Session II

\[ \text{HOL} = \text{Functional programming} + \text{Logic} \]
Proof by Term Rewriting
Term rewriting means . . .

Using equations $l = r$ from left to right as long as possible
Term rewriting means ... 

Using equations $l = r$ from left to right as long as possible

Terminology: equation $\rightsquigarrow$ rewrite rule
Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$
Example

Example:

Equation: \( 0 + n = n \)

Term: \( a + (0 + (b + c)) \)

Result: \( a + (b + c) \)
Example

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

Result: $a + (b + c)$

Rewrite rules can be conditional: $[P_1 \ldots P_n] \implies l = r$
Example

Example:

Equation: $0 + n = n$

Term: $a + (0 + (b + c))$

Result: $a + (b + c)$

Rewrite rules can be conditional: $[P_1 \ldots P_n] \implies l = r$

is used

▶ like $l = r$, but

▶ $P_1, \ldots, P_n$ must be proved by rewriting first.
Simplification in Isabelle

Goal: 1. $[ P_1; \ldots ; P_m ] \rightarrow C$

apply ($simp \ add: eq_1 \ldots eq_n$)
Simplification in Isabelle

Goal: 1. \[ [P_1; \ldots ; P_m] \Longrightarrow C \]

\textbf{apply} (simp \textit{add: eq}_1 \ldots eq_n)

Simplify \(P_1\ldots P_m\) and \(C\) using

- lemmas with attribute \textit{simp}
Simplification in Isabelle

Goal: 1. \([ P_1; \ldots ; P_m ] \implies C\)

**apply**(*simp add: eq_1 \ldots eq_n*)

Simplify \(P_1 \ldots P_m\) and \(C\) using

- lemmas with attribute \(simp\)
- additional lemmas \(eq_1 \ldots eq_n\)
Simplification in Isabelle

**Goal:**
1. \([ P_1; \ldots ; P_m ] \rightarrow C\)

**apply**(*simp add: eq_1 \ldots eq_n*)

Simplify \(P_1 \ldots P_m\) and \(C\) using

- lemmas with attribute \textit{simp}
- additional lemmas \textit{eq_1} \ldots \textit{eq_n}
- assumptions \(P_1 \ldots P_m\)
Simplification in Isabelle

Goal: 1. \[ [ P_1; \ldots ; P_m ] \implies C \]

apply\((\text{simp add: eq}_1 \ldots \text{eq}_n)\)

Simplify \(P_1 \ldots P_m\) and \(C\) using

- lemmas with attribute simp
- additional lemmas \(\text{eq}_1 \ldots \text{eq}_n\)
- assumptions \(P_1 \ldots P_m\)

Variations:

- \((\text{simp} \ldots \text{del:} \ldots)\) removes simp-lemmas
- add and del are optional
Termination

Simplification may not terminate. Isabelle uses simp-rules (almost) blindly from left to right.

Example: \( f(x) = g(x), g(x) = f(x) \)
Termination

Simplification may not terminate. Isabelle uses \texttt{simp}-rules (almost) blindly from left to right.

Example: \( f(x) = g(x), g(x) = f(x) \)

\[
\begin{align*}
[P_1 \ldots P_n] \implies l = r
\end{align*}
\]

is suitable as a \texttt{simp}-rule only if \( l \) is “bigger” than \( r \) and each \( P_i \)
Termination

Simplification may not terminate. Isabelle uses simp-rules (almost) blindly from left to right.

Example: \( f(x) = g(x), g(x) = f(x) \)

\[
[P_1 \ldots P_n] \implies l = r
\]

is suitable as a simp-rule only if \( l \) is “bigger” than \( r \) and each \( P_i \)

\[
n < m \implies (n < Suc m) = True
\]
\[
Suc n < m \implies (n < m) = True
\]
Simplification may not terminate. Isabelle uses simp-rules (almost) blindly from left to right.

Example: \( f(x) = g(x), g(x) = f(x) \)

\[
\begin{align*}
[P_1 \ldots P_n] &\implies l = r \\
\text{is suitable as a simp-rule only if } l &\text{ is “bigger” than } r \text{ and each } P_i \\
n < m &\implies (n < Suc \, m) = True \quad \text{YES} \\
Suc \, n < m &\implies (n < m) = True \quad \text{NO}
\end{align*}
\]
How to ignore assumptions

Assumptions sometimes cause problems, e.g. nontermination. How to exclude them from \textit{simp}:

\begin{itemize}
  \item \texttt{apply(simp (no_asm_simp) \ldots)}
    
    Simplify only conclusion
  \item \texttt{apply(simp (no_asm_use) \ldots)}
    
    Simplify but do not use assumptions
  \item \texttt{apply(simp (no_asm) \ldots)}
    
    Ignore assumptions completely
\end{itemize}
Tracing

Set trace mode on/off in Proof General:

Isabelle/Isar → Settings → Trace simplifier

Output in separate buffer:

Proof-General → Buffers → Trace
auto

- *auto* acts on all subgoals
- *simp* acts only on subgoal 1
- *auto* applies *simp* and more
Demo: simp
Type definitions in Isabelle/HOL

Keywords:

- **typedecl**: pure declaration (session 1)
- **types**: abbreviation
- **datatype**: recursive datatype
**types**

`types name = \tau`  

Introduces an *abbreviation* `name` for type `\tau`  

Examples:

```plaintext
  types
  name = string
  (\'a,\'b)foo = "\'a list \times \'b list"
```
**types**

`types name = τ`

Introduces an *abbreviation* `name` for type `τ`

Examples:

```plaintext
  types
  name = string
  ('a,'b)foo = ""a list × 'b list"
```

Type abbreviations are expanded after parsing.
Not present in internal representation and Isabelle output.
datatype 

\[
\text{datatype } \ 'a \ \text{list} = \text{Nil} \mid \text{Cons} \ 'a \ ''a \ \text{list} \n\]

I

Types:

Nil :: 'a list

Cons :: 'a 

Distinctness:

Nil \neq \text{Cons} \ x \ \text{xs}

Injectivity:

(\text{Cons} \ x \ \text{xs} = \text{Cons} \ y \ \text{ys}) \leftrightarrow (x = y \land \text{xs} = \text{ys})
datatype 'a list = Nil | Cons 'a "'a list"

Properties:

- **Types:**  
  - Nil :: 'a list  
  - Cons :: 'a ⇒ 'a list ⇒ 'a list

- **Distinctness:** Nil ≠ Cons x xs

- **Injectivity:** (Cons x xs = Cons y ys) = (x = y ∧ xs = ys)
case

Every datatype introduces a case construct, e.g.

$\text{(case } xs \text{ of } \text{Nil } \Rightarrow \ldots \mid \text{Cons } y \text{ ys } \Rightarrow \ldots y \ldots \text{ys} \ldots)$

- one case per constructor
- no nested patterns ($\text{Cons } x \text{ (Cons } y \text{ zs})$)
- but nested cases
Every datatype introduces a case construct, e.g.

\[
\text{(case } \text{xs of } \text{Nil } \Rightarrow \ldots \mid \text{Cons } y \text{ ys } \Rightarrow \ldots y \ldots \text{ys } \ldots)\]

- one case per constructor
- no nested patterns \((\text{Cons } x (\text{Cons } y \text{ zs}))\)
- but nested cases

apply(\text{case_tac } \text{xs}) \Rightarrow \text{one subgoal for each constructor}

\[
\begin{align*}
\text{xs} = \text{Nil} & \Rightarrow \ldots \\
\text{xs} = \text{Cons } a \text{ list} & \Rightarrow \ldots
\end{align*}
\]
Function definition schemas in Isabelle/HOL

- Non-recursive with `constdefs` (session 1)
  No problem

- Primitive-recursive with `primrec`
  Terminating by construction

- Well-founded recursion with `recdef`
  User must (help to) prove termination
consts app :: "'a list ⇒ 'a list ⇒ 'a list"
primrec
"app Nil ys = ys"
"app (Cons x xs) ys = Cons x (app xs ys)"
\textbf{primrec}

\texttt{consts app :: } "'a list \Rightarrow 'a list \Rightarrow 'a list"
\texttt{primrec}
\texttt{"app Nil ys = ys"}
\texttt{"app (Cons x xs) ys = Cons x (app xs ys)"}

- Each recursive call \textbf{structurally smaller} than lhs.
consts \textit{app} :: "'a list \Rightarrow 'a list \Rightarrow 'a list"

\begin{itemize}
  \item Each recursive call \textit{structurally smaller} than lhs.
  \item Equations used automatically in simplifier
\end{itemize}
Structural induction

$P\;xs$ holds for all lists $xs$ if

- $P\;Nil$
- and for arbitrary $x$ and $xs$, $P\;xs$ implies $P\;(Cons\;x\;xs)$
Structural induction

\( P \, xs \) holds for all lists \( xs \) if

- \( P \, \text{Nil} \)
- and for arbitrary \( x \) and \( xs \), \( P \, xs \) implies \( P \, (\text{Cons} \, x \, xs) \)

Induction theorem \textit{list.induct}:

\[
\left[ P \, \text{Nil}; \bigwedge \text{a list.} \ P \, \text{list} \implies P \, (\text{Cons} \, \text{a list}) \right] \\
\implies P \, \text{list}
\]
**Structural induction**

\( P \; x s \) holds for all lists \( x s \) if

- \( P \; Nil \)
- and for arbitrary \( x \) and \( x s \), \( P \; x s \) implies \( P \; (Cons \; x \; x s) \)

Induction theorem \( \text{list.induct} \):

\[
\begin{align*}
P \; Nil; \bigwedge a \; \text{list.} \; P \; \text{list} & \implies P \; (\text{Cons} \; a \; \text{list}) \\
\implies P \; \text{list}
\end{align*}
\]

- General proof method for induction: \( \text{(induct} \; x \text{)} \)
  - \( x \) must be a free variable in the first subgoal.
  - The type of \( x \) must be a datatype.
Theorems about recursive functions proved by induction

\textbf{consts} \textit{itrev} :: ’a list \rightarrow ’a list \rightarrow ’a list
\textbf{primrec}
\begin{align*}
\textit{itrev} \; [] \; ys &= ys \\
\textit{itrev} \; (x\#xs) \; ys &= \textit{itrev} \; xs \; (x\#ys)
\end{align*}

\textbf{lemma} \textit{itrev} \; xs \; [] &= \textit{rev} \; xs
Demo: proof attempt
Generalisation

Replace constants by variables

lemma \( \text{itrev} \; xs \; ys = \text{rev} \; xs \; @ \; ys \)
Generalisation

Replace constants by variables

\textbf{lemma} \quad \textit{itrev} \; \textit{xs} \; \textit{ys} = \textit{rev} \; \textit{xs} @ \; \textit{ys}

Quantify free variables by \forall
(except the induction variable)

\textbf{lemma} \quad \forall \; \textit{ys}. \quad \textit{itrev} \; \textit{xs} \; \textit{ys} = \textit{rev} \; \textit{xs} @ \; \textit{ys}
Function definition schemas in Isabelle/HOL

- Non-recursive with `constdefs` (session 1)
  No problem

- Primitive-recursive with `primrec`
  Terminating by construction

- Well-founded recursion with `reccdef`
  User must (help to) prove termination
recdef — examples

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"
recdef — examples

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
  \(\text{measure}\) sufficient for most cases
recdef

- 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.
Demo: recdef and induction
Sets
Notation

Type ’a set: sets over type ’a

- \{\}, \{e_1, \ldots, e_n\}, \{x. P\ x\}
- e \in A, \quad A \subseteq B
- A \cup B, \quad A \cap B, \quad A - B, \quad - A
- \bigcup_{x \in A} B\ x, \quad \bigcap_{x \in A} B\ x
- \{i..j\}
- insert :: ’a \Rightarrow ’a set \Rightarrow ’a set
- f ‘ A \equiv \{y. \ \exists\ x \in A. \ y = f\ x\}
- \ldots
Inductively defined sets: even numbers

Informally:

- 0 is even
- If $n$ is even, so is $n + 2$
- These are the only even numbers
Inductively defined sets: even numbers

Informally:

- 0 is even
- If $n$ is even, so is $n + 2$
- These are the only even numbers

In Isabelle/HOL:

```isar
cconsts Ev :: nat set
inductive Ev
intros
  0 ∈ Ev
  n ∈ Ev ⟹ n + 2 ∈ Ev
```

— The set of all even numbers
Rule induction for Ev

To prove

\[ n \in Ev \iff P\ n \]

by rule induction on \( n \in Ev \) we must prove
Rule induction for Ev

To prove

\[ n \in Ev \iff P n \]

by *rule induction* on \( n \in Ev \) we must prove

- \( P 0 \)
Rule induction for Ev

To prove

\[ n \in Ev \implies P n \]

by *rule induction* on \( n \in Ev \) we must prove

- \( P 0 \)
- \( P n \implies P(n+2) \)
Rule induction for Ev

To prove

\[ n \in Ev \implies P n \]

by *rule induction* on \( n \in Ev \) we must prove

- \( P 0 \)
- \( P n \implies P(n+2) \)

**Rule** \( Ev\text{-}induct \):

\[
\left[ n \in Ev; P 0; \bigwedge n. P n \implies P(n+2) \right] \implies P n
\]
To prove

\[ n \in Ev \implies P n \]

by rule induction on \( n \in Ev \) we must prove

- \( P 0 \)
- \( P n \implies P(n+2) \)

Rule \textbf{Ev.induct}:

\[
\left[ \begin{array}{l}
 n \in Ev; P 0; \bigwedge n. P n \implies P(n+2)
\end{array} \right] \implies P n
\]

An elimination rule
Demo: inductively defined sets
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!
A typical Isar proof

proof

  assume \( formula_0 \)
  have \( formula_1 \) by simp
  
  
  have \( formula_n \) by blast

show \( formula_{n+1} \) by 

qed
A typical Isar proof

proof

  assume \( \text{formula}_0 \)
  have \( \text{formula}_1 \)  by \text{simp}
  \\
  \\
  have \( \text{formula}_n \)  by \text{blast}
  show \( \text{formula}_{n+1} \) by \ldots

qed

proves \( \text{formula}_0 \implies \text{formula}_{n+1} \)
proof = proof [method] statement* qed
| by method
Isar core syntax

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

method = (simp ...) | (blast ...) | (rule ...) | ...
**Isar core syntax**

\[
\text{proof} = \text{proof} \ [\text{method}] \ \text{statement}^* \ \text{qed} \\
\quad \mid \ \text{by} \ \text{method}
\]

\[
\text{method} = (\text{simp} \ldots) \mid (\text{blast} \ldots) \mid (\text{rule} \ldots) \mid \ldots
\]

\[
\text{statement} = \text{fix} \ \text{variables} \ \ (\wedge) \\
\quad \mid \ \text{assume} \ \text{proposition} \ \ (\implies) \\
\quad \mid [\text{from} \ \text{name}^+] \ (\text{have} \mid \text{show}) \ \text{proposition} \ \text{proof} \\
\quad \mid \ \text{next} \ \ (\text{separates subgoals})
\]
Isar core syntax

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

method = (simp \ldots) | (blast \ldots) | (rule \ldots) | \ldots

statement = \textbf{fix} \ variables \quad (\wedge)
             \quad | \quad \textbf{assume} \ proposition \quad (\rightarrow)
             \quad | \quad [\textbf{from} \ name^+] (\textbf{have} | \textbf{show}) \ proposition \ proof
             \quad | \quad \textbf{next} \quad \text{ (separates subgoals)}

proposition = [name:] formula
Demo: propositional logic
Elimination rules / forward reasoning

- Elim rules are triggered by facts fed into a proof:
  
  \[
  \text{from } \neg a \text{ have } \text{formula proof}
  \]
Elimination rules / forward reasoning

- Elim rules are triggered by facts fed into a proof:
  from $\neg a$ have $\text{formula proof}$
- from $\neg a$ have $\text{formula proof}$ ($\text{rule rule}$)
  $\neg a$ must prove the first $n$ premises of $\text{rule}$
in the right order
the others are left as new subgoals
Elimination rules / forward reasoning

- Elim rules are triggered by facts fed into a proof:
  \( \text{from } \lnot a \text{ have } f \text{ormula } p \text{roof} \)

- \( \text{from } \lnot a \text{ have } f \text{ormula } p \text{roof } (\text{rule } \text{rule}) \)

  \( \lnot a \) must prove the first \( n \) premises of \( \text{rule} \)
in the right order
the others are left as new subgoals

- \( \text{proof alone abbreviates } \text{proof } \text{rule} \)

- \( \text{rule} \): tries elim rules first
  (if there are incoming facts \( \lnot a \)!)

IJCAR 2004, Tutorial T4 – p.35
Practical Session II

Theorem proving and sanity; Oh, my! What a delicate balance.

—Victor Carreno