A Practical Guide to Formalised Mathematics in Isabelle

Alexander Hicks, José Siqueira

February 5, 2017

Abstract

The goal of this guide is to provide an introduction to formalising mathematical theorems in the Interactive Theorem Prover (ITP) Isabelle. We review the basic features and mechanisms of Isabelle and provide some discussion from the point of view of mathematicians with no previous experience is interactive theorem proving. This guide is written for Isabelle 2016.

1 Motivation

The motivation for this guide comes following a few weeks in the Computer Laboratory (CL) at the University of Cambridge. Both authors come from a purely mathematical background, with no prior experience with interactive or automatic theorem proving.

There have been some high profile discussion on this topic in the Mathematics community. Fields medallist Vladimir Voevodsky takes care to formalise some of his work and has discussed this in [1] [2]. Another Fields medallist, Timothy Gowers, experimented with a human style output automatic theorem prover [3] [4] and presented a talk on the topic [5]. Nonetheless, one wouldn't expect the average student to hear about it in a Mathematics department and the development of formal verification comes from Computer Science departments, with not much of the documentation written by or for mathematicians — in the case of Isabelle, there exists one example: *Isabelle Primer for Mathematicians* [6] which dates back to 2010 and is worth reading as a short introduction with useful examples presented in more detail than here — do not, however, that some parts may now be out of date.

Expert use of Isabelle certainly requires some familiarity and knowledge that can only come with time. However, we believe any mathematician should be able to jump in and do *some* work with little practice and the purpose of this guide is to provide a short introduction to facilitate this. The authors do not consider themselves experts with practical advice given instead of thorough explanations. In an effort to keep this general, examples specific to some area of mathematics or library are avoided whenever possible in favour of general syntax and advice, except for the occasional illustration. The Isabelle Library provides a large amount of examples which should be looked at to illustrate the advice in an area specific to the readers work. References are included for the interested reader who may want more on a given topic, and two Appendices were added, on λ-calculus and types. Readers unfamiliar with either are suggested a quick look.

2 Formalised Mathematics and Interactive Theorem Provers

A large part of Mathematics revolves around theorems and their proofs, the latter published on the arXiv and in peer reviewed papers. The validity is checked by other mathematicians who are all human and fallible. Whether due to unintentional mistakes or the complexity of the work, errors happen, and examples through history of disagreements and controversy are plentiful, with many instances of famous problems such as the proof of Fermat's Last Theorem going through several revisions after their first announcement. With the modern ease of publishing preprints online, prestigious problems like the Navier-Stokes existence and smoothness have had multiple erroneous proofs submitted.

Some results also exist but are not considered valid for long periods of time due to the difficulty in checking them, such as the work of Shinichi Mochizuki on Inter-Universal Teichmller theory (IUT) which claims a prooof of the *abc conjecture* — IUT Mochizuki's work. Formalised mathematics would address both of the above points and there is some belief that it could become the new standard of rigour [7]. A theorem has been formalised once its proof has been written in a formal language, each statement checked and verified. In this case we do this by writing the proof within an interactive theorem proving environment such as Isabelle, Coq and HOL. These are built with the basic logical foundations needed and subsequent work is built on top of this: any prerequisite to a theorem's proof must then be proven beforehand, and thhus any proof in an ITP would be verified (even if not fully understood) as long as the ITP is built correctly, which is easier to verify. The collection of formalised results form (increasingly large) libraries that allow more and more work to be done. For a history of interactive theorem proving, see Harrison et al. [8].

In more recent times we have a few examples of important theorems whose story is linked to interactive theorem proving:

- The four colour theorem was formulated in the 19th century. Some early proofs by Alfred Kempe (1879) and Peter Guthrie Tait (1880) were not shown to be wrong for over a decade each. Appel Kenneth, Wolfgang Haken and John Koch published a proof in 1977 [9] [10]. The work was completed with the aid of a computer (checking possible counter-examples individually), before any correct traditional proof. As the first of its kind, it caused some controversy as the program used had to be verified. The theorem was later formalised in 2005 using Coq by Georges Gonthier [11];

- The Kepler conjecture was formulated in the 17th century. Much like the four colour theorem, it went through some incomplete proof by Wu-Yi Hsiang in the late 20th century. It was solved with the aid of a computer by Thomas Hales in 1998 and presumed to be correct, but the computer calculations were hard to verify. The proof was recently formalised by Hales and many collaborators in HOL Light and Isabelle [12];

- Gödel's incompleteness theorems are perhaps more famous than the previous two, and were of course proved by Gödel himself. There have been various formalisations of the first theorem, with first formalisation of both due to Lawrence Paulson [13] and completed in Isabelle. Given the history of these theorems and the work done around them, having a formalised proof that can be browsed and makes all steps explicit is of interest;

- Gödel's ontological proof for the existence of God was similarly formalised. Notably, an inconsistency can be found in Isabelle for one of the arguments but also avoided by reconstructing the argument [14].

These are only a few amongst many theorems that now benefit from being formalised. To see more examples, simply browse the libraries of existing theorem provers such as the Archive of Formal proofs for Isabelle.

Finally, note that in this section we have only talked about interactive theorem proving and not automatic theorem proving. The goal of an automatic theorem prover (ATP) is to automatically produce a full proof for a theorem whilst an ITP requires the user to write the proof himself. An ITP makes it possible to produce long and complex proofs whilst an ATP generally struggles once the proof becomes less mechanical. However an ITP can include some automatism in practice, as is the case with Isabelle due to the sledgehammer tool. Automatic theorem proving is a very interesting topic on its own, but will not be discussed in detail.

3 Isabelle

3.1 Installation and set up

Before starting anything, here are a few resources:

- Homepage: http://isabelle.in.tum.de/index.html
- Library: thttp://isabelle.in.tum.de/dist/library/
- Archive of Formal Proofs (AFP): https://www.isa-afp.org/
- Mailing list: https://lists.cam.ac.uk/mailman/listinfo/cl-isabelle-users
- StackOverflow: thttp://stackoverflow.com/questions/tagged/isabelle

Isabelle 2016 can be downloaded from the homepage. The installation is straightforward and comes with all the standard libraries which are contained in the installation folder under *Isabelle2016/src/....* Other theories (e.g from the AFP) can be downloaded and added to this folder in order to be used. Note that some entries to the AFP have already been worked into the standard libraries.
The users mailing list and stackoverflow are both used to address various issues or topics that users want to discuss. Both are active and helpful.

3.2 The Isabelle work environment

Isabelle looks and functions as an integrated development environment (IDE). Once you open up the software you should see a window split into 3 parts:

- A large text window where you will edit your theory
- Four tabs to the right. The first shows documentation (the original Isabelle manual). Sidekick allows you to browse the subsections and lemma of your theory. State shows state of the proof you are in (i.e goals, subgoals). Theories gives you a list of the theories you have imported (and their status as they load), clicking on a particular theory takes you to it.

- At the bottom, another four tabs. Output displays the result of anything you've done including error messages and completed lemma. Query will show the results of any search, such as a list of lemma. Sledgehammer show the output of the ATP tool of the same name. Symbols contains a list of commonly used symbols, you can see their abbreviation by floating your cursor over them.

To start a new theory simply use *File* → *new* to open a blank document. Before starting anything, three lines are needed:

theory *NameOfYourTheory*
imports *TheoriesYouNeed*
begin
The file should then be saved under *NameOfYourTheory.thy*. To import any theory you want, simply make sure they are located in your Isabelle installation folder as required then import them as
imports *"~~/src/FileContainingTheTheory/.../TheoryNeeded"*

Depending on the size of the theory, it may take a while to load (you can check the progress in the theories sidebar). Once your theory is complete, add the statement ***end*** as the final line.

4 Proof syntax, methods, tools

4.1 Basic proof syntax

One of the strengths of Isabelle is how readable proofs are, as you can see by browsing the Library. The syntax is very similar to what you will be familiar with from traditional mathematics. The basic outline of a proof is as follows:

lemma *NameOfYourLemma*
assumes *"assumption 1" "assumption 2" ... "assumption n"*
shows *"statement of your lemma"*
proof -
have *"intermediate result 1"*
using *SomeLemma AnotherLemma ... YetAnotherLemma assms(x) ... assms(y)*
by *SomeProofMethod*
then have *"intermediate result 2"*
...
then show *"statement of your lemma"*
using *...*
by *...*
qed

You begin by initialising a new theorem. An Isabelle theorem can be called a lemma, a proposition, a theorem or a corollary. No significance is attached to your choice but people generally stick with lemma. You next state the assumptions required to prove the lemma then the statement of the lemma. You then proceed to prove the lemma by proving any intermediate steps required to get the final result. If the intermediate statements are straightforward such that they can be dealt with in "one line", you simply indicate which already existing lemmas (in your theory or some imported theory) or assumptions (denoted *assms(1)*, *assms(2)*, ..., *assms(n)*) are

needed and use a proof method (more on this later). Note that some proof methods like the ATP that are used by Sledgehammer calls may call some lemma on their own, the syntax then becomes

*by (**ProofMethod** SomeLemma AnotherLemma ... YetAnotherLemma)*

If an intermediate step requires more effort, you simply start a sub-proof for what is essentially a sub-lemma. Every time you use the ***proof -*** command a ***qed*** statement is needed at the end to close it. It should then output the statement of your lemma. Note that in some cases, different theories can have lemmas using the same name. To differentiate both, you can use

$$TheoryName.LemmaName$$

since, if only the name of the lemma is given, Isabelle may use either lemma, leading to errors. Finally, note you can also *ctrl+click* on a lemma, object or other to go to its source, e.g. the proof of the lemma in another theory or the definition of an object.

4.2 Methods and Sledgehammer

In simple terms, a proof method is a tactic (e.g rewriting a term using some equation) to solve a goal. Isabelle comes with many methods, to see the full list simply type ***print_methods*** which will output a short description for each of them. It is unlikely that you will use all of them in practice and you do not need to know how they all work. After a bit of experience you will get some intuition as to which method might work for a given statement. The most commonly use methods are auto, blast, simp, force and the various ITP (metis, meson, ...) called by Sledgehammer. Finally, an important "method" you should remember is ***Sorry***. This is used to replace the proof, whether the statement is true or not. This comes handy if you are stuck on some lemma or step in a proof but want to be able to use it.

Sledgehammer is a fairly recent feature, adding some form of automatic theorem proving in Isabelle. It works by calling several ATP and attempting to find a proof for your current statement. It will then output suggested proofs or time-out if it finds nothing. You can use this tool by simply clicking apply in the sledgehammer window. Note that it will be applied at wherever you are in the text so you can apply it right after the statement or after the *using* line. This way you can feed it some lemma to use which may help it solve your problem. In particular if you already have all the necessary lemma needed to prove your statement it will only need to find a proof method. Secondly, it can run in the background, so you can apply it after writing a statement then try your own proof whilst it runs.

Sledgehammer can appear to work in mysterious ways as it will occasional struggle with basic statements whilst miraculously finding some weird looking proof for some complicated statement you were stuck on. In practice it is extremely useful as a tool. The most efficient way to make use of it is to input the most important lemma and assumptions needed to prove your statement then run sledgehammer to help add any lemma needed(something like commutativity of addition which you might not think about) and a proof method. To make sure you are able to use a lemma, simply check it's declaration in the theory it comes from and check that you satisfy all assumptions and are using objects of the right type. Proofs suggested by Sledgehammer will indicate how much time they take, it is good practice to pick the fastest running one which is usually the simplest.

4.3 Syntax for some specific cases

We have so far presented a very general syntax that is suited to standard, straightforward proofs. In general you start a proof using *proof* -, the dash is actually noteworthy: it means that no specific rule is applied. In some cases you may want to use for example contradiction or cases (presented below) amongst others. You can also do things other than prove statements, in particular you may want to introduce various objects inside the proof, obtain them or assume properties about them. We now list a few cases, with a short overview of their syntax.

- **Proof by contradiction**

  Writing a proof by contradiction is straightforward, with the general syntax as follows:

  lemma *NameOfYourLemma*
  assumes *...*
  shows *"statement of your lemma"*
  proof *(rule ccontr)*
  assume *"¬ (statement of your lemma)"*
  ...
  then show *False*
  using *...*
  by*...*
  qed

  This differs from the general case in a few ways. We note that we are going to use contradiction by calling the rule *ccontr* at the start. We then have to assume the logical negation of what we wish to show. Proceed to go about the proof until you reach your contradiction. The line

  then show *False*

  is used to show the contradiction coming from the starting assumption using the standard way of proving a statement.

- **Proof by cases**

  Like proofs by contradiction, this is an often used form of proof and the syntax is also simple:

  lemma *NameOfYourLemma*
  assumes *...*
  shows *"statement of your lemma"*
  proof *(cases "some statement")*
  case *True*
  ...
  then show *"statement of your lemma"*
  using *...*
  by*...*
  next
  case *False*
  ...

then show *"statement of your lemma"*
using *...*
by*...*
qed

An important thing to note is that this only deals with the binary case of *some statement* being true or false. In both cases you use this to prove the statement of your lemma, ***next*** separates both cases so only one ***qed*** is needed at the end.

- **Define**, **obtain**

In many cases you may wish to introduce new variables or objects inside a proof. To do this you can use **def** or **obtain**. When you wish to define something new, proceed using the command **def**:

def *object == "what your object is"*

Suppose you have shown (or can show) that there exists some object satisfying some property. You may want to have such an object to use explicitly. To do so, use **obtain**:

obtain *object where "property1" "property2" ... "propertyN"*
You can list as many properties as you like.
You might be tempted to use
assume *"statement"*

Note that using this can lead to an error such as failure to redefine goals once you try to end your proof.

5 Locales [1]

A mathematician or scientist will often have to reason at the level of *structures*, to be understood broadly as a collection of parameters subject to some rules, or axioms — this is the case of algebraic gadgets such as groups, monoids and modules, but also of many other vital constructions such as topological spaces, sheaves, measure spaces and so on. If one wants to refer to (and prove things about) such structures in Isabelle, there is a particular tool that establishes a "context" for theorems — a *locale declaration* tells Isabelle that certain data should be understood within a context, i.e, that when you reference that data all the specified conditions (or axioms) in the locale declaration should hold. Once that is done, we can prove things that hold for any particular instance of that structure, or *locale*, and also tell (and prove in) Isabelle that a certain particular construction just coded in is one such instance.

5.1 Locale Declarations

The general syntax of a locale declaration is as follows:

[1] More information on locales can be found in [15].

locale "name given to the locale" = "$locale_1$" + ... + "$locale_n$"+
fixes
parameter1 :: "type" and
parameter2 :: "type" and ... and
lastparameter:: "type"
assumes
"axiom1" and
"axiom2 and ... and
"lastaxiom"

Such a declaration should be read like this:

> We declare that (given name) is a locale; it assumes all the data and axioms of the locales named $locale_1$, $locale_2$, ..., $locale_n$ and, besides that, it fixes a series of parameters, with the names and types given after the command **fixes**. Such parameters will from now on be assumed to satisfy the propositions (axiom1, ..., lastaxiom) displayed after the **assumes** command.

It is often useful to give names to the axioms — this will make them more easily employable in proofs. To do so, instead of just writing down the proposition in the form

"proposition"

write

name: "proposition" .

In the following example, we create a locale designed to define "presheaves" over a topological space. In the given situation, a presheaf is just a topological space together with a mapping that associates to each open set U a ring "$objectsmap\ U$" and to each inclusion $V \subseteq U$ a ring homomorphism $objectsmap\ U \to objectsmap\ V$, named $restrictionsmap\ (U, V)$, satisfying a few properties. The formalisation goes like this:

locale presheaf = topology +
fixes
opcatopensets ::" ('a) PosetalCategory" and
objectsmap :: " 'a set => ('a, 'm) Ring_scheme" and
restrictionsmap:: "('a set 'a set) => ('a =>'a)"
assumes
"opcatopensets ≡ (| $Obj = \{x.\ x \in T\},\ Mor\ = \{(x,y) \mid xy.\ (x,y) \in\ revpo\}$,
$Dom = psheafdom,\ Cod = psheafcod,\ Id = psheafid,\ Comp = psheafcomp$ |)" and
"$\bigwedge\ y\ w.\ w \neq y\ \to\ psheafcomp\ (x,y)\ (w,z)\ =\ undefined$)" and
"$\bigwedge x.\ x \notin T\ \to\ (objectsmap\ x\ =\ undefined)$" and
homim: "$\forall x\ y.\ (restrictionsmap\ (x,y))\ \in\ rHom\ (objectsmap\ x)\ (objectsmap\ y)$" and
"$\bigwedge\ x\ y.\ (restrictionsmap\ (x,x)\ y\ =\ y)$ " and
functoriality: "$\forall\ x\ y\ z.\ (\ (restrictionsmap\ (y,z)) \circ (restrictionsmap\ (x,y)) =\ restrictionsmap\ (x,z))$"
and
objectsmaptorings: "$\bigwedge\ U.\ Ring\ (objectsmap\ U)$"

This example showcases some of the syntax associated with locales, so let's look into the details. The line

locale presheaf = topology +

gives the name "presheaf" to our locale and, since we must assume we're given a topological space, we load a previously defined locale "topology" that encompasses this notion. Next, we must create the underlying data of our presheaves, namely the maps associating a ring to each open set and a ring homomorphism to each inclusion. These are introduced to Isabelle via the lines

fixes
opcatopensets ::" ('a) PosetalCategory" and
objectsmap :: " 'a set => ('a, 'm) Ring_scheme" and
restrictionsmap:: "('a set × 'a set) => ('a =>'a)"

The parameter "opcatopensets" is just some extra information that was introduced to the benefit of some proofs and we'll ignore it. Up next comes the parameter that defines the map taking an open set to a ring, that is declared by using an additional piece of syntax: the odd-looking expression "('a, 'm) Ring_scheme". Here, "Ring" is a previously defined locale, which involves parameters of types 'a and 'm, and "('a, 'm) Ring_scheme" means "a variable with the **format** of an instance of the locale "Ring". The final portion of the locale declaration introduces the properties that "objectsmap" and "restrictionsmap" should obey. We call the reader's attention to the line

objectsmaptorings: "$\bigwedge$ *U*. *Ring* (*objectsmap U*)"

It is of vital importance and says that for each U, we assume objectsmap U is a ring. This warrants a **warning**

> A parameter *datum* : "$('a,'b,...,'z)$ *locale_scheme*" is just something of the **form** of an instance of the locale, i.e, the locale axioms are NOT assumed. The expression *locale* (*datum*) is the actual proposition that "datum" satisfies the locale conditions.

After defining a locale, it is possible to prove theorems about any structure of that kind. One way to do this is by using the syntax:

lemma (in locale) lemma_name:
assumes something: "proposition"
shows "proposition"

If a series of facts will refer to the same locale, one can write

context locale_name
begin
definition (...)
lemma (...)
lemma (...)
theorem (...)
...
end

In either case, by using the standard name given to the locale parameters/axioms, Isabelle will understand that you are referring to the whole structure, rather than to raw data of the same type (i.e to a group, rather than its underlying set, etc). It is also possible to refer to all the axioms of the locale at once by writing "locale_axioms". Here is an example of a simple theorem, showing the homomorphisms induced by the inclusions in a presheaf preserve the zero element of the rings:

lemma (in presheaf) restrictionpreserveszero:
assumes H: "$U \in T \wedge V \in T \wedge V \subseteq U$"
shows "$restrictionsmap\ (U, V)\ 0 \searrow objectsmap\ U \swarrow = 0 \searrow objectsmap\ V \swarrow$"
using $presheaf\_axioms\ H\ homim\ rHom\_def\ aHom\_def$
by $(simp\ add:\ rHom\_0\_0)$

In this case T is a parameter of the locale **topology**, which is assumed since we invoked the locale **presheaf** which presupposes it, corresponding to the set of open sets of the topological space (i.e, T is the topology). Since we're in the context of the locale, Isabelle understands what is meant by expressions such as "$U \in T$" (i.e, U is an open set), and so we can use them to build the hypothesis H. It is also appropriate to refer to any parameter of **presheaf** (which includes the parameters of any other locales in its declaration, namely **topology**). In the theorem line

"$restrictionsmap\ (U, V)\ 0_{(objectsmap\ U)} = 0_{(objectsmap\ V)}$"

the syntax $0 \searrow objectsmap\ U \swarrow$ requires further clarification: 0 is a parameter of the locale Ring, and (objectsmap U) is a particular instance of a ring — the notation tells Isabelle that one is talking about the parameter "0" of the particular ring "(objectsmap U)". It can be written by using the down-right and down-left arrows control blocks of the Isabelle interface, i.e, writing

parameter (control block) locale_instance (control block)

specifies that you're talking about the parameter for that particular instance of the locale. Notice that the proof

using $presheaf\_axioms\ H\ rHom\_def\ aHom\_def$
by $(simp\ add:\ rHom\_0\_0)$

directly appeals to the locale axioms via the theorem presheaf_axioms and previously proven facts about the locale **Ring**. Bear in mind that without using the presheaf axioms in this proof, we wouldn't have access to the axiom that tells Isabelle (**because of the hypothesis H**) that objectsmap U is a ring and thus that facts about rings hold for it. It is also possible to refer to the raw data underneath the locale (i.e, the parameters and their types) via locale_def.

Another common theme in algebra is further specialisation of structures, for example a ring is also an abelian group, which is a group, which is a monoid. Such conceptual extensions can be done quite easily in Isabelle, in the following way:

locale specialisedstructure = oldstructure +
assumes
"axiom_1" and "axiom_2" and ... and "axiom_k"

It will often be useful to work with multiple instances of the same locale at once. This will typically be needed when defining maps between structures, such as ring homomorphisms and

linear transformations (normally as a new locale), or when one needs to consider, say, two distinct topological spaces. In the following example we create a locale for the morphisms of presheaves, by fixing two instances of the locale **presheaf**:

locale presheafmorphism =
F: presheaf T opcatopensets F restrictionsmap1 +
G: presheaf T opcatopensets G restrictionsmap2
for T opcatopensets F G restrictionsmap1 restrictionsmap2 +
fixes pmorphism :: "'a set =>('a => 'a)"
assumes
homforeachopenset: "∀ U. (pmorphism U ∈ rHom (F U) (G U))" and
naturality: "∀ U V. ((U ⊆ V) → ((pmorphism V) ∘ (restrictionsmap1 (V,U)) =
(restrictionsmap2(V,U)) ∘ (pmorphism U)))"

The declaration

> F: presheaf T opcatopensets F restrictionsmap1 +
> G: presheaf T opcatopensets G restrictionsmap2
> for T opcatopensets F G restrictionsmap1 restrictionsmap2

should be read as

> F is a presheaf with parameters T, opcatopensets, F and restrictionsmap1, while G is a presheaf with parameters T opcatopensets G restrictionsmap2, both sharing the same T and opcatopensets.

Note the same name F was given both to the presheaf F and its objectsmap (similarly for G) and that we can invoke those specific instances of presheaves in the lines

> **assumes**
> homforeachopenset: "∀ U. (pmorphism U ∈ rHom (F U) (G U))" and
> naturality: "∀ U V. ((U ⊆ V) → ((pmorphism V) ∘ (restrictionsmap1 (V,U)) =
> (restrictionsmap2(V,U)) ∘ (pmorphism U)))"

and whenever the locale **presheafmorphism** is loaded.

5.2 Locale commands and proofs

Whenever a theorem is proved in the context of a locale, it can be employed as a fact in any proof within the same context. To directly invoke such a theorem, it is possible to write "locale.theorem" — the lemma **restrictionpreserveszero**, for example, can be referred to as **presheaf.restrictionpreserveszero**. There are a few other commands specific to locales that can also be employed and are worth noting:

- **print_locales**: Quite possibly your best friend. This command will print a list of every single locale in your theory, in the way they should be fed to Isabelle for usage in other commands. For example, the presheaf locale is listed as

  > presheaf T opcatopensets objectsmap restrictionsmap

where T is a parameter from the loaded locale **topology** and the rest are the remaining parameters of **presheaf**. Generally, this is just the locale name, followed by all necessary parameters (including those of other locales used in its definition) in order of appearance. Spare your time and just use **print_locales** to copy/paste everything as needed.

- **interpretation** and **interpret**: After constructing a locale and proving theorems about it, one will want to apply said theorems to particular cases (say, apply a general theorem about groups to the case of the Klein 4-group). To do so, there are two commands: **interpretation**, to be used while writing the theory, and **interpret**, to be used in the middle of a proof.

- **unfold_locales**: Suppose one constructs a certain structure and wants to tell Isabelle that what you just built is indeed an example of an instance of a locale, for example, one constructs the integers as a set, defines the operations of arithmetic, the constants 0 and 1 and wants to make an Isabelle proof that this data fits into the locale **Ring** (i.e, show the integers as defined form a ring). In order to create the subgoals for that proof, consisting of the ring axioms to be proved satisfied, one can employ the command **unfold_locales** as in the following example in which we want to show a certain construction made out of a presheaf is indeed a ring:

  lemma (in presheaf) stalk_set_is_ring:
  assumes P: "$x \in \bigcup T$"
  shows "Ring (stalk_ring x)"
  proof unfold_locales
  interpret objectimage: Ring "objectsmap ($\bigcup T$)"
  using objectsmaptorings P stalk_ring_def
  by auto

  To make it perfectly clear: this is a lemma in the context of the locale **presheaf**, which says that assuming x is a point in the topological space (here understood as the union of all the open sets, which are elements of T), we show **stalk_ring x** is an instance of the locale **Ring**. In order to prove this, we start with **unfold_locales**, which creates a subgoal for each ring axiom **stalk_ring x** has to satisfy.

- **intro_locales**: works similarly to **unfold_locales**, but restricts itself to the introduction rules only.

  Further details and more commands pertaining to locales can be found in Section 5.7 of the Isabelle/Isar Reference Manual [16].

6 Comments

This section contains general comments about working in Isabelle.

Choosing what to prove An important constraint is that you may only use results that have already been formalised which has two consequences. First, you may find yourself limited by a lack of prerequisite results you need. The only way to address this is simply to go back and add them yourself. However this can sometimes add up until you find yourself proving a lot of

small results and never get around to your original goal. Second, as you have to use work done by others you are constrained to use their notations and conventions. The conclusion of this is that you should at first try to restrict yourself to areas you know very well. Being able to find alternative proofs and understanding any notation is extremely beneficial. Understanding what everything is built on will also help. For example Isabelle has both the Lebesgue integral and the gauge integral defined, you might need properties from both. Differentiation is defined using the Fréchet derivative which has a slightly different form than what people commonly use. In some cases you may find you need to prove some result that requires knowledge from some library you don't understand (a combination of unfamiliarity with the mathematics and the syntax used in that library), you will want to limit these instances.

Readability and elegance Another aspect is the readability and elegance of proofs. An elegant proof is a great attraction for many mathematicians and it seems natural to aim for equally nice work in Isabelle, which already does a good job in terms of readability. However it is often the case that arguments from a textbook cannot be rewritten exactly and you have to work practically to make the proof work. You may have to deal with using different libraries together. For example there are two formalisations on integration which both have different notations and properties of integration. Lemma for both may be required in a proof, requiring lots of term rewriting to switch between them and leading to a loss of elegance. When dealing with something like this it can be beneficial to hide these steps that do not constitute an actual step in the proof inside subproofs that can be closed. Similarly, you may want to keep anything else requiring multiple steps which don't directly contribute to the proof, such as computations which are not efficiently dealt with, inside a subproof.

Observation. *Each function* $f : A \times B \to C$ *naturally corresponds to a function* $f^* : A \to \mathrm{Fun}(B, C)$*, where* $Fun(B, C)$ *is the set of functions* $B \to C$ *and for each* $a \in A$ *the function* $f^*(a) : B \to C$ *takes* $b \in B$ *to* $f(a, b)$*. Conversely, a map* $g^* : A \to \mathrm{Fun}(B, C)$ *corresponds naturally to* $g : A \times B \to C$*, where* $g(a, b) = g^*(a)(b)$*. This relation is called the Curry adjunction (those familiar with Category Theory will recognize this as the property of Cartesian closedness) and we refer to* $f^* : A \to \mathrm{Fun}(B, C)$ *as the curried form of* $f : A \times B \to C$*, while the process itself is called currying.*

It is good practice in Isabelle to define functions in their curried form rather than in their original form with a product in the domain, i.e define $f :: a \Rightarrow b \Rightarrow c$ *rather than* $f :: a \times b \Rightarrow c$*. Written in terms of λ-calculus* [2]*,* $f(x, y)$ *can be represented as* $(\lambda x.\ (\lambda y.\ f\ x\ y))$*.*

7 Conclusion and acknowledgements

Hopefully you now have a better idea of what Isabelle is and how it works. Browsing through previous formalisations will give you many examples you can also learn from, as well as using the mailing list and other resources listed at the start.

This work would not have happened without the chance to complete a Post Masters Placement, organised by the University of Cambridge faculty of Mathematics. In particular we wish to thank Marjorie Batchelor, Jake Rasmussen, Ruadhai Dervan and Francis Woodhouse who run the scheme, as well as the donors who provide funding for this. The project was hosted at the University of Cambridge Computer Laboratory and was supervised by Lawrence Paulson. Him and Wenda Li both provided insightful discussions and invaluable help.

[2] see appendix A.

Appendices

A λ-calculus

A.1 Introduction

The early 1900's brought to the attention of mathematicians, out of the scope of philosophy, the foundational matters of Mathematics. Cantor's inaugural paper on Set Theory (published in 1874) was still recent and so was Peano's "*Arithmetices principia, nova methodo exposita*", which introduced his axiomatic treatment of arithmetic (1889). It is mentioned by Bertrand Russel [17] that

> "*It was at the International Congress of Philosophy in Paris in the year 1900 that I became aware of the importance of logical reform for the philosophy of mathematics. (...) I was impressed by the fact that, in every discussion, [Peano] showed more precision and more logical rigour than was shown by anybody else. (...) It was [Peano's works] that gave the impetus to my own views on the principles of mathematics.*"

and his following publication, "*The Principles of Mathematics*" (1903), introduced his famous paradox (independently identified by Ernst Zermelo), leading to further investigations that culminated in mathematical work of utmost importance - Zermelo (1908) and Fraenkel's (1922) axiomatic set theory (nowadays referred to as ZF, or ZFC if the axiom of choice is included) and Russel and Whitehead's "*Principia Mathematica*" (1913). Gödel's Theorems (1931) were not yet known and logic was a much sought-after subject of enquiry. In this context, many systems were drafted in an attempt to provide consistent foundations for mathematics, some taking "function" as a first principle and others taking "sets", but there was no consensus on which approach was best. The original λ-calculus was designed to be part of a broader foundational systems based on the concept of function [18].

Ideas regarding function abstraction and application date back to at least 1889, as Peano's aforementioned "*Arithmetices principia, nova methodo exposita*" includes notation for abstraction and substitution. Similar ideas also occurred to Frege (1893) [19], Russel and Whitehead [20], although the axiomatization of both syntax and notion of substitution is characteristic of the approaches of combinatory logic and λ-calculus (1920), the former of which was actually the first to do so, even though λ-calculus expresses the concepts involved more directly [18].

Alonzo Church came to Princeton in 1924 as a brilliant undergraduate, having published his first paper [21] in 1924, the year he graduated [3]. He remained in Princeton for graduate studies, completing his PhD in 1927 under Oswald Veblen (who also supervised J.H.C Whitehead, the famous topologist nephew of Bertrand Russel's collaborator and friend Alfred Whitehead, who worked under Max Newman in Bletchley Park during World War II), with a thesis entitled "*Alternatives to Zermelos Assumption*". After many brief stays elsewhere, Church was invited to return to Princeton as an Assistant Professor in 1929. The 1930's was a rich period for logic in Princeton [22], which hosted names such as John von Neumann, Alan Turing (who was a visiting graduate student in 1936 and ended up completing his PhD under Church), Kurt Gödel (who visited the IAS in 1933-1934 and lectured about his then-recent incompleteness theorems, before deciding to stay permanently) and Church himself, alongside his students Stephen Kleene and Barkley Rosser. The λ-calculus first appeared in published work in a paper by Church in 1932 [23], but a contradiction was found shortly after and the system had to be revised, the paper

[3] Church was voted the "most likely to remain a bachelor" by his senior class [22], but was happily married for 51 years to Mary Julia Kuczinski, his nurse after an accident.

gaining a second version in 1933. This revised version brings the definition of Church numerals, an encoding of the natural numbers in λ-terms, allowing one to directly reason about arithmetic within λ-calculus. The details on the inner workings of this arithmetic were left to his student Kleene [24] and collaboration with his students proved most fruitful: many results on Church's logic and the background λ-calculus were made by Kleene and Rosser [4], including the fact that Church's logic was inconsistent [25] (1935).

Grudingly abandoning his initial project in Logic, further work was put into pure λ-calculus. At the time, there was a great interest in formalizing the intuitive notion of "effectively calculable function" and two approaches besides Church's own were taken: Herbrand-Gödel recursive functions and Turing computable functions. Both forms were shown to be equivalent to λ-definability [26, 27], but the merits of the theory came with Church's solution (1936) to Hilbert's Entscheidungsproblem [27], independently to that of Turing right afterwards. Turing had written his paper on computability (1936) following lectures at Cambridge from Max Newman, after which he undertook his PhD in Princeton under Church. Amongst his contributions to the λ-calculus is the first explicit description of a fixed-point combinator, which passed unnoticed back then. Turing machines may be considered a much more intuitive model of computation [5] and his solution to the Entscheidungsproblem was deemed more transparent, leading to a considerable decrease in general interest in λ-calculus in the 1940's and 1950's, although further developments were pursued by Kleene, Rosser and Curry. The theory resurfaced in the 1960's due to the growing interest of computer scientists interested in the development of programming languages, although the first proposal that λ-calculus itself could be used for that was made by logician Frederic Fitch in 1957 [18].

A.2 Syntactic features

To develop the λ-calculus, we shall take a (countably infinite) set of *variables* $V = \{a, b, c, ...\}$, to form an alphabet $\Sigma = \{\ (,\ ),\ \lambda,\ .\} \coprod V$ (where $A \coprod B$ denotes the disjoint union of the sets A and B as usual), and consider *formulas* [6], that is, any *finite* sequence of symbols in Σ. As with any formal system, it is necessary to distinguish amongst all formulas a subset of formulas with a structure convenient to exploit for a desired purpose - a *syntactical structure* must be determined to allow one to focus on a smaller set of formulas and in such way that it is intrinsically related to the aimed *semantics*. The formulas to be worked with shall be called *well-formed formulas* or *terms* and are defined recursively. It is also important to formalize the intuitive understanding of a variable occurring as a *free variable* or a *bound variable* in a formula - a free variable may be sensibly substituted (as x in the polynomial $x^2 + 1$, which has not been evaluated), whereas a bound variable cannot (consider x in $\int f(x)\ dx$).

Definition A.1 (Term, free variable, bound variable and closed term)**.** [7]

The language of well-formed formulas (or terms) Λ over Σ of the (untyped) λ-calculus is the

[4] Reading Church's 1936 paper "*An Unsolvable Problem of Elementary Number Theory*", one could be under the impression most of the work was actually Kleene's.

[5] Gödel in particular had much criticism in this regard to both λ-calculus and his own recursive functions theory, favouring Turing's approach.

[6] Notation: formulae names shall be denoted by bold Latin characters and, except when otherwise stated, every formula will be a name for itself.

[7] This slightly differs from the definition given by Church, in that it allows the term $(\lambda x.\mathbf{M})$ to be formed whenever $\mathbf{M}$ is a term, regardless of the way x occurs as a variable in $\mathbf{M}$. The notation is also different, but the reader should agree there is no loss of clarity, intuition nor content.

language generated by the context-free pseudogrammar [8] $(\{S\}, \Sigma, R, S)$ *with productions given by*

$$S \to \boldsymbol{x} \mid (\lambda x.S) \mid (SS) \tag{A.1}$$

for each variable x*. The set* $FV(\boldsymbol{s})$ *of free variables of a term* $\boldsymbol{s}$ *is inductively defined by*

- $FV(\boldsymbol{x}) = \{x\}$*, for each variable* $x \in V$*;*
- $FV(\boldsymbol{st}) = FV(\boldsymbol{s}) \cup FV(\boldsymbol{t})$*, for any pair of terms* $\boldsymbol{s}$ *and* $\boldsymbol{t}$*;*
- $FV(\lambda x.\boldsymbol{s}) = FV(\boldsymbol{s}) \setminus \{x\}$.

  The occurrence of a variable x *in a term* $\boldsymbol{s}$ *is free if* $x \in FV(\boldsymbol{s})$ *and is bound if* x *is in a subterm* [9] $(\lambda x.\boldsymbol{u})$ *of* $\boldsymbol{s}$ *. A term* $\boldsymbol{s}$ *is closed if* $FV(\boldsymbol{s}) = \emptyset$ *and the set of all closed terms is denoted by* Λ^0*.*

This definition gives a convenient syntactic structure and an intuitive picture may be drawn — it recursively builds terms in a way that allows variable substitutions to be made sensibly. It begins by making a distinction between merely syntactic symbols such as "(" , "." , ")" and "λ") and variables, and starts by building terms out of the simplest possible formulae excluding syntactic symbols: the formulae consisting of a single variable (this is allowed by the productions $S \to \mathbf{x}$). The occurrence of a variable x in $\mathbf{x}$ is defined to be free to state that the variable x can always be substituted in the term $\mathbf{x}$, a property of any decent notion of substitution. From there we may combine terms, introducing the syntactic symbols ( ) in a canonical way, to produce new terms while preserving variable freeness and boundness , by usage of the productions $S \to (SS)$. This may be interpreted as stating that substitutions that were possible remain possible after concatenating terms and inserting the syntactic symbols in the correct way (another process any non-trivial definition of substitution should allow for). Finally, the symbols $\lambda x.$ act as a "syntactical operator" that binds the variable x: if x occurs as a free variable in $\mathbf{M}$, then it occurs as a bound variable in the new $\lambda x.\mathbf{M}$, meaning x may no longer be substituted in this term (although other variables may remain free). Again, any notion of substitution should consider that a variable can be substituted at most once and substituting a variable has no effect on the remaining ones freedom or boundness.

Before spelling out a precise meaning for "substituting a variable in a term", we finish syntactic considerations by establishing a suitable way to compare the syntactic content of two terms and, since there are infinitely-many variables, it is always possible to pick a variable that hasn't occurred in any of the terms being dealt with - such convenient variables shall be referred to as *fresh* variables. If, in the same term, a variable occurs multiple times, but sometimes as a free variable and sometimes as a bound variable, one may substitute its occurrences as a bound variable (causing no harm to semantics) for a fresh variable. The new term obtained is syntactically equivalent to the previous one, but now any variables occur solely as a free variable or only as a bound variable. This motivates the following definition:

Definition A.2 (α-conversion)**.**
Two terms $\boldsymbol{s}$ *and* $\boldsymbol{t}$ *are* α*-convertible (written* $\boldsymbol{s} \equiv \boldsymbol{t}$*) if both terms may be transformed into the same term solely by renaming bound variables to fresh variables.*

From now on, when stating results or definitions, bound variables in a term will, without loss of generality (by applications of α-conversions if need be), be deemed different than the bound or free variables of any other terms. Additionally, the following terminology will be introduced:

[8] A *context-free pseudogrammar is defined in the same way as a context-free grammar, but relaxing the condition that the set of terminals and the set of relations need to be finite.*

[9] A subterm of $\mathbf{s}$ is a subsequence of symbols in $\mathbf{s}$ that is itself a term.

Definition A.3 (Abstraction and application)**.**
A term $\boldsymbol{s}$ *of the* λ*-calculus is an abstraction (or* λ*-abstraction) if it is of the form* $\lambda x.\boldsymbol{t}$ *for some variable* x *and term* $\boldsymbol{t}$*. Terms of the form* $\boldsymbol{st}$ *are called applications.*

By convention, applications in a λ-term shall take precedence over abstractions, applications will associate to the left (i.e, $\mathbf{rst} = (\mathbf{rs})\mathbf{t}$) and the body of an abstraction extends to the right as most as possible (so $\lambda x.\mathbf{rst} = (\lambda x.\mathbf{rst})$, and not $(\lambda x.\mathbf{r})\mathbf{st}$).

A.3 Semantics

Having set a syntatical background to work on, one might give a meaning to terms in λ-calculus and exact meaningful operations on such terms. As usual, variable substitution will be defined recursively, but being careful as to only allow free variables to be replaced:

Definition A.4 (Substitution)**.** [10]

- $\boldsymbol{x}[\boldsymbol{t}/x] = \boldsymbol{t}$ *for each variable* x*;*
- $\boldsymbol{y}[\boldsymbol{t}/x] = \boldsymbol{y}$ *if* y *is a variable and* $y \neq x$*;*
- $(\boldsymbol{su})[\boldsymbol{t}/x] = (\boldsymbol{s}[\boldsymbol{t}/x]\boldsymbol{u}[\boldsymbol{t}/x])$*;*
- $(\lambda y.\boldsymbol{s})[\boldsymbol{t}/x] = \lambda y.(\boldsymbol{s}[\boldsymbol{t}/x])$ *if* $y \neq x$ *and* $y \notin FV(\boldsymbol{t})$*.*

The established notion of substitution agrees with the previously defined syntax, carefully avoiding the binding of free variables when performing substitutions. This allows one to define an important relation on terms: the relation β on terms consists of the pairs

$$(\ (\lambda x.\mathbf{s})\mathbf{t}\ , \mathbf{s}[\mathbf{t}/x]) \in \Lambda \times \Lambda,$$

that is, $(\lambda x.\mathbf{s})\mathbf{t}\ \beta\ \mathbf{s}[\mathbf{t}/x]$.

Notice that β-convertion gives an interesting perspective for functional application: the term $\lambda x.\mathbf{M}$ may be interpreted as a function of the variable x (the variable being bound by the abstraction) and the application $(\lambda x.\mathbf{M})\mathbf{t}$ may be viewed as the evaluation of that function at $\mathbf{t}$ (which may again be a function).

Example A.5. *The term* $\lambda x.\boldsymbol{x}$ *may be identified with the identity function: if* $\boldsymbol{t}$ *is a term, then* $(\lambda x.\boldsymbol{x})\boldsymbol{t}\ \beta\ \boldsymbol{t}$*, mirroring the application of the identity function to the argument* $\boldsymbol{t}$ *to obtain the image* $\boldsymbol{t}$ *(with a suitable domain and codomain, made implicit via* λ*-abstraction).*

This point of view enables one to see λ-abstractions as models for functions whilst precluding usage of sets, i.e, by treating the notion of function as a first principle instead of sets (from which the relational definition of function derives) - an abstraction "names" a function, which may then be applied to terms in an unrestricted way. The explored syntax, alongside λ-abstraction, application and the operations of α-conversion and β-conversion forms the basis of what may be called *a* λ-calculus, but other axioms may be added. So far, the theory built, called $\lambda\beta$, is defined by the following rules:

Definition A.6. *The theory* $\lambda\beta$ *of* λ*-calculus consists of* λ*-terms together with an equivalence relation "=" on* λ*-terms, subject to the following axioms:*

- *Application: If* $\boldsymbol{s} = \boldsymbol{s'}$ *and* $\boldsymbol{t} = \boldsymbol{t'}$*, then* $\boldsymbol{st} = \boldsymbol{s't'}$*;*

[10] The notation $\mathbf{M}[\mathbf{t}/x]$ reads as "the term obtained by replacing free occurrences of x in $\mathbf{M}$ for the term $\mathbf{t}$".

- *Abstraction: If* $\boldsymbol{s} = \boldsymbol{t}$*, then* $\lambda x.\boldsymbol{s} = \lambda x.\boldsymbol{t}$*;*
- β*-conversion:* $(\lambda x.\boldsymbol{s})\boldsymbol{t} = \boldsymbol{s}[\boldsymbol{t}/\boldsymbol{x}]$.

Adding the axiom "If x is not a free variable of $\mathbf{s}$, then $\lambda x.\mathbf{sx} = \mathbf{s}$" yields another popular theory of λ-calculus called $\lambda\beta\eta$ (or *extensional* λ-calculus)[11], and the conversion of terms defined by the previous axiom is named η-conversion. It is based on the philosophy that λ-terms should be viewed as functions, and functions ought to be equal if and only if they yield the same results for all arguments. The theory $\lambda\beta$ shall be used for the remainder of this work.

Through repeated usage of the axioms in a finite number of times, it may [12] be possible to *convert* a term $\mathbf{s}$ into a term $\mathbf{t}$ in a canonical, convenient form - one in which β-conversions may no longer be made:

Definition A.7. *A term is said to be in* β *normal form (or simply in normal form) if it has no subterms of the form* $(\lambda x.\boldsymbol{M})\boldsymbol{N}$*. A term* $\boldsymbol{s}$ *is a normal form of* $\boldsymbol{t}$ *if* $\boldsymbol{s}$ *is in normal form and* $\boldsymbol{t} = \boldsymbol{s}$*.*

Keeping up with the idea that β-conversion corresponds to functional application, a term in normal form is one in which all possible functions have been evaluated. It should be clear that a term in normal form cannot be further β-converted and thus the normal form must be unique up to α-conversions (if it exists) [13], but its also true that if a term has a normal form, then so do all of its subterms and successive β-conversions done the right way will eventually lead to that normal form [28].

A.4 Impacts and further developments of the λ-calculus

The merits of λ-calculus extend far beyond what Church, Kleene and Rosser could envision in 1928-1930. The theory preceded Turing Machines as a model for "effective computability" and provided the first solution to Hilbert's Entscheidungsproblem (again a little sooner than Turing), but as mentioned in the Introduction, Church's approach was relegated in favour to that of Turing, only to resurface in the 1960's when computer scientists became interested in programming languages [18]. Since 1956, a few programming languages were developed inspired by or using λ-calculus, amongst which we may cite LISP, Algol 60 and, most notably, CUCH - a language developed by Corrado Böhm that employed both Curry's combinators (from combinatory logic) and Church's λ-terms. Many further theoretical developments followed, coming from people such as Böhm himself (in the form of Böhm's theorem, for example, that states that if two closed terms $\mathbf{s}$ and $\mathbf{t}$ are not $\lambda\beta\eta$-equivalent, then there there is some sequence of terms such that applying $\mathbf{s}$ to the sequence yields **true** and applying $\mathbf{t}$ to the sequence yields **false**), Curry and Scott (who independently proved [18] versions of slightly different generality of Theorem 2.12).

Many variants of the λ-calculus have been proposed, most notably the simply typed and the typed λ-calculi, the former being envisioned by Church himself, in an attempt to avoid paradox of the Russell kind in the original untyped λ-calculus. These variants add objects of a syntactic nature to be assigned to λ-terms and have found much fruitful use in the theory of programming languages,being the base of languages such as ML and Haskell. A much celebrated virtue was found, however, in the form of the Curry-Howard-Lambek correspondence: following observations from Curry [29] [30] (1934 and 1958) and Howard [14] [31] (1969), a correspondence

[11] Due to Haskell B. Curry.

[12] That is not always the case: the term $(\lambda x.\mathbf{xx})(\lambda x.\mathbf{xx})$ β-converts to itself in one step, thus cannot be put in β normal form.

[13] Suppose there are two distinct normal forms: no further β-conversions can be made for either and one is a normal form of the other.

[14] Dedicated to Curry in the occasion of his 80th birthday.

was drawn between the natural deduction system in Logic and the simply typed λ-calculus [32]: the so-called Curry-Howard isomorphism associates logical formulae to types, proofs to terms and proof transformations to term conversions/reductions. As the λ-calculus can be seen as a pure version of functional programming languages, this correspondence enable one to view proofs as programs. A third stage to this correspondence may yet be established.

In 1942-1945, Samuel Eilenberg and Saunders Mac Lane, inspired by the seemingly universality of some constructions and proofs within mathematics (products, the homomorphism theorem for algebraic structures, amongst other common themes), were working on a novel theory to encompass all of the mathematical thought within a common framework. Their thoughts turned out to produce a very powerful theory, called *Category Theory*, centered around the notion of a category, associated transformations called functors, transformations of transformations, etc, capturing the notion of mathematical structure. This new theory ended up providing a language to discuss all sorts of mathematical phenomena, including foundational studies, and eventually Joachim Lambek (who passed away in 2014 at 91 years old) came up with an equivalence between a particular kind of category and logic. This is done in the following way [32]: given a cartesian closed category $\mathcal{C}$ one can view formulae (or types) as objects of $\mathcal{C}$, proofs as $\mathcal{C}$-morphisms $f : A_1 \times ... \times A_k \to B$ (conversively, any such morphism is associated to a proof of B from assumption $A_1, ..., A_k$,), and all rules of natural deduction or $\beta\eta$-conversion may be obtained from the equations of the cartesian closed category.

The Curry-Howard-Lambek isomorphism is a huge advancement in that techniques developed in either three fields can be interchangeably used to attack problems in either, but it turns out that typed λ-calculi act as internal languages to other particular sorts of categories too. The list of modern topics inspired by developments in λ-calculus include Homotopy Theory, iterative proof assistants and Concurrency Theory, and is likely to expand as more connections are drawn between λ-theories, category theory and other fields within mathematics.

B Types

A particularity of Isabelle (and other ITP) that may appear unusual to some is the use of types rather than sets. We will avoid going into detail about type theory against set theory, but from a practical point of view types often work in similar ways to sets: just as we may declare some variable x to be real valued ($x \in \mathbb{R}$) we can declare it to be of type real (*x::real*).

Any object in Isabelle will have a type. This can be some abstract type denoted *'a*, *'b*, ... or some specific type such as *real*. Operators, functions and anything that takes some input or acts on something will be of some " function type " *'a* $\Longrightarrow$ *'b*, *'a* $\Longrightarrow$ *'b* $\Longrightarrow$ *'c* and so on. Isabelle automatically assigns types (which can be viewed by placing your cursor on the object) in most cases, although you can also fix them yourself when required. To fix some type in a lemma use the command ***fixes*** before your assumptions, and during a proof use ***fix***. For example if you wish to fix f to be some real function, you may write:

fixes *f :: "real* $\Longrightarrow$ *real"*
fix *f :: "real* $\Longrightarrow$ *real"*

Note that objects of type 'a and 'b are not of the same type unless 'a and 'b are somehow specified to be the same type somewhere else. You can also require that an object be of various types. For example, in the line

fixes *g :: " 'a::metric_space,ord* $\Longrightarrow$ real "

'a is declared to be of types metric_space and ord. Note that, like sets some types are more general than others. If you are fixing an object to be of some combination of types, it's a good idea to check whether there any overlaps.

Type errors can commonly happen — you will have to always make sure your objects are of the right type for the case you are working with. A good presentation of type theory is freely available online, as part of Chapter 1 of the Homotopy Type Theory book written as part of the Univalent Foundations program at the Princeton IAS, accessible via `https://hott.github.io/book/nightly/hott-online-1075-g3c53219.pdf` [15] and is extremely recommended to those who want to better understand the inner workings of Isabelle and other interactive proof assistants.

[15] As of January 22, 2017

References

[1] V. Voevodsky, "An experimental library of formalized mathematics based on the univalent foundations," *Mathematical Structures in Computer Science*, vol. FirstView, p. 1–17, February 2015.

[2] V. Voevodsky, "Computer proof assistants - the future of mathematics," *https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/2014_08_27_NUS.pdf*, *Talk*, August 2014.

[3] M. Ganesalingam and W. T. Gowers, "A fully automatic problem solver with human-style output," *CoRR*, vol. abs/1309.4501, 2013.

[4] "https://gowers.wordpress.com/2013/04/14/answers-results-of-polls-and-a-brief-description-of

[5] "https://www.microsoft.com/en-us/research/video/what-are-the-prospects-for-automatic-theorem

[6] "http://dream.inf.ed.ac.uk/projects/isabelle/Isabelle_Primer.pdf,"

[7] J. Avigad and J. Harrison, "Formally verified mathematics," *Commun. ACM*, vol. 57, pp. 66–75, Apr. 2014.

[8] J. Harrison, J. Urban, and F. Wiedijk, "History of interactive theorem proving," in *Handbook of the History of Logic vol. 9 (Computational Logic)* (J. Siekmann, ed.), pp. 135–214, Elsevier, 2014.

[9] K. Appel, W. Haken, *et al.*, "Every planar map is four colorable. part i: Discharging," *Illinois Journal of Mathematics*, vol. 21, no. 3, pp. 429–490, 1977.

[10] K. Appel, W. Haken, J. Koch, *et al.*, "Every planar map is four colorable. part ii: Reducibility," *Illinois Journal of Mathematics*, vol. 21, no. 3, pp. 491–567, 1977.

[11] "http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf,"

[12] T. Hales, M. Adams, G. Bauer, D. Tat Dang, J. Harrison, T. Le Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. Tat Nguyen, T. Quang Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. Hoai Thi Ta, T. N. Tran, D. Thi Trieu, J. Urban, K. Khac Vu, and R. Zumkeller, "A formal proof of the Kepler conjecture," *ArXiv e-prints*, Jan. 2015.

[13] L. C. Paulson, "A mechanised proof of gödel's incompleteness theorems using nominal isabelle," *Journal of Automated Reasoning*, vol. 55, no. 1, pp. 1–37, 2015.

[14] "http://www.ijcai.org/Proceedings/16/Papers/137.pdf,"

[15] "https://isabelle.in.tum.de/dist/Isabelle2016/doc/locales.pdf,"

[16] M. Wenzel, "The isabelle/ isar reference manual," 2016.

[17] B. Russell, *My Philosophical Development*. Spokesman Books, 2008.

[18] F. Cardone and J. R. Hindley, "History of Lambda-calculus and Combinatory Logic."

[19] G. Frege, *Grundgesetze der Arithmetik*, vol. II. Jena: Verlag Hermann Pohle, 1903.

[20] A. N. Whitehead and B. Russell, *Principia Mathematica*. Cambridge University Press, 1910.

[21] A. Church, "Uniqueness of Lorentz Transformation," *The American Mathematical Monthly*, vol. 31, no. 8, pp. 376–382, 1924.

[22] H. B. Enderton, "Alonzo Church: Life and Work."

[23] A. Church, "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932.

[24] S. C. Kleene, "A theory of positive integers in formal logic," *American Journal of Mathematics*, vol. 57, pp. 153–173 and 219–244, 1935.

[25] S. C. Kleene and J. B. Rosser, "The Inconsistency of Certain Formal Logics," *Annals of Mathematics*, vol. 36, pp. 630–636, 1935.

[26] S. C. Kleene, "λ-definability and recursiveness," *Duke Mathematical Journal*, vol. 2, pp. 340–353, 1936.

[27] A. Church, "An Unsolvable Problem of Elementary Number Theory," *American Journal of Mathematics*, vol. 58, no. 2, pp. 345–363, 1936.

[28] A. Church and J. B. Rosser, "Some Properties of Conversion," *Transactions of the American Mathematical Society*, vol. 39, no. 3, pp. 472–482, 1936.

[29] H. Curry, "Functionality in Combinatory Logic," *Proceedings of the National Academy of Sciences*, vol. 20, no. 11, pp. 584–590, 1934.

[30] H. B. Curry, R. Feys, and W. Craig, *Combinatory Logic*, vol. 1.

[31] W. A. Howard, "The formulae-as-types notion of construction."

[32] S. Abramsky and N. Tzevelekos, "Introduction to Categories and Categorical Logic," in *Lecture Notes in Physics, Berlin Springer Verlag* (B. Coecke, ed.), vol. 813 of *Lecture Notes in Physics, Berlin Springer Verlag*, pp. 3–642, 2011.