Automated Reasoning

Coursework lecture: **Proving and Reasoning in Isabelle/HOL**

Imogen I. Morris

20/10/2017

Coursework overview

- Part 1: Propositional and first-order proofs [40%]
- Part 2: Geometry with order and signed areas [60%]

Part 1: Propositional and first-order proofs

- Procedural proofs (sequence of rule applications).

Part 1: Propositional and first-order proofs

- Procedural proofs (sequence of rule applications).
- You are given a list of rules you may use.

Part 1: Propositional and first-order proofs

- Procedural proofs (sequence of rule applications).
- You are given a list of rules you may use.
- View them using `thm` *rule*.

Knights and Knaves problems

- You meet two inhabitants: Sue and Zippy. Sue says that Zippy is a knave. Zippy says, ‘I and Sue are knights.’
- Can you determine who is a knight and who is a knave?

Knights and Knaves problems

- You meet two inhabitants: Sue and Zippy. Sue says that Zippy is a knave. Zippy says, ‘I and Sue are knights.’
- Can you determine who is a knight and who is a knave?
- The most natural way to solve this problem is to reason by cases.

Knights and Knaves problems

- You meet two inhabitants: Sue and Zippy. Sue says that Zippy is a knave. Zippy says, ‘I and Sue are knights.’
- Can you determine who is a knight and who is a knave?
- The most natural way to solve this problem is to reason by cases.
- In Isabelle we can use `case_tac`. E.g. `(case_tac "V x")`. We then have two subgoals: $\texttt{V x} \Longrightarrow \mathit{goal}$ and $\neg\, \texttt{V x} \Longrightarrow \mathit{goal}$.

Knights and Knaves problems

- We formalise 'a is a knave' as $V\ a$ and 'a is a knight' as $G\ a$.

Knights and Knaves problems

- We formalise 'a is a knave' as $V\ a$ and 'a is a knight' as $G\ a$.
- 'Person a says statement P' is formalised as $S\ n\ a = P$, where n is some natural number.

Knights and Knaves problems

- We formalise 'a is a knave' as $V\ a$ and 'a is a knight' as $G\ a$.
- 'Person a says statement P' is formalised as $S\ n\ a = P$, where n is some natural number.
- We index the statement by n because one person may make more than one statement.

Knights and Knaves problems

- We formalise 'a is a knave' as $V\ a$ and 'a is a knight' as $G\ a$.
- 'Person a says statement P' is formalised as $S\ n\ a = P$, where n is some natural number.
- We index the statement by n because one person may make more than one statement.
- We are also assuming that the domain of the quantifiers is all the inhabitants of the island (so you, as a visitor to the island, are not included).

Knights and Knaves problems

- We can formalise the previous problem:
- `S 1 s = V z` and `S 1 z = K s` $\wedge$ `K z`.

Knights and Knaves problems

- We can formalise the previous problem:
- `S 1 s = V z` and `S 1 z = K s` $\wedge$ `K z`.
- Solution: Zippy cannot be a knight, because if what he said was true, then Sue would be telling a lie and then she is not a knight - contradiction. Hence Zippy is a knave, and as Sue is telling the truth, she is a knight.

Knights and Knaves problems

- We can formalise the previous problem:
- `S 1 s = V z` and `S 1 z = K s` $\wedge$ `K z`.
- Solution: Zippy cannot be a knight, because if what he said was true, then Sue would be telling a lie and then she is not a knight - contradiction. Hence Zippy is a knave, and as Sue is telling the truth, she is a knight.
- Suppose we get the formalisation wrong:
- `S 1 s = V z` and `S 1 z = K s` and `S 2 z = K z`.

Knights and Knaves problems

- We can formalise the previous problem:
- `S 1 s = V z` and `S 1 z = K s ∧ K z`.
- Solution: Zippy cannot be a knight, because if what he said was true, then Sue would be telling a lie and then she is not a knight - contradiction. Hence Zippy is a knave, and as Sue is telling the truth, she is a knight.
- Suppose we get the formalisation wrong:
- `S 1 s = V z` and `S 1 z = K s` and `S 2 z = K z`.
- The previous analysis holds, thus Zippy is a knave, yet he makes a true statement (that Sue is a knight), so Zippy is a knight. Hence the problem is unsolvable.

Part 2: Structured proofs & powerful reasoning tools

- Isabelle has a lot of machinery built in for presentation, interaction and automation.

Part 2: Structured proofs & powerful reasoning tools

- Isabelle has a lot of machinery built in for presentation, interaction and automation.
- Structured proofs (also called *declarative*); the name of the language is *Isar*.

Part 2: Structured proofs & powerful reasoning tools

- Isabelle has a lot of machinery built in for presentation, interaction and automation.
- Structured proofs (also called *declarative*); the name of the language is *Isar*.
- Powerful automatic tools: `simp`, `auto`, `safe`, `blast`, `fast`, `force`, `fastforce`, `linarith`, `arith`, `presburger`, `algebra`, `meson`, `metis`.

Part 2: Structured proofs & powerful reasoning tools

- Isabelle has a lot of machinery built in for presentation, interaction and automation.
- Structured proofs (also called *declarative*);
  the name of the language is *Isar*.
- Powerful automatic tools: `simp`, `auto`, `safe`, `blast`, `fast`, `force`, `fastforce`, `linarith`, `arith`, `presburger`, `algebra`, `meson`, `metis`.
- A link to external provers: **sledgehammer**.

Reasoning with equality $(=)$

Rules:

$$\frac{t = s \qquad P\,s}{P\,t}\ \text{subst} \qquad\qquad \frac{s = t \qquad P\,s}{P\,t}\ \text{ssubst}$$

$$\frac{}{t = t}\ \text{refl} \qquad\qquad \frac{s = t}{t = s}\ \text{sym} \qquad\qquad \frac{r = s \qquad s = t}{r = t}\ \text{trans}$$

$$\frac{\forall x.\ f\,x = g\,x}{f = g}\ \text{ext}$$

Reasoning with equality $(=)$

Rules:

$$\frac{t = s \qquad P\,s}{P\,t}\,\text{subst} \qquad\qquad \frac{s = t \qquad P\,s}{P\,t}\,\text{ssubst}$$

$$\frac{}{t = t}\,\text{refl} \qquad\qquad \frac{s = t}{t = s}\,\text{sym} \qquad\qquad \frac{r = s \qquad s = t}{r = t}\,\text{trans}$$

$$\frac{\forall x.\ f\,x = g\,x}{f = g}\,\text{ext}$$

Are all of these rules necessary, or can some of them be derived from the others?

Reasoning with equality $(=)$

Output:

1. $\forall c.\ \exists\, a\, b.\ a + 3 * b = c$

mult_zero_right: a * 0 = 0

add_0: 0 + a = a

add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists\, a\, b.\ a + 3 * b = c$"

Reasoning with equality (=)

Output:
1. $\bigwedge c.\ \exists a\, b.\ a + 3 * b = c$

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists a\, b.\ a + 3 * b = c$"
apply (rule allI)

Reasoning with equality (=)

Output:
1. $\bigwedge c.\ \exists b.\ c + 3 * b = c$

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists\, a\, b.\ a + 3 * b = c$"
  apply (rule allI)
  apply (rule_tac x = c in exI)

Reasoning with equality (=)

Output:
1. $\bigwedge c.\ c + 3 * 0 = c$

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists a\, b.\ a + 3 * b = c$"
  apply (rule allI)
  apply (rule_tac x = c in exI)
  apply (rule_tac x = 0 in exI)

Reasoning with equality (=)

Output:
1. $\bigwedge c.\ 3 * 0 = 0$
2. $\bigwedge c.\ c + 0 = c$

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists\, a\, b.\ a + 3 * b = c$"
apply (rule allI)
apply (rule_tac x = c in exI)
apply (rule_tac x = 0 in exI)
apply (rule_tac s = 0 and t = "3 * 0" in ssubst)

Reasoning with equality (=)

Output:
1. $\bigwedge c.\ c + 0 = c$

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists a\, b.\ a + 3 * b = c$"
apply (rule allI)
apply (rule_tac x = c in exI)
apply (rule_tac x = 0 in exI)
apply (rule_tac s = 0 and t = "$3 * 0$" in ssubst)
apply (rule mult_zero_right)

Reasoning with equality (=)

Output:

1. $\bigwedge c.\ c + 0 = 0 + c$
2. $\bigwedge c.\ 0 + c = c$

mult_zero_right: a * 0 = 0

add_0: 0 + a = a

add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists a\, b.\ a + 3 * b = c$"

apply (rule allI)

apply (rule_tac x = c in exI)

apply (rule_tac x = 0 in exI)

apply (rule_tac s = 0 and t = "$3 * 0$" in ssubst)

apply (rule mult_zero_right)

apply (rule_tac s = "$0 + c$" and t = "$c + 0$" in ssubst)

Reasoning with equality (=)

Output:
1. $\bigwedge c.\ 0 + c = c$

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists a\, b.\ a + 3 * b = c$"
apply (rule allI)
apply (rule_tac x = c in exI)
apply (rule_tac x = 0 in exI)
apply (rule_tac s = 0 and t = "$3 * 0$" in ssubst)
apply (rule mult_zero_right)
apply (rule_tac s = "$0 + c$" and t = "$c + 0$" in ssubst)
apply (rule add.commute)

Reasoning with equality (=)

Output:
No subgoals!

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists a\, b.\ a + 3 * b = c$"
apply (rule allI)
apply (rule_tac x = c in exI)
apply (rule_tac x = 0 in exI)
apply (rule_tac s = 0 and t = "$3 * 0$" in ssubst)
apply (rule mult_zero_right)
apply (rule_tac s = "$0 + c$" and t = "$c + 0$" in ssubst)
apply (rule add.commute)
apply (rule add_0)

Reasoning with equality (=)

Output:
No subgoals!

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

lemma "$\forall c :: \texttt{int}.\ \exists a\, b.\ a + 3 * b = c$"
apply (rule allI)
apply (rule_tac x = c in exI)
apply (rule_tac x = 0 in exI)
apply (rule_tac s = 0 and t = "$3 * 0$" in ssubst)
apply (rule mult_zero_right)
apply (rule_tac s = "$0 + c$" and t = "$c + 0$" in ssubst)
apply (rule add.commute)
apply (rule add_0)
done

Reasoning with equality (=)

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

Output:
No subgoals!

lemma "$\forall c :: \texttt{int}.\ \exists a\, b.\ a + 3 * b = c$"
  apply (rule allI)
  apply (rule_tac x = c in exI)
  apply (rule_tac x = 0 in exI)
  apply (subst mult_zero_right)
  apply (subst add.commute)
  apply (rule add_0)
  done

We can save all that variable instantiation using `subst`: rewriting.

But we will be using Isar:

Without subst:

```
lemma "∀c :: int. ∃a b. a + 3 * b = c"
proof
  fix c :: int
  have "c + 3 * 0 = c + 0"
    by (rule_tac s = 0 and
    t = "3 * 0" in ssubst,
    rule mult_zero_right, rule refl)
  also have "... = 0 + c"
    by (rule add.commute)
  also have "... = c" by (rule add_0)
  finally have "c + 3 * 0 = c"
    by (rule trans, rule_tac refl)
  then have "∃b. c + 3 * b = c"
    by (rule exI)
  then show "∃a b. a + 3 * b = c"
    by (rule exI)
qed
```

With subst:

```
lemma "∀c :: int. ∃a b. a + 3 * b = c"
proof
  fix c :: int
  have "c + 3 * 0 = c + 0"
    by (subst mult_zero_right,
    rule refl)
  also have "... = 0 + c"
    by (rule add.commute)
  also have "... = c" by (rule add_0)
  finally have "c + 3 * 0 = c"
    by (rule trans, rule_tac refl)
  then have "∃b. c + 3 * b = c"
    by (rule exI)
  then show "∃a b. a + 3 * b = c"
    by (rule exI)
qed
```

Useful attributes to use with `subst`

`symmetric`: This swaps the left and right hand sides of the equality in *theorem*.

Usage: `subst` *theorem*`[symmetric]`

`asm`: This allows substitution into the assumption rather than the conclusion.

Usage: `subst(asm)` *theorem*

n, where n is a natural number: This allows substitution with the n^{th} occurrence in the goal of an expression that can be unified with the left-hand side of *theorem*.

Usage: `subst(`n`)` *theorem*

Reasoning with equality (=)

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

Output:
No subgoals!

lemma "$\forall c :: \texttt{int}.\ \exists\, a\, b.\ a + 3 * b = c$"
apply (rule allI)
apply (rule_tac x = c in exI)
apply (rule_tac x = 0 in exI)
apply (simp only: mult_zero_right add.commute add_0)
done

Method `simp` does substitution automatically (given the right rules!).

Reasoning with equality (=)

mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a

Output:
No subgoals!

lemma "$\forall c :: int.\ \exists a\, b.\ a + 3 * b = c$"
  apply (rule allI)
  apply (rule_tac x = c in exI)
  apply (rule_tac x = 0 in exI)
  apply simp
  done

Method `simp` does substitution automatically (given the right rules!).
...and the right rules are already in the Main library.

Isabelle's powerful tools

- **simp**: rewriting using equations.
  Uses:
  ```
  apply simp
  apply (simp add: eq_1 ... eq_n)
  apply (simp only: eq_1 ... eq_n)
  apply (simp del: eq_1 ... eq_n)
  ```

Isabelle's powerful tools

- **simp**: rewriting using equations.

  Uses:
  ```
  apply simp
  apply (simp add: eq1 ... eqn)
  apply (simp only: eq1 ... eqn)
  apply (simp del: eq1 ... eqn)
  ```

- **auto**: rewriting + proof search (using classical logic).

  Uses:
  ```
  apply auto
  apply (auto simp add: eq1 ... eqn)
  apply (auto simp only: eq1 ... eqn)
  apply (auto simp del: eq1 ... eqn)
  ```

Isabelle's powerful tools

- **simp**: rewriting using equations.
  Uses:
  ```
  apply simp
  apply (simp add: eq1 ... eqn)
  apply (simp only: eq1 ... eqn)
  apply (simp del: eq1 ... eqn)
  ```
- **auto**: rewriting + proof search (using classical logic).
  Uses:
  ```
  apply auto
  apply (auto simp add: eq1 ... eqn)
  apply (auto simp only: eq1 ... eqn)
  apply (auto simp del: eq1 ... eqn)
  ```
- Others: **blast**, **fast**, **force**, **fastforce**, **safe**, **algebra**, **linarith**, **arith**, **presburger**, **meson**, **metis**.

Isabelle's powerful tools

Sledgehammer

Sledgehammer

- Tool for invoking external provers.

Sledgehammer

- Tool for invoking external provers.
- Isabelle should not just trust external provers.

Isabelle's powerful tools

Sledgehammer

- Tool for invoking external provers.
- Isabelle should not just trust external provers.
- Sledgehammer tries to reconstruct proof inside Isabelle.

Sledgehammer

- Tool for invoking external provers.
- Isabelle should not just trust external provers.
- Sledgehammer tries to reconstruct proof inside Isabelle.
- Usually, `metis` will do the job, given a list of lemmas suggested by sledgehammer.

Sledgehammer

- Tool for invoking external provers.
- Isabelle should not just trust external provers.
- Sledgehammer tries to reconstruct proof inside Isabelle.
- Usually, `metis` will do the job, given a list of lemmas suggested by sledgehammer.

lemma "$inj\_on\ f\ A \Longrightarrow$
$\exists g.\ g' f' A \subseteq A \wedge (\forall a \in A.\ g(f\ a) = a) \wedge (\forall b \in A.\ f(g(f\ b)) = f\ b)$"
sledgehammer

Sledgehammer

- Tool for invoking external provers.
- Isabelle should not just trust external provers.
- Sledgehammer tries to reconstruct proof inside Isabelle.
- Usually, `metis` will do the job, given a list of lemmas suggested by sledgehammer.

lemma "$inj\_on\ f\ A \Longrightarrow$
$\exists g.\ g' f' A \subseteq A \wedge (\forall a \in A.\ g(f\ a) = a) \wedge (\forall b \in A.\ f(g(f\ b)) = f\ b)$"
by (metis order_refl the_inv_into_f_f the_inv_into_onto)

Isabelle's powerful tools

Useful commands:

Isabelle's powerful tools

Useful commands:

- **try0**: tries a bunch of internal provers (`auto`, `simp`, . . .).

Isabelle's powerful tools

Useful commands:

- **try0**: tries a bunch of internal provers (`auto`, `simp`, . . .).
- **try**: `try0` + `sledgehammer` + counterexample checkers!

Isabelle's powerful tools

Useful commands:

- **try0**: tries a bunch of internal provers (`auto`, `simp`, ...).
- **try**: `try0` + `sledgehammer` + counterexample checkers!
- Use them just like `sledgehammer`.

Part 2: Geometry with order and signed areas [60%]

Part 2: Geometry with order and signed areas [60%]

- Getting familiar with axiomatic systems.

Part 2: Geometry with order and signed areas [60%]

- Getting familiar with axiomatic systems.
- In particular, Isabelle's *locales*.

Part 2: Geometry with order and signed areas [60%]

- Getting familiar with axiomatic systems.
- In particular, Isabelle's *locales*.
- We will define familiar geometric objects in terms of new concepts (order, signed area).

Part 2: Geometry with order and signed areas [60%]

- Getting familiar with axiomatic systems.
- In particular, Isabelle's *locales*.
- We will define familiar geometric objects in terms of new concepts (order, signed area).
- It will help if we relate the formal statements to our geometric intuition.

Signed area

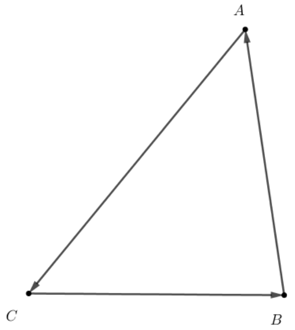
- You will be given a locale defining a function Δ.

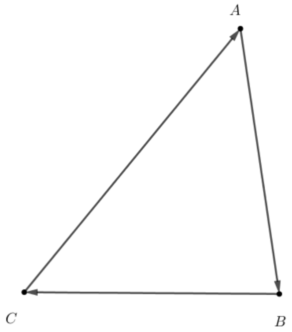
Signed area

- You will be given a locale defining a function Δ.
- We can interpret $\Delta\, x\, y\, z$ as the signed area of a triangle defined by the three arguments, x, y and z, of Δ.

Signed area

- You will be given a locale defining a function Δ.
- We can interpret $\Delta\, x\, y\, z$ as the signed area of a triangle defined by the three arguments, x, y and z, of Δ.
- The signed area of a triangle is just the area of that triangle, multiplied by -1 if the points of that triangle are traversed clockwise, and by 1 otherwise.





Relating the formal statement to geometry

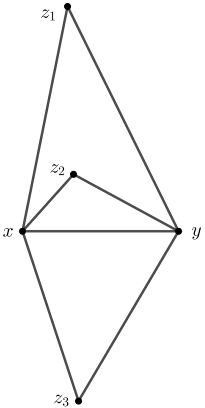
- Take as an example Axiom 2 from the locale:
  `"x ≠y ⟹∃z. (R::real)= Δx y z"`.

Relating the formal statement to geometry

- Take as an example Axiom 2 from the locale:
  `"x ≠y ⟹∃z. (R::real)= Δx y z"`.
- Geometrically it says given two distinct points we can construct a triangle with any area (even negative)

Relating the formal statement to geometry

- Take as an example Axiom 2 from the locale:
  `"x ≠y ⟹∃z. (R::real)= Δx y z"`.
- Geometrically it says given two distinct points we can construct a triangle with any area (even negative)



Hints for proving together with Isabelle

- Always solve the problems in your head (or on paper), before applying rules!

Hints for proving together with Isabelle

- Always solve the problems in your head (or on paper), before applying rules!
- If in your proof in paper it's clear that results P and Q are used in the proof, then try **using** P Q **sledgehammer**
- This gives the provers a hint.

Hints for proving together with Isabelle

- Always solve the problems in your head (or on paper), before applying rules!
- If in your proof in paper it's clear that results P and Q are used in the proof, then try **using** P Q **sledgehammer**
- This gives the provers a hint.
- Preinstantiate variables when trying to use a result in a proof: **using** P[**where** x = "some term"] Q **sledgehammer**.

Hints for proving together with Isabelle

- Always solve the problems in your head (or on paper), before applying rules!
- If in your proof in paper it's clear that results P and Q are used in the proof, then try **using** P Q **sledgehammer**
- This gives the provers a hint.
- Preinstantiate variables when trying to use a result in a proof: **using** P[**where** x = "some term"] Q **sledgehammer**.
- When in doubt *add brackets*.

Hints for proving together with Isabelle

- Always solve the problems in your head (or on paper), before applying rules!
- If in your proof in paper it's clear that results P and Q are used in the proof, then try **using** P Q **sledgehammer**
- This gives the provers a hint.
- Preinstantiate variables when trying to use a result in a proof: **using** P[**where** x = “some term”] Q **sledgehammer**.
- When in doubt *add brackets*.
- When in doubt *add type constraints*.

Hints for proving together with Isabelle

- Always solve the problems in your head (or on paper), before applying rules!
- If in your proof in paper it's clear that results P and Q are used in the proof, then try **using** P Q **sledgehammer**
- This gives the provers a hint.
- Preinstantiate variables when trying to use a result in a proof: **using** P[**where** x = "some term"] Q **sledgehammer**.
- When in doubt *add brackets*.
- When in doubt *add type constraints*.
- During a proof, if you know your goal is unprovable (e.g., false), go back one step!

Hints for proving together with Isabelle

- Always solve the problems in your head (or on paper), before applying rules!
- If in your proof in paper it's clear that results P and Q are used in the proof, then try **using** P Q **sledgehammer**
- This gives the provers a hint.
- Preinstantiate variables when trying to use a result in a proof: **using** P[**where** x = "some term"] Q **sledgehammer**.
- When in doubt *add brackets*.
- When in doubt *add type constraints*.
- During a proof, if you know your goal is unprovable (e.g., false), go back one step!
- Counterexample checkers (Quickcheck, Nitpick) can help you realise you made a wrong turn. Either call them directly (typing **quickcheck** or **nitpick**), or simply type **try**. (Especially important for knights and knaves).

More hints

More hints

- Start early.

More hints

- Start early.
- Go to the lab sessions.

More hints

- Start early.
- Go to the lab sessions.
- Contact me by email: I.I.Morris@sms.ed.ac.uk.