Isabelle Primer for Mathematicians

B. Grechuk

Abstract

This is a quick introduction to the Isabelle/HOL proof assistant, aimed for mathematicians who would like to use it for formalization of mathematical results.

1 Introduction

Interactive proof assistants are special programs, which make it possible to check mathematical results up to a nearly absolute level of certainty. Clearly, computers cannot read and understand usual human language, and even if they could, a typical textbook proof usually omits some details and cannot be treated as absolutely rigorous. To check the proof in an automated proof assistant, you need to write it using special language, understandable by computer. This ”translation” to computer language is called the formalization of the proof.

There are many proof assistants with different languages and underlying logics, and the first step is to choose a particular one. Suppose that you chose the Isabelle/HOL proof assistant and want to learn how to perform formalization using it. The obvious first step is to download the tutorial [1], and this should certainly be done. However, not everyone will want to read a 218-page tutorial before formalizing the first simple lemma. In contrast to the tutorial, this primer is aimed to help you start formalization immediately, and learn the Isabelle language (called Isar) in the process, based on examples. Sometimes, however, we will still refer to the tutorial.

You can work with Isabelle directly, or use a graphical user interface. It is highly advised to use a graphical user interface, and the recommended interface is the Emacs-based Proof General. From now on I will assume that you use this interface.

First, you need to install Isabelle with Proof-General on your computer. The installation process depends on the computer platform you use. All the programs and instructions can be found on the Isabelle website http://isabelle.in.tum.de/ and in the tutorial. Supposing you have installed the program correctly, you are ready to work. Let us start!

2 A First Lemma

If you open Proof General, you see an empty window where it is possible to enter text. This file is automatically called Scratch.thy. The whole Isabelle library consists of files with the extension ”thy”, which are called theories. A theory in Isabelle is just a collection of definitions, lemmas and theorems, like chapters in a book. Dozens of theories are installed with Isabelle on your computer, and you can use any lemma from these theories to derive your results. More important at this stage is that you can use these theories to study Isar - the language of Isabelle. Instead of reading hundreds pages of tutorial, let’s just open some theory and see how it looks.

To open an existing theory, you use the File $>$ Open File command in the menu, then choose the folder where you installed Isabelle, then $>$ src $>$ HOL, and here is a huge theory collection. Let us, for example, open theory Fun.thy, in which basic notions about functions are introduced. The theory starts with some comments and explanations of what is done here, embraced by open parenthesis-star and star-closed parenthesis, like this:

```
(*comment*)
```

As you have probably guessed, in such comments you can write whatever you want, and this will not affect your proofs. Other possible syntax for comments are `text{*comment*}`, `--{*comment*}`, `--"comment"`. Because large formal proofs are sometimes hard to understand, it is desirable to write such comments, but for now we can omit them together with the optional `header` command and look further, where the theory actually starts.

```
theory Fun
```

Ok, this is easy. So, every theory should start with keyword `theory` followed by the theory name. We can open Isabelle in a new window, and write similarly:

```
theory MyFirst
```

The next lines in Fun.thy are

```
imports Complete_Lattice
begin
```

Clearly, the command `begin` just starts the theory. The command `imports` requires a little bit more explanation. Theories in Isabelle form a huge directed graph - some theories ”import” other ones to refer to their results and definitions. You will need to import at least one theory, otherwise you will have to prove everything from scratch. You can import several theories using a command like

```
imports <theoryName1> <theoryName2> <theoryName3>
```

but importing a theory obviously means importing the whole hierarchy behind it, therefore it is usually enough to import just one theory with a rich enough parent hierarchy. For a start, it may be a good choice to import Main.thy, which accumulates all the basic theories in arithmetic. So, let us write

```
imports Main
begin
```

and look at the example Fun.thy again. After the command `begin`, we can see some comment of the form `text {* ... *}` , which we will ignore, and then finally we see the first lemma of the theory

```
lemma expand_fund_eq:  "f = g ⇔ (∀x. f x = g x)"
```

From this example we see, that every lemma starts with keyword `lemma` followed by the lemma’s name, a colon `:`, and the lemma formulation in quotes. Clearly, the name of the lemma is given for future references. Let us ignore this lemma for now, and start with something which seems to be easier to prove, and write

```
lemma two_two:  "2 + 2 = 4"
```

Now, we want to prove this lemma. The statement which we want to prove is called a goal. If we look at Fun.thy, we can guess that text following every lemma is the proof. First lemma expand_fund_eq has a 4-line proof, which looks like

```
apply(...)
...
apply(...)
done
```

This is one of the general proof strategies in Isabelle. Command `apply` means that we want to apply some method to prove our goal. After this, the goal is usually simplified, and we use another method to proceed. When the goal is proved, we write command `done`. But for our simple lemma we do not expect a long proof, we would prefer to prove such a statements in one step. So, let us look at the second lemma in Fun.thy

```
lemma apply_inverse:
"f x = u → (∀ x.  P x → g (f x) = x) → P x → x = g u"
by auto
```

You can see, that the whole proof here is just the phrase `by auto`. `auto` is a method which tries to prove the statement automatically. This method combines logical reasoning and arithmetic transformations to simplify the goal and ideally prove it completely. Command `by` is just a replacement for two commands `apply` and `done`, namely `by(method)` is the same as `apply(method) done`. It requires the method to solve a goal completely, and will fail otherwise. Also, from this example we can see, that `by auto` can be written without parentheses. Ok, let us try to prove our simple lemma similarly:

```
lemma two_two:  "2+2=4" by auto
```

Next step is to check in Isabelle that the proof is correct. We can click on Proof-General > Next Step in the menu. This is the same as clicking on the right arrow on the tool-bar panel just below the menu. If you click once, you see that the start of your theory has a different color, which means that this part is checked and no misprints found. We will call this checking process "execution". If you click second time, the lemma formulation is accepted, and if you click the third time, you will see the message `Failed to finish proof At command by` in the window below, which is called the Proof General response buffer.

The usual strategy in this case is to replace `by` by `apply` and see how far `auto` can proceed. But in this case we will see the message `empty result sequence -- proof command failed At command "apply"`. This message indicates that method cannot make any progress in solving the goal and hence cannot be applied here at all.

So, does it mean that we cannot prove even such a simple lemma automatically in Isabelle? Clearly, in this case formalizing serious mathematical results would be completely impossible in Isabelle. Fortunately, this is not the case. To see the problem, let us go back (left arrow on the tool-bar panel), then try again, but execute only the lemma formulation and look at the response buffer (window below).

```
proof (prove):  step 0
goal (1 subgoal):
1.  (2::'a)+(2::'a)=(4::'a)
```

The line `proof (prove): step 0` just indicates that proof starts, then Isabelle shows us how it understands our lemma. Symbol `::` indicates type of the object. The point is that we often use the same symbols in mathematics to indicate formally different notions. Of course, this is completely unacceptable in formal proof system, and therefore every constant or variable has a type, which clearly indicates what we mean. For example, we use symbol "2" in mathematical text to indicate natural number, real number, complex number, or even 2 in a non-decimal system, in which 2+2 may not be 4. Symbol `'a` denotes a *type variable*, meaning that the system understands that 2, 2, and 4 are meant to have the same type, but it has no idea what this type is. Sometimes this works: for example, if we write the lemma `2=2`, we can easily prove it by auto, because this lemma is true for every type. So, formally, we wanted to state our lemma about natural numbers, but we stated a much more general lemma, which in general is incorrect, and this was the reason that we could not prove it.

The type for natural numbers is `nat`, and we can correct the lemma as follows

```
lemma two_two:  "(2::nat)+2=4" apply auto
```

Notice that we need to specify the type only once, and Isabelle automatically understands that the second 2 and the 4 are also natural numbers. Now, if we execute the lemma formulation, it shows the correct goal `2+2=4`,[1] and after executing `apply auto` we get a message `No subgoals!` which means that the proof is finished, and we can either type `done` or replace `apply` with `by`, finally getting

[1] Now no types are indicated in the goal `2+2=4`. The type-related error described above is very typical when working in Isabelle, so if you have error and do not know what is the reason, choose Isabelle > Settings > Show types in the menu, and try to execute again. Now you will see all the types in the response buffer, in particular, in our case, the goal will be `(2::nat)+(2::nat)=(4::nat)`

```
lemma two_two:  "(2::nat)+2=4" by auto
```

Now, we can write `end` at the end of file, and save the file choosing File > Save as. There is a special condition that theory name is the same as file name, so we should save our theory with name "MyFirst.thy".

2.1 Summary

- Mathematical knowledge formalized by Isabelle consists of theory files *.thy, which form a theory hierarchy by importing each other using the command `imports`. Syntax: each theory starts with keyword `theory` followed by theory name, the command `imports`, then `begin`, followed by the body of the theory, ending with `end`.
- The body of a theory is a collection of definitions, lemmas and theorems. Every lemma consists of its statement and proof. Syntax: keyword `lemma`, then lemma name, then `:`, then the lemma's statement in quotes, then the proof.
- The proof may look like `apply(<method_1>) ... apply(<method_n>) done`. As we incrementally execute such a proof we can see what is left to prove in the response buffer. Syntax: the last expression `apply(<method_n>) done'` can be replaced by `by(<method_n>)`.
- `auto` is a method which tries to prove the statement automatically.
- Every constant or variable in Isabelle has a type. `nat` is the type used for natural numbers (syntax example: `(2::nat)`). We can use Isabelle > Settings > Show types in the menu to see all the types in the response buffer.

3 Main Notations

Now, let us have a closer look at lemmas in Fun.thy to learn the main notations of the Isabelle language. Let us start with the first lemma expand_fun_eq:

```
lemma expand_fun_eq:  "f = g ↔ (∀x. f x = g x)"
```

First of all, you can see that the lemma's statement uses some mathematical symbols which you cannot see on your keyboard. You can type such symbols using the Math menu, but I would not say that this is convenient. In the appendix of the tutorial there is a very useful table which shows the way to write most symbols using the keyboard in natural way. For example, the arrow $\Longrightarrow$ can be written as ==>, or even \<Longrightarrow>. Notations like ==>, which is called ASCII notation, are more convenient, and will be used in the rest of the primer. If menu option Proof-General > Options > Unicode Tokens is turned on, "==>" will be automatically transformed to "$\Longrightarrow$" each time you type it.[2]. You will get used to such notation very quickly and it will be absolutely no problem to understand mathematical text in this form. I would recommend to print the table with ASCII notation from the tutorial appendix now, and have it available until you get used to it.

In this notation $\leftrightarrow$ is written as <->, $\forall$ is denoted `ALL`. So, the lemma expand_fun_eq can be written as

```
lemma expand_fun_eq:  "f=g <-> (ALL x.  f x = g x)"
```

Now all the symbols are present on the keyboard and it is easy to type this lemma in. The meaning of the lemma is obvious: two functions f and g are equal if and only if $f(x) = g(x)$ for every argument x. Notice that $f(x)$ can be written without parenthesis. More complicated expressions with functions can also be written

[2]This is your choice, but I personally prefer to see on the screen exactly what I typed, therefore I usually turn off Unicode Tokens in the menu, and just use ASCII notations

without parenthesis, but you should be careful here: for example, expression `f t u` means `(f t) u` (and not `f (t u)`), and this expression can be used to denote a function of two variables f(t,u). Also, you should know the priorities of different operations: expression f x + y means (f x) + y and not f(x+y); equality has a high priority and `A & B = B & A` (symbol & means "and") means `A & (B=B) & A`, not `(A & B) = (B & A)`. In general, the priorities of the main logical binary connectives in decreasing order are &, | ("or"), `-->`, and they are associative to the right: `A --> B --> C` means `A --> (B --> C)`. You can find all these and many more rules and examples of this kind in the Isabelle tutorial, but I would recommend you to use parenthesis when you are not sure. If you write `(A & B) = (B & A)` this clearly indicates what you mean, and moreover looks nice.

Let us return to the lemma's statement. The phrase "for all x, we have f(x)=g(x)" is written as `ALL x. f x = g x`, where the full stop "`.`" replaces "we have". Full stop is always used after any quantifier, although we can write `ALL x y z. x+(y+z)=(x+y)+z` instead of `ALL x. ALL y. ALL z. x+(y+z)=(x+y)+z`. Another important use of full stop is in function definitions. For example, to define function $f(x) = x+1$ we can write `f=(%x. x+1)`. It is important to always leave a space after the full stop, or expressions like `ALL x.x` or `%x.x` will be understood incorrectly. Moreover, to be safe, I would recommend leaving a space after every special character. For example, the existential quantifier, $\exists$, denoted[3] as `?` should be followed by a space "`? x`" because the expression "`?x`" has a completely different meaning. It is used for *schematic variables*, or free variables, which can be instantiated arbitrarily. For example, the mathematical theorem `x=x` is represented in Isabelle as `?x=?x`, which means that you can instantiate this variable to any term of the given type.

Now we understand all of the notation in the statement of the lemma expand_fun_eq, so let us move to the next lemma in the Fun.thy theory, the lemma apply_inverse, which in ASCII notation takes the form

```
lemma apply_inverse:
"f x = u --> (ALL x.  P x --> g (f x) = x) --> P x --> x = g u"
```

This formulation is a little bit less intuitive. Expression `P x` here can be understood as a predicate, i.e. for every *x* expression `P x` is either true or false. Formally, this can be a function with Boolean values, or equivalently just a set, and in this case the notation `P x` is equivalent to `x:P`, which is the ASCII abbreviation for `x∈P`. In Isabelle there is a special *type constructor*, called `set`, which can be used to define relevant types: for example, any set of natural numbers has the type `nat set`. To define a set explicitly, we can use notation of the form "{x. x is such that...}", or even of the form "{f(x,y) | x y. x and y are such that...}". For example, strings `P::nat set` and `P = {x. x>10}` define the set of all natural numbers greater than 10.

Now all the symbols are understandable, but it may still be not obvious how to read this lemma, which has the form of a long logical formula. The point is that expression `A --> B --> C` is logically equivalent to `(A & B) --> C`, and the same is true for longer expressions. Thus, lemmas with assumptions A_1, A_2, ..., A_n which proves B can be written as $A_1 --> A_2 --> ... --> A_n --> B$. Now we can easily read the above lemma as follows: assume that (1) f(x)=u, (2) for every $x \in P$ we have g(f(x))=x, and (3) $x \in P$. Then g(u)=x. Now we can see that lemma is obvious, and it is not a surprise that it is proved automatically.

After two auxiliary lemmas, theory Fun.thy contains some definitions, the first of which looks like

```
definition id ::  " 'a ⇒ 'a " where "id = (λx. x)"
```

This defines an identity function f(x)=x where x is a variable of any type. In ACSII notation ⇒ becomes `=>` and λ becomes %. We can see, that the keyword `definition` is followed by the name of an object we want to define, then after `::` we indicate the type of this object, and then after `where` we list the defining equation of this object. We already know that arbitrary type is denoted by `'a`, thus the function from `'a` to `'a` has a type `'a => 'a`[4]. Then, after `where` we specify that id(x)=x by writing `id = (%x. x)`.

[3] Recommended ASCII notation of ∃ is `EX`

[4] Please notice that implication is denoted as `==>`, and the function type generator is `=>`, with a single equality sign.

Let us use this example to construct our own definition, for example, to define a function from real to real, which is nondecreasing on some interval S. First, we need to understand that this is actually a Boolean functional, which for every function f and set S will have value "True" or "False". The Boolean type in Isabelle is one of the base types and is denoted by `bool`. The formal definition of real numbers is somewhat complicated, so we will not discuss it here, but the corresponding type is called `real`. Now S has the type `real set`, `f` has the type `(real => real)`, so the type of our functional will be `real set => (real => real) => bool`. Let us give it the name `nondecreasing_on`. Now, `nondecreasing_on(S,f)` has a value "True" if and only if for all $x,y \in S$ such that $x \leq y$ we have $f(x) \leq f(y)$. We already know all the necessary notation from the previous examples, and can write this as

```
definition nondecreasing_on ::  "real set => (real => real) => bool"
  where "nondecreasing_on S f <-> (ALL x:S. ALL y:S. x<=y --> f x <= f y)"
```

However, if we try to execute this definition, we will obtain an error: `Undeclared type constructor: "real" At command "definition"`. The obvious guess after such an error would be that the type of real number is not denoted `real` in Isabelle. To check this, we can go to the directory with Isabelle's theory files (the folder you installed Isabelle in), then > src > HOL, and look for the corresponding theory. Fortunately, this is easy in this case, because there is a theory with name RealDef.thy, which is obviously what we want. If we look inside this theory, we will see that the type name is indeed `real`. Usually, if you see a definition or lemma in some theory, you can use it. Then if the system does not recognize it, it may be that you did not import the corresponding theory. Indeed it turns out that real numbers are not included by Main.thy, and one option is to add RealDef.thy to `imports` command at the beginning of your theory. But, to avoid similar problems with a next lemma, it is desirable to import the "latest" theory, accumulating as much of the library as possible. However, I cannot tell you the name of such a theory once and for all, because the library is growing continuously. The current theory dependencies graph can be found on Isabelle website: http://www.cl.cam.ac.uk/research/hvg/isabelle/dist/library/HOL/index.html. For now, let us replace `imports Main` by `imports Complex_Main`: theory Complex_Main.thy imports Main.thy together with many more theories containing, for example, all the basic properties of real and complex numbers. The definition can now be executed.

As you can see, reading just a few lemmas and one definition from a randomly chosen theory, Fun.thy, gives us enough notation to create our own nontrivial definitions. If we look at this theory further, we will be able to guess the meaning of almost all new mathematical notation in Isabelle. For example, one of the next lemmas is `image_ident` stating that `(%x. x) ' Y = Y`. We can guess that it states that image of the set Y under the identity function will be Y, whence symbol `'` is an important notation of the *image* of a set under a function (`f ' X` is by definition the set `{y. EX x:X. y = f x }`). Looking at the next few lemmas we can see that `-'` is a notation for *inverse image*, and the next definition introduces a useful notation for function *composition* `f o g` which means function "f(g(.))", or, in Isabelle notation, `(%x. f (g x))`. All the important functions and notations in `Main.thy` are listed in [3]. But it is impossible to learn all the notation at once. We now know more than enough to start proving theorems, and will learn more notation in the process.

3.1 Summary

- All the main mathematical symbols also have ASCII notation, as a way to type them on a keyboard. For example, $\Longrightarrow$ is `==>`, $\longrightarrow$ is $--$ >, $\leftrightarrow$ is <->, $\forall$ is `ALL`, $\exists$ is `EX`, $\leq$ is <=, `x:P` is ASCII abbreviation for `x`$\in$`P`, the symbol & means "and"; the symbol | means "or".

- Equality has a high priority; the priorities of the main logical binary connectives in decreasing order are &, |, $--$ >, and they associate to the right. Use parentheses when you are not sure.

- `bool` is the type for Boolean variables. `set` is the type constructor for sets, for example `nat set` is the set of natural numbers. Function types may be constructed using `=>`, for example `nat => bool`. Many standard types are defined in Isabelle, for example `real` is the type of real numbers.
- Full stop is used (1) with quantifiers, like `ALL x. f(x)=g(x)`; (2) to define functions, for example `f=(%x. x+1)`; and (3) to define sets, eg. `P = {x. x>10}`. In every case, a space should follow immediately after the full stop.
- The symbol ‘ is a notation of the image of a set under a function (`f ‘ X` is by definition set {`y. EX x:X. y = f x `}). Notation `f o g` means function composition f(g(.)).
- All the important functions and notations in `Main.thy` are listed in [3].

4 Automatic Proofs

The most important skill in proving mathematical theorems in Isabelle is the ability to prove simple lemmas with almost no effort. Every arbitrarily long proof can be represented as a chain of simple steps, and this representation is an interesting and fully mathematical task. But proving every simple step may be a bit problematic for new Isabelle users, as we saw on the example of lemma ”2+2=4”.

One of the main problems here is that the user often does not know what relevant lemmas exist in the library. If you want to prove some result from a particular area of mathematics, it is useful to look at the existing theories in this area before start. However, Isabelle also provides us with several ways to search through the library.

Given that we want to prove some lemma in Isabelle, the first question is what if exactly such lemma already exists in the library and we just did not know about it? For example, assume that we do not know the lemma `id_apply` in Fun.thy, stating that `id x = x`, and want to prove exactly the same lemma with name `apply_id` (recall that the function `id` is defined in `Fun.thy`, and it is the identity function).

```
lemma apply_id:  "id x = x"
```

If we execute this lemma statement, we see the following message in the response buffer:

```
The current goal could be solved directly with:
Fun.id_apply:  id ?x = ?x
```

This important mechanism of lemma suggestions can prevent you from reproving results which already exist in the library. If such a message does not appear, you may be sure that your result is new[5]. The search tries to look on the *meaning* of the lemma, not just symbol-by-symbol coincidence, for example if you change the name of the variable and write:

```
lemma apply_id:  "id y = y"
```

Isabelle will still suggest you use lemma `id_apply` to solve the goal directly. To use an existing lemma in your proof you can use the `using` command and write:

```
lemma apply_id:  "id y = y" using id_apply by auto
```

and the lemma will be proved. But, clearly, proving this lemma again has no sense: if you wanted to prove some lemma and the system found this lemma in the library, it is better just delete your lemma and use the existing one. Notice that we can use it just by lemma name, like `id_apply`, writing the full name like `Fun.id_apply`

[5] At least in theories that you have imported.

is possible but unnecessary, and moreover will make your theory unstable. It can stop working in the future versions of Isabelle if the library is reorganized.

Unfortunately, the lemma suggestion mechanism is currently very sensitive to your lemma's formulation. In particular, Isabelle regards equations as directed, and if we write

```
lemma apply_id:  "x = id x"
```

no lemma is suggested. This statement is actually a combination of two lemmas: id_apply and the fact that "a=b" is equivalent to "b=a". Fortunately, there is a tool in Isabelle, called Sledgehammer, which tries existing lemmas to automatically prove your goal. After executing the formulation of your lemma, choose Isabelle > Commands > Sledgehammer in the menu, and you will see the message

```
Try this command:  apply (metis id_apply)
```

which suggest to apply lemma id_apply to solve our lemma. If we click at this command in the response buffer, it will be added to the proof, and we get message `goal: No subgoals!`. So the proof is finished, we can write `done` to get

```
lemma apply_id:  "x = id x" apply (metis id_apply) done
```

Method `metis`, similar to `auto`, tries to prove the statement automatically. The difference is that metis uses only logical reasoning, but it is very strong in proving logical statements. To prove the correct logical formula, it is often enough to write `apply metis`. To prove that our statement is a logical consequence of some lemma, we need to write `apply(metis <lemma_name>)`, in our case `apply (metis id_apply)`. If we want to prove corollary from several lemmas we should write all of them, for example `apply (metis id_apply two_two)`. But the point is that we wrote the proof above using sledgehammer having no idea about metis, and (theoretically) without knowledge that the relevant lemma "id_apply" already exists in the library.

Sometimes sledgehammer needs some time to find a proof, but it can work in the background, and you can continue to work on your proof in parallel. Sometimes, it gives back a message that it cannot find a proof. This may indicate that your lemma is nontrivial and new, which would be the ideal case. Unfortunately, sledgehammer is not always helpful even in simple cases, especially if lemmas involve quantifiers. For example, if we formulate the lemma

```
lemma expand:  "h = t <-> (ALL y.  h y = t y)"
```

which is just a reformulation of lemma `expand_fun_eq` in `Fun.thy`, the lemma suggestion mechanism immediately tells you that

```
The current goal could be solved directly with:
Fun.expand_fun_eq:  (?f = ?g) = (ALL x.  ?f x = ?g x)
```

But now, if you replace `h y = t y` by `t y = h y` and write

```
lemma expand:  "h = t <-> (ALL y.  t y = h y)"
```

no lemma is suggested automatically, and sledgehammer also gives the message `External prover failed`. For this reasons it is desirable to formulate all your lemmas in the most natural way, without unnecessary changes of order in equalities, etc. Hopefully, the lemma suggestion mechanism and sledgehammer will be significantly improved in future versions of Isabelle, and all the results that are trivial consequences from the existing ones will be proved automatically, without any effort from the user.

4.1 Summary

- When we execute the formulation of a lemma, we may get the message `The current goal could be solved directly with:...` which implies that this lemma is not new but is just an instance of an existing result in the Library.
- Sledgehammer is a mechanism which tries to combine two or more existing lemmas to prove the goal. After executing the formulation of your lemma, choose Isabelle > Commands > Sledgehammer in the menu, and, if successful, you will see the message like `Try this command: ...`
- To use an existing lemma in your proof you can use the `using` command and write `using <lemma_name> apply(<method>)`.
- The proof method `metis`, similarly to `auto`, tries to prove statements automatically. It is very good at proving logical statements.

5 Interactive Proof

In this section, we start writing proofs, which Isabelle cannot do completely automatically. Let us start from a simple example: suppose we want to prove a simple formula from school algebra: $(a+b)^2 = a^2 + 2ab + b^2$.

First, we should formulate the lemma carefully. We know from the example `2+2=4` that the type of variables should be specified explicitly. Furthermore, we cannot omit the multiplication symbol `*` and write `ab`, because Isabelle will understand this as a new variable. With this in mind, it is easy to formulate the lemma:

```
lemma sum_square:  "(a+b)^2=a^2+(2::real)*a*b+b^2"
```

If we execute this lemma statement, no suggestion appears in the response buffer. Sledgehammer also cannot help here (message `Interrupted (reached timeout)` appears), so we need to prove the lemma by hand.

So far, we know only two automatic proof methods `auto` and `metis`. To prove the lemma above, we will need a third one: `simp`. This method is a powerful simplifier, which tries to simplify your expression using hundreds of lemmas and simplification rules. Actually, the lemma `id_apply` from `Fun.thy`, considered above, is formulated as follows:

```
lemma id_apply [simp]:  "id x = x" by (simp add: id_def)
```

The attribute `[simp]` after lemma formulation states that this lemma will be automatically added to those ones which the `simp` method will use. Namely, if `simp` finds expression like `id <expression>`, it will simplify it and rewrite as just `<expression>` using this lemma. Your can easily use the `[simp]` attribute to add any of your lemmas to those which are used by `simp`. But this should be done very carefully: if you add lemma like `a+b=b+a`, `simp` may use it again and again, working forever. It is recommended to add only those lemmas which really simplify expressions, in the sense that the right-hand side is simpler that the left-hand side. If you want the simplifier to use some other lemma(s) in a particular case, you can use the `add` command. For example, expression `by (simp add: id_def)` in the example above proves lemma `id_apply` by the simp method, which uses all the lemmas with the `[simp]` attribute plus the definition of the identity function `id_def`. In general, if we write any definition in Isabelle, the corresponding lemma with suffix `_def` is automatically created, and we can use this lemma in future proofs by writing commands like `by (simp add: id_def)` or `using id_def by auto`.

There are many lemmas which are not added to `simp` by default, but which are very useful in some particular cases. For example, to simplify expressions involving addition and multiplication (or, more generally, any

group, ring, or field equalities), it is useful to add algebra_simps to the simplifier by writing apply (simp add: algebra_simps).

Let us return to proving our lemma sum_square. It involves addition and multiplication, but immediately writing apply (simp add: algebra_simps) fails to prove the lemma, because it involves also operation of power. First, we should explain to Isabelle, that a^2 means a*a and so on. This fact is so simple, that it should be in the library for sure, but how to find it? One way is to use the lemma suggestion mechanism and write somewhere above

```
lemma "a^2=(a::real)*a"
```

execute this, and the system will suggest you that

```
The current goal could be solved directly with:
Nat_Numeral.monoid_mult_class.power2_eq_square:  ?a^2 = ?a * ?a
```

Sometimes, however, it is hard to guess the exact lemma formulation. In this case you can use ProofGeneral > Find Theorems item in the menu, where you can search for theorems, say, by name, writing name: <name_to_find> in the string below. For example, in our case, we can guess to search for name: square and find all the lemmas containing square in the name. In this case we will get the following message:

```
found 57 theorems (40 displayed) in 0.146 secs:
```

This message indicates that not all found lemmas are displayed. We can modify the search and state explicitly how many lemmas we want to be displayed: if we search for (100) name: square, now all the lemmas containing square in the name are displayed, including the relevant power2_eq_square lemma.

If we cannot guess part of lemma's name, we can also search for part of its formulation. For example, to find all the lemmas containing product of some number by itself we can (after choosing ProofGeneral > Find Theorems item in the menu) search for "?a*?a", and now 44 theorems are found, including the power2_eq_square lemma. If too many lemmas are found, we can combine these approaches: search for name: square "?a*?a" result in just 20 lemmas which have both "square" in the name and expression like "a*a" in the formulation. Another relevant attempt is the query "?b^2" "?a*?a" which searches for lemmas containing both these expressions at the same time: now only three lemmas are found.

After finding lemma power2_eq_square, we are almost done. First, tell the simplifier to transform squares to multiplication by writing apply (simp add: power2_eq_square)

```
lemma sum_square:  "(a+b)^2=a^2+(2::real)*a*b+b^2"
   apply (simp add: power2_eq_square)
```

After executing this, we can see the following in the response buffer:

```
goal (1 subgoal):  1.  (a + b) * (a + b) = a * a + 2 * a * b + b * b
```

This means that, after simplification, it is left to prove the statement above. Now it uses only addition and multiplication, we can write apply (simp add:algebra_simps), execute to see message goal: No subgoals!, and finish the proof by command done. The resulting proof looks like

```
lemma sum_square:  "(a+b)^2=a^2+(2::real)*a*b+b^2"
   apply (simp add: power2_eq_square)
   apply (simp add: algebra_simps)
done
```

The resulting proof is called *backward proof*. At every step we apply the relevant method to simplify the goal, and see what is left to prove in the response buffer. But, first, the resulting proof script is hard to understand: after reading the long sequence of `apply` commands which proves hard theorem, it is hard to say what was the main idea of the proof. Second, most of the proofs in mathematical papers rarely use the argument "it is left to prove that". The typical proof usually looks like "From definition we have Statement 1. Also, from well-known lemma it follows Statement 2. Now, from these 2 statements we can conclude Statement 3, and the proof follows". This is called *forward proof*, or *declarative proof*, and it is also supported in Isabelle. Let us use the lemma above to prove that $x^2+6x+9 \geq 0$.

```
lemma expression_nonneg:  "x^2+(6::real)*x+9 >= 0"
```

As a first step of proof, we want to say that $x^2+6x+9 = (x+3)^2$. For Isabelle, this will be a sublemma inside the proof of our lemma. To formulate such a sublemma, command `have` is used, and to indicate that we are inside the proof of lemma `expression_nonneg`, we should write `proof-` before `have`:

```
lemma expression_nonneg:  "x^2+(6::real)*x+9 >= 0"
proof-
   have aux:  "x^2+(6::real)*x+9 = (x+3)^2"
```

Here `aux` is the name of the sublemma which will be used for future references. To prove this, we need to substitute `x` and `3` into the lemma `sum_square`. This can be done using the attribute `of` which has format `<lemma_name> [of <var1> <var2> ... <varn>]`:

```
have aux:  "x^2+(6::real)*x+9 = (x+3)^2" using sum_square [of x 3] by auto
```

Now we need to find a lemma stating that full square is nonnegative. It is logical to assume that such a lemma has `square` as a part of name. To find it, go to ProofGeneral > Find Theorems in the menu,then write `name: square`, and in resulting list we can see the lemma we need:

```
Rings.linordered_ring_strict_class.zero_le_square:  (0::?'a) <= ?a * ?a
```

In our case variable `?a` should be $x+3$. To see the result of the substitution we can use `thm` command

```
thm zero_le_square [of "x+3"]
```

Please notice that expressions like `x+3` should be in quotes when used after `of`. Executing the string above, we can see in the response buffer

```
0 <= (x + 3) * (x + 3)
```

as expected. So we can write

```
have "(x+3)^2 >= 0" using zero_le_square [of "x+3"] by auto
```

and the auxiliary `thm` command can now be erased.

Alternatively, we can try to use Sledgehammer to prove any of our subgoals. After executing the expression after have, we can choose Isabelle > Commands > Sledgehammer in the menu, and for the statement `(x+3)^2 >= 0` it works: you can see the message

```
Try this command: apply (metis zero_le_power2)
```

which also finishes the proof of the statement immediately.

Now we want to say that the lemma expression_nonneg follows from these two statements. The first one has name `aux`, and the last one can be referred using keywords `from this`. The phrase "and lemma follows" in Isabelle language looks like `show ?thesis`, and we can write:

```
from this show ?thesis using aux by auto
```

After executing we see that the proof is correct, and then the declarative proof should be finished by the command qed. The resulting proof looks like:

```
lemma expression_nonneg:  "x^2+(6::real)*x+9 >= 0"
proof-
   have aux:  "x^2+(6::real)*x+9 = (x+3)^2" using sum_square [of x 3] by auto
   have "(x+3)^2 >= 0" using zero_le_square [of "x+3"] by auto
   from this show ?thesis using aux by auto
qed
```

In contrast to the backward proof of lemma sum_square, this proof is clear for the reader. For readability of large proofs, it is important to indent the proof text between proof- and qed. Moreover, you can improve the language of the proof by replacing from this by then, from this have by then have or hence, from this show by thus, etc. However, backward proof can sometimes be easier to write: you just formulate the goal and see how far the appropriate automated method can proceed. So, maybe the best strategy would be to combine forward and backward proof methods. For example, you may prefer to state some major sublemmas in your proof using have and then prove these sublemmas by the backward strategy using apply.

Sometimes it is convenient to exchange the order of proving sublemmas in a forward proof. For example, if the proof of statement B has the form have A hence B, we may prefer first to prove the second statement (namely, that A implies B) and then return to proving A. Isabelle provides you with this opportunity with the help of sorry command, which ”proves” everything. For example, if we would like to use lemma expression_nonneg first, and then return to proving it, we could temporary ”prove” it in one line:

```
lemma expression_nonneg:  "x^2+(6::real)*x+9 >= 0" sorry
```

However, sorry command should be used with care, because you can ”prove” a false statement with it, and then build your proof based on this statement.

5.1 Summary

- Method simp is a simplification method which tries to simplify the goal automatically. It uses all the lemmas in the library marked with the [simp] attribute. To make simp also use another lemma, we should add it explicitly, by writing apply(simp add: <lemma_name>). For example, apply (simp add: algebra_simps) is useful to simplify expressions involving addition and multiplication.

- If you can guess a part of the name of the lemma you want to use, choose ProofGeneral > Find Theorems item in the menu, and then write name: <name_to_find> in the string below. You can also indicate how many found lemmas to display: for example search for (100) name: square will result in up to 100 lemmas with ”square” as a part of the name.

- Similarly, if you can guess a part of the lemma statement, choose ProofGeneral > Find Theorems, and then search for any expression, or several expressions, using schematic variables. For example, search for "?b^2" "?a*?a" will result in the list of lemmas containing both a square and a product of a term with itself.

- There are two main strategies in proving results. Proof in the form apply(<method_1>) ... apply (<method_n>) done is called backward proof. Alternatively, you can use the forward strategy which looks like proof- have <statement_1 > ... have <statement_*n* > qed, where the statements after have should, in turn, be proved using backward or forward strategies, and, perhaps, the earlier proved statements. Also, at every step you can use Sledgehammer to try to prove the statement automatically.

- To substitute particular values for the variables in an earlier proved lemma, you can use the attribute `of`, for example `...using sum_square [of x 3] by auto`.
- To use the last proven fact you can write `from this have`, or `then have`, or `hence`.
- At the end of forward proof, you should write `show ?thesis`, finish the proof, and then write `qed`.
- To exchange the order of reasoning in a forward proof, you can temporarily "prove" any statement or lemma with command `sorry`.

6 Assumptions and Local Variables

In a usual human proof, the same variables and notations are used to denote, strictly speaking, different objects, and in this section we explain how this works in Isabelle. For example, let us write down the proof of a simple well-known result, that point x in metric space belongs to the interior[6] of set S if and only if S contains a ball with center x.

Proof: If x belongs to interior S, then, by definition, x belongs to some open subset T of S. Because T is open, it contains a ball B with center x, and we have $B \subseteq T \subseteq S$. Vice versa, if S contains a ball with center x, this ball is an open subset of S which contains x, hence, by definition, x belongs to interior S.

We can see that the proof consists of two parts, and in first part x is an arbitrary point belonging to interior of S, while in the second one we assume that "S contains a ball with center x", and this creates no confusion. Let us try to formalize the proof above.

First, we can find if the notion of interior is defined in Isabelle. If you import `Complex_Main` only, searching for `name: interior_def` or even `name: interior` will show no results. Fortunately, basic notions of topology was recently formalized in the context of a big project of multivariate analysis formalization in Isabelle. If you add `imports Multivariate_Analysis` [7] at the beginning of your theory, searching for `name: interior_def` will lead you to the result:

```
interior ?S = {x.  EX T. open T & x:T & T <= ?S}
```

Similarly, search for `name: ball_def` will result in `ball ?x ?e = {y. dist ?x y < ?e}`, so we can conclude that `ball x e` is the ball with center x and radius e. Now we can formulate the lemma.

```
lemma interior_ball:  "x:interior S <-> (EX e.  0 < e & (ball x e) <= S)"
```

There is no need to specify that x is an element of a metric space, and e is a real number: Isabelle understands this from string `ball x e`. If we execute this lemma formulation, we will see that `The current goal could be solved directly with: Topology_Euclidean_Space.mem_interior`, i.e. that this lemma already exists in the library, but we will ignore this and prove it by hand.

The proof starts with the assumption "If x belongs to interior S...". In Isabelle, there is a corresponding command `assume`. To tell Isabelle which part of proof this assumption affects, we should enclose it in braces {}. The general way to prove the implication `A --> B` is to write {`assume A...  have B`}. Also, we can write several assumptions $A_1, ..., A_n$ to derive $(A_1 \& A_2 \& ... \& A_n) --> B$.

So, let us start the proof by writing

```
proof-
{ assume "x:interior S"
```

[6] By definition, interior of S is the set of all points x, such that there exists an open subset of S which contains x

[7] If you get an error `Could not find theory file "Multivariate_Analysis.thy" in...`, please go to the folder in which you have installed Isabelle and execute the command `./build -m HOL-Multivariate_Analysis HOL`

The next step is to define T such that $x \in T$, T is open, and $T \subseteq S$. This can be done by commands `obtain` and `where`:

```
from this obtain T where T_def:  "open T & x:T & T <= S" using interior_def by
auto
```

The line above does not just introduce a new notation T, but it proves that such a T actually exists. This fact follows directly (`by auto`) from the assumption (`from this`) and the definition of interior (`using interior_def`), and we gave it a name (`T_def`) for future reference.

Next, we want to say that "Because T is open, it contains a ball B with center x". The corresponding lemma can be easily found, say, by searching for "name: ball" and it is called `open_contains_ball`. Thus, we can write

```
hence "EX e.  e>0 & ball x e <= T" using open_contains_ball by auto
```

Here `hence` is the same as `from this have`, so we have proved this statement by auto using lemma `open_contains_ball` and the previous line stating that T is open. Now we can use the fact `T <= S` to conclude that `ball x e <= S`, and then finish the first part of proof

```
  hence "EX e.  e>0 & ball x e <= S" using T_def by auto
} note imp1 = this
```

All the statements inside the block are conditional with respect to the assumption. The last statement before closing parenthesis } should not contain any local notation like T. Then we can close the block, and Isabelle automatically derives the unconditional statement of the form "assumption –> last statement inside the block", which can be referred to using `this`. Command `note imp1 = this` gives it a name `imp1` for future references.

To prove the converse statement, we assume that S contains a ball with center x, denote it T, and claim that this T is exactly what we need to prove that `x : interior S` by definition.

```
{ assume "EX e.  e>0 & ball x e <= S"
  from this obtain e where e_def:  "e>0 & (ball x e) <= S" by auto
  def T == "ball x e"
  hence "open T & x:T & T <= S"
```

From the assumption we "obtain" radius $e > 0$ such that `(ball x e) <= S`, and then just introduce a notation T for (ball x e) in command `def`. The syntax is `def <name> == "<description>"`. In contrast to `obtain`, command `def` does not require any proof of existence. Now, to prove that T is open we need to find the corresponding lemma `open_ball` in the library, statement `T <= S` follows from `e_def`, and the obvious fact `x : T` can be proved automatically. So, the rest of the proof is easy:

```
  hence "open T & x:T & T <= S" using open_ball e_def by auto
  hence "x:interior S" using interior_def by auto
} from this show ?thesis using imp1 by auto
qed
```

The last `this` here refers to the fact that from assumption `EX e.  e>0 & ball x e <= S` the last statement of the block `x:interior S` follows. This together with `imp1` finishes the proof.

The proof of this lemma in Isabelle library is shorter, but this proof illustrates how to use assumptions in Isabelle, which will be very useful in other proofs. For example, the general method to prove that `A <-> B` is to derive B from A using format {`assume A...  have B`}, and then derive B from A using {`assume B...`

have A}. Also, to prove that, say, two sets S and T are equal, the straightforward way is {assume "x:S"... have "x:T"} to derive (x:S) --> (x:T), then prove the opposite statement to derive (x:T) --> (x:S), and finally use lemma expand_set_eq which states that (?A = ?B) = (ALL x. (x:?A) = (x:?B)).

For example, our lemma interior_ball can be used to obtain the following equivalent characterization of the interior:

```
lemma interior_def2: "interior S = {x. EX e. e>0 & (ball x e) <= S}"
```

For a proof, we need to use lemma interior_ball for a particular S, but for arbitrary x. This can be done using the of attribute in the following format: interior_ball[of _ S]. To derive lemma interior_def2 from this statement, it is left to apply lemma expand_set_eq.

```
lemma interior_def2: "interior S = {x. EX e. e>0 & (ball x e) <= S}"
   using interior_ball[of _ S] by (simp add: expand_set_eq)
```

6.1 Summary

- To use some additional assumption during the proof, one can use command assume. In this case, we should enclose the block where this assumption applies in braces {}. The general way to prove the implication A --> B is to write {assume A... have B}.
- Command obtain <object_name> where <thm_name>: ... allows us to obtain an object with a particular properties. To finish this command, we need to prove that such an object exists, and then we can use this object as <object_name> during the proof (say, substitute it as a parameter after of attribute), referring to its properties as to <thm_name>.
- Command def with format def <name> == "<description>" just introduces a notation and does not require any proof of existence. We can refer to this definition using <name>_def.
- All statements inside a block enclosed in braces {} are conditional with respect to assumptions, and all variables defined inside such a block are local ones. No such variable can participate in the last statement of the block. We can use variables with the same names in different blocks.
- To prove that two sets S and T are equal, one can first prove that (x:S) <=> (x:T), and then use lemma expand_set_eq which states that (?A = ?B) = (ALL x. (x:?A) = (x:?B)).

7 Introducing New Notations and Concepts

Convenient definitions and notation are crucial in proving mathematical results. For example, suppose we want to prove, that for any convex sets A, B, C, the set

```
{x + y + z |x y z. x:A & y:B & z:C}
```

is also convex. Search in Isabelle (for example, by name: convex, provided that you import theory Convex) results in lemma convex_sums stating that for two convex sets A and B set {x + y |x y. x:A & y:B} is convex. Now it is natural to define sum A+B of two sets, and then argue that for convex sets A, B, C, set A+B, and whence A+B+C = (A+B)+C is convex by lemma convex_sums.

Now, how to introduce new notation? As usual, you can read the tutorial, but it is easier to look at the existing theories. For example, our favorite theory Fun.thy introduces a notation for function composition

```
definition
  comp ::  "('b => 'c) => ('a => 'b) => ('a => 'c)" (infixl "o" 55)
  where "f o g == (%x.  f (g x))"
```

We see, that composition is actually a function `comp(f g)` with two arguments: f of type `('b => 'c)` and g of type `('a => 'b)`, which returns the result of type `('a => 'c)`. String `(infixl "o" 55)` introduces notation `f o g` for this function. Using this example, we can try to define sum of two sets: it should be a function like `set_add(A B)`, where A and B are sets with elements of the same type, and we want a notation, say, `A +s B`

```
definition   set_add ::  "'a set => 'a set => 'a set" (infixl "+s" 55)
  where "A +s B == {x + y |x y. x:A & y:B}"
```

But, if we try to execute this definition, we get an error

```
Type unification failed:  Variable 'a::type not of sort plus
```

This is quite common error in Isabelle, stating that the types are not appropriate (not of correct "sort") for this context. For example, in our case Isabelle tells us that addition can not be defined for arbitrary sets, and elements of our sets should have a type with a special sort `plus`. To correct this error, we can easily specify sort explicitly

```
definition   set_add ::  "('a::plus) set => 'a set => 'a set" (infixl "+s" 55)
  where "A +s B == {x + y |x y.  x:A & y:B}"
```

and now the definition executes correctly[8].

Next, let us formulate and prove lemma about sum of two convex sets in the new notation. As usual, to learn the corresponding syntax, let us first look at the statement and beginning of the proof of the existing lemma `convex_sums` in theory `Convex.thy`.

```
lemma convex_sums:
  assumes "convex s" "convex t"
  shows "convex {x + y |x y. x:s & y:t}"
  using assms unfolding convex_def image_iff
proof- ...
```

We can see, that assumptions of the lemma can be formulated after keyword `assumes`, and then we can refer to them during the proof by writing `using assms`. The statement of the lemma in this case follows the keyword `shows`. Let us formulate the corresponding lemma with notation `"+s"`.

```
lemma convex_sums2:
  assumes "convex A" "convex B"
  shows "convex (A +s B)"
```

Since it is just a reformulation of lemma `convex_sums`, it is natural to try to prove this lemma automatically, using the existing lemma, assumptions, and definition of `"+s"`. However, an attempt

[8] It may be confusing that we still do not understand the meaning of string `(infixl "+s" 55)`. Actually, now it is easy to use the Tutorial's Index to find `infixl` there, and read that `infixl` means that operation is associative to the left, and `55` indicates the priority of the operation

```
using set_add_def assms convex_sums by auto
```

does not work, so more details are required. One way is to tell Isabelle the exact parameters to substitute in `set_add_def`, namely

```
using set_add_def[of A B] assms convex_sums by auto
```

Alternatively, we can proceed by analogy with the proof of lemma `convex_sums` above, and use `unfolding` command:

```
lemma convex_sums2:
  assumes "convex A" "convex B"
  shows "convex (A +s B)"
  unfolding set_add_def using assms convex_sums by auto
```

After executing the statement of the lemma we can see goal `convex (A +s B)`. Then, after executing `unfolding set_add_def` the remaining goal is `convex {x + y |x y. x:A & y:B}`, i.e. this command "unfolded" the definition. Next we can execute `using assms convex_sums by auto` and the proof is successfully finished.

Now we can prove the initial lemma

```
lemma convex_sums3:
  assumes "convex A" "convex B" "convex C"
  shows "convex {x + y + z |x y z. x:A & y:B & z:C}"
```

First, we want to claim that the set in question is exactly `A +s B +s C` in the new notation.

```
proof-
  have "{x + y + z |x y z. x:A & y:B & z:C} = (A +s B +s C)"
  unfolding set_add_def by auto
```

Then we can claim that set `A +s B +s C` is convex, prove this using `convex_sum2`, and the lemma will follow from these two statements. To tell Isabelle that one statement follows from *several* previous ones, we can use `moreover` and `ultimately` commands:

```
moreover have "convex (A +s B +s C)" using convex_sum2 assms by auto
ultimately show ?thesis by auto qed
```

In this case, convenient notation for sum of sets helped us prove the desired result easily. However, you should be very careful when introducing new definitions and notation in Isabelle. The problem is that notation for many natural concepts already exists somewhere in Isabelle, but it is often nontrivial to find it. Introducing several different notations for the same concept will often result in double work in theorem proving. For example, in our case, definition of sum of two sets really exists in Isabelle library, in theory `SetsAndFunctions.thy`, but the formulation is slightly different

```
definition
  set_plus ::  "('a::plus) set => 'a set => 'a set"(infixl "⊕" 65)
  where "A ⊕ B == {c.  EX a:A. EX b:B. c = a + b}",
```

and it is nontrivial to find it there.[9] For this reason, it helps to spend some time to look through the library to be aware of what concepts it offers. You can also ask the Isabelle mailing list, if you are going to introduce some important concept and do not see it in the library. In general, your are welcome to ask this mailing list all kinds of questions, somebody will answer you soon and in details. You can subscribe to it at http://www.cl.cam.ac.uk/research/hvg/Isabelle/community.html. At the same page, there is a useful link to Isabelle FAQ, and also links to various materials (slides, demos, and exercises) for learning Isabelle.

7.1 Summary

- Types in Isabelle have `sort`, which tells us some properties of this type. For example, addition is defined for types of sort `plus`. Syntax example: `('a::plus) set` is a set of arbitrary elements, for which addition is defined.
- Assumptions of a lemma can be formulated after keyword `assumes`, and then we can refer to them in the proof by writing `using assms`. The statement of the lemma in this case follows the keyword `shows`.
- Command `unfolding` with format `unfolding` <`lemma_name`>, where <`lemma_name`> is often a definition, tries to "unfold" symbol or term in the goal using <`lemma_name`>.
- Isabelle mailing list is the place where you can ask any questions about Isabelle. You can subscribe at http://www.cl.cam.ac.uk/research/hvg/Isabelle/community.html

8 Summary

This Primer is aimed for mathematicians, who want to start working with Isabelle. We have discussed only a tiny portion of Isabelle here, but it is enough to start formalization of some simple mathematical results. We have tried to concentrate on topics, which are especially useful for a beginner: main notations, search in Isabelle, sledgehammer, organization of blocks inside the proof, etc. More importantly, we have tried not just tell you "how it works", but tell you how to *learn* Isabelle, looking at the existing theories. Instead of providing you with correct proofs immediately, we often start with intuitive, but incorrect versions, and describe how to correct the resulting errors.

Obviously, you will often need some methods which are not described here. In this case, one general strategy is to look at the existing theories formalizing the same area of mathematics, and maybe you will find relevant methods there. Reading selected sections from Isabelle tutorial [1], which corresponds to your particular formalization, is also useful. Proof by induction, as well as some other useful proof methods, is described very well in a short Isar tutorial [2]. Many basic types, functions, and notations are listed in [3]. These and other useful documents are available at Documentation section of the Isabelle website http://www.cl.cam.ac.uk/research/hvg/Isabelle/documentation.html. Finally, many Isabelle users are ready to help you, if you send your question to the Isabelle mailing list.

In conclusion, formalization of mathematics in Isabelle is a little bit difficult to start, but very exciting. After some time, you become comfortable with Isabelle, and then enjoy proving nontrivial theorems to the strongest opponent in the world, who would never overlook your error or non-strict argument. And maybe, after some time with Isabelle, you also begin to feel, that only formalized theorems are really *proved* in mathematics. All the other proofs are just proof outlines.

[9] Well, you can guess import Library, and then search for, say, `"?x + ?y" "{x. ?P x}"`, to find all the expressions which contains sum, and also defines a set in the form "set consists of all x such that P(x)", and in this case you would found 24 theorems including `SetsAndFunctions.set_plus_def`. But it is a nontrivial guess to perform such a search.

References

[1] Nipkow, T, Paulson, C., Wenzel, M., Isabelle/HOL: A proof Assistant for Higher-Order Logic, http://www.cl.cam.ac.uk/research/ hvg/Isabelle/documentation.html

[2] Nipkow, T, A tutorial Introduction to Structured Isar Proofs, http://www.cl.cam.ac.uk/research/ hvg/Isabelle/documentation.html

[3] Nipkow, T, What's in Main, http://www.cl.cam.ac.uk/research/ hvg/Isabelle/documentation.html