128
- [22] Lawrence C. Paulson, *The Isabelle Reference Manual*, available at <http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/ref.pdf>

- 
- [23] L. C. Paulson, *Verifying the SET protocol: overview*, Invited lecture, FASec 2002: Formal Aspects of Security. Royal Holloway College, University of London, England, December 2002
- [24] Tom Ridge, *A Mechanically Verified, Efficient, Sound and Complete Theorem Prover for First Order Logic*, February 2005, available at <http://www.cl.cam.ac.uk/~tjr22/doc/prover.pdf>
- [25] R. Rivest, A. Shamir, L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, Vol. 21 (2), pp. 120-126. 1978
- [26] Markus Wenzel, *Using Axiomatic Type Classes in Isabelle, a tutorial* 1995, available at <http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/Isabelle/doc/axclass.pdf>
- [27] <http://www.cl.cam.ac.uk/research/hvg/Isabelle/>