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.
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:**

\[
\begin{align*}
\frac{t = s}{P t} & \quad \text{subst} \\
\frac{P s}{P t} & \quad \text{ssubst}
\end{align*}
\]

\[
\begin{align*}
\frac{t = t}{s = t} & \quad \text{sym} \\
\frac{s = t}{r = s} & \quad \text{trans}
\end{align*}
\]

\[
\begin{align*}
\forall x. \ f x = g x & \quad \text{ext}
\end{align*}
\]
Reasoning with equality (=)

Rules:

\[
\begin{align*}
\frac{t = s}{P_t} & \quad \text{subst} \\
\frac{s = t}{P_t} & \quad \text{ssubst}
\end{align*}
\]

\[
\begin{align*}
\frac{t = t}{P_t} & \quad \text{refl} \\
\frac{s = t}{t = s} & \quad \text{sym} \\
\frac{r = s}{s = t} & \quad \text{trans}
\end{align*}
\]

\[
\begin{align*}
\forall x. \ f \ x = g \ x & \quad \text{ext}
\end{align*}
\]

Are all of these rules necessary, or can some of them be derived from the others?
Reasoning with equality (=)

Output:
1. \( \forall c. \exists a b. \ a + 3 \times b = c \)

\[
\begin{align*}
\text{mult}_\text{zero}_\text{right}: & \quad a \times 0 = 0 \\
\text{add}_0: & \quad 0 + a = a \\
\text{add}_\text{commute}: & \quad a + b = b + a
\end{align*}
\]

\textbf{lemma} \quad "\( \forall c :: \text{int.} \ \exists a b. \ a + 3 \times b = c \)"
Reasoning with equality (=)

Output:
1. $\forall c. \exists a \ b. \ a + 3 \cdot b = c$

lemma “$\forall c :: \text{int}. \exists a \ b. \ a + 3 \cdot b = c$”
apply (rule allI)

mult_zero_right: \[ a \cdot 0 = 0 \]
add_0: \[ 0 + a = a \]
add.commute: \[ a + b = b + a \]
Reasoning with equality (\(=\))

Output:
1. \(\forall c. \exists b. \ c + 3 \times b = c\)

\[\begin{align*}
\text{lemma} & \quad \forall c :: \text{int. } \exists a \ b. \ a + 3 \times b = c \\
\text{apply} & \quad \text{(rule allI)} \\
\text{apply} & \quad \text{(rule_tac x = c in exI)} \\
\text{apply} & \quad \text{(rule_tac s = 0 and t = } "3 \times 0" \text{ in ssubst)} \\
\text{apply} & \quad \text{(rule mult_zero_right)} \\
\text{apply} & \quad \text{(rule_tac s = } "0 + c" \text{ and t = } "c + 0" \text{ in ssubst)} \\
\text{apply} & \quad \text{(rule add.commute)} \\
\text{apply} & \quad \text{(rule add_0)} \\
\text{done}
\end{align*}\]
Reasoning with equality (=)

Output:
1. \( \forall c. \ c + 3 \times 0 = c \)

\[
\begin{align*}
\text{lemma} \ & \ “\forall c :: \text{int}. \ \exists a \ b. \ a + 3 \times b = c” \\
\text{apply} \ & \ (\text{rule allI}) \\
\text{apply} \ & \ (\text{rule_tac} \ x = c \ \text{in exI}) \\
\text{apply} \ & \ (\text{rule_tac} \ x = 0 \ \text{in exI})
\end{align*}
\]

\[
\begin{align*}
\text{mult_zero_right:} \ & \ a \times 0 = 0 \\
\text{add_0:} \ & \ 0 + a = a \\
\text{add.commute:} \ & \ a + b = b + a
\end{align*}
\]
Reasoning with equality (=)

Output:

1. $\forall c. 3 \times 0 = 0$
2. $\forall c. c + 0 = c$

**Lemma** “$\forall c :: \text{int}. \exists a b. a + 3 \times b = c$”

- 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 \times 0" in ssubst)
- apply (rule add.commute)
- apply (rule add_0)
- done

**Rules**

- **mult_zero_right**: $a \times 0 = 0$
- **add_0**: $0 + a = a$
- **add.commute**: $a + b = b + a$
Reasoning with equality (=)

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

```
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 add.commute)
  apply (rule add_0)
  done
```

```
mult_zero_right: a * 0 = 0
add_0: 0 + a = a
add.commute: a + b = b + a
```
Reasoning with equality (=)

Output:
1. $\forall c. \ c + 0 = 0 + c$
2. $\forall c. \ 0 + c = c$

```
lemma "\forall c :: int. \exists a \ b. \ a + 3 \times 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 \times 0" in ssubst)
  apply (rule mult_zero_right)
  apply (rule_tac s = "0 + c" and t = "c + 0" in ssubst)
```

```
mult_zero_right: a * 0 = 0
add_0: \ 0 + a = a
add.commute: a + b = b + a
```
Reasoning with equality (=)

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

lemma “$\forall c :: \text{int}. \exists a b. a + 3 \times b = c$”
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 \times 0" in ssubst)
apply (rule mult_zero_right)
apply (rule_tac s = "0 + c" and t = "c + 0" in ssubst)
apply (rule add.commute)

mult_zero_right: \quad a \times 0 = 0
add_0: \quad 0 + a = a
add.commute: \quad a + b = b + a
Reasoning with equality (=)

Output:
No subgoals!

\[ \text{mult\_zero\_right: } a \times 0 = 0 \]
\[ \text{add\_0: } 0 + a = a \]
\[ \text{add\_commute: } a + b = b + a \]

**lemma** "\( \forall c :: \text{int. } \exists a \ b. \ a + 3 \times b = c \)"
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 \times 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!

**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 (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)
done

\[
\begin{align*}
\text{mult_zero_right:} & \quad a * 0 = 0 \\
\text{add_0:} & \quad 0 + a = a \\
\text{add.commute:} & \quad a + b = b + a
\end{align*}
\]
Reasoning with equality (\(=\))

**Output:**

No subgoals!

- **mult\_zero\_right:** \(a \times 0 = 0\)
- **add\_0:** \(0 + a = a\)
- **add.\_commute:** \(a + b = b + a\)

**lemma** "\(\forall c :: \text{int}. \exists a \ b. \ a + 3 \times b = c\)"

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!

\[
\begin{align*}
\text{mult\_zero\_right:} & \quad a \times 0 = 0 \\
\text{add\_0:} & \quad 0 + a = a \\
\text{add\_commute:} & \quad a + b = b + a
\end{align*}
\]

**lemma** “\(\forall c :: \text{int}. \exists a b. a + 3 \times b = c\)”

apply \((\text{rule allI})\)
apply \((\text{rule\_tac } x = c \text{ in exI})\)
apply \((\text{rule\_tac } x = 0 \text{ in exI})\)
apply \((\text{simp only: } \text{mult\_zero\_right add\_commute add\_0})\)
done

Method \text{simp} does that automatically (given the right rules!).
Reasoning with equality (\(=\))

Output:
No subgoals!

\[
\begin{align*}
\text{mult\_zero\_right:} & \quad a \times 0 = 0 \\
\text{add\_0:} & \quad 0 + a = a \\
\text{add\_commute:} & \quad a + b = b + a
\end{align*}
\]

**Lemma**  \(\forall c :: \text{int. } \exists a \ b. \ a + 3 \times b = c\)

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: eq₁ ... eqₙ)`
  
  `apply (simp only: eq₁ ... eqₙ)`
  
  `apply (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)`
Isabelle’s powerful tools

- **simp**: rewriting using equations.
  Uses: `apply simp`
  `apply (simp add: eq\_1 \ldots eq\_n)`
  `apply (simp only: eq\_1 \ldots eq\_n)`
  `apply (simp del: eq\_1 \ldots eq\_n)`

- **auto**: rewriting + proof search (using classical logic).
  Uses: `apply auto`
  `apply (auto simp add: eq\_1 \ldots eq\_n)`
  `apply (auto simp only: eq\_1 \ldots eq\_n)`
  `apply (auto simp del: eq\_1 \ldots eq\_n)`

- Others: **blast, fast, force, fastforce, safe, algebra, linarith, arith, presburger, meson, metis.**
Isabelle’s powerful tools

Sledgehammer

\[
\text{lemma} \quad \text{inj on } f \ A = \implies \exists g. g' f A \subseteq A \land (\forall a \in A. g(f a) = a) \land (\forall b \in A. f(g(f b)) = f b)
\]
Sledgehammer

▶ Tool for invoking external provers.

\[
\text{lemma}\ "\text{inj}\ on\ f\ A = \exists g. g' f A \subseteq A \land (\forall a \in A. g(f a) = a) \land (\forall b \in A. f(g(f b)) = f b)"
\]
Isabelle’s powerful tools

Sledgehammer

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

```
lemma "inj on f A = \exists g. g' f A \subseteq A \land (\forall a \in A. g (f a) = a) \land (\forall b \in A. f (g (f b)) = f b)"
```
Isabelle’s powerful tools

**Sledgehammer**

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

```
lemma "inj on f A = \exists g. g' f A \subseteq A \land (\forall a \in A. g(f a) = a) \land (\forall b \in A. f(g(f b)) = f b)"
```
Isabelle’s powerful tools

**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.
Isabelle’s powerful tools

**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** "\( \text{inj\_on } f A \implies \exists g. g'f' A \subseteq A \land (\forall a \in A. g(f(a)) = a) \land (\forall b \in A. f(g(f(b))) = f(b)) \)"

```
lemma
sledgehammer
```
Sledgehammer

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

\textbf{lemma} \textit{“inj\_on \( f \) \( A \) \( \implies \)}
\[
\exists g. \ g'f' \subseteq A \land (\forall a \in A. \ g(f(a)) = a) \land (\forall b \in A. \ f(g(f(b))) = f(b))
\]
\textbf{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** “∀ c :: int. ∃ 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 “∃ a b. a + 3 * b = c” by blast

**qed**
Structured proof

Example (using keywords assume, obtain and hence/then have/from ⋯ 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 ⋯ 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 · · · have):

**lemma** “∀ c :: int. (∃a. 4 * a = c) → (∃b. 2 * b = c)”

**proof** (rule allI, rule impI)

1. fix c :: int
2. assume “∃a. 4 * a = c”
3. then obtain a where P: “4 * a = c” by auto
4. then have “2 * (2 * a) = c” by simp
5. thus “∃b. 2 * b = c” by blast

Qed
Structured proof

Example (using keywords assume, obtain and hence/then have/from · · · have):

**Lemma** “∀c :: int. (∃a. 4 * a = c) −→ (∃b. 2 * b = c)”

**Proof** (rule allI, rule impI)

1. **fix c :: int**
2. **assume** “∃a. 4 * a = c”
3. **then obtain** a where P: “4 * a = c” **by** auto
4. **from this have** “2 * (2 * a) = c” **by** simp
5. **thus** “∃b. 2 * b = c” **by** blast

**Qed**
Structured proof

Example (using keywords assume, obtain and hence/then have/from · · · 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
  from P have “2 * (2 * a) = c” by simp
  thus “∃b. 2 * b = c” by blast
qed
Yet another way to write the proof (with \texttt{from} \texttt{assms} or \texttt{using} \texttt{assms}): 

\begin{verbatim}
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
\end{verbatim}
Yet another way to write the proof (with \texttt{from \textit{assms}} or \texttt{using \textit{assms}}):

\begin{verbatim}
lemma mylemma:
    fixes \( c :: \text{int} \)
    assumes "\( \exists a. 4 \ast a = c \)"
    shows "\( \exists b. 2 \ast b = c \)"
proof -
    obtain a where "\( 4 \ast a = c \)" using \textit{assms} by auto
    hence "\( 2 \ast (2 \ast a) = c \)" by simp
    thus "\( \exists b. 2 \ast b = c \)" by blast
qed
\end{verbatim}
Structured proof

There is also calculation using ‘...’, **moreover** and **ultimately**.

\[
\begin{align*}
\text{have} & \quad \text{“a = b” by <some method>} \\
\text{moreover have} & \quad \text{“... = c” by <some method>} \\
\text{moreover have} & \quad \text{“... = d” by <some method>} \\
\text{ultimately show} & \quad \text{“a = d” by auto}
\end{align*}
\]

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

\[
\begin{align*}
a &= b \\
&= c \\
&= d.
\end{align*}
\]
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!

\[ P(0) \]
\[ \forall n. (P(n) \rightarrow P(Suc(n))) \]

induction (of \( \mathbb{N} \))
Recursion and Induction

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

\[
P(0) \quad \forall n. (P \ n \rightarrow P \ (Suc \ n)) \quad \forall n. \ P \ n
\]

induction (of \(\mathbb{N}\))
Recursion and Induction

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

```isabelle
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

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

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

Isabelle adds simplification rules automatically for recursive definitions. For this example: `listsum.simps` and `mymult.simps`.
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 :: 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 :: nat) + 1)^2 + x + 1 = 2 * (y + x + 1)" by simp
    thus "even ((Suc x)^2 + 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 $l$. 

- *: Projective Geometry
- **: Hyperbolic Geometry
Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:
  
  \[
  \text{For every line } l, \text{ if a point } p \text{ lies outside of } l, \\
  \text{then there is a line that passes through } p, \text{ parallel to } l.
  \]

- Is it provable from the \textit{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?*

* Projective Geometry

** Hyperbolic 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?**
Part 3: Reasoning about Geometries [60%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:
  
  \[\text{For every line } l, \text{ if a point } p \text{ lies outside of } l,\]
  
  \[\text{then there is a line that passes through } p, \text{ parallel to } l.\]
  
- 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%]

- Choosing a representation has consequences.
- Surprising models are common.
- Think about axiom:
  
  \[
  \text{For every line } l, \text{ if a point } \ p \ \text{lies outside of } l, \text{ then there is a line that passes through } p, \text{ parallel to } l. 
  \]

- 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
In this assignment we ask the question:

Are there finite models of geometry?
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:

\textit{Are there finite models of geometry?}
The Featherless biped

- Define human = featherless biped (Socrates)

Diogenes of Sinope's criticism:
Are finite geometries just shaved chickens?
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 $P \land Q$ sledgehammer
- This gives the provers a hint.
- Preinstantiate variables when trying to use a result in a proof: using $P[\text{where } x = \text{“some term”}] Q$ 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

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

- Start early.

Contact me by email: D.Raggi@sms.ed.ac.uk.
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.