Intro to interactive theorem proving in Isabelle/HOL

Filip Marić

*Faculty of Mathematics, University of Belgrade

PAT 2023
Table of contents

1 Introduction

2 Basic syntax and automated provers

3 Natural deduction

4 Isar: a language of structured proofs

5 Functional programming
Overview

1. Introduction
2. Basic syntax and automated provers
3. Natural deduction
4. Isar: a language of structured proofs
5. Functional programming
Isabelle/HOL

- Isabelle is a generic interactive theorem prover, developed by Lawrence Paulson (Cambridge) and Tobias Nipkow (Munich). First released in 1986.
- https://isabelle.in.tum.de/
- Archive of formal proofs (https://www.isa-afp.org/)
It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.

Integrated tool support for:
- Automated provers
- Sledgehammer: powerful proof search
- Counter-example finding
- Code generation
- \LaTeX\ document generation
Isabelle/HOL

- Isabelle/HOL – Isabelle’s incarnation for Higher-Order Logic
- FOL extended with functions and sets, polymorphic types, …
- ML-style functional programming
- “HOL = functional programming + logic”
A Course at University of Belgrade

- Introduction to interactive theorem proving
- Elective course on the 4. year of undergraduate studies of informatics
- 12 weeks of teaching (weekly 1.5 hours lectures and 1.5 hours labs)
- Students have some background knowledge in logic and functional programming
- Two parts:
  - Logic and mathematics
  - Functional programming and verification
Approach

- Recapitulation of many concept that students have informally used, but through the lens of interactive theorem proving
- Students need to understand precise, rigorous communication and reasoning (both in abstract mathematics, and in computer programming)
- Slow pace
- Proof assistant is a tool, and not the aim
- Students need not become experts in using some concrete proof assistant
- Concepts are introduced only when necessary, usually through examples
- Using automation is welcome (except in the beginning, when the concept of proof is introduced)
This summer school

- We have 4 hours total
- The main aim is to give a brief introduction to Isabelle/HOL to those who did not have any experience with it
- The other aim is to offer a slightly different didactic to proof assistants than the usual one
  - How to teach proof assistants to support better understanding and provide rigour to elementary high-school/undergraduate mathematics and computer science?
  - As close to everyday mathematics as possible (focus only on the concepts that students do in elementary mathematics)
  - Focus is not on what proof assistants can do and how are they professionally used, but on how to use them “without tears” as a supplement to introductory math/cs curriculum
- Many exercises that we can do together
The course starts with an advertisement of interactive theorem proving
- A brief history
- Major successes

Instead of the standard “bottom-up” approach where we strictly define notions before using them, we use a “hands-on” approach where we try to give intuition and formally define notions along the way, only when necessary.
Example: absolute value

What is an Isabelle/HOL theory?

We define some mathematical concept.

We state some of its properties (in form of lemmas).

We prove those lemmas:
- using automated theorem provers or
- we write the proof in some language and the system checks that proof.

We start with a very simple example of absolute value function.

We formalize the beginning of https://en.wikipedia.org/wiki/Absolute_value
Definition:

\[ |x| = \begin{cases} 
  x, & \text{if } x \geq 0, \\
  -x, & \text{if } x < 0.
\end{cases} \]

Properties:

\[ |a| \geq 0 \]  
Non-negativity

\[ |a| = 0 \iff a = 0 \]  
Positive-definiteness

\[ |ab| = |a| \cdot |b| \]  
Multiplicativity

\[ |a + b| \leq |a| + |b| \]  
Subadditivity, specifically the triangle inequality
“Non-negativity, positive definiteness, and multiplicativity are readily apparent from the definition. To see that subadditivity holds, first note that $|a + b| = s(a + b)$ where $s = \pm 1$, with its sign chosen to make the result positive. Now, since $-1 \cdot x \leq |x|$ and $+1 \cdot x \leq |x|$, it follows that, whichever of $\pm 1$ is the value of $s$, one has $s \cdot x \leq |x|$ for all real $x$. Consequently, $|a + b| = s \cdot (a + b) = s \cdot a + s \cdot b \leq |a| + |b|$, as desired.”
Demo

DEMO 1: AbsoluteValue.thy
Main takeaways

- Isabelle definitions are very similar to functional programming
- Powerful automation
- Declarative proof language makes proofs very similar to everyday mathematical proofs
- Details of syntax are going to be given throughout the course
Overview

1 Introduction

2 Basic syntax and automated provers

3 Natural deduction

4 Isar: a language of structured proofs

5 Functional programming
Syntax

- Students must be comfortable in translating natural language formulations into formal statements
  - Syntax of the proof assistant
  - Typing symbols
  - Writing logically correct formulae (much deeper skill)
- First lab exercises require to write given statements and prove them using automation (by `auto`).
Examples

1. If everyone who lies also steals, and there is someone who lies, then there is someone who steals.

2. If no homework is fun, and some reading is homework, then some reading is not fun.

3. If there is a shoe that fits every leg, then for every leg there is a shoe that fits it. Does the opposite hold?

4. In one village knights always tell the truth, and knaves always lie. Visitor asks the person A if he is a knight, but did not understand his answer. Person B explains that A said that he is a knave, but then C tells that B lies. Prove that C must be a knight.

5. If everyone loves a lover and John loves Mary, then does Iago love Othello?
1. Let $f$ be a binary operation that is associative, has a left-identity element, and all elements have a left inverse. Show that left inverse is always also the right inverse.

2. Is every symmetric and transitive relation also reflexive? Is there some additional condition that guarantees it?
Assume that Pinocchio always lies and says: “All my hats are green”. Which of the following must be true.

1. Pinocchio has no green hats.
2. Pinocchio has only one green hat.
3. Pinocchio has no hats.
4. Pinocchio has at least one hat.
5. Pinocchio has at least one green hat.
Intro to interactive theorem proving in Isabelle/HOL

Basic syntax and automated provers

Demo

DEMO 2: BasicSyntax.thy
Overview

1. Introduction
2. Basic syntax and automated provers
3. Natural deduction
4. Isar: a language of structured proofs
5. Functional programming
Two styles of proof in Isabelle/HOL:
- Tactics (apply style proofs)
- Isar (readable, structured proofs)

Although readable proofs are desirable (easier to read, write and maintain), under the hood everything boils down to applying natural deduction rules.

Natural deduction is like an assembly language of interactive theorem proving.

It is good if the students have some understanding of what is happening “under the hood”.
Rules for propositional logic

**Conjunction**

\[
\begin{align*}
A & \quad B \\
\hline
A \land B & \quad \land I \\
A & \quad A \land B \\
B & \quad A \land B \\
\hline
A & \quad \land E_1 \\
B & \quad \land E_2
\end{align*}
\]

**Disjunction**

\[
\begin{align*}
A & \quad \lor I_1 \\
A & \quad A \lor B \\
B & \quad A \lor B \\
\hline
A \lor B & \quad \lor I_2 \\
\hline
A \lor B & \quad \lor E_{1,2}
\end{align*}
\]

**Negation**

\[
\begin{align*}
\neg I_1 & \quad [A]_1 \\
\vdots & \\
\vdots & \\
\vdots & \\
\hline
\neg A & \quad \neg I_1 \\
\hline
\neg E & \quad A \quad \neg A
\end{align*}
\]
Rules for propositional logic

Implication

\[ [A]^1 \quad \vdash \quad B \quad \Rightarrow \quad A \Rightarrow B \Rightarrow E \]

Logic constants

\[ \perp \quad \vdash \quad \perp \quad \quad \top \quad \top \]

Intro to interactive theorem proving in Isabelle/HOL

Natural deduction
Intro to interactive theorem proving in Isabelle/HOL

Natural deduction

Natural deduction i Isabelle - rules


- **notI**: \((P \Rightarrow False) \Rightarrow \neg P\)
- **notE**: \([\neg P; P] \Rightarrow R\)
- **conjI**: \([P; Q] \Rightarrow P \land Q\)
- **conjunct1**: \(P \land Q \Rightarrow P\)
- **conjunct2**: \(P \land Q \Rightarrow Q\)
- **conjE**: \([P \land Q; [P; Q] \Rightarrow R] \Rightarrow R\)
- **disjI1**: \(P \Rightarrow P \lor Q\)
- **disjI2**: \(Q \Rightarrow P \lor Q\)
- **disjE**: \([P \lor Q; P \Rightarrow R; Q \Rightarrow R] \Rightarrow R\)
- **impl**: \((P \Rightarrow Q) \Rightarrow P \rightarrow Q\)
- **impE**: \([P \rightarrow Q; P; Q \Rightarrow R] \Rightarrow R\)
- **mp**: \([P \rightarrow Q; P] \Rightarrow Q\)
Applying rules

- Introduction rules apply (rule <rule_name>).
- Elimination rules apply (erule <rule_name>).
Example proof

lemma "¬(A | B) --> ¬A & ¬B"
apply (rule impI)
apply (rule conjI)
apply (rule notI)
apply (erule notE)
apply (rule disjI1)
apply assumption
apply (rule notI)
apply (erule notE)
apply (rule disjI2)
apply assumption
done
DEMO 3: NaturalDeduction.thy
Rules for first order logic

\[ allI : \forall x \cdot P x \implies \forall x \cdot P x \]

\[ allE : \left[ \forall x \cdot P x ; P ?x \implies R \right] \implies R \]

\[ exI : P ?x \implies \exists x \cdot P x \]

\[ exE : \left[ \exists x \cdot P x ; \forall x \cdot P x \implies Q \right] \implies Q \]
DEM0 3: NaturalDeduction.thy  (cont.)
Rules for classical logic

\[ ccontr \quad : \quad (\neg P \implies \text{False}) \implies P \]

\[ \text{classical} \quad : \quad (\neg P \implies P) \implies P \]
Demo

DEMO 2: NaturalDeduction.thy (cont.)
Natural deduction rules for sets

\[ UnI1 : \quad c \in A \implies c \in A \cup B \]
\[ UnI2 : \quad c \in B \implies c \in A \cup B \]
\[ UnE : \quad \begin{array}{l} [c \in A \cup B; \ c \in A \implies P; \ c \in B \implies P] \implies P \end{array} \]
\[ Intl : \quad \begin{array}{l} [c \in A; \ c \in B] \implies c \in A \cap B \end{array} \]
\[ IntE : \quad \begin{array}{l} [c \in A \cap B; \ [c \in A; \ c \in B] \implies P] \implies P \end{array} \]
\[ subsetI : \quad (\bigwedge x. \ x \in A \implies x \in B) \implies A \subseteq B \]
\[ subsetD : \quad \begin{array}{l} [A \subseteq B; \ c \in A] \implies c \in B \end{array} \]
\[ ComplI : \quad (c \in A \implies False) \implies c \in \neg A \]
\[ ComplD : \quad c \in \neg A \implies c \notin A \]
DEMO 3: NaturalDeduction.thy (cont.)
Overview

1. Introduction
2. Basic syntax and automated provers
3. Natural deduction
4. Isar: a language of structured proofs
5. Functional programming
Intro to interactive theorem proving in Isabelle/HOL

Isar: a language of structured proofs

Isar

- Imitate “pen-and-paper” proofs as much as possible
- All proofs must be understandable by reading their text, without running the prover
- Combine with powerful automation
  - Proofs should provide justification
  - Proofs should provide explanation
  - Make a balance – write readable proofs and automate trivial parts
How to teach Isar?

- Language Isar allows to express the same proof in many different ways
- Two main styles:
  - backward proofs (from the goal towards assumptions)
  - forward proofs (from the assumptions towards goals)
- Some patterns can be recognized in many proofs
- Introduce the proof language through carefully chosen examples that show different techniques and patterns
- In their previous courses students did not prove formulae of pure logic, but they have experience in proving facts about sets, relations, and functions
- Use those domains to introduce Isar
A set example

- Prove \((A \cup B)^c = A^c \cap B^c\).

Top-level proof structure:

```
lemma "\(-(A \cup B) = \neg A \cap \neg B"}
proof
  show "\(\neg (A \cup B) \subseteq \neg A \cap \neg B""
  sorry
  show "\(\neg A \cap \neg B \subseteq \neg (A \cup B)""
  sorry
qed
```
Backward proof

- Proof is either application of an automatic proof method by \ldots or consists of a \texttt{proof...qed} block.
- The keyword \texttt{proof} can be followed by a proof method that transforms the goal.
- If method is not specified the system chooses a method based on the structure of the current goal.
- In this example the method \texttt{rule equalityI} is chosen.

\[
equalityI : [A \subseteq B; B \subseteq A] \implies A = B
\]

- The proof state becomes:
  proof (state)
  goal (2 subgoals):
    1. \(-(A \cup B) \subseteq -A \cap -B
    2. \(-A \cap -B \subseteq -(A \cup B)
- The proof continues by explicitly stating and proving each goal.
Each subgoal can be proved by using a backward proof

lemma "\( - (A \cup B) = -A \cap -B \)"
proof
  show "\( - (A \cup B) \subseteq -A \cap -B \)"
  proof
    fix x
    assume "\( x \in - (A \cup B) \)"
    show "\( x \in -A \cap -B \)"
      sorry
  qed
next
  show "\( -A \cap -B \subseteq -(A \cup B) \)"
  proof
    fix x
    assume "\( x \in -A \cap -B \)"
    show "\( x \in -(A \cup B) \)"
      sorry
  qed
qed
In both subgoals the proof keywords triggers the rule subsetI method.

\[ \text{subsetI} : (\forall x. x \in A \implies x \in B) \implies A \subseteq B \]

The first subgoal becomes:

\[ \bigwedge x. x \in -(A \cup B) \implies x \in -A \cap -B \]

In Isar this becomes fix ... assume ... show ...
Let us focus on the first subgoal

At this point we have both an assumption that we can use
\( (x \in \neg (A \cup B)) \), and the goal that should be proved
\( (x \in \neg A \cap \neg B) \)

From the assumption we can deduce some new facts

We can state them using the keyword \texttt{have}:

\begin{verbatim}
fix x
assume "x \in \neg (A \cup B)"
have "x \notin A \cup B"
...
show "x \in \neg A \cap \neg B"
sorry
\end{verbatim}
Proof context

- If not stated otherwise, in each (sub)goal the prover "sees" only the formula that should be proved.
- In order to use available assumptions they must be brought in the proof context.
- There are various (equivalent) ways to do that (e.g., from, using, ...)

```isar
fix x
assume "x ∈ −(A ∪ B)"
from 'x ∈ −(A ∪ B)'
have "x ∉ A ∪ B"
  by (rule complD)
show "x ∈ −A ∩ −B"
  sorry
```

```isar
fix x
assume "x ∈ −(A ∪ B)"
have "x ∉ A ∪ B"
  using 'x ∈ −(A ∪ B)'
  by (rule complD)
show "x ∈ −A ∩ −B"
  sorry
```
Many abbreviations (some are deprecated)

this the last statement
then from this
hence then have
thus then show
with ... from this and ...

fix x
assume "x ∈ −(A ∪ B)"
then have "x ∉ A ∪ B"
    by (rule complD)
show "x ∈ −A ∩ −B"
sorry
DEMO 4: Isar.thy
Let us focus on the second subgoal

show "\(-A \cap -B \subseteq -(A \cup B)\)"
proof
  fix \(x\)
  assume "\(x \in -A \cap -B\)"
  then have "\(x \notin A\)" "\(x \notin B\)"
    by auto
  show "\(x \in -(A \cup B)\)"
proof
  assume "\(x \in A \cup B\)"
  show False
    sorry
qed
qed
Reasoning by cases (disjunction elimination)

- Currently we have that \( x \notin A \), \( x \notin B \), and \( x \in A \cup B \), and we need to derive a contradiction.

- It easily follows by considering cases \( x \in A \) and \( x \in B \) that follow from \( x \in A \cup B \).

- If we bring a disjunctive fact such as \( x \in A \cup B \) as the first fact into the proof context, the proof applies reasoning by cases (elimination rules such as UnE, disjE, ...).
... 
assume "\( x \in A \cup B \)"
then show False
proof
  assume \( x \in A \)"
  with \( x \notin A \) show False
    by - (erule notE)
next
  assume \( x \in B \)"
  with \( x \notin B \) show False
    by - (erule notE)
qed
DEMO 4: Isar.thy (cont.)
Intro to interactive theorem proving in Isabelle/HOL

Isar: a language of structured proofs

Introducing universal quantifiers

- Prove that every symmetric transitve relation, with no isolated elements is reflexive.
- Since the goal starts with the universal quantifier, `proof` automatically chooses the `allI` rule – prove the statement for an arbitrary element $x$.

```
lemma
  assumes "∀x. ∃y. R x y"
  sym: "∀x y. R x y → R y x"
  trans: "∀x y. R x y → R y x"
  shows "∀x. R x x"
proof
  fix x
  show "R x x"
    sorry
qed
```
Eliminating existential quantifiers

- We can always name an element for which we know it exists using the keyword obtain

lemma
  assumes "∀x. ∃y. R x y"
  sym: "∀x y. R x y → R y x"
  trans: "∀x y. R x y → R y x"
  shows "∀x. R x x"

proof
  fix x
  from assms(1) obtain y where "R x y"
    by auto
  with sym have "R y x"
    by auto
  from "R x y" "R y x" show "R x x"
    using trans
    by auto
qed
Proof by contradiction (rule ccontr)

- Prove the “drinkers paradox”:
  \( \exists d. \text{drinks } d \rightarrow (\forall x. \text{drinks } x) \).

lemma "\( \exists d. \text{drinks } d \rightarrow (\forall x. \text{drinks } x) \)"
proof (rule ccontr)
  assume "\( \neg \triangleleft \text{thesis} \)"
  then have "\( \forall d. \neg(\text{drinks } d \rightarrow (\forall x. \text{drinks } x)) \)"
    by auto
  then have "\( \forall d. \text{drinks } d \land \neg(\forall x. \text{drinks } x) \)"
    by auto
  then have "\((\forall d. \text{drinks } d) \land \neg(\forall x. \text{drinks } x) \)"
    by auto
  show False
    by auto
qed
Proof by cases (cases)

- Alternative proof considers cases when everybody drinks and when there is someone who does not drink.

```
lemma "\exists d. drinks d \rightarrow (\forall x. drinks x)"
proof (cases "\forall x. drinks x")
  case True
  then show ?thesis
    by auto
next
  case False
  then obtain d where "\neg drinks d"
  by auto
  then have "drinks d \rightarrow (\forall x. drinks x)"
    by auto
  then show ?thesis
    by auto
qed
```
Let us do some exercises about functions

Library predicates inj, surj, bij hold for injective, surjective, and bijective functions

$\circ$ denotes the function composition

$f \cdot A$ denotes the image of the set $A$ under the function $f$

$f \prec 'A$ denotes the inverse image of the set $A$ under the function $f$
Moreover—ultimately

- Often the final statement follows from several intermediate statements.
- A special moreover...ultimately syntax can abbreviate such proofs so that there is no need to explicitly recall intermediate statements and insert them into the proof context of the final statement.
Moreover-ultimately

lemma
  assumes "surj f" "surj g"
  shows "surj (f ◦ g)"
  unfolding surj_def
proof
  fix y
  obtain z where "f z = y"
    using 'surj f' unfolding surj_def by metis
moreover
  obtain x where "g x = z"
    using 'surj g' unfolding surj_def by metis
ultimately
  show "∃x. y = (f ◦ g)x"
    unfolding comp_def by auto
qed
Intro to interactive theorem proving in Isabelle/HOL

Isar: a language of structured proofs

Demo

DEMO 4: Isar.thy (cont.)
Numbers

- Several number types (nat, int, rational, real, complex)
- Choosing appropriate number type is sometimes essential
- Be careful to specify number type, since Isabelle often cannot automatically deduce exact number type
- You can apply all field laws on rational, real and complex numbers
- You can apply usual subtraction laws on integers, but not on naturals
- Proving inequalities is usually much harder than proving equalities
- Proving rational expressions is usually harder than proving polynomials
- ...
Many equational theorems about rational and real numbers can be proved by using simplifier with algebra_simps and field_simps collections of theorems.

lemma
  fixes x y :: real
  shows "(x + y)^2 = x^2 + 2 * x * y + y^2"
  by (simp add: power2_eq_square algebra_simps)
Equational reasoning (also-finally)

- In classical mathematics we usually reason by writing chains of equalities (or consistently oriented inequalities, mixed with equalities)
- This reasoning is implicitly based on transitivity of $=, \leq, \geq$
- Isabelle has special syntax also...finally for this type of reasoning
Equational reasoning (also-finally)

lemma
    fixes x y :: real
    shows "\((x + y)^2 = x^2 + 2 \times x \times y + y^2\)"
proof-
    have "\((x + y)^2 = (x + y) \times (x + y)\"
      sorry
    also have ". . . = x \times (x + y) + y \times (x + y)"
      sorry
    ...
    also have "\(. . . = x^2 + 2 \times x \times y + y^2\)"
      sorry
    finally show ?thesis
.
qed
- method subst ... makes a substitution of an equation (rewrite rule) within the current goal
- method subst (asm) ... makes substitution of an equation (rewrite rule) within the current assumptions
- Some basic theorems that can be used as rewrite rules:
  - add.assoc: \((x + y) + z = x + (y + z)\)
  - add.commute: \(x + y = y + x\)
  - mult.assoc: \((x * y) * z = x * (y * z)\)
  - mult.commute: \(x * y = y * x\)
  - distrib_left: \((x + y) * z = x * z + y * z\)
  - distrib_right: \(z * (x + y) = z * x + z * y\)
- theorems can be instantiated
  - add.assoc[of 1 2] gives \(1 + 2 + z = 1 + (2 + z)\)
  - add.assoc[where x=1 and y=2]
DEMOS 5: Numbers.thy
Induction over natural numbers

- Induction is the fundamental property of naturals
- Special syntactic support for induction proofs

```isar
lemma
  n :: nat
  shows "(∑ x ∈ {0.. < n + 1}) = n * (n + 1) div 2"
proof (induction n)
  case 0
  show ?case by simp
next
case (Suc n)
  then show ?case by simp
qed
```
Demo

DEMO 5: Numbers.thy (cont.)
Definition of natural numbers

- Natural numbers can be defined as an algebraic datatype

```haskell
datatype nat = Zero | Suc nat
```

- Functions can then be defined using primitive recursion

```haskell
primrec add :: "nat ⇒ nat" where
  "add x Zero = x"
| "add x (Suc y) = Suc (add x y)"
```

- Proofs can use induction over the algebraic datatype
Overview

1. Introduction
2. Basic syntax and automated provers
3. Natural deduction
4. Isar: a language of structured proofs
5. Functional programming
Algebraic datatypes and primitive recursion can be used to define classic programming datastructures (lists, trees, ...)

```haskell
datatype 'a List =
    Empty
    | Cons 'a (List 'a)
```

```haskell
datatype 'a Tree =
    Nil
    | Node 'a (Tree 'a) (Tree 'a)
```
Demo

DEMO 6: ListAndTrees.thy
Generalizing induction hypothesis

- Induction hypothesis can sometimes be too weak and must be generalized to become useful.
- Example – reverse list in linear time:

```isabelle
primrec reverse' where
  reverse' [] acc = acc |
  reverse' (x # xs) acc = reverse' xs (x # acc)
definition reverse where
  "reverse xs = reverse' xs []"
```
The induction hypothesis holds for acc, but we need it for the term \( x \# \text{acc} \).

```isabelle
lemma "\text{reverse'} xs acc = reverse xs @ acc"
apply(induction xs)
...
using this:
  reverse' xs acc = reverse xs @ acc
goal (1 subgoal):
  1. reverse' (x # xs) acc = reverse (x # xs) @ acc
...
  1. reverse' xs (x # acc) = (reverse xs @ [x]) @ acc
...
  1. reverse' xs (x # acc) = reverse xs @ (x # acc)
```

Keyword arbitrary:

```isabelle
lemma "\text{reverse'} xs acc = reverse xs @ acc"
  by (induction xs arbitrary: acc) auto
```
DEMO 6: ListAndTrees.thy (cont.)
General recursion and induction

- Not all recursion patterns correspond to primitive recursion over algebraic datatypes
- Isabelle supports general recursion
- Defining general recursive functions requires proving their termination (that can sometimes be done automatically)
  - `fun` define a general recursive function and prove its termination automatically
  - `function` define a general recursive function and leave proving its termination to the user
- It is sometimes possible to define partial functions, that terminate only for some values in their domain
fun (sequential) power :: "nat ⇒ nat ⇒ nat" where
  "power x 0 = 1"
| "power x n =
  (if n mod 2 = 0 then
   power (x * x) (n div 2)
  else
   x * power x (n - 1))"
Intro to interactive theorem proving in Isabelle/HOL

- Functional programming

Demo

DEMO 7: GeneralRecursion.thy
Axiomatic reasoning

- Two ways to specify axioms
  - axiomatization
  - locale (fixes constants and assumes their properties)
- Locales can be interpreted
  - ensures that axioms are consistent
  - theorems proved abstractly become available for a concrete interpretation
locale Geometry =
  fixes cong :: \'a ⇒ \'a ⇒ \'a ⇒ \'a ⇒ bool
  assumes cong_refl: "∀ x y. cong x y y x"
  assumes cong_id: "∀ x y z. cong x y z z ⇒ x = y"
...
begin
  definition ...
  lemma ...
end
type_synonym point_R2 = "real × real"

fun dist :: "point_R2 ⇒ point_R2" where
dist (x₁, y₁) (x₂, y₂) = (x₂ − x₁)² + (y₂ − y₁)²

definition cong_R2 :: "point_R2 ⇒ point_R2 ⇒ point_R2 ⇒ point_R2 ⇒ bool" where
cong_R2 x y z w ←→ dist x y = dist z w

interpretation Geometry_R2: Geometry cong_R2
proof
  
  qed
Demo

DEMO 8: Locales.thy