NICTA Advanced Course

Theorem Proving
Principles, Techniques, Applications
CONTENT

➜ Intro & motivation, getting started with Isabelle

➜ Foundations & Principles
  • Lambda Calculus
  • Higher Order Logic, natural deduction
  • Term rewriting

➜ Proof & Specification Techniques
  • Inductively defined sets, rule induction
  • Datatypes, recursion, induction
  • Calculational reasoning, mathematics style proofs
  • Hoare logic, proofs about programs
LAST TIME ON HOL

→ Defining HOL
LAST TIME ON HOL

- Defining HOL
- Higher Order Abstract Syntax
LAST TIME ON HOL

→ Defining HOL
→ Higher Order Abstract Syntax
→ Deriving proof rules
LAST TIME ON HOL

➔ Defining HOL
➔ Higher Order Abstract Syntax
➔ Deriving proof rules
➔ More automation
THE THREE BASIC WAYS OF INTRODUCING THEOREMS

➔ Axioms:

Example: \texttt{axioms refl: "t = t"}
THE THREE BASIC WAYS OF INTRODUCING THEOREMS

→ Axioms:

Example: \texttt{axioms refl: } "\(t = t\)"

Do not use. Evil. Can make your logic inconsistent.
The Three Basic Ways of Introducing Theorems

➔ Axioms:

Example: \texttt{axioms refl: }"t = t"

Do not use. Evil. Can make your logic inconsistent.

➔ Definitions:

Example: \texttt{defs inj_def: }"inj f \equiv \forall x \ y. f \ x = f \ y \rightarrow x = y"
The Three Basic Ways of Introducing Theorems

→ Axioms:

Example: \[ \text{axioms refl: } "t = t" \]

Do not use. Evil. Can make your logic inconsistent.

→ Definitions:

Example: \[ \text{defs inj_def: } "\text{inj } f \equiv \forall x y. f x = f y \rightarrow x = y" \]

→ Proofs:

Example: \[ \text{lemma } "\text{inj } (\lambda x. x + 1)" \]
The Three Basic Ways of Introducing Theorems

→ Axioms:

Example: **axioms** refl: "\( t = t \)"

Do not use. Evil. Can make your logic inconsistent.

→ Definitions:

Example: **defs** inj\_def: "\( \text{inj } f \equiv \forall x \ y. f \ x = f \ y \rightarrow x = y \)"

→ Proofs:

Example: **lemma** "\( \text{inj } (\lambda x. x + 1) \)"

The harder, but safe choice.
THE THREE BASIC WAYS OF INTRODUCING TYPES

→ **typedef**: by name only

Example: **typedef** names
THE THREE BASIC WAYS OF INTRODUCING TYPES

→ **typedef**: by name only

Example:    **typedef** names
Introduces new type *names* without any further assumptions
THE THREE BASIC WAYS OF INTRODUCING TYPES

→ **typedecl**: by name only

Example:       **typedecl** names
Introduces new type *names* without any further assumptions

→ **types**: by abbreviation

Example:       **types** $\alpha$ rel = "$\alpha \Rightarrow \alpha \Rightarrow bool$"
THE THREE BASIC WAYS OF INTRODUCING TYPES

→ **typedecl**: by name only

Example: **typedecl** names
Introduces new type *names* without any further assumptions

→ **types**: by abbreviation

Example: **types** $\alpha$ rel = ”$\alpha \Rightarrow \alpha \Rightarrow \text{bool}$”
Introduces abbreviation *rel* for existing type $\alpha \Rightarrow \alpha \Rightarrow \text{bool}$
Type abbreviations are immediately expanded internally
THE THREE BASIC WAYS OF INTRODUCING TYPES

→ **typedef**: by name only

Example: `typedef names`  
Introduces new type `names` without any further assumptions

→ **types**: by abbreviation

Example: `types α rel = "α ⇒ α ⇒ bool"`  
Introduces abbreviation `rel` for existing type `α ⇒ α ⇒ bool`  
**Type abbreviations are immediately expanded internally**

→ **typedef**: by definition as a set

Example: `typedef new_type = "{some set}" <proof>`
THE THREE BASIC WAYS OF INTRODUCING TYPES

→ **typedecl**: by name only

Example: **typedecl** names
Introduces new type *names* without any further assumptions

→ **types**: by abbreviation

Example: **types** \( \alpha \) rel = ”\( \alpha \Rightarrow \alpha \Rightarrow bool \)”
Introduces abbreviation *rel* for existing type \( \alpha \Rightarrow \alpha \Rightarrow bool \)
*Type abbreviations are immediately expanded internally*

→ **typedef**: by definition as a set

Example: **typedef** new_type = ”\{some set\}” <proof>
Introduces a new type as a subset of an existing type.
The proof shows that the set on the rhs in non-empty.
HOW TYPEDEF WORKS

new type
How typedef Works

new type

existing type
HOW TYPEDEF WORKS

new type

existing type
HOW TYPEDEF WORKS

new type

existing type

Rep

Abs
HOW TYPEDEF WORKS

new type

existing type

Rep

Abs
EXAMPLE: PAIRS

\[(\alpha, \beta) \text{ Prod}\]

1. Pick existing type:
EXAMPLE: Pairs

\((\alpha, \beta)\) Prod

1. Pick existing type: \(\alpha \Rightarrow \beta \Rightarrow \text{bool}\)
2. Identify subset:
**Example: Pairs**

\((\alpha, \beta)\) Prod

1. Pick existing type: \(\alpha \Rightarrow \beta \Rightarrow \text{bool}\)

2. Identify subset:
   
   \[(\alpha, \beta) \text{ Prod} = \{ f. \exists a b. f = \lambda(x :: \alpha) (y :: \beta). x = a \land y = b\}\]

3. We get from Isabelle:
**Example: Pairs**

\[(\alpha, \beta) \text{ Prod}\]

1. Pick existing type: \(\alpha \Rightarrow \beta \Rightarrow \text{bool}\)

2. Identify subset:
   \[ (\alpha, \beta) \text{ Prod} = \{f. \exists a \ b. f = \lambda(x :: \alpha) (y :: \beta). x = a \land y = b\} \]

3. We get from Isabelle:
   - functions Abs_Prod, Rep_Prod
   - both injective
   - \(\text{Abs}_{\text{Prod}}(\text{Rep}_{\text{Prod}} x) = x\)

4. We now can:
Example: Pairs

\((\alpha, \beta) \text{ Prod}\)

1. Pick existing type: \(\alpha \Rightarrow \beta \Rightarrow \text{bool}\)

2. Identify subset:
\[
(\alpha, \beta) \text{ Prod} = \{f. \exists a \; b. \; f = \lambda (x :: \alpha) \; (y :: \beta). \; x = a \land y = b\}
\]

3. We get from Isabelle:
   - functions Abs_Prod, Rep_Prod
   - both injective
   - \(\text{Abs}_{\text{Prod}} (\text{Rep}_{\text{Prod}} x) = x\)

4. We now can:
   - define constants Pair, fst, snd in terms of Abs_Prod and Rep_Prod
   - derive all characteristic theorems
   - forget about Rep/Abs, use characteristic theorems instead
DEMO: INTRODUCTING NEW TYPES
TERM REWRITING
THE PROBLEM

Given a set of equations

\[ l_1 = r_1 \]
\[ l_2 = r_2 \]
\[ \vdots \]
\[ l_n = r_n \]
THE PROBLEM

Given a set of equations

\[ l_1 = r_1 \]
\[ l_2 = r_2 \]
\[ \vdots \]
\[ l_n = r_n \]

does equation \( l = r \) hold?
THE PROBLEM

Given a set of equations

\[ l_1 = r_1 \]
\[ l_2 = r_2 \]
\[ \vdots \]
\[ l_n = r_n \]

does equation \( l = r \) hold?

Applications in:

- **Mathematics** (algebra, group theory, etc)
- **Functional Programming** (model of execution)
- **Theorem Proving** (dealing with equations, simplifying statements)
TERM REWRITING: THE IDEA

use equations as reduction rules

\[ l_1 \rightarrow r_1 \]
\[ l_2 \rightarrow r_2 \]
\[ \vdots \]
\[ l_n \rightarrow r_n \]

**decide** \( l = r \) **by deciding** \( l \stackrel{*}{\rightarrow} r \)
\[ \rightarrow^0 = \{(x, y) | x = y\} \quad \text{identity} \]
0 \to = \{(x, y) | x = y\} \quad \text{identity}

n+1 \to = \underbrace{n \to \circ \to}_{n+1 \text{ fold composition}}
\[ \begin{align*}
0 & \quad = \quad \{(x, y) \mid x = y\} \quad \text{identity} \\
1 & \quad = \quad \underbrace{n \circ \cdots \circ n}_{n+1} \quad \text{n+1 fold composition} \\
+ & \quad = \quad \bigcup_{i > 0} i \quad \text{transitive closure}
\end{align*} \]
\[ 0 \rightarrow = \{(x, y) | x = y\} \quad \text{identity} \]
\[ n+1 \rightarrow = \overset{n}{\rightarrow} \circ \overset{1}{\rightarrow} \quad \text{n+1 fold composition} \]
\[ + \rightarrow = \bigcup_{i > 0} \overset{i}{\rightarrow} \quad \text{transitive closure} \]
\[ * \rightarrow = \overset{1}{\rightarrow} \bigcup \overset{0}{\rightarrow} \quad \text{reflexive transitive closure} \]
\begin{align*}
\rightarrow^0 &= \{(x, y) \mid x = y\} \quad \text{identity} \\
\rightarrow^{n+1} &= \rightarrow^n \circ \rightarrow \\ &= n+1 \text{ fold composition} \\
\rightarrow^+ &= \bigcup_{i > 0} \rightarrow^i \\ &= \text{transitive closure} \\
\rightarrow^* &= \rightarrow^+ \cup \rightarrow^0 \\ &= \text{reflexive transitive closure} \\
\rightarrow^\rightarrow &= \rightarrow^+ \cup \rightarrow^0 \\ &= \text{reflexive closure}
\end{align*}
\[
\begin{align*}
0 & \rightarrow \{ (x, y) \mid x = y \} \quad \text{identity} \\
n+1 & \rightarrow = \underbrace{n \circ \cdots \circ}_{n \text{ fold}} \quad \text{n+1 fold composition} \\
+ & \rightarrow = \bigcup_{i > 0} \underbrace{i}_{i \text{ times}} \quad \text{transitive closure} \\
* & \rightarrow = + \cup 0 \quad \text{reflexive transitive closure} \\
\rightarrow & = \rightarrow \cup 0 \quad \text{reflexive closure} \\
\rightarrow^{-1} & = \{ (y, x) \mid x \rightarrow y \} \quad \text{inverse}
\end{align*}
\]
\[
\begin{align*}
\overrightarrow{0} &= \{(x, y) | x = y\} & \text{identity} \\
\overrightarrow{n+1} &= \overrightarrow{n} \circ \overrightarrow{n} & \text{n+1 fold composition} \\
\overrightarrow{+} &= \bigcup_{i > 0} \overrightarrow{i} & \text{transitive closure} \\
\overrightarrow{*} &= \overrightarrow{+} \cup \overrightarrow{0} & \text{reflexive transitive closure} \\
\overrightarrow{=} &= \bigcup \overrightarrow{0} & \text{reflexive closure} \\
\overrightarrow{-1} &= \{(y, x) | x \rightarrow y\} & \text{inverse} \\
\overleftarrow{} &= \overrightarrow{-1} & \text{inverse}
\end{align*}
\]
\[
\begin{align*}
\rightarrow^0 & = \{(x, y)| x = y\} & \text{identity} \\
\rightarrow^{n+1} & = \rightarrow^n \circ \rightarrow & \text{n+1 fold composition} \\
\rightarrow^+ & = \bigcup_{i > 0} \rightarrow^i & \text{transitive closure} \\
\rightarrow^* & = \rightarrow^+ \cup \rightarrow^0 & \text{reflexive transitive closure} \\
\rightarrow & = \rightarrow^+ \cup \rightarrow^0 & \text{reflexive closure} \\
\rightarrow^{-1} & = \{(y, x)| x \rightarrow y\} & \text{inverse} \\
\leftarrow & = \rightarrow^{-1} & \text{inverse} \\
\leftrightarrow & = \leftarrow \cup \rightarrow & \text{symmetric closure}
\end{align*}
\]
0 \rightarrow = \{(x, y) | x = y\} \quad \text{identity}

n + 1 \rightarrow = n \circ \rightarrow \quad \text{n+1 fold composition}

+ \rightarrow = \bigcup_{i > 0} i \rightarrow \quad \text{transitive closure}

* \rightarrow = + \cup 0 \rightarrow \quad \text{reflexive transitive closure}

\rightarrow = \rightarrow \cup 0 \rightarrow \quad \text{reflexive closure}

-1 \rightarrow = \{(y, x) | x \rightarrow y\} \quad \text{inverse}

\leftarrow = -1 \rightarrow \quad \text{inverse}

\leftrightarrow = \leftarrow \cup \rightarrow \quad \text{symmetric closure}

\leftrightarrow = \bigcup_{i > 0} i \leftrightarrow \quad \text{transitive symmetric closure}

\leftrightarrow = \leftrightarrow \cup 0 \leftrightarrow \quad \text{reflexive transitive symmetric closure}
HOW TO DECIDE $l \leftrightarrow^* r$

Same idea as for $\beta$: 

---

Fact: $\beta$ is Church-Rosser if it is confluent.
**How to Decide** $l \leftrightarrow r$

Same idea as for $\beta$: look for $n$ such that $l \rightarrow^* n$ and $r \rightarrow^* n$

Does this always work?
How to Decide $l \leftrightarrow^* r$

Same idea as for $\beta$: look for $n$ such that $l \rightarrow^* n$ and $r \rightarrow^* n$

Does this always work?

If $l \rightarrow^* n$ and $r \rightarrow^* n$ then $l \leftrightarrow^* r$. Ok.
How to Decide $l \leftrightarrow^* r$

Same idea as for $\beta$: look for $n$ such that $l \rightarrow^* n$ and $r \rightarrow^* n$

Does this always work?
- If $l \rightarrow^* n$ and $r \rightarrow^* n$ then $l \leftrightarrow^* r$. Ok.
- If $l \leftrightarrow^* r$, will there always be a suitable $n$?
How to Decide \( l \leftrightarrow^* r \)

Same idea as for \( \beta \): look for \( n \) such that \( l \rightarrow^* n \) and \( r \rightarrow^* n \)

Does this always work?

If \( l \rightarrow^* n \) and \( r \rightarrow^* n \) then \( l \leftrightarrow^* r \). Ok.

If \( l \leftrightarrow^* r \), will there always be a suitable \( n \)? No!

Example:

Rules: \( f\ x \rightarrow a, \ g\ x \rightarrow b, \ f\ (g\ x) \rightarrow b \)
How to Decide $l \leftrightarrow^* r$

Same idea as for $\beta$: look for $n$ such that $l \rightarrow^* n$ and $r \rightarrow^* n$

Does this always work?

If $l \rightarrow^* n$ and $r \rightarrow^* n$ then $l \leftrightarrow^* r$. Ok.

If $l \leftrightarrow^* r$, will there always be a suitable $n$? No!

Example:

Rules:

\[
\begin{align*}
  f \; x & \rightarrow a, \\
  g \; x & \rightarrow b, \\
  f \; (g \; x) & \rightarrow b \\
  f \; x & \leftrightarrow^* g \; x & \text{because} & \quad f \; x & \rightarrow a & \leftarrow f \; (g \; x) & \rightarrow b & \leftarrow g \; x
\end{align*}
\]
HOW TO DECIDE \( l \leftrightarrow^* r \)

Same idea as for \( \beta \): look for \( n \) such that \( l \rightarrow^* n \) and \( r \rightarrow^* n \)

Does this always work?

If \( l \rightarrow^* n \) and \( r \rightarrow^* n \) then \( l \leftrightarrow^* r \). Ok.

If \( l \leftrightarrow^* r \), will there always be a suitable \( n \)? **No!**

Example:

Rules: \( f \, x \rightarrow a, \quad g \, x \rightarrow b, \quad f \,(g \, x) \rightarrow b \)

\( f \, x \leftrightarrow^* g \, x \) because \( f \, x \rightarrow a \leftarrow f \,(g \, x) \rightarrow b \leftarrow g \, x \)

**But:** \( f \, x \rightarrow a \) and \( g \, x \rightarrow b \) and \( a, b \) in normal form
How to Decide $l \leftrightarrow^* r$

Same idea as for $\beta$: look for $n$ such that $l \rightarrow^* n$ and $r \rightarrow^* n$

Does this always work?

If $l \rightarrow^* n$ and $r \rightarrow^* n$ then $l \leftrightarrow^* r$. Ok.

If $l \leftrightarrow^* r$, will there always be a suitable $n$? No!

Example:

Rules: $f x \rightarrow a$, $g x \rightarrow b$, $f (g x) \rightarrow b$

$f x \leftrightarrow^* g x$ because $f x \rightarrow a \leftarrow f (g x) \rightarrow b \leftarrow g x$

But: $f x \rightarrow a$ and $g x \rightarrow b$ and $a, b$ in normal form

Works only for systems with Church-Rosser property:

$l \leftrightarrow^* r \Rightarrow \exists n. l \rightarrow^* n \land r \rightarrow^* n$
**HOW TO DECIDE** $l \leftrightarrow^* r$

Same idea as for $\beta$: look for $n$ such that $l \rightarrow^* n$ and $r \rightarrow^* n$

Does this always work?

- If $l \rightarrow^* n$ and $r \rightarrow^* n$ then $l \leftrightarrow^* r$. Ok.
- If $l \leftrightarrow^* r$, will there always be a suitable $n$? **No!**

Example:

Rules: 

$$
\begin{align*}
  f \, x & \rightarrow a, & g \, x & \rightarrow b, & f \, (g \, x) & \rightarrow b \\
  f \, x & \leftrightarrow^* g \, x & \text{because} & f \, x & \rightarrow a & \leftarrow f \, (g \, x) & \rightarrow b & \leftarrow g \, x
\end{align*}
$$

**But:** $f \, x \rightarrow a$ and $g \, x \rightarrow b$ and $a, b$ in normal form

Works only for systems with **Church-Rosser** property:

$$
  l \leftrightarrow^* r \iff \exists n. \, l \rightarrow n \land r \rightarrow n
$$

**Fact:** $\rightarrow$ is Church-Rosser iff it is confluent.
Problem:
is a given set of reduction rules confluent?
Problem:

is a given set of reduction rules confluent?

undecidable
**Problem:**
is a given set of reduction rules confluent?

**Local Confluence**

```
x *-------------------* s
|                   /|
|                  / |
|                 /  |
y *------------* t *```
**Problem:** is a given set of reduction rules confluent?  

*undecidable*

**Local Confluence**

**Fact:** local confluence and termination $\iff$ confluence
TERMINATION

→ is **terminating** if there are no infinite reduction chains
→ is **normalizing** if each element has a normal form
→ is **convergent** if it is terminating and confluent

Example:
TERMINATION

$\rightarrow$ is **terminating** if there are no infinite reduction chains

$\rightarrow$ is **normalizing** if each element has a normal form

$\rightarrow$ is **convergent** if it is terminating and confluent

**Example:**

$\rightarrow_\beta$ in $\lambda$ is not terminating, but confluent
TERMINATION

\[ \rightarrow \text{ is terminating} \] if there are no infinite reduction chains

\[ \rightarrow \text{ is normalizing} \] if each element has a normal form

\[ \rightarrow \text{ is convergent} \] if it is terminating and confluent

Example:

\[ \rightarrow_\beta \text{ in } \lambda \text{ is not terminating, but confluent} \]

\[ \rightarrow_\beta \text{ in } \lambda \rightarrow \text{ is terminating and confluent, i.e. convergent} \]
is terminating if there are no infinite reduction chains

is normalizing if each element has a normal form

is convergent if it is terminating and confluent

Example:

$\longrightarrow_\beta$ in $\lambda$ is not terminating, but confluent

$\longrightarrow_\beta$ in $\lambda\rightarrow$ is terminating and confluent, i.e. convergent

Problem: is a given set of reduction rules terminating?
TERMINATION

\[ \rightarrow \text{ is terminating} \text{ if there are no infinite reduction chains} \]

\[ \rightarrow \text{ is normalizing} \text{ if each element has a normal form} \]

\[ \rightarrow \text{ is convergent} \text{ if it is terminating and confluent} \]

Example:

\[ \rightarrow_\beta \text{ in } \lambda \text{ is not terminating, but confluent} \]

\[ \rightarrow_\beta \text{ in } \lambda \text{ is terminating and confluent, i.e. convergent} \]

**Problem:** is a given set of reduction rules terminating?

undecidable
Basic Idea:
When is \( \rightarrow \) terminating?

**Basic Idea:** when the \( r_i \) are in some way simpler than the \( l_i \).
WHEN IS \( \rightarrow \) TERMINATING?

**Basic Idea:** when the \( r_i \) are in some way simpler than the \( l_i \)

**More formally:** \( \rightarrow \) is terminating when there is a well founded order \(<\) in which \( r_i < l_i \) for all rules.

(Well founded = no infinite decreasing chains \( a_1 > a_2 > \ldots \))

**Example:**

---

WHEN IS \( \rightarrow \) TERMINATING?  17-B
When is → Terminating?

**Basic Idea:** when the $r_i$ are in some way simpler than the $l_i$

**More formally:** → is terminating when
there is a well founded order $<$ in which $r_i < l_i$ for all rules.
(Well founded = no infinite decreasing chains $a_1 > a_2 > \ldots$)

**Example:** $f \ (g \ x) \rightarrow g \ x$, $g \ (f \ x) \rightarrow f \ x$

This system always terminates. Reduction order:
**Basic Idea:** when the $r_i$ are in some way simpler than the $l_i$

**More formally:** $\rightarrow$ is terminating when there is a well founded order $<$ in which $r_i < l_i$ for all rules.

(well founded = no infinite decreasing chains $a_1 > a_2 > \ldots$)

**Example:** $f\,(g\,x) \rightarrow g\,x$, $g\,(f\,x) \rightarrow f\,x$

This system always terminates. Reduction order:

$s <_r t$ iff $\text{size}(s) < \text{size}(t)$ with

$\text{size}(s) =$ numer of function symbols in $s$
**Basic Idea:** when the $r_i$ are in some way simpler then the $l_i$

**More formally:** is terminating when there is a well founded order $<$ in which $r_i < l_i$ for all rules. (well founded = no infinite decreasing chains $a_1 > a_2 > \ldots$)

**Example:** $f \; (g \; x) \rightarrow g \; x$, $g \; (f \; x) \rightarrow f \; x$

This system always terminates. Reduction order:

$s <_r t$ iff $\text{size}(s) < \text{size}(t)$ with

$\text{size}(s) =$ numer of function symbols in $s$

1. $g \; x <_r f \; (g \; x)$ and $f \; x <_r g \; (f \; x)$
**When is —— Terminating?**

**Basic Idea:** when the \( r_i \) are in some way simpler than the \( l_i \)

**More formally:** is terminating when there is a well founded order \( < \) in which \( r_i < l_i \) for all rules. (well founded = no infinite decreasing chains \( a_1 > a_2 > \ldots \))

**Example:** \( f \left( g \ x \right) \longrightarrow g \ x, \ g \left( f \ x \right) \longrightarrow f \ x \)

This system always terminates. Reduction order:

\[ s <_r t \iff \text{size}(s) < \text{size}(t) \text{ with} \]
\[ \text{size}(s) = \text{numer of function symbols in } s \]

1. \( g \ x <_r f \left( g \ x \right) \) and \( f \ x <_r g \left( f \ x \right) \)
2. \( <_r \) is well founded, because \( < \) is well founded on \( \mathbb{N} \)
Term rewriting engine in Isabelle is called **Simplifier**
Term rewriting engine in Isabelle is called **Simplifier**

```
apply simp
```

→ uses simplification rules
Term rewriting engine in Isabelle is called Simplifier

apply simp

→ uses simplification rules

→ (almost) blindly from left to right
Term rewriting engine in Isabelle is called **Simplifier**

```
apply simp
```

- uses simplification rules
- (almost) blindly from left to right
- until no rule is applicable.
Term rewriting engine in Isabelle is called **Simplifier**

\[ \text{apply simp} \]

- uses simplification rules
- (almost) blindly from left to right
- until no rule is applicable.

**termination:** not guaranteed
(may loop)
Term rewriting engine in Isabelle is called **Simplifier**

- `apply simp` 
- uses simplification rules 
- (almost) blindly from left to right 
- until no rule is applicable. 

**termination:** not guaranteed  
(may loop) 

**confluence:** not guaranteed  
(result may depend on which rule is used first)
Equations turned into simplification rules with \texttt{[simp]} attribute
Equations turned into simplification rules with \texttt{[simp]} attribute

Adding/deleting equations locally:
\begin{itemize}
  \item \texttt{apply (simp add: \{rules\})}
  \item \texttt{apply (simp del: \{rules\})}
\end{itemize}
Equations turned into simplification rules with [simp] attribute

Adding/deleting equations locally:
apply (simp add: <rules>) and apply (simp del: <rules>)

Using only the specified set of equations:
apply (simp only: <rules>)
DEMO
ISAR

A LANGUAGE FOR STRUCTURED PROOFS
apply scripts

→ unreadable
apply scripts

➔ unreadable
➔ hard to maintain
apply scripts

→ unreadable
→ hard to maintain
→ do not scale
apply scripts

→ unreadable
→ hard to maintain
→ do not scale

No structure.
apply scripts

⇒ unreadable
⇒ hard to maintain
⇒ do not scale

What about...

⇒ Elegance?

No structure.
<table>
<thead>
<tr>
<th>apply scripts</th>
<th>What about..</th>
</tr>
</thead>
<tbody>
<tr>
<td>unreadable</td>
<td>Elegance?</td>
</tr>
<tr>
<td>hard to maintain</td>
<td>Explaining deeper insights?</td>
</tr>
<tr>
<td>do not scale</td>
<td></td>
</tr>
</tbody>
</table>

No structure.
<table>
<thead>
<tr>
<th>apply scripts</th>
<th>What about..</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ unreadable</td>
<td>➔ Elegance?</td>
</tr>
<tr>
<td>➔ hard to maintain</td>
<td>➔ Explaining deeper insights?</td>
</tr>
<tr>
<td>➔ do not scale</td>
<td>➔ Large developments?</td>
</tr>
</tbody>
</table>

**No structure.**
**apply scripts**

- unreadable
- hard to maintain
- do not scale

**What about..**

- Elegance?
- Explaining deeper insights?
- Large developments?

---

**No structure.**

**Isar!**
A typical Isar proof

proof

assume \( \text{formula}_0 \)

have \( \text{formula}_1 \) by simp

\vdots

have \( \text{formula}_n \) by blast

show \( \text{formula}_{n+1} \) by \ldots

qed
proof

assume $formula_0$

have $formula_1$ by simp

:::

have $formula_n$ by blast

show $formula_{n+1}$ by \ldots

qed

proves $formula_0 \implies formula_{n+1}$
proof

assume \( \text{formula}_0 \)

have \( \text{formula}_1 \) by simp

:::

have \( \text{formula}_n \) by blast

show \( \text{formula}_{n+1} \) by \( \ldots \)

qed

proves \( \text{formula}_0 \implies \text{formula}_{n+1} \)

(analogous to assumes/shows in lemma statements)
proof = \textbf{proof} [method] \texttt{statement}* \texttt{qed} \\
\textbf{by} method
proof = proof [method] statement* qed
    | by method

method = (simp ...) | (blast ...) | (rule ...) | ...
**Isar Core Syntax**

proof = \textbf{proof} [method] statement* \textbf{qed}

| \textbf{by} method |

method = (simp . . .) | (blast . . .) | (rule . . .) | . . .

statement = \textbf{fix} variables (\&)

| \textbf{assume} proposition (\implies)

| [\textbf{from name}^+] (\textbf{have} | \textbf{show}) proposition proof

| \textbf{next} (separates subgoals)
**ISAR CORE SYNTAX**

proof = **proof** [method] statement* **qed**

   | **by** method

method = (simp . . . ) | (blast . . . ) | (rule . . . ) | . . .

statement = **fix** variables (Λ)

   | assume proposition (⇒⇒)

   | [from name*] (**have** | **show**) proposition proof

   | **next** (separates subgoals)

proposition = [name:] formula
proof [method] statement* qed

lemma ”[A; B] \implies A \land B”
**Proof and QED**

proof [method] statement* qed

lemma "[A; B] \implies A \land B"

proof (rule conjI)
proof [method] statement* qed

lemma 

proof (rule conjI)

assume A: ”A”

from A show ”A” by assumption
proof [method] statement* qed

lemma 

proof (rule conjl)
  
  assume A: "A"
  
  from A show "A" by assumption

next
proof [method] statement* qed

lemma "[A; B] \implies A \land B"

proof (rule conjl)
  assume A: "A"
  from A show "A" by assumption

next
  assume B: "B"
  from B show "B" by assumption
proof [method] statement* qed

lemma "[A; B] \implies A \land B"

proof (rule conjI)
    assume A: "A"
    from A show "A" by assumption

next
    assume B: "B"
    from B show "B" by assumption

qed
proof [method] statement* qed

lemma 

proof (rule conjI)
  assume A: "A"
  from A show "A" by assumption

next
  assume B: "B"
  from B show "B" by assumption

qed

→ proof (<method>) applies method to the stated goal
**PROOF AND QED**

proof [method] statement* qed

lemma "[A; B] \implies A \land B"

proof (rule conjI)
  assume A: "A"
  from A show "A" by assumption

next
  assume B: "B"
  from B show "B" by assumption

qed

→ proof (<method>) applies method to the stated goal
→ proof applies a single rule that fits
proof [method] statement* qed

lemma "[A; B] → A ∧ B"

proof (rule conjI)
  assume A: "A"
  from A show "A" by assumption

next
  assume B: "B"
  from B show "B" by assumption

qed

→ proof (<method>) applies method to the stated goal
→ proof applies a single rule that fits
→ proof - does nothing to the goal
Look at the proof state!

**Lemma**: $[A; B] \implies A \land B$

**Proof** (rule conjI)
How do I know what to Assume and Show?

Look at the proof state!

**lemma** "\([A; B] \implies A \land B\)"

**proof** (rule conjI)

→ **proof** (rule conjI) changes proof state to

1. \([A; B] \implies A\)
2. \([A; B] \implies B\)
**How do I know what to Assume and Show?**

Look at the proof state!

**Lemma** \([A; B] \Rightarrow A \land B\)

**Proof** (rule conjI)

→ **Proof** (rule conjI) changes proof state to

1. \([A; B] \Rightarrow A\)
2. \([A; B] \Rightarrow B\)

→ so we need 2 shows: **show** "A" and **show** "B"
HOW DO I KNOW WHAT TO ASSUME AND SHOW?

Look at the proof state!

**lemma** "\([A; B] \implies A \land B\)"

**proof** (rule conjI)

→ **proof** (rule conjI) changes proof state to
  1. \([A; B] \implies A\)
  2. \([A; B] \implies B\)

→ so we need 2 shows: **show** "A" and **show** "B"

→ We are allowed to **assume** \(A\),
  because \(A\) is in the assumptions of the proof state.
The Three Modes of Isar

⇒ [prove]:
  goal has been stated, proof needs to follow.
THE THREE MODES OF ISAR

→ **[prove]:**
  goal has been stated, proof needs to follow.

→ **[state]:**
  proof block has opened or subgoal has been proved,
  new *from* statement, goal statement or assumptions can follow.
THE THREE MODES OF ISAR

➔ [prove]:
  goal has been stated, proof needs to follow.

➔ [state]:
  proof block has opened or subgoal has been proved,
  new from statement, goal statement or assumptions can follow.

➔ [chain]:
  from statement has been made, goal statement needs to follow.
THE THREE MODES OF ISAR

→ [prove]:
goal has been stated, proof needs to follow.

→ [state]:
proof block has openend or subgoal has been proved,
new from statement, goal statement or assumptions can follow.

→ [chain]:
from statement has been made, goal statement needs to follow.

lemma "\([A; B] \implies A \land B\)"
THE THREE MODES OF ISAR

➔ [prove]:
goal has been stated, proof needs to follow.

➔ [state]:
proof block has openend or subgoal has been proved,
new from statement, goal statement or assumptions can follow.

➔ [chain]:
from statement has been made, goal statement needs to follow.

**Lemma** "\([A; B] \implies A \land B\)" [prove]
THE THREE MODES OF ISAR

→ **[prove]:**
goal has been stated, proof needs to follow.

→ **[state]:**
proof block has opened or subgoal has been proved,
new *from* statement, goal statement or assumptions can follow.

→ **[chain]:**
*from* statement has been made, goal statement needs to follow.

**lemma** "\[ A; B \] \implies A \land B" **[prove]**
**proof** (rule conjI) **[state]**
The Three Modes of Isar

→ [prove]:
goal has been stated, proof needs to follow.

→ [state]:
proof block has opened or subgoal has been proved,
new from statement, goal statement or assumptions can follow.

→ [chain]:
from statement has been made, goal statement needs to follow.

lemma \[[A; B] \implies A \land B\] [prove]
proof (rule conjI) [state]
    assume A: ”A” [state]
THE THREE MODES OF ISAR

→ [prove]:
goal has been stated, proof needs to follow.

→ [state]:
proof block has opened or subgoal has been proved,
new from statement, goal statement or assumptions can follow.

→ [chain]:
from statement has been made, goal statement needs to follow.

lemma \([A; B] \implies A \land B\) [prove]
proof (rule conjl) [state]
assume A: ”A” [state]
from A [chain]
THE THREE MODES OF ISAR

→ [prove]:
  goal has been stated, proof needs to follow.

→ [state]:
  proof block has opened or subgoal has been proved,
  new from statement, goal statement or assumptions can follow.

→ [chain]:
  from statement has been made, goal statement needs to follow.

lemma "\([A; B] \implies A \land B\)" [prove]
proof (rule conjI) [state]
  assume A: "A" [state]
next [state] …
H ave

Can be used to make intermediate steps.

Example:
Can be used to make intermediate steps.

Example:

lemma "(x :: nat) + 1 = 1 + x"

proof -
  have A: "x + 1 = Suc x" by simp
  have B: "1 + x = Suc x" by simp
  show "x + 1 = 1 + x" by (simp only: A B)

qed
Demo: Isar Proofs
WE HAVE LEARNED TODAY ...

- Introducing new Types
WE HAVE LEARNED TODAY ...

- Introducing new Types
- Equations and Term Rewriting
WE HAVE LEARNED TODAY ...

- Introducing new Types
- Equations and Term Rewriting
- Confluence and Termination of reduction systems
WE HAVE LEARNED TODAY ...

- Introducing new Types
- Equations and Term Rewriting
- Confluence and Termination of reduction systems
- Term Rewriting in Isabelle
We have learned today...

- Introducing new Types
- Equations and Term Rewriting
- Confluence and Termination of reduction systems
- Term Rewriting in Isabelle
- First structured proofs (Isar)
Exercises

→ use **typedef** to define a new type \( v \) with exactly one element.

→ define a constant \( u \) of type \( v \)

→ show that every element of \( v \) is equal to \( u \)

→ design a set of rules that turns formulae with \( \land, \lor, \rightarrow, \neg \) into disjunctive normal form
  (= disjunction of conjunctions with negation only directly on variables)

→ prove those rules in Isabelle

→ use **simp only** with these rules on \( (\neg B \rightarrow C) \rightarrow A \rightarrow B \)