\[
a = b \leq c \leq \ldots
\]
 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
  • More recursion, Calculational reasoning
  • Hoare logic, proofs about programs
  • Locales, Presentation
LAST WEEK

➔ Constructive Logic & Curry-Howard-Isomorphism
LAST WEEK

- Constructive Logic & Curry-Howard-Isomorphism
- The Coq System
LAST WEEK

➔ Constructive Logic & Curry-Howard-Isomorphism
➔ The Coq System
➔ The HOL4 system
LAST WEEK

- Constructive Logic & Curry-Howard-Isomorphism
- The Coq System
- The HOL4 system
- Before that: datatypes, recursion, induction
GENERAL Recursion

The Choice
GENERAL RECURSION

The Choice

→ Limited expressiveness, automatic termination
  • primrec
The Choice

- Limited expressiveness, automatic termination
  - primrec

- High expressiveness, prove termination manually
  - recdef
consts sep :: "'a × 'a list ⇒ 'a list"
recdef sep "measure (λ(a, xs). size xs)"
  "sep (a, x # y # zs) = x # a # sep (a, y # zs)"
  "sep (a, xs) = xs"
consts sep :: "'a × 'a list ⇒ 'a list"
recdef sep "measure (λ(a, xs). size xs)"
  "sep (a, x # y # zs) = x # a # sep (a, y # zs)"
  "sep (a, xs) = xs"

consts ack :: "nat × nat ⇒ nat"
recdef ack "measure (λm. m) <*lex*> measure (λn. n)"
  "ack (0, n) = Suc n"
  "ack (Suc m, 0) = ack (m, 1)"
  "ack (Suc m, Suc n) = ack (m, ack (Suc m, n))"
The definition:

- one parameter
- free pattern matching, order of rules important
- termination relation
  (measure sufficient for most cases)
The definition:

- one parameter
- free pattern matching, order of rules important
- termination relation
  (measure sufficient for most cases)

Termination relation:

- must decrease for each recursive call
- must be well founded
The definition:

- one parameter
- free pattern matching, order of rules important
- termination relation
  (measure sufficient for most cases)

Termination relation:

- must decrease for each recursive call
- must be well founded

Generates own induction principle
→ Each recdef definition induces an induction principle
Each recdef definition induces an induction principle

For each equation:

show that the property holds for the lhs provided it holds for each recursive call on the rhs
Each recdef definition induces an induction principle

For each equation:

show that the property holds for the lhs provided it holds for each recursive call on the rhs

Example sep.induct:

\[
\begin{align*}
\[ & \land a. \ P \ a \ []; \\
& \land a \ w. \ P \ a \ [w] \\
& \land a \ x \ y \ zs. \ P \ a \ (y#zs) \implies P \ a \ (x#y#zs); \\
\] \implies \ P \ a \ xs
\end{align*}
\]
Isabelle tries to prove termination automatically

→ For most functions and termination relations this works.
TERMINATION

Isabelle tries to prove termination automatically

⇒ For most functions and termination relations this works.
⇒ Sometimes not

recdef quicksort "measure length"
quicksort [] = quicksort []
quicksort (x # xs) = quicksort [y # xs | x < y] @ [x]@

(hints recdef simp: less Suc eq le)

For exploration:
⇒ allow failing termination proof
⇒ recdef (permissive) quicksort "measure length"
⇒ termination conditions as assumption in simp and induct rules
**TERMINATION**

Isabelle tries to prove termination automatically

- For most functions and termination relations this works.
- Sometimes not \(\Rightarrow\) error message with unsolved subgoal
TERMINATION

Isabelle tries to prove termination automatically

⇒ For most functions and termination relations this works.
⇒ Sometimes not ⇒ error message with unsolved subgoal
⇒ You can give hints (additional lemmas) to the recdef package:

\[\text{recdef quicksort "measure length"}
\text{quicksort \([] = []\)
\text{quicksort \((x \# xs) = quicksort \([y \in xs.y \leq x]@[x]@[quicksort \([y \in xs.x < y]\]
\text{(hints recdef_simp: less_Suc_eq_le)\]
}
Isabelle tries to prove termination automatically

- For most functions and termination relations this works.
- Sometimes not \(\Rightarrow\) error message with unsolved subgoal
- You can give hints (additional lemmas) to the recdef package:

```
recdef quicksort "measure length"
quicksort [] = []
quicksort (x#xs) = quicksort [y ∈ xs.y ≤ x]@[x]@ quicksort [y ∈ xs.x < y] (hints recdef_simps: less_Suc_eq_le)
```

For exploration:

- allow failing termination proof
Isabelle tries to prove termination automatically

→ For most functions and termination relations this works.
→ Sometimes not ⇒ error message with unsolved subgoal
→ You can give hints (additional lemmas) to the recdef package:

recdef quicksort "measure length"
quicksort [] = []
quicksort (x#xs) = quicksort [y ∈ xs. y ≤ x]@[x]@ quicksort [y ∈ xs. x < y]
(hints recdef_simp: less_Suc_eq_le)

For exploration:

→ allow failing termination proof
→ recdef (permissive) quicksort "measure length"
TERMINATION

Isabelle tries to prove termination automatically

→ For most functions and termination relations this works.
→ Sometimes not ⇒ error message with unsolved subgoal
→ You can give **hints** (additional lemmas) to the recdef package:

```isar
recdef quicksort "measure length"
quicksort [] = []
quicksort (x#xs) = quicksort [y ∈ xs. y ≤ x]@[x]@ quicksort [y ∈ xs. x < y]
(hints recdef_simp: less_Suc_eq_le)
```

**For exploration:**

→ allow failing termination proof
→ **recdef (permissive)** quicksort "measure length"
→ termination conditions as assumption in simp and induct rules
HOW DOES RECDEF WORK?

We need:  general recursion operator
HOW DOES RECDEF WORK?

We need: general recursion operator

something like: \( \text{rec } F = F (\text{rec } F) \)
HOW DOES RECDEF WORK?

We need: general recursion operator

something like: \( \text{rec } F = F \ (\text{rec } F) \)  
\((F \text{ stands for the recursion equations})\)

Example:
**How does recdef work?**

**We need:** general recursion operator

something like: $rec\ F = F\ (rec\ F)$

($F$ stands for the recursion equations)

**Example:**

$\rightarrow$ recursion equations: $f = 0$  
$x (SUC\ n) = fn$
How does recdef work?

We need: general recursion operator

something like: \( \text{rec } F = F \ (\text{rec } F) \)

(\( F \) stands for the recursion equations)

Example:

\( \rightarrow \) recursion equations: \( f = 0 \quad f \ (\text{Suc} \ n) = fn \)

\( \rightarrow \) as one \( \lambda \)-term: \( f = \lambda n'. \ case \ n' \ of \ 0 \Rightarrow 0 \ | \ Suc \ n \Rightarrow f \ n \)
How does recdef work?

We need: general recursion operator

something like: \(\text{rec} \ F = F \ (\text{rec} \ F)\)

\((F\) stands for the recursion equations\)

Example:

\(\rightarrow\) recursion equations: \(f = 0 \quad f \ (\text{Suc} \ n) = fn\)

\(\rightarrow\) as one \(\lambda\)-term: \(f = \lambda n'. \ \text{case} \ n' \ \text{of} \ 0 \Rightarrow 0 \mid \text{Suc} \ n \Rightarrow f \ n\)

\(\rightarrow\) functor: \(F = \lambda f. \ \lambda n'. \ \text{case} \ n' \ \text{of} \ 0 \Rightarrow 0 \mid \text{Suc} \ n \Rightarrow f \ n\)
HOW DOES RECDATA WORK?

We need: general recursion operator

something like: \( \text{rec } F = F \ (\text{rec } F) \)

\( (F \text{ stands for the recursion equations}) \)

Example:

→ recursion equations: \( f = 0 \quad f \ (\text{Suc } n) = f n \)
→ as one \( \lambda \)-term: \( f = \lambda \ n'. \ \text{case } n' \text{ of } 0 \Rightarrow 0 | \text{Suc } n \Rightarrow f \ n \)
→ functor: \( F = \lambda f. \ \lambda \ n'. \ \text{case } n' \text{ of } 0 \Rightarrow 0 | \text{Suc } n \Rightarrow f \ n \)

→ \( \text{rec } :: ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta)) \Rightarrow (\alpha \Rightarrow \beta) \) like above cannot exist in HOL (only total functions)
→ But 'guarded' form possible:
  \( \text{wfrec } :: (\alpha \times \alpha) \text{ set } \Rightarrow ((\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta)) \Rightarrow (\alpha \Rightarrow \beta) \)
→ \( (\alpha \times \alpha) \text{ set a well founded order, decreasing with execution} \)
Why \( \text{rec } F = F (\text{rec } F) \)?
HOW DOES recdef WORK?

Why \( rec \ F = F \ (rec \ F) \)?

Because we want the recursion equations to hold.

Example:

\[
\begin{align*}
F & \equiv \ \lambda g. \ \lambda n'. \ \text{case } n' \text{ of } 0 & \Rightarrow & 0 \mid \text{Suc } n \Rightarrow g \ n \\
f & \equiv \ \text{rec } F
\end{align*}
\]
HOW DOES RECDEF WORK?

Why $rec\ F = F\ (rec\ F)$?

Because we want the recursion equations to hold.

Example:

$F \equiv \lambda g.\ \lambda n'.\ \text{case}\ n'\ \text{of}\ 0 \Rightarrow 0\ |\ \text{Suc}\ n \Rightarrow g\ n$

$f \equiv rec\ F$

$f\ 0 = rec\ F\ 0$
HOW DOES RECDEF WORK?

Why \( \text{rec} \, F = F \,(\text{rec} \, F) \)?

Because we want the recursion equations to hold.

Example:

\[
\begin{align*}
F & \equiv \lambda g. \, \lambda n'. \ \text{case} \ n' \ \text{of} \ 0 & \Rightarrow & 0 \ | \ \text{Suc} \ n \Rightarrow g \ n \\
f & \equiv \text{rec} \, F \\
\text{rec} \, F \ 0 & \equiv \text{rec} \, F \ 0 \\
\ldots & \equiv F \,(\text{rec} \, F) \ 0
\end{align*}
\]
**How does recdef work?**

Why $rec\ F = F\ (rec\ F)$?

Because we want the recursion equations to hold.

Example:

\[
\begin{align*}
F & \equiv \lambda g.\ \lambda n'.\ \text{case } n' & (0 \Rightarrow 0) & | & (\text{Suc } n \Rightarrow g\ n) \\
\quad f & \equiv rec\ F \\
\quad f\ 0 & \equiv rec\ F\ 0 \\
\quad \ldots & \equiv F\ (rec\ F)\ 0 \\
\quad \ldots & \equiv (\lambda g.\ \lambda n'.\ \text{case } n' & (0 \Rightarrow 0) & | & (\text{Suc } n \Rightarrow g\ n))\ (rec\ F)\ 0
\end{align*}
\]
How does recdef work?

Why \( \text{rec } F = F (\text{rec } F) \)?

Because we want the recursion equations to hold.

Example:

\[
\begin{align*}
F & \equiv \lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 | \text{Suc } n \Rightarrow g \ n \\
f & \equiv \text{rec } F \\
f \ 0 & = \text{rec } F \ 0 \\
\ldots & = F (\text{rec } F) \ 0 \\
\ldots & = (\lambda g. \lambda n'. \text{case } n' \text{ of } 0 \Rightarrow 0 | \text{Suc } n \Rightarrow g \ n) (\text{rec } F) \ 0 \\
\ldots & = (\text{case } 0 \text{ of } 0 \Rightarrow 0 | \text{Suc } n \Rightarrow \text{rec } F \ n)
\end{align*}
\]
HOW DOES RECDEF WORK?

Why \( rec\, F = F\, (rec\, F)\)?

Because we want the recursion equations to hold.

Example:

\[
\begin{align*}
F & \equiv \lambda g. \lambda n'. \text{ case } n' \text{ of } 0 \Rightarrow 0 | \text{ Suc } n \Rightarrow g\, n \\
f & \equiv rec\, F
\end{align*}
\]

\[
\begin{align*}
f\, 0 & = rec\, F\, 0 \\
\ldots & = F\, (rec\, F)\, 0 \\
\ldots & = (\lambda g. \lambda n'. \text{ case } n' \text{ of } 0 \Rightarrow 0 | \text{ Suc } n \Rightarrow g\, n)\, (rec\, F)\, 0 \\
\ldots & = (\text{ case } 0 \text{ of } 0 \Rightarrow 0 | \text{ Suc } n \Rightarrow rec\, F\, n) \\
\ldots & = 0
\end{align*}
\]
Well Founded Orders

Definition

$\prec_r$ is well founded if well founded induction holds

$\text{wf } r \equiv \forall P. (\forall x. (\forall y <_r x. P y) \rightarrow P x) \rightarrow (\forall x. P x)$
Well Founded Orders

Definition

$<_r$ is well founded if well founded induction holds

\[
\text{wf } r \equiv \forall P. (\forall x. (\forall y <_r x. P y) \rightarrow P x) \rightarrow (\forall x. P x)
\]

Well founded induction rule:

\[
\begin{align*}
\text{wf } r & \quad \bigwedge x. (\forall y <_r x. P y) \rightarrow P x \\
\text{Pa} & \quad \frac{}{P x}
\end{align*}
\]
Well Founded Orders

Definition

\( <_r \) is well founded if well founded induction holds

\[
\text{wf } r \equiv \forall P. (\forall x. (\forall y <_r x. P y) \rightarrow P x) \rightarrow (\forall x. P x)
\]

Well founded induction rule:

\[
\begin{array}{c}
\text{wf } r \\
\forall x. (\forall y <_r x. P y) \\
\hline
\exists a
\end{array}
\]

Alternative definition (equivalent):

there are no infinite descending chains, or (equivalent):
every nonempty set has a minimal element wrt \( <_r \)

\[
\begin{align*}
\text{min } r Q x & \equiv \forall y \in Q. y \not<_r x \\
\text{wf } r & = (\forall Q \neq \{\}. \exists m \in Q. \text{min } r Q m)
\end{align*}
\]
Well Founed Orders: Examples

- < on \( \mathbb{N} \) is well founded
  well founded induction = complete induction
Well Founded Orders: Examples

- < on \(\mathbb{N}\) is well founded
  well founded induction = complete induction

- > and \(\leq\) on \(\mathbb{N}\) are **not** well founded
Well Founded Orders: Examples

→ < on \( \mathbb{N} \) is well founded
   well founded induction = complete induction

→ > and \( \leq \) on \( \mathbb{N} \) are not well founded

→ \( x <_r y = x \text{ dvd } y \land x \neq 1 \) on \( \mathbb{N} \) is well founded
   the minimal elements are the prime numbers
Well Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well founded
  well founded induction = complete induction

- $>$ and $\leq$ on $\mathbb{N}$ are **not** well founded

- $x <_r y = x \text{ dvd } y \land x \neq 1$ on $\mathbb{N}$ is well founded
  the minimal elements are the prime numbers

- $(a, b) <_r (x, y) = a <_1 x \lor a = x \land b <_1 y$ is well founded
  if $<_1$ and $<_2$ are
Well Founded Orders: Examples

- $<$ on $\mathbb{N}$ is well founded
  well founded induction = complete induction

- $>$ and $\leq$ on $\mathbb{N}$ are not well founded

- $x <_r y = x \text{ dvd } y \land x \neq 1$ on $\mathbb{N}$ is well founded
  the minimal elements are the prime numbers

- $(a, b) <_r (x, y) = a <_1 x \lor a = x \land b <_1 y$ is well founded
  if $<_1$ and $<_2$ are

- $A <_r B = A \subset B \land \text{finite } B$ is well founded
Well Founded Orders: Examples

→ < on \( \mathbb{N} \) is well founded
   well founded induction = complete induction

→ > and \( \leq \) on \( \mathbb{N} \) are not well founded

→ \( x <_r y = x \text{ dvd } y \land x \neq 1 \) on \( \mathbb{N} \) is well founded
   the minimal elements are the prime numbers

→ \( (a, b) <_r (x, y) = a <_1 x \lor a = x \land b <_1 y \) is well founded
   if \( <_1 \) and \( <_2 \) are

→ \( A <_r B = A \subset B \land \text{ finite } B \) is well founded

→ \( \subseteq \) and \( \subset \) in general are not well founded

More about well founded relations: Term Rewriting and All That
Back to recursion: \( \text{rec } F = F (\text{rec } F) \) not possible

Idea:
**The Recursion Operator**

**Back to recursion:** $\text{rec } F = F (\text{rec } F)$ not possible

**Idea:** have $\text{wfrec } R F$ where $R$ is well founded
The Recursion Operator

Back to recursion: \( \text{rec } F = F (\text{rec } F) \) not possible

Idea: have \( \text{wfrec } R F \) where \( R \) is well founded

Cut:
- only do recursion if parameter decreases wrt \( R \)
- otherwise: abort
The Recursion Operator

Back to recursion: \( rec\ F = F\ (rec\ F) \) not possible

Idea: have \( wf\ rec\ R\ F \) where \( R \) is well founded

Cut:

- only do recursion if parameter decreases wrt \( R \)
- otherwise: abort

- arbitrary \( \alpha \)
  
  cut \( \colon (\alpha \Rightarrow \beta) \Rightarrow (\alpha \times \alpha) \) set \( \Rightarrow \alpha \Rightarrow (\alpha \Rightarrow \beta) \)
  
  cut \( G\ R\ x \equiv \lambda y. \) if \( (y, x) \in R \) then \( G\ y \) else arbitrary
THE RECURSION OPERATOR

Back to recursion: \( \text{rec } F = F \ (\text{rec } F) \) not possible

Idea: have \( \text{wfrec } R \ F \) where \( R \) is well founded

Cut:

→ only do recursion if parameter decreases wrt \( R \)
→ otherwise: abort

→ arbitrary :: \( \alpha \)
  cut :: \( (\alpha \Rightarrow \beta) \Rightarrow (\alpha \times \alpha) \ \text{set} \Rightarrow \alpha \Rightarrow (\alpha \Rightarrow \beta) \)
  cut \( G \ R \ x \equiv \lambda y. \ \text{if } (y, x) \in R \ \text{then } G \ y \ \text{else arbitrary} \)

\[ \text{wf } R \implies \text{wfrec } R \ F \ x = F \ (\text{cut } (\text{wfrec } R \ F) \ R \ x) \ x \]
Admissible recursion

- recursive call for $x$ only depends on parameters $y <_R x$
- describes exactly one function if $R$ is well founded
The Recursion Operator

Admissible recursion

- recursive call for $x$ only depends on parameters $y <_R x$
- describes exactly one function if $R$ is well founded

\[
\text{adm}_{\text{wf}} \ R \ F \equiv \forall f \ g \ x. \ (\forall z. \ (z, x) \in R \rightarrow f \ z = g \ z) \rightarrow F \ f \ x = F \ g \ x
\]
The Recursion Operator

Admissible recursion

- recursive call for $x$ only depends on parameters $y <_R x$
- describes exactly one function if $R$ is well founded

$$\text{adm}_{wf} R F \equiv \forall f \; g \; x. \; (\forall z. \; (z, x) \in R \rightarrow f \; z = g \; z) \rightarrow F \; f \; x = F \; g \; x$$

Definition of $\text{wf}_\text{rec}$: again first by induction, then by epsilon

$$\text{(x, \quad ) } \in \text{wfrec}_{rel} \; R \; F$$
The Recursion Operator

Admissible recursion

- recursive call for $x$ only depends on parameters $y <_R x$
- describes exactly one function if $R$ is well founded

$$\text{adm}_w f R F \equiv \forall f g x. \ (\forall z. \ (z, x) \in R \rightarrow f z = g z) \rightarrow F f x = F g x$$

Definition of \texttt{wf}_rec: again first by induction, then by epsilon

$$\ (x, F g x) \in \text{wfrecrel} \ R \ F$$
THE RECURSION OPERATOR

Admissible recursion

- recursive call for \( x \) only depends on parameters \( y <_R x \)
- describes exactly one function if \( R \) is well founded

\[
\text{adm}_\text{wf} \ R \ F \equiv \forall f \ g \ x. \ (\forall z. \ (z, x) \in R \rightarrow f \ z = g \ z) \rightarrow F \ f \ x = F \ g \ x
\]

Definition of \( \text{wf}_\text{rec} \): again first by induction, then by epsilon

\[
\forall z. \ (z, x) \in R \rightarrow (z, g \ z) \in \text{wfrec}_\text{rel} \ R \ F
\]

\[
(x, F \ g \ x) \in \text{wfrec}_\text{rel} \ R \ F
\]
**The Recursion Operator**

### Admissible recursion

- recursive call for $x$ only depends on parameters $y <_R x$
- describes exactly one function if $R$ is well founded

\[
\text{adm}_\text{wf} \ R \ F \equiv \forall f \ g \ x. (\forall z. (z, x) \in R \rightarrow f \ z = g \ z) \rightarrow F \ f \ x = F \ g \ x
\]

**Definition of wf_rec**: again first by induction, then by epsilon

\[
\forall z. (z, x) \in R \rightarrow (z, g \ z) \in \text{wfrec}_\text{rel} \ R \ F
\]

\[
(x, F \ g \ x) \in \text{wfrec}_\text{rel} \ R \ F
\]

\[
\text{wfrec} \ R \ F \ x \equiv \text{THE} \ y. (x, y) \in \text{wfrec}_\text{rel} \ R \ (\lambda f \ x. F \ (\text{cut} \ f \ R \ x) \ x)
\]

More: John Harrison, *Inductive definitions: automation and application*
DEMO
CALCULATIONAL REASONING
THE GOAL

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]
THE GOAL

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]

Can we do this in Isabelle?
THE GOAL

\[ x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1}) \]
\[ \ldots = 1 \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1} \]
\[ \ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1}) \]
\[ \ldots = (x^{-1})^{-1} \cdot x^{-1} \]
\[ \ldots = 1 \]

Can we do this in Isabelle?

⇒ Simplifier: too eager
THE GOAL

\[x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1})\]
\[\ldots = 1 \cdot x \cdot x^{-1}\]
\[\ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1}\]
\[\ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1}\]
\[\ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1}\]
\[\ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1})\]
\[\ldots = (x^{-1})^{-1} \cdot x^{-1}\]
\[\ldots = 1\]

Can we do this in Isabelle?

→ Simplifier: too eager
→ Manual: difficult in apply stile
THE GOAL

\[
x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1})
\]
\[
\ldots = 1 \cdot x \cdot x^{-1}
\]
\[
\ldots = (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1}
\]
\[
\ldots = (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1}
\]
\[
\ldots = (x^{-1})^{-1} \cdot 1 \cdot x^{-1}
\]
\[
\ldots = (x^{-1})^{-1} \cdot (1 \cdot x^{-1})
\]
\[
\ldots = (x^{-1})^{-1} \cdot x^{-1}
\]
\[
\ldots = 1
\]

Can we do this in Isabelle?

⇒ Simplifier: too eager
⇒ Manual: difficult in apply stile
⇒ Isar: with the methods we know, too verbose
The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of \( = \)

CHAINS OF EQUATIONS
The Problem

\[ a = b \]

\[ \ldots = c \]

\[ \ldots = d \]

shows \( a = d \) by transitivity of \( = \)

Each step usually nontrivial (requires own subproof)
The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of \( = \)

Each step usually nontrivial (requires own subproof)

Solution in Isar:

⇒ Keywords also and finally to delimit steps
Chains of equations

The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of equality.

Each step usually nontrivial (requires own subproof)

Solution in Isar:

- Keywords also and finally to delimit steps
- \( \ldots : \) predefined schematic term variable, refers to right hand side of last expression
The Problem

\[ a = b \]
\[ \ldots = c \]
\[ \ldots = d \]

shows \( a = d \) by transitivity of =

Each step usually nontrivial (requires own subproof)

Solution in Isar:

- Keywords \textbf{also} and \textbf{finally} to delimit steps
- \ldots: predefined schematic term variable, refers to right hand side of last expression
- Automatic use of transitivity rules to connect steps
have $t_0 = t_1$ [proof]

also
also have "$t_0 = t_1$" [proof]
also calculation register
"$t_0 = t_1$"
also / finally

\begin{align*}
\text{have } & "t_0 = t_1" \quad \text{[proof]} \\
\text{also} & \\
\text{have } & "\ldots = t_2" \quad \text{[proof]} \\
\text{calculation register } & \quad "t_0 = t_1"
\end{align*}
have \( t_0 = t_1 \) [proof]

also

have \( \ldots = t_2 \) [proof]

also

calculation register

\( t_0 = t_1 \)

\( t_0 = t_2 \)
also
have "\( t_0 = t_1 \)" [proof]
also
have "... = t_2" [proof]
also
: 
also
: 
also
calculation register
"\( t_0 = t_1 \)"
"\( t_0 = t_2 \)"
: 
: 
"\( t_0 = t_{n-1} \)"
have "t_0 = t_1" [proof]
also
have "\ldots = t_2" [proof]
also
\vdots
also
have "\ldots = t_n" [proof]
calculation register
"t_0 = t_1"
"t_0 = t_2"
\vdots
"t_0 = t_{n-1}"

also
FINALLY
have "$t_0 = t_1$" [proof]
also
have "$\ldots = t_2$" [proof]
also
::
also
have "$\ldots = t_n$" [proof]
finally
calculation register
"$t_0 = t_1$"
"$t_0 = t_2$"
::
"$t_0 = t_{n-1}$"
$t_0 = t_n
have "\(t_0 = t_1\)" [proof]
also
have "\(\ldots = t_2\)" [proof]
also
\[ \vdots \]
also
have "\(\ldots = t_n\)" [proof]
finally

\(t_0 = t_n\)  

—’finally’ pipes fact "\(t_0 = t_n\)" into the proof
MORE ABOUT ALSO

→ Works for all combinations of \(=\), \(\leq\) and \(<\).
MORE ABOUT ALSO

⇒ Works for all combinations of =, ≤ and <.

⇒ Uses all rules declared as [trans].
Works for all combinations of \(=\), \(\leq\) and \(<\).

Uses all rules declared as \([\text{trans}]\).

To view all combinations in Proof General:

\text{Isabelle/Isar} \rightarrow \text{Show me} \rightarrow \text{Transitivity rules}
calculation = "l₁ ⊕ r₁"

have "... ⊕ r₂" [proof]

also ←
calculation = "l_1 \odot r_1"
have "... \odot r_2" [proof]
also \iff

Anatomy of a [trans] rule:

\[
\rightarrow \text{ Usual form: plain transitivity } [l_1 \odot r_1; r_1 \odot r_2] \rightarrow l_1 \odot r_2
\]
**DESIGNING [trans] RULES**

**calculation** = "\( l_1 \odot r_1 \)"

**have** ”... \( \odot r_2 \)“ [proof]

**also** \( \leftarrow \)

**Anatomy of a [trans] rule:**

- **Usual form:** plain transitivity \( [l_1 \odot r_1; r_1 \odot r_2] \rightarrow l_1 \odot r_2 \)
- **More general form:** \( [P l_1 r_1; Q r_1 r_2; A] \rightarrow C l_1 r_2 \)

**Examples:**
calculation = "l₁ ⊙ r₁"
have "... ⊙ r₂" [proof]
also ⇐

Anatomy of a [trans] rule:

¬ Usual form: plain transitivity \([l_1 ⊙ r_1; r_1 ⊙ r_2] \implies l_1 ⊙ r_2\)
¬ More general form: \([P l_1 r_1; Q r_1 r_2; A] \implies C l_1 r_2\)

Examples:

¬ pure transitivity: \([a = b; b = c] \implies a = c\)
calculation = "l_1 \circ r_1"

have "\ldots \circ r_2" [proof]

also \Leftarrow

Anatomy of a [trans] rule:

- Usual form: plain transitivity $[l_1 \circ r_1; r_1 \circ r_2] \Rightarrow l_1 \circ r_2$
- More general form: $[P l_1 r_1; Q r_1 r_2; A] \Rightarrow C l_1 r_2$

Examples:

- pure transitivity: $[a = b; b = c] \Rightarrow a = c$
- mixed: $[a \leq b; b < c] \Rightarrow a < c$
calculation = "l₁ ⊙ r₁"

have "... ⊙ r₂" [proof]

also ⇐

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \([l₁ ⊙ r₁; r₁ ⊙ r₂] \implies l₁ ⊙ r₂\)
→ More general form: \([P \ l₁ \ r₁; Q r₁ r₂; A] \implies C l₁ r₂\)

Examples:

→ pure transitivity: \([a = b; b = c] \implies a = c\)
→ mixed: \([a \leq b; b < c] \implies a < c\)
→ substitution: \([P a; a = b] \implies P b\)
Designing [trans] Rules

calculation = "l₁ ⊕ r₁"
have "... ⊕ r₂" [proof]
also ⟷

Anatomy of a [trans] rule:

→ Usual form: plain transitivity \([l_1 ⊕ r_1; r_1 ⊕ r_2] \implies l_1 ⊕ r_2\)
→ More general form: \([P l_1 r_1; Q r_1 r_2; A] \implies C l_1 r_2\)

Examples:

→ pure transitivity: \([a = b; b = c] \implies a = c\)
→ mixed: \([a ≤ b; b < c] \implies a < c\)
→ substitution: \([P a; a = b] \implies P b\)
→ antisymmetry: \([a < b; b < a] \implies P\)
**DESIGNING [TRANS] RULES**

**calculation** = ""l₁ ⊕ r₁"

**have** "... ⊕ r₂" [proof]

**also** ↔

**Anatomy of a [trans] rule:**

- **Usual form:** plain transitivity \([l₁ ⊕ r₁; r₁ ⊕ r₂] → l₁ ⊕ r₂\)
- **More general form:** \([P l₁ r₁; Q r₁ r₂; A] → C l₁ r₂\)

**Examples:**

- **pure transitivity:** \([a = b; b = c] → a = c\)
- **mixed:** \([a ≤ b; b < c] → a < c\)
- **substitution:** \([P a; a = b] → P b\)
- **antisymmetry:** \([a < b; b < a] → P\)
- **monotonicity:** \([a = f b; b < c; \wedge x y. x < y → f x < f y] → a < f c\)
We have seen today ...

- Recdef
- More induction
- Well founded orders
- Well founded recursion
- Calculations: also/finally
- [trans]-rules
Define a predicate **sorted** over lists

Show that **sorted** (**quicksort** \(xs\)) holds

Look at [http://isabelle.in.tum.de/library/HOL/Wellfounded_Recursion.html](http://isabelle.in.tum.de/library/HOL/Wellfounded_Recursion.html)

Show that in groups, the left-one is also a right-one: \(x \cdot 1 = x\)
(you can use the right_inv lemma from the demo)

Take an algebra textbook and formalize a simple theorem over groups in Isabelle.