Formal Proofs in Isabelle/HOL

Research Seminar Cryptography Spring 2018

Sara Chiers

Concept

Proof assistants can be of great help to cryptographers since they can check whether a mathematical proof is correct. Of course their use is by no means limited to cryptographers. They are of great use to cryptographers though, since cryptographic proofs are often quite error prone and the real world consequences could easily be more severe than those of let us say a mistake in the classification theorem for finite simple groups. Parts of the latter were computer checked in 2012 as well though, using another proof assistant called Coq. 30 years and several mistakes after the first proof was believed to be complete.
After sharing a first version of this project, the suggestion came to expand a little more on proof assistants in general. Like what alternatives there are and how they compare, I thought it would be interesting to add indeed. However, since my experience with proof assistents is limmited to Isabelle, I could not make a good comparison, so I browsed some litterature. After noticing the amount of writing on the topic, and especially upon finding the (by now slightly dated) book titled "The Seventeen Provers of the World" [1] and the over 300 names listed in [2], I concluded however, that there are many many things to say, but I am not knowledgeable enough to quickly give a good summary.
If I were to give my impression though, after working with Isabelle and learning about it during this project, I would say the other proof assistant that kept popping up was Coq. Coq is built quite differently, since it was originally built for checking programs more than for mathematical proofs. Coq is often liked for its customizability, but you will likely not find anyone claiming what proof assistant is better or best though since it is hard to compare any of them. They are designed in different ways, and depending on your use for them, different features will be more or less important to you. Isabelle, for example, scores favors with Isar. This way of interacting with Isabelle starts to bridge the gap between textbook, human readable, proofs and the typical machine readable code. This is a quite rare feature since most proof assistants result in a script of commands to be read interactively and hence are nothing like a readable proof text.
Rather than discuss the pros and cons though, the aim of this project is to show, using an example, a little bit about how Isabelle functions. More specifically we will be working with Isabelle/HOL, the instance of Isabelle equiped for higher order logic. To illustrate how one can use Isabelle, we will be proving the correctness of the ElGamal encryption scheme.

1 El Gamal Encryption System

Before jumping into the Isabelle language, a quick recap of the El Gamal encryption scheme.
For ElGamal, in the key generation step, we fix a finite cyclic group G, a generator $g \in G$ and an integer $x \in \{0, \ldots, |G| - 1\}$ at random. Finally we compute $h = g^x$ to have our public and private keys: $\mathrm{pk} = (g, h)$ and $\mathrm{sk} = (g, x)$.
Then, to actually encrypt a message $m \in G$ we pick another integer $r \in \{0, \ldots, |G| - 1\}$ at random and compute the ciphertext as follows: $c = (c_1||c_2) = (g^r||m \cdot h^r)$.
Decryption on the other hand can be done by computing $c_2 \cdot (c_1^x)^{-1}$.
Correctness of an encryption scheme means that $\mathrm{Dec}(\mathrm{Enc}(m)) = m$ for any message m. So in this case, we want to prove:

Theorem 1. *For any finite cyclic group G, any generator $g \in G$, any integers $x, r \leq |G| - 1$ and any group element $m \in G$ holds $Dec((g, x), Enc((g, g^x), m) = m$ where* Dec *and* Enc *are as described above.*

Proof. To prove this, we start by implementing all these definitions which gives us the new goal of proving that $Dec((g, x), (g^r, m \cdot (g^x)^r) = m$ or still $(m \cdot (g^x)^r) \cdot ((g^r)^x)^{-1} = m$
Next, we can use some of the things we know about group multiplication and exponentiation to see that indeed the two powers of g cancel out on the left side and we are left with $m = m$ which is of course true for any m. □

If we want to be really critical though, we can easily point out some things that are left implicit. For example, our non-formal definition of Dec and Enc or the assumption that the usual rules for exponentiation hold in arbitrary groups ...
This happens in almost any mathematical proof for ease of reading, and few people will argue this is inherently bad. However, it does leave room for error, especially since the more complicated proofs get the more of these handwavy steps people tend to use (and the less likely readers are to double check them).

2 Defining things

So, let us take a look at how using a proof assistant helps us to be more explicit and not make accidental mistakes. For starters, no computer program takes ambiguous definitions, and the Isabelle framework is no different.

2.1 Theories

Since all Isabelle text must be in a theory, we start by creating a file ElGamal.thy and start a theory with the same name, as shown below. Theories work a lot like modules in programming languages. They contain a collection of definitions, types, functions and theorems and they can build on each other using import statements. Here we import only the theory `Main`, which is a theory that mainly exists to group the basic theories one will often need. Main contains basic types like `set, bool, list, nat, int ...` just like any programming language, however, since this is a tool for mathematical proving, there are also some basic algebraic structures already implemented like fields, functions, relations, groups, lattices ... All of these come with a theory full of related definitions, theorems ... We will touch on how to use these theorems in a bit, but to get a quick look at what is already there, there is a great summary "What is in main" in the documentation that comes with Isabelle.

```
theory ElGamal
imports Main
begin
```

Okay, so now we have started our theory, how do we define the necessary things?

2.2 Type system

Like many programming language, the Isabelle framework has a type system. As in programming languages this serves to 'check if operations make sense'. Hereby we mean things like checking we apply a function (say addition) only to things for which this function is defined (say to two numbers, not to two lists).
In Isabelle there are base types like `bool, nat` and `int`. From these we can make new types using type constructors like `list` and `set`. We can also make function types from them using `=>`, we could for example consider things of the type `(int => nat)`. A concrete example of this would be the absolute value function. Lastly, instead of using concrete types in these constructions we can also use type variables denoted by `'name`. One example where this comes in handy is if we want to define a operation on a list, no matter the type of its content, say we want to take its first element.
Isabelle requires that all formulae are well-typed, however, this does not mean we have to explicitly type everything. In most cases the type inference takes care of it, however, in case of overloaded operators an explicit type annotation like `2::nat` might be needed.

In practise type errors may also show up due to precedence rules ... for example: `'a=>'b list` which is interpreted as `'a => ('b list)` or a function mapping a thing of type `a` to a list of things of type `b`, has a different type then `('a=>'b) list` which is a list of functions that go from type `a` objects to type `b` objects.

Defining new types

type_synonym Things like `type_synonym string = "char list"` are basically a short hand notation for writing a script or Isar document. They are not actually types since Isabelle just expands them when parsing, so type synonyms will not be in any output or internal representation.

datatype Things like

```
datatype color = blue | yellow | red | green |orange | purple
```

or

```
datatype ('a, 'b,'c) triple = Triple 'a 'b 'c
```

or

```
datatype 'a tree =Tip | Node "'a tree" 'a "'a tree"
```

however do define proper datatypes (and datatype constructors) which are used by Isabelle internally and will also appear in output. Defining new datatypes can be as simple as giving a list of possible values or pasting together existing datatypes. More advanced options are available though, for example the definition of tree here is recursive. You have quite extensive options to define datatypes, the formal requirement for this construction is just that all of the possible constructors (separated by `|`) are disjunct and each of them should also be injective. These two requirements guarantee that each instance of this type will have been built in a unique way.

typedecl One more option would be to make a new type without defining it right away, using something like `typedecl elephant` albeit preferably with a more reasonable name. This allows you to add requirements for this type later, giving you some more freedom.

locale and class Another more intricate way to define structures might be using `locale` and `class`. The precise working of these two constructions are outside of the scope of a quick introduction, however, you might see them whilst wandering around `Main` quite a lot. They seem the preferred method to define algebraic structures since they allow you to easily add new axioms to an existing structure to create a new one. Another advantage of these constructs, compared to defining an empty datatype and giving axioms explicitly, is that any possible inconsistencies stay within this subject. Say you have two axioms that hold in the locale "zet" like all elements of a "zet" are equal, and there exist two different elements in a "zet". The result is that there exist no "zets". You can prove inconsistant results for "zets", but since no "zets" exist, these results do not 'escape' this local bubble and your overall theory stays consistent. Now say you have made an empty datatype "zetelement", write down these two axioms for "zetelements", then since they contradict each other, you can prove anything from them.

Back to the example For our example, all our calculations involve group elements, so we need a structure of groups. `Main` actually contains a commutative group structure, however, it is noted additive, which is at times quite confusing for us humans when combined with exponentials. Isabelle does not care of course, but for readability purposes I wrote a similar definition as is in `Main` with multiplicative notation.
As noted, how `class` really works is outside of the scope of a quick introduction, so it is just here for completeness. In what follows, we will just assume it is a type restriction on our elements with some properties we can use, similar to how we will use natural numbers and for example assume there is an addition which is known to be commutative.

```
class inverse = fixes inverse :: "'a =>'a"
class group_mult = inverse + comm_monoid_mult + finite +
assumes left_inverse: "(inverse a)  * a = 1"
begin
```

Quick remark though, `group_mult` defines a finite commutative multiplicative group. ElGamal uses a finite cyclic group, but since every cyclic group is commutative, we will actually just prove a slightly stronger theorem. Similarly, for correctness g does not have to be a generator, so we will prove the theorem for an arbitrary element g.

UNIV One more handy thing the type system gives us is `UNIV`. We will only use `UNIV` as a way to get the cardinality of our group, because of ways `class` and `locale` work, but as an idea `UNIV` is quite an interesting thing. `UNIV` is an element, or rather a name for one element for each type we have. `UNIV` for a given type is the set of all things of that type. Besides being handy if we want to iterate over all natural numbers for example, this also deals with some more serious logical issues. In intuitive untyped set theory, we could think about the set of all sets that do not contain themselves, which would be a set that does and does not contain itself …Russell's paradox. A type system is one of the two ways that were proposed to resolve Russell's paradox (and others like it), the other being an axiomatic set theory like the Zermelo-Fraenkel set theory. ZF set theory does not allow to define a set by giving a property all elements should satisfy, unless we limit the elements to a known set, thereby avoiding this paradox. A type system solves this issue differently. The set of all natural numbers would be `UNIV` of type `nat set`. This `UNIV` itself is contained in `UNIV`, but this time `UNIV` is of type `nat set set`. This way, none of these sets of all things of a certain type are contained within themselves because they have a diferent ('longer') type, solving the inconsistency that way. If this sounds a little technical, base line would be that self-reference is a tricky thing, and having a type system makes sure we do not get stuck in loops of self-reference.
All of this to say that type systems are not just a byproduct of working with computers. Actually, type theory has long been studied also within formal logic as a possible alternative foundation compared to set theory.

2.3 Function definitions

Now for some more definitions, we need exponentiation in groups, and also a more rigorous definition of encryption and decryption.
There are some different ways to define functions in Isabelle, each with their uses:

abbreviation Things like `abbreviation squares' ::"nat => nat => nat" where "squares' a b == a*a+b*b"` introduce syntactic sugar much like `type_synonym` does.

definition Writing instead `definition squares ::"nat => nat => nat" where "squares a b = a*a+b*b"` actually makes it into the internal workings of Isabelle. Both of these can only be used for defining non-recursive functions though.

fun For more involved functions, we have to use `fun`, for example recursive definitions like the exponential below cannot be done using `definition`. Another situation where `fun` has to be used instead of `definition` is when we want to differentiate between the different types of values a datatype can have. For example going back to the tree example, there, we said a tree can be either just a `tip`, or a combination of two trees and an element. Now we might want to define the size of the tree for both cases, 1 in case it is just a `tip`, the sum of all components' size if the tree is bigger.

```
fun size::"'a tree => nat" where
"size Tip = 1"|
"size (Node t1 n t2) = 1+ size t1 + size t2"
```

inductive When we want to define a certain property of things inductively, we can use something like

```
inductive even :: "nat => bool" where
"even 0"|
"even n ==> even(Suc(Suc(n)))"
```

giving the base case, and an inductive rule. Basically, we are giving all the cases where the output of this function `even` should be `True`.

and more Other options for recursive definitions exist, like *primrec*.
For inductive definitions, `inductive_set` works similar to `inductive` but for sets instead of propositions.
There is a extension of `fun` called `function` which allows you to prove the necessary requirements (like termination of a recursive definition) manually in case Isabelle cannot resolve them automatically.

In other words, too much to cover, however the ones above should get you started.

Back to our example Here we use `fun` recursively for exponentiation, and a simple definition for encryption and decryption. We could have used abbreviation rather than definition, but since this is only syntactic sugar it seems inappropriate.

```
fun exp :: "'b::group_mult => nat => 'b" where
"exp q 0 = 1 "|
"exp q k =q * exp q (k-1)"

definition Enc1 ::
"'b::group_mult => 'b => 'b => nat => 'b" where
"Enc1 g h m r = exp g r"

definition Enc2 ::
"'b::group_mult => 'b => 'b => nat => 'b" where
"Enc2 g h m r = m * exp h r"

definition Dec ::
"'b::group_mult => nat => 'b => 'b => 'b" where
"Dec g x c1 c2 = c2*inverse(exp c1 x)"
```

3 Apply scripts

3.1 Stating a theorem

We have our definitions, so next question is, how do we formulate our theorem ... although, let us start with a little simpler lemma.

```
lemma "g = exp g 1"
```

This would be the most simple way to state a lemma. First remark here: as in most mathematics writings, we can call this lemma, theorem, corollary or proposition as we like, technically there is no difference, it just indicates to the reader how important this result may be. Another thing; as noted, type inference is at play: Isabelle has noticed that this g must be an element of our group, so although it looks like a general result, it is not. This also makes sense, since we defined `exp` only for group elements.
We might like to add to this though, e.g. we may want to name this result, to refer to it later.

```
lemma version2 : "g = exp g 1"
```

Another handy argument one could add is `[simp]`, this basically adds the theorem (once it is proven) to the toolbox of the basic automated proof methods `simp` and `auto`, more on those below. Here there is an obvious

simplification going on, so it might be a good idea, however, we want to turn it around. Logically version 2 and 3 are equivalent, but since the automated proof methods try to simplify things, they use these lemmas only in one direction, assuming that they are given in such a way that the simplest form is on the right. Adding everything to this toolbox is not a good idea though, neither is adding them the wrong way, because then these methods might get stuck in endless loops of expanding and reducing formulas.

```
lemma version3 [simp]: "exp g 1 = g"
```

We can also add explicit typing in case we want/need to, this can be just a personal reminder, or have real purpose in case of type overloading.

```
lemma version4 : "(g::'b::group_mult) = exp g 1"
```

A nicer format for this, offered by Isar, especially for more involved theorems, might be this though.

```
lemma version5:
fixes g::"'b::group_mult"
shows "g = exp g 1"
```

A different line for assumptions might also be added to this for involved theorems, like our goal of proving correctness, which can be written as follows.

```
theorem
fixes g::"'b::group_mult"
fixes x:: nat
fixes r:: nat
assumes "h =exp g x"
assumes "x< card (UNIV::'b set)-1"
assumes "r< card (UNIV::'b set)-1"
shows "Dec g x (Enc1 g h m r ) (Enc2 g h m r ) = m"
```

As noted though, there are options, and it may sometimes be a case of personal preference.

3.2 Simp & auto

Now we have stated our lemma, how can we prove it? In Isabelle the idea is to apply proof methods, amongst which the creatively named `auto`, like so:

```
lemma exp_g_1:
fixes g::"'b::group_mult"
shows "g = exp g 1"
apply (auto)
done
```

Once you have typed your lemma in your prefered style, or any time later when you put your cursor on those lines the output will read something of this style, basically just saying you are in a proof now, this is your goal.

```
proof (prove)
goal (1 subgoal):
1. g = exp g 1
```

Any `apply()` statement you type next will use a certain method to modify these goals until you hopefully get

```
proof (prove)
goal:
No subgoals!
```

in which case you can type `done` to save the theorem as a real fact in Isabelle and close this proof environment. This will also generate an output like

```
theorem exp_g_1:
?g = exp ?g 1
```

saying it saved this theorem (Isabelle calls everything theorems internally), its name if you gave it one, and the internal representation, using this `?name` notation to show which things are placeholders and which things are actually constants. For this simple lemma, `auto` did the trick, so we are already done.

How does this work? How `auto` works is a bit complicated, but its key component `simp` is fairly understandable, and nice to know. You can also use `simp` as a method on its own, if `auto` would modify the goals too much for example.
`simp` simplifies both sides of a goal (formulated as an equality of some sort) as much as possible using the facts it has available. These facts include simplification rules, that is theorems stated using `[simp]` (including those in imported theories), the assumptions of the theorem and simplification rules automatically generated with the constructions `fun` and `datatype`. Remark: definitions are not automatically included in `simp`, since they can be abstract concepts we do not want to expand. They do make facts though, named `name_def` that can be added manually to proof methods.
Often times though, `auto/simp` will not have enough power, like in this next lemma:

```
lemma sum_exp:
fixes g::"'b::{group_mult}"
fixes x:: nat
fixes y:: nat
shows "exp g x * exp g y= exp g (x+y)"
apply auto
```

it might reduce the theorem to different goal(s) that will be outputted, or it might not accomplish anything like in this case

```
proof (prove)
goal (1 subgoal):
1. exp g x * exp g y = exp g (x + y)
Failed to apply proof method:
goal (1 subgoal):
1. exp g x * exp g y = exp g (x + y)
```

This means we have to find another method that may help. Remembering our recursive definition of `exp`, we might want to try induction, using, unsurprisingly `apply (induction y)`. This gives us new goals corresponding to induction base and induction step.

```
proof (prove)
goal (2 subgoals):
1. exp g x * exp g 0 = exp g (x + 0)
2. ⋀y. exp g x * exp g y = exp g (x + y)
    ==> exp g x * exp g (Suc y) = exp g (x + Suc y)
```

The first goal is taken care of by `auto`, the second one stays unchanged though. Another option we have is to add theorems to the set of facts available to `simp`. We can do this locally using

```
apply (auto simp add: mult.left_commute)
```

or

```
apply (simp add: mult.left_commute)
```

which solves it in this case.

But wait a moment, we did not define this fact, so how can we find this fact and what does it say?

To find out what a theorem says we can type something like `thm mult.left_commute` which shows us that this is the fact: `?b * (?a * ?c) = ?a * (?b * ?c)`.
To find a theorem, there are some options. If you have a quite specific idea of what sort of theorem you are looking for, you can use `find_theorems` followed by a name of a type or function like `nat` to get a list of theorems that talk about this, or you can follow it up with something like `"_*0"` to search for theorems that contain something of this form.
If interested, you can also click through on types, theorems and other names by holding ctrl/cmd, which will redirect you to the place where this thing was introduced. Whether that is in your own theory or somewhere in `Main` or other imported theories. You may have a scroll through and find something interesting, although, fair warning, many of these theories are not written for readability.
If you have less of an idea you can also use a tool called `sledgehammer`, more on that in a bit.

3.3 Stronger automated proof methods

`simp` is mainly limited to equalities and `auto` adds some simple logic and set theory capabilities. These two are useful, but a lot remains that they cannot do, thus, what other options are there?

In goals with quantifiers, interesting proofmethods are `fastforce` and `force`. They do very similar things, `force` is mainly a slower version of `fastforce`, that sometimes find some results that `fastforce` gave up on. There are two main caveats with this and all following automated methods compared to `auto` and `simp`. Firstly, these methods act only on the first subgoal, secondly, they succeed or fail, but they do not show you where they got stuck.

For more complex logical goals, the proof method `blast` can be tried. This proof method attempts to be complete for first order formulas, however, in practise there are search bounds of course. `blast` also works on higher order logic, but it is far from a complete method. This is to be expected for all proof methods though, because even in theory there does not exist any complete proof method for higher order logic. `blast` is no cure all though, rather, these methods complement each other quite nicely since this one is quite weak in terms of equalities, `blast` does no rewriting and focusses mainly on logic, sets and relations.

Another appropriately named method, although quite different in its workings, is `sledgehammer`.
Typing `sledgehammer` calls many external automatic theorem provers, using the full library of facts, not just a limited set like e.g. `simp` does. This means you do not need to add in lemmas explicitly. `sledgehammer` also functions on a success/fail basis without feedback about where it failed. `sledgehammer` does however output something different than the methods above, and does not actually finish the proof the way `apply (...)` does. Since `sledgehammer` also calls external provers, it is not fully trusted by Isabelle. So it will return which of the internal methods with which added facts will solve the goal, it might also return something like `proof found : try using by(metis some_lemma another_lemma)`. `Metis` is a specific method that uses only pure logic and its explicit arguments, so it is not easy to use manually, but it allows `sledgehammer` to give the lemmas used by an external prover in such a way that Isabelle can check the proof that was found. Replacing `sledgehammer` by the recommended methods in your document, should finish the proof.
Remark, the suggestion is written in slightly different syntax, so if you want to stick with apply script, you can use `apply(metis some_lemma another_lemma)` instead and follow it up by `done`. Using `by(...)` just combines the two in one, you can not use `by(...)` if you have new subgoals, but for these succes/fail methods it is your choice. There are arguments for the `by(...)` version though, since it closes the proof environment by default, even if something failed, which can be a good thing in keeping issues local.

Although `sledgehammer` sometimes recommends internal proof methods like `simp`, it does not call all of them fully. It is thus recommended to apply at least `auto` or `simp` before calling `sledgehammer`. Furthermore, if you just want to trial out all of the methods, you may be more interested in the methods `try` and `try0`, that try all of the other ones shortly. The latter omits `sledgehammer` and you can also give extra facts as input to these two

methods.

3.4 The rest is your job

For any interesting theorems though, applying one of these methods alone will likely not solve the entire proof.
One option to try and fix this, is to use lemmas that tackle important steps in proving the bigger theorem. The intuitive way to do this is to state and prove these lemmas before proving the bigger result and then add them to our methods.
Another option is to apply rules manually to modify the goals into something these methods can prove. Isabelle has a few rules by default, they are grouped in the command `apply rule` but can be invoked specifically just like other facts. These rules are part of the logical system of natural deduction. They include rules like `conjI` or conjuction introduction, which says that if one knows predicates A and B to be true, then one also knows the predicate $A \wedge B$ to be true. Similar rules exist on how to introduce different other logical constructs like universal quantifier, implication, equivalence ...Apply scripts sort of work in the opposite direction though, they modify the goal, unlike many written proofs that accumulate already known facts. This means that applying the rule `conjI` can be done on a goal of the form $A \wedge B$ to split it up into two subgoals A and B on which to apply other methods. In general, this applying means, replacing the goal, equal to the conclusion of the rule, by as many new subgoals as there are assumptions for the applied rule.
If you have a lemma you want to be used by `apply rule` or other (non-equality) proof methods discussed above, you can give it the attribute `[intro]` when stating it. This comes with the same caveat as for `[simp]` though, it increases the possibility of nontermination when calling these methods. A better option often is to modify a specific call as follows `apply(blast intro: my_rule)`.
`apply rule` or `apply (rule my_rule)` give new subgoals, unlike adding them to an automated proof method. Hence, although you can add them to an automated proof method (later), applying rules in steps gives you more control and feedback when searching for a correct proof.
A final, more forward suggestion, would be to apply rules to facts. This is in contrast to the backchaining that `apply(rule)` does by modifing the goals. You can build theorems from other theorems by using the constructions `my_theorem[of "value"]`, `my_theorem[where ?x = "value"]` or `my_rule[OF my_theorem]`. The first two fill in the placeholders with concrete values for some or all of the terms, the last one applies `my_rule` to `my_theorem`. If you want this forward approach to be used in an automated proofmethod, there is also an argument `dest: some_theorem` that can be added.

3.5 Example

To illustrate these last things, let us look at another lemma about exponentiation, without using automated provers.

```
lemma exp_plus_1:
fixes g::"'b::{group_mult}"
fixes x:: nat
shows "g*exp g x = exp g (Suc x)"
apply (rule trans[where ?s = "g*exp g (Suc x - 1)"])
apply (rule HOL.sym)
apply (rule HOL.arg_cong[where ?f="op * g"])
apply (rule HOL.arg_cong[where ?f="exp g"])
apply (rule diff_Suc_1)
apply (rule HOL.sym[OF exp.simps(2)])
done
```

So what is happening here?
First let us look at the one odd thing in the theorem statement: Suc. $Suc(n)$ or the successor of a natural number n is $n + 1$. But why the complication? Because in formal logic, we cannot really define natural

numbers as ”0,1,2,3 and so on” that is not very formal, so instead, we define them as 0 with all of these successors. The details vary slightly, which is why we do not note it as $+1$, but this is where the function Suc comes from.
To our proof; the initial goal this time is

```
goal (1 subgoal):
1. g * exp g x = exp g (Suc x)
```

The first thing we want to do is apply transitivity to introduce a suitable intermediate step.

```
apply (rule trans[where ?s = "g*exp g (Suc x - 1)"])
```

applies the following theorem

```
?r=?s ==> ?s=?t ==> ?r=?t
```

where we substitute $?s$ to get

```
r=g*exp g (Suc x - 1) ==> g * exp g (Suc x - 1) = ?t ==> ?r = ?t
```

Isabelle does unification behind the scenes to fill in the other terms in such a way that it can be applied to our current goal. The requirement for a rule to be directly applicable to a goal is that the conclusion of the theorem is exactly the goal. This means the theorem used is

```
g * exp g x = g * exp g (Suc x - 1) ==> g * exp g (Suc x - 1) = exp g (Suc x)
                                        ==> g * exp g x = exp g (Suc x)
```

What applying a theorem does then, is to replace the goal by the assumptions of the theorem. In this case there are two so we get two new goals.

```
goal (2 subgoals):
1. g * exp g x = g * exp g (Suc x - 1)
2. g * exp g (Suc x - 1) = exp g (Suc x)
```

Any further rule applications only work on the first of the two goals. For this first goal, we have found a theorem `diff_Suc_1`: `Suc ?n - 1 = ?n` but it goes in the opposite direction compared to our goal, so we mirror our goal using

```
apply (rule HOL.sym)
```

Where `HOL.sym` is the fact `?s = ?t ==> ?t = ?s`. This gives us the new goals

```
goal (2 subgoals):
1. g * exp g (Suc x - 1) = g * exp g x
2. g * exp g (Suc x - 1) = exp g (Suc x)
```

Now, to apply this theorem there are still some things in the way. To get rid of the multiplication by g and the exponentiation we can use the theorem `HOL.arg_cong`: `?x = ?y ==> ?f ?x =?f ?y` by instantiating the function $?f$ by those two operations.

```
apply (rule HOL.arg_cong[of _ _"op * g"])
apply (rule HOL.arg_cong[where ?f="exp g"])
```

The new first goal looks exactly like the theorem mentioned before.

```
goal (2 subgoals):
1. Suc x - 1 = x
2. g * exp g (Suc x - 1) = exp g (Suc x)
```

Therefor we can apply it to solve this goal entirely.

```
apply (rule diff_Suc_1)
```

```
goal (1 subgoal):
1. g * exp g (Suc x - 1) = exp g (Suc x)
```

With the second goal, we can demonstrate the other forward option: `OF`.

```
apply (rule HOL.sym[OF exp.simps(2)])
```

Here `exp.simps` are the simplification rules automatically generated by defining exponentiation using `fun`. They look like this

```
exp ?q 0 = 1
exp ?q (Suc ?v) =
?q*exp ?q (Suc ?v - 1)
```

Slightly different than the original definition, since Isabelle automatically translates them into a forward stepping induction rather than the recursive $q^k = q * q^{k-1}$ we defined.
Very similar to our goal though, only mirrored. Hency we want to use `HOL.sym` again. This time instead of applying it to the goal, we opted to apply it to the simplification rule to compose the theorem

```
?q*exp ?q (Suc ?v - 1) = exp ?q (Suc ?v)
```

which is exactly our goal with no extra assumptions and therefore solves the goal entirely.

```
goal:
No subgoals!
```

Now we can write `done` to save this lemma for later use.

4 Isar

As you may have been thinking by now, applying rule after rule becomes quite long, and unless you are running the script, it is very hard to follow what is happening. Applying all these theorems using an automated method looks less messy, but they are bit of a black box and it might not work in one step. Hence, since most of them are success/fail only and thus can only be used as the last proof method to finish a proof, that is not always an option. Resolving this by making a separate and named lemma for each of the steps you need ...maybe not the most readable either.
Another way to write in the Isabelle framework is using Isar (short for intelligible semi-automated reasoning), a more intelligible language for structured proofs, as the name suggests. We already saw how we can state our theorems in a more ordered way, but what about proofs?
And when are we going to prove this correctness example? Well, now and now, or ...let us start by trying to prove the theorem in ways we already know, for comparison.
First, two cheats, since this is a relatively simple example, we could just give this:

```
theorem
fixes g::"'b::group_mult"
fixes x:: nat
fixes r:: nat
assumes "h =exp g x"
assumes "x< card (UNIV::'b set)-1"
assumes "r< card (UNIV::'b set)-1"
shows "Dec g x (Enc1 g h m r ) (Enc2 g h m r ) = m"
apply(auto simp add: Enc1_def Enc2_def Dec_def assms prod_exp
    mult.assoc mult.commute left_inverse)
```

or this:

```
by (smt Dec_def Enc1_def Enc2_def ab_semigroup_mult_class.mult.commute
    ab_semigroup_mult_class.mult_ac(1) assms(1) group_mult_class.left_inverse
    monoid_mult_class.mult.left_neutral prod_exp)
```

The second one, unsurprisingly, is a product of `sledgehammer`, the only automated prover in `try` that came up with a proof for this theorem. In reality `sledgehammer` is of course a valid and useful tool, and for this level of complexity, it might well come up with a proof, however, it is not very educating and will not always work, so let us forget about that. The first one, was made after finding a step by step proof, and testing whether given all the lemmas `auto` would find it in one go. Less of a cheat, and a clean final version like this might be quite desirable. Still, let us look at a more realistic apply script.

```
apply (auto simp add: Enc1_def Enc2_def Dec_def)
apply (auto simp add: assms)
apply (auto simp add: prod_exp)
apply (rule trans[where ?s = "m*(exp g (r*x) * inverse (exp g (x*r)))"])
apply (simp add: mult.assoc)
apply (auto simp add: mult.commute)
apply (simp add: mult.commute[of "exp g (r * x)""inverse (exp g (r * x))"]
                    left_inverse[of "exp g (r*x)"])
done
```

Interesting note though, some of these subgoals do not get solved by `auto` in one go if given all the facts in remaining steps. This can happen because `simp` simplifies both sides as much as possible, whilst your intermediate step might not be on its path to the best simplification. This also relates to another weakness of these apply style scripts. Since the strength of these proof methods may evolve over time, after an update `simp` might simplify something further. This does not sound like a bad thing. However it means that if in some proof you use `simp` in a case where a goal is not solved completely, all further steps that were written to proof a certain remaining goal will likely not work on the new remaining goal. Quite an issue in maintaining these scripts.
Quick run through: the first apply statement expands our definitions (remember, unlike functions, definitions are not automatically considered simplification rules). Next we fill in h using the fact `assms`, the default name for the assumptions. Then, we simplify the double exponential using another lemma (A full proof with all definitions and necessary lemmas in order can be found in appendix) and have by now this goal.

```
1. m*exp g (r*x)*ElGamal.inverse_class.inverse(exp g (x*r)) = m
```

This would seem a fair simplification, as said though, if we add the fact `left_inverse` now, even adding associativity and commutativity, it does not get solved by `simp`, so we add an intermediate step, using the same transitivity rule as before.
Giving `simp` access to associativity then solves the first of the two goals, the second stays, and adding the axiom `left_inverse` still does not help. Maybe it is due to the difference between $x * r$ and $r * x$, commutativity helps there.
The inverse still does not resolve itself, with some trial and error, or `sledgehammer`, you could probably find which theorem about the unit element you should add, but we said we forgot about `sledgehammer`. Another option to nudge Isabelle in the correct direction, is to tell the simplifier exactly on what to use inverse and commutativity, which in this case works ...no goals left ...great.

Not too bad, right? Right, but this is a very simple example, and we are using some previous lemmas. And the other remark still holds, it is still quite hard to follow allong without running the script, even with additional notes.
So, how does Isar solve these issues? For starters, as you see in the example below, Isar documents read somewhat like a normal proof. We state a subgoal each time, and follow that with by which method we prove this. Secondly, if a step is not solved by one method, Isar also allows to replace the `by(...)` statement by an entire proof block like this. You can take this too far of course, lemmas exist for a reason, but for small steps it can be easier to follow along then to go find a lemma in a long document for each non-trivial step.

```
proof -
have "Dec g x (Enc1 g h m r ) (Enc2 g h m r ) = m * exp h r * inverse (exp (exp g r) x)"
 by (auto simp add: Enc1_def Enc2_def Dec_def)
also have "...= m * exp (exp g x) r *inverse(exp (exp g r) x)"
 by (simp add: assms)
also have "...= m * exp g (r * x) * inverse(exp g (x * r)) "
 by (simp add: prod_exp)
also have "...= m *(exp g (r*x) * inverse(exp g (x*r)))"
 by (simp add: mult.assoc)
also have "...= m *(inverse(exp g (x * r)) * exp g (r*x))"
 by (simp add: mult.commute)
also have "...= m *((inverse(exp g (x*r))*(exp g (x*r))))"
 by (simp add: mult.commute)
also have "...= m " by (simp add: left_inverse)
finally show "Dec g x (Enc1 g h m r ) (Enc2 g h m r ) = m".
qed
```

As before, quick runthrough: we start a proof of with the keyword `proof` and then, in general, a proof method, in this case we write '-'. This indicates we do not use any proof method, and no modifications of the goals should happen right away. Other options like `proof (induction r)` would break up the goal in two cases for induction base and induction step, where once you prove both cases Isabelle will consider the proof correctly finished. As always, `induction` and also the similar looking `cases` have many options and uses, about which there has been written. Another proof method option is to leave a blank, which tells the system to use one of the standard introduction rules. Similar to what `apply rule` does, which we discussed before. In practise these seem to be the most commonly used things here, but in theory anything that can be in a `apply(...)` or `by(...)` is a proof method and can thus be used after `proof`. No matter the proof method we choose here, the goal(s) will be outputted now as well, and they are the ones that should be after `show` in the end to finish the proof.
Next, we want to apply the definitions of encryption and decryption again. You will notice we use some of the same intermediate steps as for the apply script. However, even though you can make a literal translation, from an apply script to an Isar document and back, that is not how writing both of these happens in reality. Apply scripts incentivise working backwards from your goal, since they work by reducing a goal to new (sub)goals, Isar style writing on the other hand incentivises to have more of a forward approach, stating a fact, proving it, using that fact to prove the next etc. due to its structure. Both of these are valid proof methods, and depending on personal preference and the topic at hand either might be more desirable. Both systems allow both styles, but in my opinion there is really a factor of the syntax influencing ones writing style.
If we look at the syntax we see that applying the definitions requires writing `have` – the foreseen result – `by` – a proofmethod. Alternatively we could replace `by(...)` with a proof block, i.e. another proof like this one. This is our first step. In general a step in an Isar proof might include a few more things, like in this made up example.

```
fix k::nat
assume "k<x"
from this diff_Suc_1[of "k"] have "Suc k-1 < Suc x-1" using diff_Suc_1 by auto
(*diff_Suc_1: Suc ?n - 1 = ?n*)
```

First we can possibly fix some variables and make some extra assumptions. Then we can name facts we want to use. Here `this` is $k < x$, our assumption, if we do not make assumptions, `this` is the proposition proved in the previous step. We can also use other facts like previously proved lemmas or theorems from `Main`. An equivalent way to add facts we want to use appears after the `have` statement, with the keyword `using`. Finally, we give the proof of our claim, either with a proof method like here, or with a proof block. The last line is just a comment showing what this theorem from main says, something you may expect in further examples.
In general a proof will have a series of steps like these, with the last one being of the same form but using `show` to indicate this result is our goal. In case there are several goals due to our initial proof method, then there will be several steps with `show`. After the last goal is tackled, we put `qed` to close the proof environment and

save the theorem. A few more examples of Isar style proofs and explanations of handy syntax will follow, but let us go back to our example here, because since this is a series of equalities, a few things are quite different. For equalities we can make use of another internal theorem variable `calculation`. Similar to `this` which is always the last fact, `calculation` is a way to use what we did before. `Calculation` is started when we use `also` for the first time, at which point it is initialised as `this`, our first equality we just proved. Every following time we use `also`, Isabelle adds `this` to `computation` using some basic transitivity rules. (The same scheme works for inequalities as well.) In the end we can use `finally` which is short for `also from computation` to use all of the previous steps to prove our goal. The proof method of choice here is '.' which is proving only using the assumptions. We could use something like `by auto` here as well, but it is unnecessary and '.' looks quite neat. Thus, in our example, we want to add to the previous equality by expanding h and we start a new step with `also have`. We could now write the new equation all out, but for readability we can use '…', another default variable, that is always set to the right hand term in the previous (in)equality. Proof method is as before using the assumptions, the next step simplifies the double exponential adding another equality to `computation`. Some more steps follow to rearrange factors so we can apply the `left_inverse` axiom and finish the proof.

4.1 Example

As a bonus, we will take a look at the entire proof for correctness again, using no automated proof methods. We will logically only use the basic steps, in an attempt to illustrate what might happen behind the scenes. However since this is not a running script, we will do so using the Isar syntax and take the opportunity to also see some of its features.
Although this is probably not exactly what the automated provers would do in this case, these automated provers internally also break down proofs to individual theorem applications. Since each theorem has been build in this way from the priorly available theorems, this means that only a very small core of the system has to be trusted. Provers that use an approach like this are said to use a foundational approach.

For both `sum_exp` and `prod_exp` we want to use the definition of `exp` in a slightly different way, so we made this lemma.

```
lemma exp_plus_1:
fixes g::"'b::group_mult"
fixes x:: nat
shows "g*exp g x = exp g (Suc x)"
proof -
have "exp g (Suc x) = g* exp g x"
 proof(rule trans) (*thm trans ?r = ?s ==> ?s = ?t ==>?r = ?t*)
  show "exp g (Suc x) = g*exp g (Suc(x)-1)"
   by (rule exp.simps(2))(*thm exp.simps(2) exp ?q (Suc ?v) = ?q*exp ?q (Suc ?v - 1)*)
  have "Suc x - 1 = x"
   by (rule diff_Suc_1) (*thm diff_Suc_1 Suc ?n - 1 = ?n*)
  from this show "g*exp  g (Suc x -1) = g* exp g x"
   by (rule HOL.arg_cong) (*thm HOL.arg_cong ?x = ?y ==>?f ?x = ?f ?y*)
 qed
from this show ?thesis
 by (rule HOL.sym) (*thm HOL.sym ?s = ?t ==> ?t = ?s*)
qed
```

Some interesting things here:

- As hinted at before, we can nest proof blocks. After the `qed` of the inner proof block, `this` is the statement for which we started the proof block, not the line right before. This makes sense in a way with `this` always being the last proven fact.
- `proof(rule trans)` demonstrates that any method that fits a apply or by statement can also be used after proof. We also see that as in an apply script, proof method `rule trans` causes the original goal to split

up in two different goals. This is why there are two show statements in this proof block. Unlike in apply scripts though, since you write down what is being shown, you can work in either order (or on both goals at the same time).

- Another default name that is used here is `?thesis`, this is, as one would expect, a proposition variable that contains the goal of the current proof. In case this is uniquely defined of course.

```
lemma sum_exp:
fixes g::"'b::group_mult"
fixes x y:: nat
shows "exp g x * exp g y= exp g (x+y)"
proof (induction y)
case 0
  have a: "exp g x * exp g 0 = exp g x * 1"
   by (rule HOL.arg_cong[OF exp.simps(1)]) (*thm exp.simps(1) exp ?q 0 = 1*)
  have b: "exp g x * 1 = exp g x"
   by (rule mult_1_right) thm mult_1_right (*thm mult_1_right ?a * 1 = ?a*)
  have "exp g (x+0) = exp g x"
   by (rule HOL.arg_cong[OF Nat.add_0_right]) (*thm Nat.add_0_right ?m + 0 = ?m*)
  from this have c: "exp g x = exp g (x+0)"
   by (rule HOL.sym)
  have "exp g x * exp g 0 = exp g x" by (rule trans [OF a b])
  from this show ?case by (rule trans [OF _ c])
next
case (Suc y)
  have "exp g (Suc y) = g*exp g y"
   proof(rule trans)
    show  "exp g (Suc y) = g*exp g (Suc y - 1 ) "
     by (rule exp.simps(2)[of g y])
    show "g*exp g (Suc y - 1) = g* exp g y"
     by (rule HOL.arg_cong[OF  diff_Suc_1[of y]])
   qed
  then have a: "exp g x * exp g (Suc y) = (exp g x) * (g * (exp g y))"
   by (rule HOL.arg_cong)
  have "(exp g x) * g * (exp g y) = (exp g x) * (g * (exp g y))"
   by (rule mult.assoc) (*thm mult.assoc:  ?a * ?b * ?c = ?a * (?b * ?c)*)
  then have b: "(exp g x) * (g * (exp g y)) =(exp g x) * g* (exp g y)"
   by (rule HOL.sym)
  have c: "(exp g x) * g* (exp g y) =g * exp g x *(exp g y)"
   proof -
    have "exp g x * g = g * exp g x"
    by (rule mult.commute) (*thm mult.commute ?a * ?b = ?b * ?a*)
    then show ?thesis by (rule HOL.arg_cong)
   qed
  have d: "g * exp g x *(exp g y) =g * (exp g x *exp g y)"
   by (rule mult.assoc)
  have e: "g * (exp g x *exp g y) = g * exp g (x+y)"
   by (rule HOL.arg_cong[OF Suc]) (*thm Suc exp g x * exp g y = exp g (x + y)*)
```

```
  have f: "g * exp g (x+y) = exp g (x+Suc(y))"
   proof (rule trans)
    show "g* exp g (x+y) = exp g (Suc (x+y))"
     by (rule exp_plus_1)
    have "Suc (x+y) = x+Suc y"
     by (rule HOL.sym[OF Nat.add_Suc_right]) (*Nat.add_Suc_right ?m + Suc ?n = Suc(?m+?n)*)
    thus "exp g (Suc (x+y)) = exp g (x+Suc y)"
     by (rule HOL.arg_cong)
   qed
  have "exp g x * exp g (Suc y) =(exp g x) * g* (exp g y)"
   using a b  by (rule trans)
  hence " exp g x * exp g (Suc y) =g * exp g x *(exp g y)"
   using c by (rule trans)
  hence " exp g x * exp g (Suc y) =g * (exp g x *exp g y)"
   using d by (rule trans)
  hence " exp g x * exp g (Suc y) = g * exp g (x+y)"
   using e by (rule trans)
  thus ?case using f
   by (rule trans)
qed
```

A few things we can note here:

- `proof (induction)` is a proof method used for induction that finds the correct induction rule for the type of the element specified. With this proof method (in Isar) the subgoals are formatted as cases. You can still opt to not use these, and just make two correct show statements for the two goals. These cases do not do much differently formally, but they create some overview and also some automatically generated names. Namely, within each case the variable `?case` will contain the appropriate subgoal. In the second case / the induction step, this proof structure will also split the goal $P(n) \Rightarrow P(Suc(n))$ into the assumption $P(n)$ with name `Suc`, available to use as a fact, and a goal $P(Suc(n))$. Once one goal is proven, we can move to the second case by putting `next`.

- To avoid making named external lemmas for little things in case we need for example more than one previous fact at a time and therefore `this` is not enough, we can also name intermediate facts. We can do this by adding `name:` after `have`, and this allows us to reference to this fact the same way as all other theorems, for example with the construct `using`. We can however only do this within the current proof, so once it is finished, we can not reference this intermediate step anymore, nor can we reference steps from an inner proof block within the main proof.

- To make the proofs easier to read, several words are available, for example these shortcuts
  `from this` = `then`
  `from this have` = `hence`
  `from this show` = `thus`
  these words do not make a difference internally, they are just expanded upon parsing, but they can help with readability, and they can give a clearer indication of the structure then a long sequence of words might.

```
lemma prod_exp:
fixes g::"'b::group_mult"
fixes x r:: nat
shows "exp (exp g x) r = exp g (r*x)"
proof (induction r)
case 0
  have 1: "exp(exp g x) 0 = 1" by (rule exp.simps(1))
  have "exp g (0*x) = exp g 0" by (rule HOL.arg_cong[OF Nat.mult_0])
  hence 2: "exp g (0*x) = 1" by (rule _ trans[OF _ exp.simps(1)])
  show ?case using 1 2 by (rule trans_sym)
next
case (Suc r)
  have "exp (exp g x) (Suc r) = (exp g x) * exp(exp g x) r" by (rule HOL.sym[OF
    exp_plus_1])
  hence "exp (exp g x) (Suc r) = exp g x * exp g (r*x)"
   using HOL.arg_cong[OF Suc, where ?f= "op * (exp g x)"]by (rule trans)
    (*thm Suc exp (exp g x) r = exp g (r * x)*)
  hence "exp(exp g x) (Suc r) = exp g (x + r*x)" using sum_exp  by (rule trans)
  thus "exp(exp g x) (Suc r) = exp g (Suc r *x)"
   using HOL.arg_cong[OF HOL.sym[OF Nat.times_nat.simps(2)]] by (rule trans)
    (*thm Nat.times_nat.simps(2) Suc ?m * ?n = ?n + ?m * ?n*)
qed
```

If you are by now wondering what the difference between `[OF ...]` and `using` is, in our case they do the same. When we are using `using` Isabelle tries to figure out behind the scenes where this fact is usable. Since we are only using basic rules and no automated provers, it basically tries to use `[OF...]`-style instantiation. We still use `[OF...]` sometimes, because it allows us to do several nested instantiation steps which the rule method does not try automatically. Also interesting: if you give an automated prover too many arguments, it might not terminate but more often it does not make much of a difference. If we are working with such basic steps like `rule` though, it does, any arguments given with `from` or `using` are just filled in, in order, as if they were given with `[OF ...]`.

```
theorem correctness:
fixes g::"'b::group_mult"
fixes x r:: nat
assumes "h =exp g x"
assumes "x< card (UNIV::'b set)-1"
assumes "r< card (UNIV::'b set)-1"
shows "Dec g x (Enc1 g h m r ) (Enc2 g h m r ) = m"
proof -
have a: "Dec g x (Enc1 g h m r ) (Enc2 g h m r ) =m*exp h r*inverse(exp(exp g r) x)"
 (is "?la = ?ra")
 proof (rule trans)
  have "?la= (Enc2 g h m r) * inverse(exp ( Enc1 g h m r) x)" by (rule Dec_def)
  moreover have "(Enc2 g h m r)*inverse(exp(Enc1 g h m r) x)=
      (Enc2 g h m r)*inverse(exp(exp g r) x)" using Enc1_def by (rule HOL.arg_cong)
  ultimately show "?la = (Enc2 g h m r)*inverse(exp(exp g r) x)" by (rule trans)
  show "(Enc2 g h m r)*inverse(exp(exp g r) x) = ?ra" using Enc2_def by (rule HOL.arg_cong)
 qed
```

```
have b: "...= m * exp (exp g x) r * inverse(exp (exp g r) x)" (is "?lb = ?rb")
 using assms(1)  by (rule HOL.arg_cong)
have c: "...= m * exp g (r*x) * inverse(exp (exp g r) x) " (is "?lc = ?rc")
 using prod_exp by (rule HOL.arg_cong)
have d: "...= m * exp g (r*x) * inverse(exp g (x*r)) " (is "?ld = ?rd")
 using prod_exp by (rule HOL.arg_cong)
have e: "...= m * (exp g (r*x) * inverse(exp g (x*r)))" (is "?le = ?re")
 by (rule mult.assoc)
have f: "...= m * (inverse(exp g (x*r)) * exp g (r*x))" (is "?lf = ?rf")
 using mult.commute by (rule HOL.arg_cong)
have g: "...= m * ((inverse(exp g (x * r)) * (exp g (r*x))))" (is "?lg = ?rg")
 using mult.assoc by (rule HOL.arg_cong)
have h: "...= m * ((inverse(exp g (x*r))*(exp g (x*r))))" (is "?lh = ?rh")
 by (rule HOL.arg_cong[OF mult.commute[of "r" "x"]])
have i:  "...= m * 1 " (is "?li = ?ri") using left_inverse by (rule HOL.arg_cong)
have j: "... = m"  by (rule mult_1_right) (*mult_1_right:  ?a * 1 = ?a*)
have "?la=?rb" using a b by (rule trans)
hence "?la=?rc" using c by (rule trans)
hence "?la=?rd" using d by (rule trans)
hence "?la=?re" using e by (rule trans)
hence "?la=?rf" using f by (rule trans)
hence "?la=?rg" using g by (rule trans)
hence "?la=?rh" using h by (rule trans)
hence "?la=?ri" using i by (rule trans)
thus ?thesis using j by (rule trans)
qed
```

Two more space savers:

- `moreover` and `ultimately` are a few more quite useful words. These allow you to gather several facts in `this` to use. This is a nice substitute for explicit names in case you need more then one previous fact to prove your next statement.
- We already know the `...` to refer to the right hand side of the last statement, but we can also give terms our own names. This is done using `(is ...)` and works analoguously to the unification taking place when we apply rules. You are also by no means limited to equalities.

And to finish this list of tricks, three more neat commands.
When using Isabelle, you can type `quickcheck` after writing a theorem down. This does a quick search for counterexamples, which may catch a mistake before you spend hours trying to prove it.
If you have written a false theorem, or did not finish a proof for some other reason but do want to keep it in your document, you can write `oops` to let Isabelle know something was off, and that environment should be closed. This prevents your next proof from generating all kinds of syntax errors.
And if you made it through this whole report without getting the feeling that Isabelle is a friendly but stubborn lady ... you can also excuse yourself to her in case you do not want to prove something (preferably for good reasons, otherwise you might end up with a false proof after all) you can type `sorry` to save a fact without proving it, sort of as an axiom. Just an observation.

Take aways

We argued why proof assistents like Isabelle are valuable. We took a look at how Isabelle works with goals, using known theorems to reduce them step by step. We also saw that, thanks to Isar, intelligible for the computer no longer has to mean completely unreadable to humans. Lastly I hope you got inspired or curious to try out Isabelle for yourself because besides ensuring the correctness of the proof you are actually checking, working with a proof assistant also teaches you where you may need to pay some more attention in general.

I also decided not to shorten the examples by neglecting syntactical details, so if you do decide to give it a try, I believe these pieces of example code and the little practical tricks throughout this report will prove quite useful.

References

[1] F. Wiedijk (ed.), The Seventeen Provers of the World, foreword by Dana S. Scott, Springer LNAI 3600, 2006.

[2] `http://www.cs.ru.nl/~freek/digimath/index.html` by Freek Wiedijk

[3] Many tutorials and documentations included with the Isabelle download and/or on `http://isabelle.in.tum.de/` notably the tutorial "Programming and Proving in Isabelle/HOL".

[4] `https://proofcraft.org/blog/isabelle-style.html` by Gerwin Klein

Apendix: the whole theory in Isar using simp/auto – to give an idea

```
theory ElGamal
imports Main
begin

class inverse = fixes inverse :: "'a=>'a"
class group_mult = inverse + comm_monoid_mult + finite +
assumes left_inverse: "(inverse a)  * a = 1"

fun exp :: "'b::group_mult=>nat =>'b" where
  "exp q 0 = 1 "|
  "exp q k =q * exp q (k-1)"
definition Enc1 :: "'b::group_mult=>'b=>'b=> nat=>'b"
  where "Enc1 g h m r = exp g r"
definition Enc2 ::"'b::group_mult=>'b=>'b=> nat =>'b"
  where "Enc2 g h m r = m * exp h r"
definition Dec :: "'b::group_mult=>nat=>'b=>'b=>'b"
  where "Dec g x c1 c2 = c2*inverse(exp c1 x)"

lemma sum_exp:
fixes g::"'b::group_mult" and x y:: nat
shows "exp g x * exp g y= exp g (x+y)"
proof (induction y)
case 0
  thus ?case by auto
next
case (Suc y)
  have  "exp g x * exp g (Suc y) = (exp g x) * (g * (exp g y))" using Suc by auto
  also have "... =(exp g x) * g * (exp g y)" by (simp add: mult.assoc)
  also have "... =g * exp g x * exp g y" by (simp add: mult.commute)
  also have "... =g * (exp g x * exp g y)" by (simp add: mult.assoc)
  also have "... = g * exp g (x+y)" using Suc by (auto)
  also have "... = exp g (x + Suc y)" by auto
  finally show ?case.
qed

lemma prod_exp:
fixes g::"'b::group_mult" and x r:: nat
shows "exp (exp g x) r = exp g (r*x)"
proof (induction r)
case 0
  thus ?case by auto
next
case (Suc r)
  thus ?case by (simp add: sum_exp)
qed

theorem correctness:
fixes g::"'b::group_mult" and x r:: nat
assumes "h =exp g x" and "x< card (UNIV::'b set)-1" and "r< card (UNIV::'b set)-1"
shows "Dec g x (Enc1 g h m r ) (Enc2 g h m r ) = m"
proof -
have "Dec g x (Enc1 g h m r ) (Enc2 g h m r ) = m*exp(exp g x) r * inverse(exp(exp g r) x)"
 by (auto simp add: Enc1_def Enc2_def Dec_def assms)
also have "...= m * exp g (r*x) * inverse(exp g (x*r)) " by (simp add: prod_exp)
also have "...= m * (exp g (r*x) * inverse(exp g (x*r)))" by (simp add: mult.assoc)
also have "...= m * ((inverse(exp g (x*r)) * (exp g (x*r))))" by (simp add: mult.commute)
also have "...= m " by (simp add: left_inverse)
finally show ?thesis.
qed
end
```