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

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: $[\![P_1 \dots P_n]\!] \Longrightarrow l = r$

Example

Example:

Equation: $0 + n = n$

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

Result: $a + (b + c)$

Rewrite rules can be conditional: $[\![P_1 \ldots P_n]\!] \Longrightarrow l = r$ is used

- like $l = r$, but
- $P_1, \ldots, P_n$ must be proved by rewriting first.

Simplification in Isabelle

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

apply(*simp add: eq*$_1$ $\ldots$ *eq*$_n$)

Simplification in Isabelle

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

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

Simplify $P_1 \ldots P_m$ and C using

- lemmas with attribute *simp*

Simplification in Isabelle

Goal: *1.* $[\![$ *P*$_1$; $\ldots$; *P*$_m$ $]\!] \Longrightarrow$ *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*$_m$ $]\!] \Longrightarrow$ *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.* $[\![P_1; \ldots ; P_m]\!] \Longrightarrow C$

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

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 *simp*-rules (almost) blindly from left to right.

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

$$\llbracket P_1 \ldots P_n \rrbracket \Longrightarrow l = r$$

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: $f(x) = g(x), g(x) = f(x)$

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

n < m $\Longrightarrow$ *(n < Suc m) = True*
Suc n < m $\Longrightarrow$ *(n < m) = True*

Termination

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

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

$$[\![P_1 \dots P_n]\!] \Longrightarrow l = r$$

is suitable as a *simp*-rule only
if l is "bigger" than r and each P_i

n < *m* $\Longrightarrow$ (*n* < *Suc m*) = *True* YES
Suc n < *m* $\Longrightarrow$ (*n* < *m*) = *True* NO

How to ignore assumptions

Assumptions sometimes cause problems, e.g. nontermination. How to exclude them from *simp*:

apply*(simp (no_asm_simp) ...)*
Simplify only conclusion

apply*(simp (no_asm_use) ...)*
Simplify but do not use assumptions

apply*(simp (no_asm) ...)*
Ignore assumptions completely

Tracing

Set trace mode on/off in Proof General:

Isabelle/Isar $\rightarrow$ Settings $\rightarrow$ Trace simplifier

Output in separate buffer:

Proof-General $\rightarrow$ Buffers $\rightarrow$ 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 τ

Examples:

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

types

types $name = \tau$

Introduces an *abbreviation* $name$ for type τ

Examples:

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

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

datatype

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

datatype

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

Properties:

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

  Cons :: *'a* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*
- Distinctness: *Nil* $\neq$ *Cons x xs*
- Injectivity: *(Cons x xs = Cons y ys) = (x = y* $\wedge$ *xs = ys)*

case

Every datatype introduces a *case* construct, e.g.

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

- one case per constructor
- no nested patterns (*Cons x (Cons y zs)*)
- but nested cases

case

Every datatype introduces a *case* construct, e.g.

(case xs of Nil $\Rightarrow \ldots$ | *Cons y ys* $\Rightarrow$ *... y ... ys ...)*

- one case per constructor
- no nested patterns (*Cons x (Cons y zs)*)
- but nested cases

apply*(case_tac xs)* $\Rightarrow$ one subgoal for each constructor

xs = *Nil* $\Longrightarrow \ldots$
xs = *Cons a list* $\Longrightarrow \ldots$

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

primrec

consts *app :: "'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list"*
primrec
"app Nil ys = ys"
"app (Cons x xs) ys = Cons x (app xs ys)"

primrec

consts *app :: "'a list $\Rightarrow$ 'a list $\Rightarrow$ 'a list"*
primrec
"app Nil ys = ys"
"app (Cons x xs) ys = Cons x (app xs ys)"

- Each recursive call structurally smaller than lhs.

primrec

consts *app :: "'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list"*
primrec
"app Nil ys = ys"
"app (Cons x xs) ys = Cons x (app xs ys)"

- Each recursive call 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:

$$\llbracket P\ Nil;\ \bigwedge a\ list.\ P\ list \Longrightarrow P\ (Cons\ a\ list) \rrbracket$$

$$\Longrightarrow P\ list$$

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:

  $$\llbracket P\ Nil;\ \bigwedge a\ list.\ P\ list \Longrightarrow P\ (Cons\ a\ list)\rrbracket$$

  $$\Longrightarrow P\ list$$

- 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 list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*
primrec
itrev [] ys = ys
itrev (x#xs) ys = itrev xs (x#ys)

lemma *itrev xs [] = rev xs*

Demo: proof attempt

Generalisation

Replace constants by variables

lemma *itrev xs ys = rev xs @ ys*

Generalisation

Replace constants by variables

lemma *itrev xs ys = rev xs @ ys*

Quantify free variables by ∀
(except the induction variable)

lemma ∀ *ys. itrev xs ys = rev xs @ 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 **recdef**
  User must (help to) prove termination

recdef — examples

consts *sep :: "'a $\times$ 'a list $\Rightarrow$ '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 definiton:
  - one parameter
  - free pattern matching, order of rules important
  - termination relation
    (*measure* sufficient for most cases)

recdef

- The definiton:
  - 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

recdef

- The definiton:
  - 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$_1$,...,e$_n$}*, *{x. P x}*
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, *A - B*, *- A*
- $\bigcup_{x \in A} B\ x$, $\bigcap_{x \in A} B\ x$
- *{i..j}*
- *insert :: 'a* $\Rightarrow$ *'a set* $\Rightarrow$ *'a set*
- *f ` A* $\equiv$ *{y.* $\exists x \in A$*. y = f x}*
- ...

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:

consts *Ev :: nat set* — The set of all even numbers

inductive *Ev*

intros

$0 \in Ev$

$n \in Ev \Longrightarrow n + 2 \in Ev$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

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

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

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

- $P\ 0$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

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

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

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

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule `Ev.induct`:

$$\llbracket\ n \in Ev;\ P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P(n+2)\ \rrbracket \Longrightarrow P\ n$$

Rule induction for Ev

To prove

$$n \in Ev \Longrightarrow P\ n$$

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

- $P\ 0$
- $P\ n \Longrightarrow P(n+2)$

Rule `Ev.induct`:

$$[\![\ n \in Ev;\ P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P(n+2)\ ]\!] \Longrightarrow P\ n$$

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

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

Isar core syntax

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

$\quad\quad\quad$ | **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 $\qquad (\bigwedge)$

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

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

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

Isar core syntax

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

  | **by** method

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

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

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

  | [**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** $formula$ **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:
  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
- **proof** alone abbreviates **proof** *rule*
- *rule*: tries elim rules first
  (if there are incoming facts $\vec{a}$!)

Practical Session II

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

—Victor Carreno