Session II

HOL = Functional programming + 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 $\leadsto$ 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:

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$(simp add: eq_1 \ldots eq_n)$
Goal: 1. \[[ P_1; \ldots ; P_m \]\] \implies C

\textbf{apply} (\textit{simp 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. \([ P_1; \ldots ; P_m ] \implies C\)

\textbf{apply}(simp add: eq_1 \ldots eq_n)

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

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

Variations:

- \((simp \ldots del: \ldots)\) removes \(simp\)-lemmas
- \(add\) and \(del\) are optional
Simplification may not terminate. Isabelle uses \textit{simp}-rules (almost) blindly from left to right.

Example: $f(x) = g(x), g(x) = f(x)$
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 \( P_i \).
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 \( P_i \)

\[
\begin{align*}
n < m & \implies (n < Suc m) = True \\
Suc n < m & \implies (n < m) = True
\end{align*}
\]
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 \( P_i \)

\[
n < m \implies (n < Suc m) = True \quad \text{YES}
\]
\[
Suc n < m \implies (n < m) = True \quad \text{NO}
\]
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)}
  \begin{itemize}
  \item Simplify only conclusion
  \end{itemize}
\item \texttt{apply(simp (no_asm_use) \ldots)}
  \begin{itemize}
  \item Simplify but do not use assumptions
  \end{itemize}
\item \texttt{apply(simp (no_asm) \ldots)}
  \begin{itemize}
  \item Ignore assumptions completely
  \end{itemize}
\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
Keywords:

- \texttt{typedefcl}: pure declaration \hspace{1cm} (session 1)
- \texttt{types}: abbreviation
- \texttt{datatype}: recursive datatype
**types**

**types** name = \( \tau \)

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

Examples:

**types**

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

**types** name = τ

Introduces an *abbreviation* name for type τ

Examples:

**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 'a list = Nil | Cons 'a ""a list"
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.

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

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

\textbf{apply}(\text{case\_tac} \ xs) \Rightarrow \text{one subgoal for each constructor}

\[
xs = \text{Nil} \ \Rightarrow \ldots
\]
\[
xs = \text{Cons } a \text{ list} \ \Rightarrow \ldots
\]
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)"
consts app :: "'a list ⇒ 'a list ⇒ 'a list"
primrec
"app Nil ys = ys"
"app (Cons x xs) ys = Cons x (app xs ys)"

▶ Each recursive call structurally smaller than lhs.
consts \( \text{app} :: \text{"a list} \Rightarrow \text{'}a\text{ list} \Rightarrow \text{'}a\text{ list}" \)

primrec

"\text{app \text{Nil} } ys = ys"

"\text{app \text{(Cons} x \text{ xs) } } ys = \text{Cons} x \text{ (app} \text{ xs} \text{ ys)}""

- Each recursive call \textbf{structurally smaller} than lhs.
- Equations used automatically in simplifier
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

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

Induction theorem list.induct:

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

$P \, \text{xs}$ holds for all lists $\text{xs}$ if

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

Induction theorem list.induct:

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

General proof method for induction: (induct $x$)

- $x$ must be a free variable in the first subgoal.
- The type of $x$ must be a datatype.
Induction heuristics

Theorems about recursive functions proved by induction

consts \( itrev :: 'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ list} \)

primrec

\( \text{itrev \;[]} \quad \text{ys} = \text{ys} \)
\( \text{itrev \;(x#xs) \;ys} = \text{itrev \;xs \;(x#ys)} \)

lemma \( \text{itrev \;xs \;[]} = \text{rev \;xs} \)
Demo: proof attempt
Generalisation

Replace constants by variables

\textbf{lemma} \ itrev \ xs \ ys = \ rev \ xs \ @ \ ys
Generalisation

Replace constants by variables

\textbf{lemma} \quad \textit{itrev} \; \textit{x}s \; \textit{y}s = \textit{rev} \; \textit{x}s \; \texttt{@} \; \textit{y}s

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

\textbf{lemma} \quad \forall \; \textit{y}s. \; \textit{itrev} \; \textit{x}s \; \textit{y}s = \textit{rev} \; \textit{x}s \; \texttt{@} \; \textit{y}s
Function definition schemas in Isabelle/HOL

► Non-recursive with \texttt{constdefs} (session 1)
   No problem

► Primitive-recursive with \texttt{primrec}
   Terminating by construction

► Well-founded recursion with \texttt{recdef}
   User must (help to) prove termination
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))"
recdef

The definition:

- one parameter
- free pattern matching, order of rules important
- termination relation
  \((\textit{measure} \textit{sufficient for most cases})\)
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\}
- \text{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
consts Ev :: nat set — The set of all even numbers
inductive Ev
intros
  0 ∈ Ev
  $n$ ∈ Ev ⇒ $n + 2$ ∈ Ev
```
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

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

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

\begin{itemize}
\item \( P\ 0 \)
\item \( P\ n \implies P(n+2) \)
\end{itemize}

Rule \text{Ev.induct}:

\[ \left[ n \in Ev; P\ 0; \bigwedge n. P\ n \implies P(n+2) \right] \implies P\ n \]
Rule induction for Ev

To prove

\[ n \in \text{Ev} \implies P(n) \]

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

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

Rule \text{Ev.induct}:

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

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 \ldots

qed
A typical Isar proof

proof
  assume $formula_0$
  have $formula_1$ by simp
  
  have $formula_n$ by blast
  show $formula_{n+1}$ by \ldots
  
  qed

proves $formula_0 \Longrightarrow formula_{n+1}$
Isar core syntax

\[
\text{proof} = \text{proof} \ [\text{method}] \ \text{statement}^* \ \text{qed} \\
\text{by} \ \text{method}
\]
Isar core syntax

\[
\text{proof} = \text{proof} \ [\text{method}] \ \text{statement}^{*} \ \text{qed} \\
\quad | \ \text{by} \ \text{method} \\
\text{method} = (\text{simp} \ \ldots) \ | \ (\text{blast} \ \ldots) \ | \ (\text{rule} \ \ldots) \ | \ \ldots
\]
Isar core syntax

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

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

statement = fix variables (\forall)
        | assume proposition (\Rightarrow)
        | [from name+] (have | show) proposition proof
        | next (separates subgoals)
Isar core syntax

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

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

statement = fix variables (\wedge)
    | assume proposition (\Rightarrow)
    | [from name+] (have | show) proposition proof
    | next (separates subgoals)

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

- Elim rules are triggered by facts fed into a proof:
  from \( \vec{a} \) have \( \text{formula} \) proof
Elimination rules / forward reasoning

- Elim rules are triggered by facts fed into a proof:
  from \( \vec{a} \) have formula proof

- from \( \vec{a} \) have formula proof (rule rule)

  \( \vec{a} \) must prove the first \( n \) premises of 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:
  \[ \text{from } \vec{a} \text{ have formula proof} \]

- \[ \text{from } \vec{a} \text{ have formula proof (rule rule)} \]
  - \( \vec{a} \) must prove the first \( n \) premises of \( \text{rule} \)
  - in the right order
  - the others are left as new subgoals

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

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

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

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

— Victor Carreno