Automated Reasoning

Coursework lecture:

Proving and Reasoning in Isabelle/HOL

Daniel Raggi

18/10/2016

Coursework overview

- Part 1: Propositional and first-order proofs [15%]
- Part 2: Structured proofs & powerful reasoning tools [25%]
- Part 3: Reasoning about Geometries [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).
- Introduction and elimination rules.

Part 1: Propositional and first-order proofs

- Procedural proofs (sequence of rule applications).
- Introduction and elimination rules.
- You should be reasonably skilled with these things by now.

Part 2: Structured proofs & powerful reasoning tools

- You will solve complex problems.

Part 2: Structured proofs & powerful reasoning tools

- You will solve complex problems.
- Isabelle has a lot of machinery built in for presentation, interaction and automation.

Part 2: Structured proofs & powerful reasoning tools

- You will solve complex problems.
- 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

- You will solve complex problems.
- 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

- You will solve complex problems.
- 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:

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

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

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

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 "$\forall c$ :: `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 (=)

Output:
No subgoals!

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

```
lemma "∀c :: int. ∃ 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.

Reasoning with equality (=)

Output:
No subgoals!

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

lemma "$\forall c$ :: 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 that automatically (given the right rules!).

Reasoning with equality (=)

Output:
No subgoals!

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

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

Method `simp` does that 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**.

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 "$\mathit{inj\_on}\ f\ A \Longrightarrow$
$\exists g.\ g`f`A \subseteq A \wedge (\forall a \in A.\ g(f\ a) = a) \wedge (\forall b \in A.\ f(g(f\ b)) = f\ b)$"
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`.

Structured proof

- The proofs you can build right now read like a list of instructions.

Structured proof

- The proofs you can build right now read like a list of instructions.
- Mathematical proofs (and written arguments in general) don't look like that!

Structured proof

- The proofs you can build right now read like a list of instructions.
- Mathematical proofs (and written arguments in general) don't look like that!
- Structured proofs look much more like maths.

Structured proof

- The proofs you can build right now read like a list of instructions.
- Mathematical proofs (and written arguments in general) don't look like that!
- Structured proofs look much more like maths.

lemma "$\forall c :: \texttt{int}.\ \exists\, a\, b.\ a + 3 * b = c$"
proof (writing nothing after 'proof' applies a default rule; e.g. `allI`)
  fix c::`int`
  have "$c + 3 * 0 = c$" **by** simp
  thus "$\exists\, a\, b.\ a + 3 * b = c$" **by** blast
qed

Structured proof

Example (using keywords **assume**, **obtain** and **hence**/**then** **have**/**from** $\cdots$ **have**):

```
lemma "∀c :: int. (∃a. 4 * a = c) ⟶ (∃b. 2 * b = c)"
proof (rule allI, rule impI)
  fix c :: int
  assume "∃a. 4 * a = c"
  then obtain a where P: "4 * a = c" by auto
  hence "2 * (2 * a) = c" by simp
  thus "∃b. 2 * b = c" by blast
qed
```

Structured proof

Example (using keywords **assume**, **obtain** and **hence**/**then** **have**/**from** $\cdots$ **have**):

lemma "$\forall c :: \texttt{int}.\ (\exists a.\ 4 * a = c) \longrightarrow (\exists b.\ 2 * b = c)$"
proof `(rule allI, rule impI)`
  fix $c :: \texttt{int}$
  assume "$\exists a.\ 4 * a = c$"
  then obtain a **where** P: "$4 * a = c$" **by** `auto`
  $\boxed{\textbf{hence}}$ "$2 * (2 * a) = c$" **by** `simp`
  thus "$\exists b.\ 2 * b = c$" **by** `blast`
qed

Structured proof

Example (using keywords **assume**, **obtain** and **hence**/**then have**/**from** $\cdots$ **have**):

lemma "$\forall c :: \texttt{int}.\ (\exists a.\ 4 * a = c) \longrightarrow (\exists b.\ 2 * b = c)$"
proof `(rule allI, rule impI)`
  fix $c :: \texttt{int}$
  assume "$\exists a.\ 4 * a = c$"
  then obtain a **where** P: "$4 * a = c$" **by** `auto`
  $\boxed{\textbf{then have}}$ "$2 * (2 * a) = c$" **by** `simp`
  thus "$\exists b.\ 2 * b = c$" **by** `blast`
qed

Structured proof

Example (using keywords **assume**, **obtain** and **hence**/**then** **have**/**from** $\cdots$ **have**):

lemma "$\forall c :: \texttt{int}.\ (\exists a.\ 4 * a = c) \longrightarrow (\exists b.\ 2 * b = c)$"
proof `(rule allI, rule impI)`
  fix $c :: \texttt{int}$
  assume "$\exists a.\ 4 * a = c$"
  then obtain a **where** P: "$4 * a = c$" **by** `auto`
  $\boxed{\textbf{from}\ \texttt{this}\ \textbf{have}}$ "$2 * (2 * a) = c$" **by** `simp`
  thus "$\exists b.\ 2 * b = c$" **by** `blast`
qed

Structured proof

Example (using keywords **assume**, **obtain** and **hence**/**then** **have**/**from** $\cdots$ **have**):

lemma "$\forall c :: \texttt{int}.\ (\exists a.\ 4 * a = c) \longrightarrow (\exists b.\ 2 * b = c)$"
proof `(rule allI, rule impI)`
  fix $c :: \texttt{int}$
  assume "$\exists a.\ 4 * a = c$"
  then obtain a **where** P: "$4 * a = c$" **by** `auto`
  $\boxed{\textbf{from}\ \texttt{P}\ \textbf{have}}$ "$2 * (2 * a) = c$" **by** `simp`
  thus "$\exists b.\ 2 * b = c$" **by** `blast`
qed

Structured proof

Yet another way to write the proof (with **from** `assms` or **using** `assms`):

lemma mylemma:
  fixes c :: `int`
  assumes “$\exists a.\ 4 * a = c$”
  shows “$\exists b.\ 2 * b = c$”
proof -
  from `assms` **obtain** `a` **where** “$4 * a = c$” **by** `auto`
  hence “$2 * (2 * a) = c$” **by** `simp`
  thus “$\exists b.\ 2 * b = c$” **by** `blast`
qed

Structured proof

Yet another way to write the proof (with **from** `assms` or **using** `assms`):

lemma mylemma:
  fixes c :: `int`
  assumes "$\exists a.\ 4 * a = c$"
  shows "$\exists b.\ 2 * b = c$"
proof -
  obtain a **where** "$4 * a = c$" **using** `assms` **by** `auto`
  hence "$2 * (2 * a) = c$" **by** `simp`
  thus "$\exists b.\ 2 * b = c$" **by** `blast`
qed

Structured proof

There is also calculation using '**...**', **moreover** and **ultimately**.

have "a = b" **by** <*some method*>
moreover have "... = c" **by** <*some method*>
moreover have "... = d" **by** <*some method*>
ultimately show "a = d" **by** auto

Moreover *collects* results and **ultimately** uses them (like using keyword **from**). This corresponds to:

$$\begin{aligned} a &= b \\ &= c \\ &= d. \end{aligned}$$

Recursion and Induction

Recursion and Induction

- Recursive *datatypes* are part of Isabelle. Naturals and Lists are represented this way.

Recursion and Induction

- Recursive *datatypes* are part of Isabelle. Naturals and Lists are represented this way.
- Functions can be defined recursively (e.g., using **primrec**).

Recursion and Induction

- Recursive *datatypes* are part of Isabelle. Naturals and Lists are represented this way.
- Functions can be defined recursively (e.g., using **primrec**).
- Induction can be used to prove things about recursive datatypes!

Recursion and Induction

- Recursive *datatypes* are part of Isabelle. Naturals and Lists are represented this way.
- Functions can be defined recursively (e.g., using **primrec**).
- Induction can be used to prove things about recursive datatypes!

$$\frac{P(0) \qquad \forall n.\ (P\ n \longrightarrow P\,(\mathtt{Suc}\ n))}{\forall n.\ P\ n}\ \text{induction (of } \mathbb{N})$$

Recursion and Induction

Defining a function recursively: **primrec** or **fun**.

Recursion and Induction

Defining a function recursively: **primrec** or **fun**.

primrec `listsum` :: "`nat list` ⇒ `nat`" **where**
  "`listsum` (h::t) = h + `listsum` t"
| "`listsum` [] = 0"

Recursion and Induction

Defining a function recursively: **primrec** or **fun**.

primrec `listsum` :: "`nat list` ⇒ `nat`" **where**
  "`listsum` (h::t) = h + `listsum` t"
| "`listsum` [] = 0"

primrec `mymult` :: "`nat` ⇒ `nat` ⇒ `nat`" **where**
  "`mymult` (Suc n) m = m + `mymult` n m"
| "`mymult` 0 m = 0"

Recursion and Induction

Defining a function recursively: **primrec** or **fun**.

primrec `listsum` :: "`nat list` ⇒ `nat`" **where**
"`listsum` (h::t) = h + `listsum` t"
| "`listsum` [] = 0"

primrec `mymult` :: "`nat` ⇒ `nat` ⇒ `nat`" **where**
"`mymult` (Suc n) m = m + `mymult` n m"
| "`mymult` 0 m = 0"

Isabelle adds simplification rules automatically for recursive definitions. For this example: `listsum.simps` and `mymult.simps`

Recursion and Induction

A proof by induction:

lemma "even ($x^2 + x$::nat)"
proof (induction x)
  case 0
   show "even ($0^2 + 0$ :: nat)" **by** simp
  case (Suc x)
   from Suc **obtain** y **where**
     P: "$2 * y = x^2 + x$" **by** (metis evenE)
   have "$((x :: \mathtt{nat}) + 1)^2 + x + 1 = (x^2 + 2 * x + 1) + x + 1$" **by** algebra
   moreover have "... $= x^2 + 2 * x + x + 2$" **by** simp
   moreover have "... $= x^2 + x + (2 * x + 2)$" **by** simp
   moreover have "... $= 2 * y + 2 * x + 2$" **using** P **by** simp
   moreover have "... $= 2 * (y + x + 1)$" **by** simp
   ultimately have "$((x :: \mathtt{nat}) + 1)^2 + x + 1 = 2 * (y + x + 1)$" **by** simp
   thus "even ($(\mathtt{Suc}\, x)^2 + \mathtt{Suc}\, x$)" **by** simp
qed

Part 3: Reasoning about Geometries [60%]

Part 3: Reasoning about Geometries [60%]

- Getting familiar with axiomatic systems.

Part 3: Reasoning about Geometries [60%]

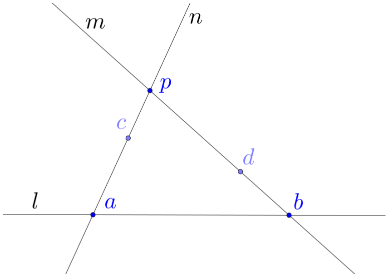
- Getting familiar with axiomatic systems.
- In particular, Isabelle's *locales*.

Part 3: Reasoning about Geometries [60%]

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

Part 3: Reasoning about Geometries [60%]

- Getting familiar with axiomatic systems.
- In particular, Isabelle's *locales*.
- Formalising geometry.
- Proving things about geometric constructions.



Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.

Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.

Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:

For every line l, if a point p lies outside of l,
then there is a line that passes through p, parallel to l.

Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:

  For every line l, if a point p lies outside of l, then there is a line that passes through p, parallel to l.

- Is it provable from the *other* axioms?

Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:

  For every line l, if a point p lies outside of l,
  then there is a line that passes through p, parallel to l.
- Is it provable from the *other* axioms?
- Is there a universe (model) where there are no parallel lines?*

Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:

  For every line l, if a point p lies outside of l,
  then there is a line that passes through p, parallel to l.
- Is it provable from the *other* axioms?
- Is there a universe (model) where there are no parallel lines?*
- Is there a universe where there are many parallel lines?**

Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:

  For every line l, if a point p lies outside of l,
  then there is a line that passes through p, parallel to l.

- Is it provable from the *other* axioms?
- Is there a universe (model) where there are no parallel lines?*
- Is there a universe where there are many parallel lines?**
- $*$: Projective Geometry

Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:

  For every line l, if a point p lies outside of l,
  then there is a line that passes through p, parallel to l.

- Is it provable from the *other* axioms?
- Is there a universe (model) where there are no parallel lines?*
- Is there a universe where there are many parallel lines?**
- $*$: Projective Geometry
- $**$: Hyperbolic Geometry

Part 3: Reasoning about Geometries [60%]

Part 3: Reasoning about Geometries [60%]

In this assignment we ask the question:

Part 3: Reasoning about Geometries [60%]

In this assignment we ask the question:

Are there finite models of geometry?

The Featherless biped

- Define human = featherless biped (Socrates)

The Featherless biped

- Define human = featherless biped (Socrates)
- Can you think of any other featherless biped?

The Featherless biped

- Define human = featherless biped (Socrates)
- Can you think of any other featherless biped?

Diogenes of Sinope's criticism:

The Featherless biped

- Define human = featherless biped (Socrates)
- Can you think of any other featherless biped?

Diogenes of Sinope's criticism:

Are finite geometries just shaved chickens?

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**.

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: D.Raggi@sms.ed.ac.uk.

More hints

- Start early.
- Go to the lab sessions.
- Contact me by email: D.Raggi@sms.ed.ac.uk.
- That's it.