Intro to interactive theorem proving in Isabelle/HOL

Filip Marić

*Faculty of Mathematics,
University of Belgrade

PAT 2023

Table of contents

Overview

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/`)

Isabelle/HOL

- 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 understaning and provide rigour to elementary high-school/undergraduate mathematics and computer science?
    - As close to every day 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

Introductory example

- 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:

$\lvert a\rvert \geq 0$	Non-negativity
$\lvert a\rvert = 0 \iff a = 0$	Positive-definiteness
$\lvert ab\rvert = \lvert a\rvert \cdot \lvert b\rvert$	Multiplicativity
$\lvert a + b\rvert \leq \lvert a\rvert + \lvert b\rvert$	Subadditivity, specifically the triangle inequality

Wikipedia proof

"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 ± 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

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 somene 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.

Demo

DEMO 2: BasicSyntax.thy

Overview

Natural deduction

- 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

$$\frac{A \qquad B}{A \wedge B}\ \wedge I \qquad\qquad \frac{A \wedge B}{A}\ \wedge E1 \qquad\qquad \frac{A \wedge B}{B}\ \wedge E2$$

Disjunction

$$\frac{A}{A \vee B}\ \vee I1 \qquad\qquad \frac{B}{A \vee B}\ \vee I2 \qquad\qquad \frac{A \vee B \qquad \begin{array}{c}[A]^1\\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B]^2\\ \vdots \\ C\end{array}}{C}\ \vee E^{1,2}$$

Negation

$$\frac{\begin{array}{c}[A]^1\\ \vdots \\ \bot\end{array}}{\neg A}\ \neg I^1 \qquad\qquad \frac{A \qquad \neg A}{\bot}\ \neg E$$

Rules for propositional logic

Implication

$$\dfrac{\begin{matrix}[A]^1 \\ \vdots \\ B\end{matrix}}{A \Rightarrow B} \Rightarrow I^1$$

$$\dfrac{A \qquad A \Rightarrow B}{B} \Rightarrow E$$

Logic constants

$$\dfrac{\bot}{A} \bot E$$

$$\dfrac{}{\top} \top I$$

Natural deduction i Isabelle - rules

$$
\begin{array}{rcl}
notI & : & (P \Longrightarrow False) \Longrightarrow \neg P \\
notE & : & [\![\neg P; P]\!] \Longrightarrow R \\
conjI & : & [\![P; Q]\!] \Longrightarrow P \wedge Q \\
conjunct1 & : & P \wedge Q \Longrightarrow P \\
conjunct2 & : & P \wedge Q \Longrightarrow Q \\
conjE & : & [\![P \wedge Q; [\![P; Q]\!] \Longrightarrow R]\!] \Longrightarrow R \\
disjI1 & : & P \Longrightarrow P \vee Q \\
disjI2 & : & Q \Longrightarrow P \vee Q \\
disjE & : & [\![P \vee Q; P \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R \\
impI & : & (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q \\
impE & : & [\![P \longrightarrow Q; P; Q \Longrightarrow R]\!] \Longrightarrow R \\
mp & : & [\![P \longrightarrow Q; P]\!] \Longrightarrow Q
\end{array}
$$

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

DEMO 3: NaturalDeduction.thy

Rules for first order logic

$$
\begin{aligned}
allI &: \bigwedge x.\ P\ x \Longrightarrow \forall x.\ P\ x \\
allE &: [\![\forall x.\ P\ x;\ P\ ?x \Longrightarrow R]\!] \Longrightarrow R \\
exI &: P\ ?x \Longrightarrow \exists x.\ P\ x \\
exE &: [\![\exists x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q]\!] \Longrightarrow Q
\end{aligned}
$$

Demo

DEMO 3: NaturalDeduction.thy (cont.)

Rules for classical logic

$$
\begin{aligned}
\mathit{ccontr} &: (\neg P \Longrightarrow \mathit{False}) \Longrightarrow P \\
\mathit{classical} &: (\neg P \Longrightarrow P) \Longrightarrow P
\end{aligned}
$$

Demo

DEMO 2: NaturalDeduction.thy (cont.)

Natural deduction rules for sets

$$
\begin{aligned}
\mathit{UnI1} &: \quad c \in A \Longrightarrow c \in A \cup B \\
\mathit{UnI2} &: \quad c \in B \Longrightarrow c \in A \cup B \\
\mathit{UnE} &: \quad [\![c \in A \cup B; c \in A \Longrightarrow P; c \in B \Longrightarrow P]\!] \Longrightarrow P \\
\mathit{IntI} &: \quad [\![c \in A; c \in B]\!] \Longrightarrow c \in A \cap B \\
\mathit{IntE} &: \quad [\![c \in A \cap B; [\![c \in A; c \in B]\!] \Longrightarrow P]\!] \Longrightarrow P \\
\mathit{subsetI} &: \quad (\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B \\
\mathit{subsetD} &: \quad [\![A \subseteq B; c \in A]\!] \Longrightarrow c \in B \\
\mathit{ComplI} &: \quad (c \in A \Longrightarrow \mathit{False}) \Longrightarrow c \in -A \\
\mathit{ComplD} &: \quad c \in -A \Longrightarrow c \notin A
\end{aligned}
$$

Demo

DEMO 3: NaturalDeduction.thy (cont.)

Overview

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 an patterns
- In their previous courses student 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 "$-(A \cup B) = -A \cap -B$"
proof
  show "$-(A \cup B) \subseteq -A \cap -B$"
    sorry
  show "$-A \cap -B \subseteq -(A \cup B)$"
    sorry
qed

Backward proof

- Proof is either applcation of an automatic proof method `by ...` or consists of a `proof...qed` block
- The keyword `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 `rule equalityI` is chosen

$$equalityI \quad : \quad [\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B$$

- 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 ∪ B) = −A ∩ −B"
proof
  show "−(A ∪ B) ⊆ −A ∩ −B"
  proof
    fix x
    assume "x ∈ −(A ∪ B)"
    show "x ∈ −A ∩ −B"
      sorry
  qed
next
  show "−A ∩ −B ⊆ −(A ∪ B)"
  proof
    fix x
    assume "x ∈ −A ∩ −B"
    show "x ∈ −(A ∪ B)"
      sorry
  qed
qed
```

- In both subgoals the `proof` keywords triggers the `rule subsetI` method.

$$subsetI \quad : \quad (\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$$

The first subgoal becomes:

$$\bigwedge x.\ x \in -(A \cup B) \Longrightarrow x \in -A \cap -B$$

In Isar this becomes `fix ... assume ... show ...`

Forward proof

- Let us focus on the first subgoal
- At this point we have both an assumption that we can use ($x \in -(A \cup B)$), and the goal that should be proved ($x \in -A \cap -B$)
- From the assumption we can deduce some new facts
- We can state them using the keyword `have`:

```
fix x
assume "x ∈ −(A ∪ B)"
have "x ∉ A ∪ B"
    ...
show "x ∈ −A ∩ −B"
    sorry
```

Proof context

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

fix x
assume "$x \in -(A \cup B)$"
from '$x \in -(A \cup B)$'
have "$x \notin A \cup B$"
    by (rule complD)
show "$x \in -A \cap -B$"
    sorry

fix x
assume "$x \in -(A \cup B)$"
have "$x \notin A \cup B$"
    using '$x \in -(A \cup B)$'
    by (rule complD)
show "$x \in -A \cap -B$"
    sorry

Abbreviations

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 "$x \in -(A \cup B)$"
then have "$x \notin A \cup B$"
    by (rule complD)
show "$x \in -A \cap -B$"
    sorry

Demo

DEMO 4: Isar.thy

Reasoning by cases

- Let us focus on the second subgoal

```
show "−A ∩ −B ⊆ −(A ∪ B)"
proof
  fix x
  assume "x ∈ −A ∩ −B"
  then have "x ∉ A" "x ∉ B"
    by auto
  show "x ∈ −(A ∪ B)"
  proof
    assume "x ∈ A ∪ 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

DEMO 4: Isar.thy (cont.)

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 "$\forall x.\ \exists y.\ R\ x\ y$"
    sym: "$\forall x\, y.\ R\ x\ y \longrightarrow R\ y\ x$"
    trans: "$\forall x\, y.\ R\ x\ y \longrightarrow R\ y\ x$"
  shows "$\forall x.\ R\ x\ x$"
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 "$\forall x.\ \exists y.\ R\ x\ y$"
        sym: "$\forall x\, y.\ R\ x\ y \longrightarrow R\ y\ x$"
        trans: "$\forall x\, y.\ R\ x\ y \longrightarrow R\ y\ x$"
    shows "$\forall x.\ R\ x\ x$"
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":
  $\exists d.\ drinks\ d \longrightarrow (\forall x.\ drinks\ x)$.

```
lemma "∃d. drinks d ⟶ (∀x. drinks x)"
proof (rule ccontr)
   assume "¬ ?thesis"
   then have "∀d.¬(drinks d ⟶ (∀x. drinks x))"
      by auto
   then have "∀d.drinks d ∧ ¬(∀x. drinks x)"
      by auto
   then have "(∀d.drinks d) ∧ ¬(∀x. 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 "$\exists d.\ drinks\ d \longrightarrow (\forall x.\ drinks\ x)$"
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 "$drinks\ d \longrightarrow (\forall x.\ drinks\ x)$"
        by auto
    then show ?thesis
        by auto
qed

Exercises

- Let us do some exercises about functions
- Library predicates `inj`, `surj`, `bij` hold for injective, surjective, and bijective functions
- $f \circ g$ denotes the function composition
- f ‘ A denotes the image of the set A under the function f
- f $-$‘ A denotes the inverse image 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 $(f \circ g)$"
    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 "$\exists x.\ y = (f \circ g) x$"
        unfolding comp_def by auto
qed

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 then proving equalities
- Proving rational expressions is usually harder than proving polynomials
- ...

Field simps

- 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 "$(x+y)^2 = x^2 + 2*x*y + y^2$"
proof-
  have "$(x+y)^2 = (x+y)*(x+y)$"
    sorry
  also have "$\ldots = x*(x+y) + y*(x+y)$"
    sorry
  ...
  also have "$\ldots = x^2 + 2*x*y + y^2$"
    sorry
  finally show ?thesis
  .
qed

- method `subst ...` makes a substitution of an eqation (rewrite rule) within the current goal
- method `subst (asm) ...` makes substitution of an eqation (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]

Demo

DEMO 5: Numbers.thy

Induction over natural numbers

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

lemma

  n :: nat

  shows "$(\sum x \in \{0.. < n+1\}) = n * (n+1)$ 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.)

Defining natural numbers

- Natural numbers can be defined as an algebraic datatype

  datatype nat = Zero | Suc nat

- Functions can then be defined using primitive recursion

  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

Datastructures

- Algebraic datatypes and primitive recursion can used to define classic programming datastructures (lists, trees, . . .)

```
datatype 'a List =
    Empty
|   Cons 'a "'a List"

datatype 'a Tree =
    Nil
|   Node 'a "'a Tree" "'a Tree"
```

Demo

DEMO 6: ListAndTrees.thy

Generalizing induction hypothesis

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

```
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 # acc`.

  lemma "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`:

  lemma "reverse' xs acc = reverse xs @ acc"
      by (induction xs arbitrary: acc) auto

Demo

DEMO 6: ListAndTrees.thy (cont.)

General recursion and induction

- Not all recursion patters 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))"
```

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

Locales

```
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 (x_1, y_1) $(x_2, y_2) = (x_2 - x_1)^2 + (y_2 - y_1)^2$

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