NICTA Advanced Course

Theorem Proving

Principles, Techniques, Applications

$$\longrightarrow$$

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**:

Expample: **axioms** refl: "$t = t$"

THE THREE BASIC WAYS OF INTRODUCING THEOREMS

➜ **Axioms**:

Expample: **axioms** refl: "$t = t$"

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

The Three Basic Ways of Introducing Theorems

➜ **Axioms**:

Expample: **axioms** refl: "$t = t$"

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

➜ **Definitions:**

Example: **defs** inj_def: "inj $f \equiv \forall x\ y.\ f\ x = f\ y \longrightarrow x = y$"

THE THREE BASIC WAYS OF INTRODUCING THEOREMS

➜ **Axioms**:

Expample: **axioms** refl: "$t = t$"

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

➜ **Definitions:**

Example: **defs** inj_def: "inj $f \equiv \forall x\ y.\ f\ x = f\ y \longrightarrow x = y$"

➜ **Proofs:**

Example: **lemma** "inj $(\lambda x.\ x + 1)$"

The Three Basic Ways of Introducing Theorems

- **Axioms**:

  Expample: **axioms** refl: ”$t = t$”

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

- **Definitions:**

  Example: **defs** inj_def: ”inj $f \equiv \forall x\ y.\ f\ x = f\ y \longrightarrow x = y$”

- **Proofs:**

  Example: **lemma** ”inj $(\lambda x.\ x + 1)$”

  The harder, but safe choice.

The Three Basic Ways of Introducing Types

➜ **typedecl**: by name only

Example: **typedecl** names

THE THREE BASIC WAYS OF INTRODUCING TYPES

➜ **typedecl**: by name only

Example: **typedecl** 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** α 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** α rel = "$\alpha \Rightarrow \alpha \Rightarrow bool$"

Introduces abbreviation *rel* for existing type $\alpha \Rightarrow \alpha \Rightarrow bool$

Type abbreviations are immediatly expanded internally

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** α rel = "$\alpha \Rightarrow \alpha \Rightarrow bool$"

Introduces abbreviation *rel* for existing type $\alpha \Rightarrow \alpha \Rightarrow bool$

Type abbreviations are immediatly expanded internally

➜ **typedef**: by definiton as a set

Example: **typdef** 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** α rel = "$\alpha \Rightarrow \alpha \Rightarrow bool$"

  Introduces abbreviation *rel* for existing type $\alpha \Rightarrow \alpha \Rightarrow bool$

  Type abbreviations are immediatly expanded internally

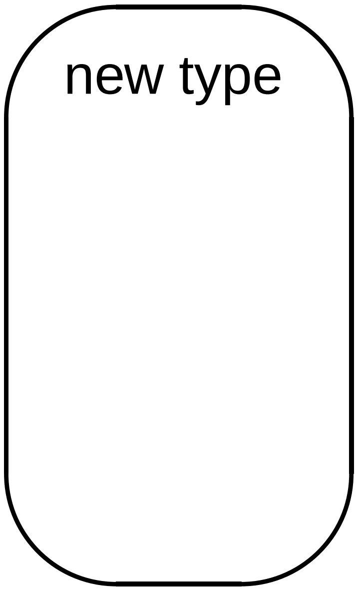
- **typedef**: by definiton as a set

  Example: **typdef** new_type = "$\{$some set$\}$" $<$proof$>$

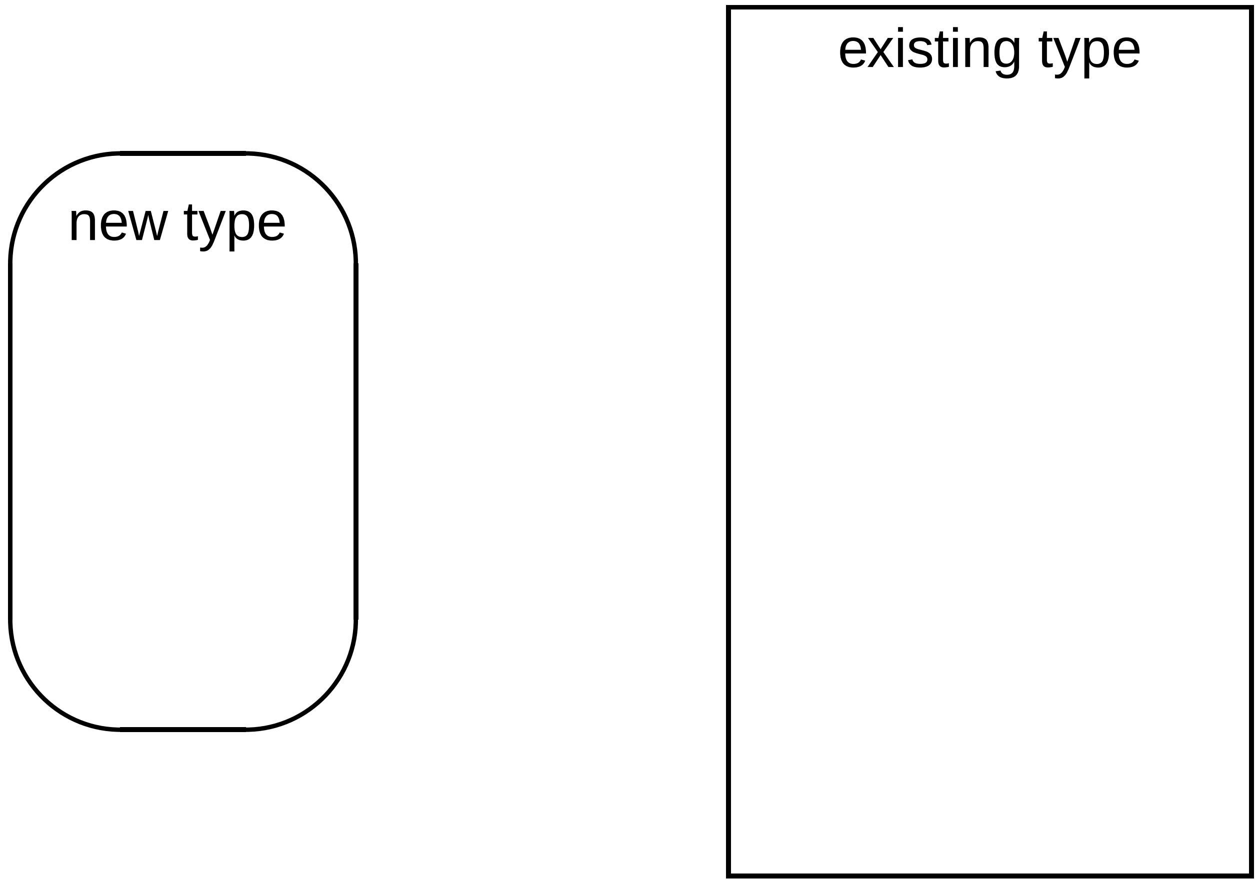
  Introduces a new type as a subset of an existing type.

  The proof shows that the set on the rhs in non-empty.

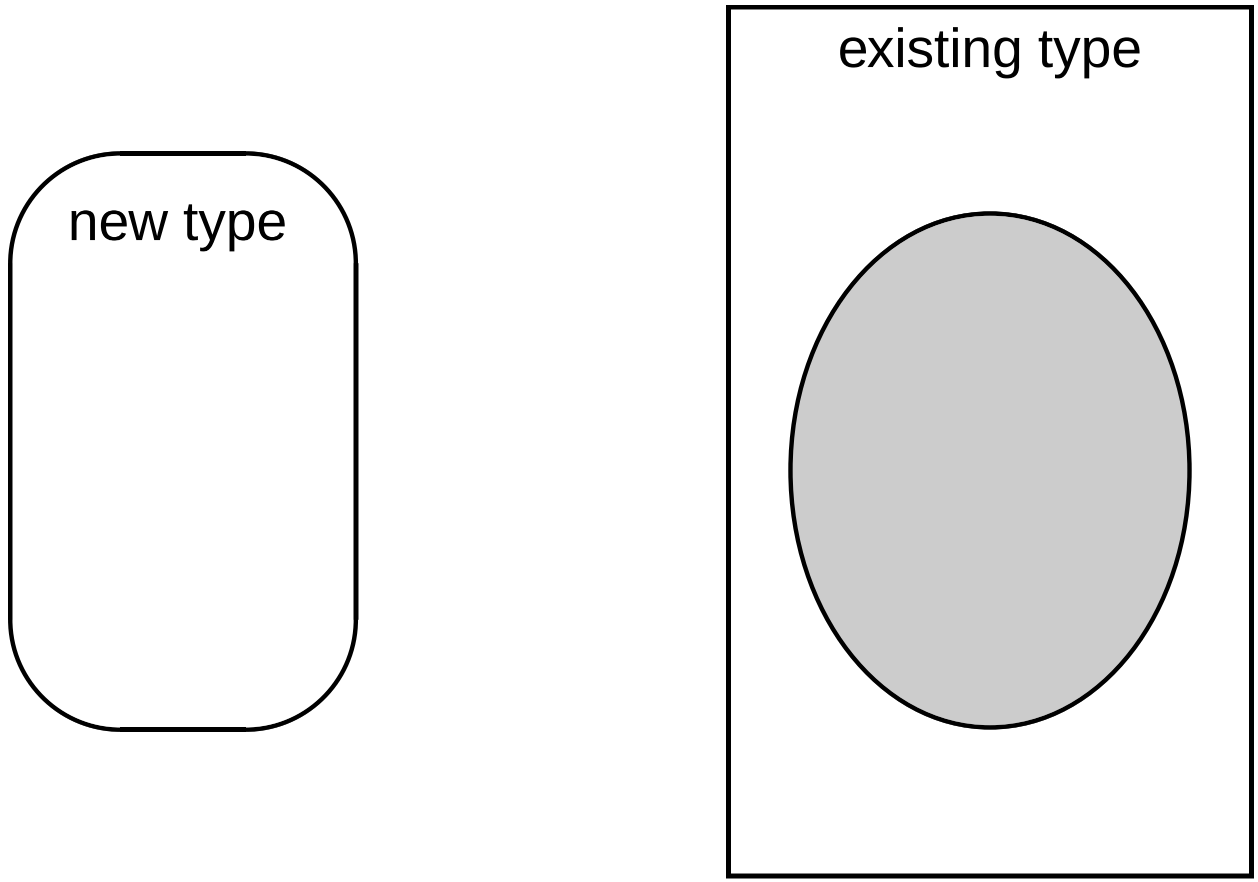
How typedef Works



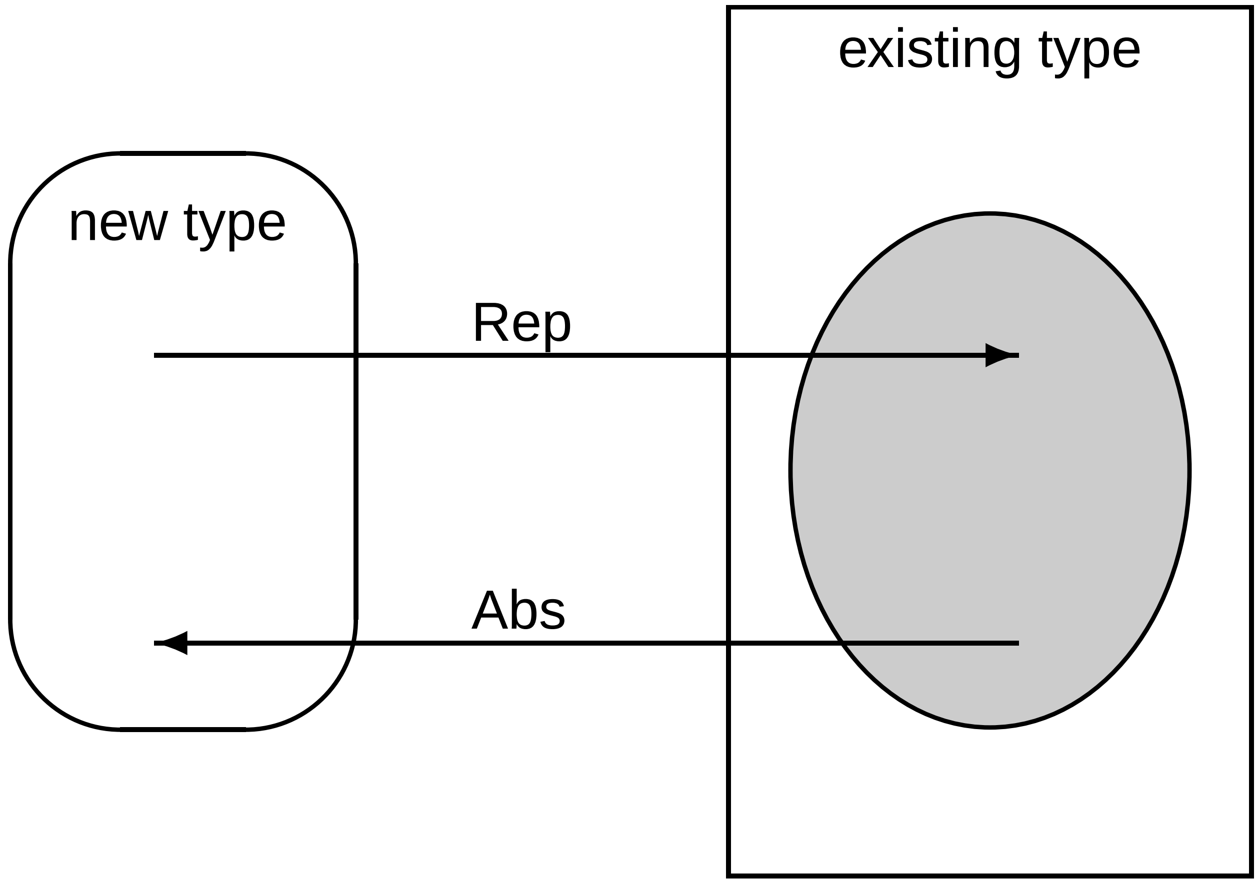
HOW TYPEDEF WORKS



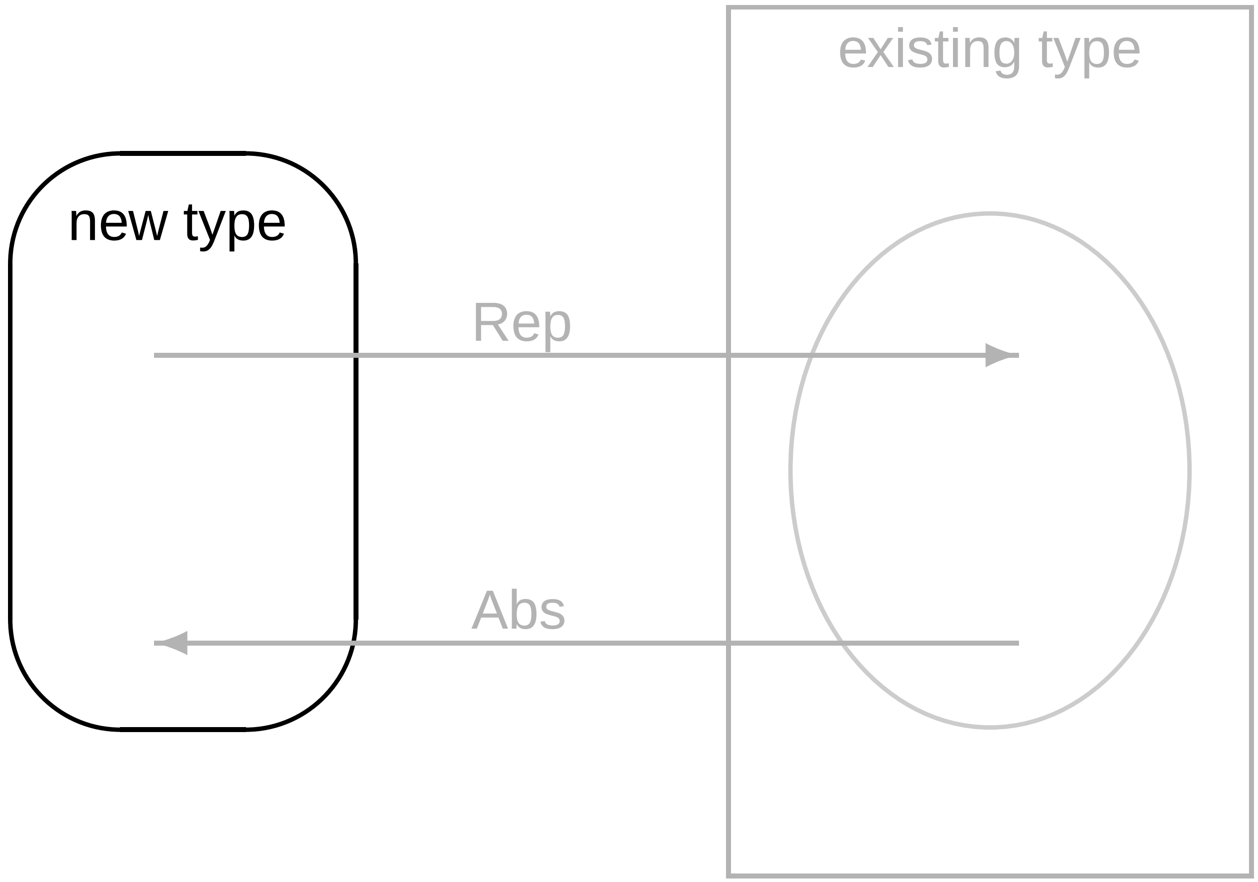
How typedef Works



HOW TYPEDEF WORKS



How typedef Works



EXAMPLE: PAIRS

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

① Pick existing type:

EXAMPLE: PAIRS

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

① Pick existing type: $\alpha \Rightarrow \beta \Rightarrow \mathsf{bool}$

② Identify subset:

Example: Pairs

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

① Pick existing type: $\alpha \Rightarrow \beta \Rightarrow \mathsf{bool}$

② Identify subset:
(α, β) Prod $= \{f.\ \exists a\ b.\ f = \lambda(x :: \alpha)\ (y :: \beta).\ x = a \wedge y = b\}$

③ We get from Isabelle:

EXAMPLE: PAIRS

(α, β) Prod

① Pick existing type: $\alpha \Rightarrow \beta \Rightarrow \mathsf{bool}$

② Identify subset:
(α, β) Prod $= \{f.\ \exists a\ b.\ f = \lambda(x :: \alpha)\ (y :: \beta).\ x = a \wedge y = b\}$

③ We get from Isabelle:

- functions Abs_Prod, Rep_Prod
- both injective
- Abs_Prod (Rep_Prod x) $= x$

④ We now can:

Example: Pairs

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

① Pick existing type: $\alpha \Rightarrow \beta \Rightarrow \mathsf{bool}$

② Identify subset:
(α, β) Prod $= \{f.\ \exists a\ b.\ f = \lambda(x :: \alpha)\ (y :: \beta).\ x = a \wedge y = b\}$

③ We get from Isabelle:

- functions Abs_Prod, Rep_Prod
- both injective
- Abs_Prod (Rep_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

$$l_1 \longrightarrow r_1$$

$$l_2 \longrightarrow r_2$$

$$\vdots$$

$$l_n \longrightarrow r_n$$

decide $l = r$ **by deciding** $l \stackrel{*}{\longleftrightarrow} r$

Arrow Cheat Sheet

$$\xrightarrow{0} \quad = \quad \{(x,y) | x = y\} \qquad \text{identity}$$

ARROW CHEAT SHEET

$\xrightarrow{0}$	$=$	$\{(x,y)\|x=y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y) \mid x = y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y) \mid x = y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y) \mid x = y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=}$	$=$	$\longrightarrow \cup \xrightarrow{0}$	reflexive closure

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y) \mid x = y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=}$	$=$	$\longrightarrow \cup \xrightarrow{0}$	reflexive closure
$\xrightarrow{-1}$	$=$	$\{(y,x) \mid x \longrightarrow y\}$	inverse

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y) \mid x = y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=}$	$=$	$\longrightarrow \cup \xrightarrow{0}$	reflexive closure
$\xrightarrow{-1}$	$=$	$\{(y,x) \mid x \longrightarrow y\}$	inverse
$\longleftarrow$	$=$	$\xrightarrow{-1}$	inverse

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y) \mid x = y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=}$	$=$	$\longrightarrow \cup \xrightarrow{0}$	reflexive closure
$\xrightarrow{-1}$	$=$	$\{(y,x) \mid x \longrightarrow y\}$	inverse
$\longleftarrow$	$=$	$\xrightarrow{-1}$	inverse
$\longleftrightarrow$	$=$	$\longleftarrow \cup \longrightarrow$	symmetric closure

Arrow Cheat Sheet

$\xrightarrow{0}$	$=$	$\{(x,y) \mid x = y\}$	identity
$\xrightarrow{n+1}$	$=$	$\xrightarrow{n} \circ \longrightarrow$	n+1 fold composition
$\xrightarrow{+}$	$=$	$\bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*}$	$=$	$\xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=}$	$=$	$\longrightarrow \cup \xrightarrow{0}$	reflexive closure
$\xrightarrow{-1}$	$=$	$\{(y,x) \mid x \longrightarrow y\}$	inverse
$\longleftarrow$	$=$	$\xrightarrow{-1}$	inverse
$\longleftrightarrow$	$=$	$\longleftarrow \cup \longrightarrow$	symmetric closure
$\overset{+}{\longleftrightarrow}$	$=$	$\bigcup_{i>0} \overset{i}{\longleftrightarrow}$	transitive symmetric closure
$\overset{*}{\longleftrightarrow}$	$=$	$\overset{+}{\longleftrightarrow} \cup \overset{0}{\longleftrightarrow}$	reflexive transitive symmetric closure

How to Decide $l \stackrel{*}{\longleftrightarrow} r$

Same idea as for β:

How to Decide $l \xleftrightarrow{*} r$

Same idea as for β: look for n such that $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$

Does this always work?

How to Decide $l \xleftrightarrow{*} r$

Same idea as for β: look for n such that $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$

Does this always work?

If $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$ then $l \xleftrightarrow{*} r$. Ok.

How to Decide $l \xleftrightarrow{*} r$

Same idea as for β: look for n such that $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$

Does this always work?

If $l \xrightarrow{*} n$ and $r \xrightarrow{*} n$ then $l \xleftrightarrow{*} r$. Ok.

If $l \xleftrightarrow{*} r$, will there always be a suitable n?

How to Decide $l \stackrel{*}{\longleftrightarrow} r$

Same idea as for β: look for n such that $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$

Does this always work?

If $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$ then $l \stackrel{*}{\longleftrightarrow} r$. Ok.

If $l \stackrel{*}{\longleftrightarrow} r$, will there always be a suitable n? **No!**

Example:

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

How to Decide $l \stackrel{*}{\longleftrightarrow} r$

Same idea as for β: look for n such that $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$

Does this always work?

If $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$ then $l \stackrel{*}{\longleftrightarrow} r$. Ok.

If $l \stackrel{*}{\longleftrightarrow} r$, will there always be a suitable n? **No!**

Example:

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

$f\ x \stackrel{*}{\longleftrightarrow} g\ x$ because $f\ x \longrightarrow a \longleftarrow f\ (g\ x) \longrightarrow b \longleftarrow g\ x$

HOW TO DECIDE $l \stackrel{*}{\longleftrightarrow} r$

Same idea as for β: look for n such that $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$

Does this always work?

If $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$ then $l \stackrel{*}{\longleftrightarrow} r$. Ok.

If $l \stackrel{*}{\longleftrightarrow} r$, will there always be a suitable n? **No!**

Example:

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

$f\ x \stackrel{*}{\longleftrightarrow} g\ x$ because $f\ x \longrightarrow a \longleftarrow f\ (g\ x) \longrightarrow b \longleftarrow g\ x$

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

HOW TO DECIDE $l \stackrel{*}{\longleftrightarrow} r$

Same idea as for β: look for n such that $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$

Does this always work?

If $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$ then $l \stackrel{*}{\longleftrightarrow} r$. Ok.

If $l \stackrel{*}{\longleftrightarrow} r$, will there always be a suitable n? **No!**

Example:

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

$f\ x \stackrel{*}{\longleftrightarrow} g\ x$ because $f\ x \longrightarrow a \longleftarrow f\ (g\ x) \longrightarrow b \longleftarrow g\ x$

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

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

$$l \stackrel{*}{\longleftrightarrow} r \Longrightarrow \exists n.\ l \stackrel{*}{\longrightarrow} n \wedge r \stackrel{*}{\longrightarrow} n$$

How to Decide $l \stackrel{*}{\longleftrightarrow} r$

Same idea as for β: look for n such that $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$

Does this always work?

If $l \stackrel{*}{\longrightarrow} n$ and $r \stackrel{*}{\longrightarrow} n$ then $l \stackrel{*}{\longleftrightarrow} r$. Ok.

If $l \stackrel{*}{\longleftrightarrow} r$, will there always be a suitable n? **No!**

Example:

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

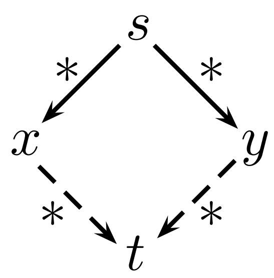
$f\ x \stackrel{*}{\longleftrightarrow} g\ x$ because $f\ x \longrightarrow a \longleftarrow f\ (g\ x) \longrightarrow b \longleftarrow g\ x$

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

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

$$l \stackrel{*}{\longleftrightarrow} r \Longrightarrow \exists n.\ l \stackrel{*}{\longrightarrow} n \wedge r \stackrel{*}{\longrightarrow} n$$

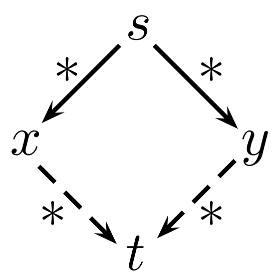
Fact: $\longrightarrow$ is Church-Rosser iff it is confluent.



Problem:

is a given set of reduction rules confluent?

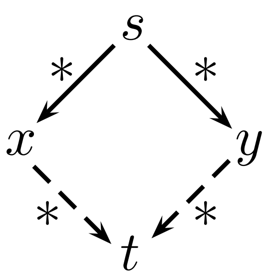
CONFLUENCE



Problem:

is a given set of reduction rules confluent?

undecidable

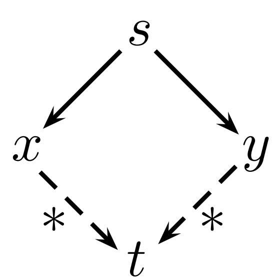


Problem:

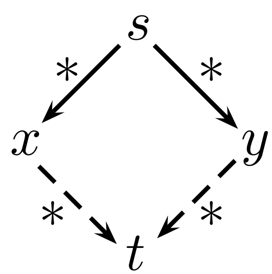
is a given set of reduction rules confluent?

undecidable

Local Confluence



CONFLUENCE

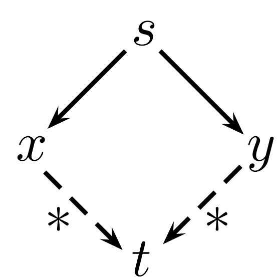


Problem:

is a given set of reduction rules confluent?

undecidable

Local Confluence



Fact: local confluence and termination $\Longrightarrow$ confluence

TERMINATION

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

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

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

Example:

TERMINATION

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

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

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

Example:

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

TERMINATION

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

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

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

Example:

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

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

Termination

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

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

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

Example:

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

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

Problem: is a given set of reduction rules terminating?

TERMINATION

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

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

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

Example:

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

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

Problem: is a given set of reduction rules terminating?

undecidable

WHEN IS $\longrightarrow$ TERMINATING?

Basic Idea:

When is $\longrightarrow$ Terminating?

Basic Idea: when the r_i are in some way simpler then the l_i

When is $\longrightarrow$ Terminating?

Basic Idea: when the r_i are in some way simpler then the l_i

More formally: $\longrightarrow$ 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 $\longrightarrow$ TERMINATING?

Basic Idea: when the r_i are in some way simpler then the l_i

More formally: $\longrightarrow$ 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: $f\ (g\ x) \longrightarrow g\ x$, $g\ (f\ x) \longrightarrow f\ x$

This system always terminates. Reduction order:

WHEN IS $\longrightarrow$ TERMINATING?

Basic Idea: when the r_i are in some way simpler then the l_i

More formally: $\longrightarrow$ 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: $f\ (g\ x) \longrightarrow g\ x$, $g\ (f\ x) \longrightarrow f\ x$

This system always terminates. Reduction order:

$s <_r t$ iff $size(s) < size(t)$ with
$size(s) =$ numer of function symbols in s

WHEN IS $\longrightarrow$ TERMINATING?

Basic Idea: when the r_i are in some way simpler then the l_i

More formally: $\longrightarrow$ 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: $f\ (g\ x) \longrightarrow g\ x$, $g\ (f\ x) \longrightarrow f\ x$

This system always terminates. Reduction order:

$s <_r t$ iff $size(s) < size(t)$ with
$size(s) =$ numer of function symbols in s

① $g\ x <_r f\ (g\ x)$ and $f\ x <_r g\ (f\ x)$

When is $\longrightarrow$ Terminating?

Basic Idea: when the r_i are in some way simpler then the l_i

More formally: $\longrightarrow$ 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: $f\ (g\ x) \longrightarrow g\ x$, $g\ (f\ x) \longrightarrow f\ x$

This system always terminates. Reduction order:

$s <_r t$ iff $size(s) < size(t)$ with
$size(s) =$ numer of function symbols in s

① $g\ x <_r f\ (g\ x)$ and $f\ x <_r g\ (f\ x)$
② $<_r$ is well founded, because $<$ is well founded on $\mathbb{N}$

Term Rewriting in Isabelle

Term rewriting engine in Isabelle is called **Simplifier**

TERM REWRITING IN ISABELLE

Term rewriting engine in Isabelle is called **Simplifier**

apply simp

➜ uses simplification rules

Term Rewriting in Isabelle

Term rewriting engine in Isabelle is called **Simplifier**

apply simp

➜ uses simplification rules

➜ (almost) blindly from left to right

Term Rewriting in Isabelle

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 IN ISABELLE

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)

TERM REWRITING IN ISABELLE

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)

CONTROL

➜ Equations turned into simplifaction rules with **[simp]** attribute

CONTROL

- ➜ Equations turned into simplifaction rules with **[simp]** attribute

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

CONTROL

- ➜ Equations turned into simplifaction 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

No structure.

What about..

- ➜ Elegance?

apply scripts

- ➜ unreadable
- ➜ hard to maintain
- ➜ do not scale

What about..

- ➜ Elegance?
- ➜ Explaining deeper insights?

No structure.

apply scripts	**What about..**
➜ unreadable	➜ Elegance?
➜ hard to maintain	➜ Explaining deeper insights?
➜ do not scale	➜ Large developments?

No structure.

apply scripts

- ➜ unreadable
- ➜ hard to maintain
- ➜ do not scale

No structure.

What about..

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

Isar!

A TYPICAL ISAR PROOF

proof

  assume $formula_0$

  have $formula_1$ **by** simp

  $\vdots$

  have $formula_n$ **by** blast

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

qed

A TYPICAL ISAR PROOF

proof

  assume $formula_0$

  have $formula_1$ **by** simp

  $\vdots$

  have $formula_n$ **by** blast

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

qed

proves $formula_0 \Longrightarrow formula_{n+1}$

proof

  assume $formula_0$

  have $formula_1$ **by** simp

  $\vdots$

  have $formula_n$ **by** blast

  show $formula_{n+1}$ **by** $\dots$

qed

proves $formula_0 \Longrightarrow formula_{n+1}$

(analogous to **assumes/shows** in lemma statements)

ISAR CORE SYNTAX

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

| **by** method

ISAR CORE SYNTAX

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

$\quad|$ **by** method

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

ISAR CORE SYNTAX

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

$\quad\quad$ | **by** method

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

statement = **fix** variables $\quad\quad$ ($\bigwedge$)

$\quad\quad$ | **assume** proposition $\quad\quad$ ($\Longrightarrow$)

$\quad\quad$ | [**from** name$^+$] (**have** | **show**) proposition proof

$\quad\quad$ | **next** $\quad\quad$ (separates subgoals)

ISAR CORE SYNTAX

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

$\quad\quad$ | **by** method

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

statement = **fix** variables $\quad$ ($\bigwedge$)

$\quad\quad$ | **assume** proposition $\quad$ ($\Longrightarrow$)

$\quad\quad$ | [**from** name$^+$] (**have** | **show**) proposition proof

$\quad\quad$ | **next** $\quad$ (separates subgoals)

proposition = [name:] formula

PROOF AND QED

proof [method] statement* **qed**

lemma "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"

PROOF AND QED

proof [method] statement* **qed**

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
proof (rule conjI)

PROOF AND QED

proof [method] statement* **qed**

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
proof (rule conjI)
  assume A: "A"
  from A **show** "A" **by** assumption

proof [method] statement* **qed**

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"

proof (rule conjI)

  assume A: "A"

  from A **show** "A" **by** assumption

next

PROOF AND QED

proof [method] statement* **qed**

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
proof (rule conjI)
  assume A: "A"
  from A **show** "A" **by** assumption
next
  assume B: "B"
  from B **show** "B" **by** assumption

proof [method] statement* **qed**

lemma "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"
proof (rule conjI)
    assume A: "A"
    from A **show** "A" **by** assumption
next
    assume B: "B"
    from B **show** "B" **by** assumption
qed

PROOF AND QED

proof [method] statement* **qed**

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
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 AND QED

proof [method] statement* **qed**

lemma "$[\![A; B]\!] \Longrightarrow A \wedge B$"
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 AND QED

proof [method] statement* **qed**

lemma "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"

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

How do I know what to Assume and Show?

Look at the proof state!

lemma "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"
proof (rule conjI)

HOW DO I KNOW WHAT TO ASSUME AND SHOW?

Look at the proof state!

lemma "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"
proof (rule conjI)

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

1. $\llbracket A; B \rrbracket \Longrightarrow A$
2. $\llbracket A; B \rrbracket \Longrightarrow B$

HOW DO I KNOW WHAT TO ASSUME AND SHOW?

Look at the proof state!

lemma "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"
proof (rule conjI)

- **proof** (rule conjI) changes proof state to
  1. $\llbracket A; B \rrbracket \Longrightarrow A$
  2. $\llbracket A; B \rrbracket \Longrightarrow B$
- 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 "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$"
proof (rule conjI)

- **proof** (rule conjI) changes proof state to
  1. $\llbracket A; B \rrbracket \Longrightarrow A$
  2. $\llbracket A; B \rrbracket \Longrightarrow B$
- 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 openend 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 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.

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 "$[\![A; B]\!] \Longrightarrow A \wedge B$"

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 "$[\![A; B]\!] \Longrightarrow A \wedge B$" **[prove]**

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 "$[\![A; B]\!] \Longrightarrow A \wedge B$" **[prove]**
proof (rule conjI) **[state]**

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 "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$" **[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 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 "$\llbracket A; B \rrbracket \Longrightarrow A \wedge B$" **[prove]**
proof (rule conjI) **[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 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 "$[\![A; B]\!] \Longrightarrow A \wedge B$" **[prove]**
proof (rule conjI) **[state]**
  assume A: "A" **[state]**
  from A **[chain]** **show** "A" **[prove]** **by** assumption **[state]**
next **[state]** **. . .**

HAVE

Can be used to make intermediate steps.

Example:

HAVE

Can be used to make intermediate steps.

Example:

lemma "$(x :: \mathsf{nat}) + 1 = 1 + x$"

proof -

have A: "$x + 1 = \mathsf{Suc}\ x$" **by** simp

have B: "$1 + x = \mathsf{Suc}\ x$" **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 $\wedge, \vee, \longrightarrow, \neg$
  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 $(\neg B \longrightarrow C) \longrightarrow A \longrightarrow B$