Preuves Interactives et Applications

Burkhart Wolff

www.lri.fr/~wolff/teach-material/2020-2021/M2-CSMR/index.html

Université Paris-Saclay

Automated Proof Techniques in Isabelle/HOL: An Introduction

Revisions

- Elementary apply-style (backward) proofs
- Elementary attributed (forward) proofs
- Advanced apply-style proof techniques

Introduction to more Advanced Proof Techniques

- Induction and case-splitting
- Rewriting
- Tableaux provers
- Paramodulation prover
- Presburger arithmetics prover
- A magic device: sledgehammer

Revision: Proof Commands

- Simple (Backward) Proofs:

```
lemma  <thmname> :
  [ <contextelem>+ shows ]"<φ>"
  <proof>
```

  – where <contextelem> declare elements of a proof context Γ (list of assumptions)
  – where <proof> are
    – high-level proof method by(simp), by(auto), by(metis), by(arith) or the ellipses sorry and oops
    – apply-style ("imperative") proofs, and
    – structured ("declarative") proofs.

Revision: Proof Commands

- Core of structured proofs:

```
proof (<method>)
  [case - fix - assumes - defs- have-]
  show "<goal>" <proof>
next
   ...
next
   [case - fix - assumes - defs- have-]
  show "<goal>" <proof>
qed
```

- … a switch from procedural to declarative style can be done by rephrasing the goals

A Summary of Proof Methods

- low-level procedures and versions with explicit substitution:
  - assumption
  - rule_tac <subst> in <thmname>
  - erule_tac <subst> in <thmname>
  - drule_tac <subst> in <thmname>

- … where <subst> is of the form:

  x_1="φ_1" and x_n="φ_n

A Summary of Proof Methods

low-level methods:

- assumption (unifies conclusion vs. a premise)
- subst [(asm)] <thmname>

  does one rewrite-step

  (by instantiating the HOL subst-rule)

- rule <thmname>, rule_tac <subst> in <thmname>

  PROLOG - like resolution step using HO-Unification

- erule <thmname>, erule_tac <subst> in <thmname>

  elimination resolution (for ND elimination rules)

- drule <thmname>, drule_tac <subst> in <thmname>,

  destruction resolution (for ND destriction rules)

A Summary of Proof Methods

- Local forward proof constructions by attributes

  - <thm>[THEN <thm>]   (unifies conclusion vs. premise)
  - <thm>[OF <thm>]   (unifies premise vs. conclusion)
  - <thm>[symmetric]   (flips an equation)
  - <thm>[of (<term> | _)*] (instantiates variables)
  - <thm>[simp]   (simplifies a thm)
  - <thm>[simp only: <thm>]   (simplifies a thm)

A Summary of Proof Methods

- advanced methods:
  - insert <thmname>, insert <thmname>[„[„ of <subst>“]“]

    inserts local and global facts into assumptions

  - induct_tac “ϕ”, induct “ϕ” [arbitrary : „<variable>“]

    searches for appropriate induction scheme using type information and instantiates it

  - case_tac “ϕ”, cases “ϕ”,

    searches for appropriate case splitting scheme using type information and instantiates it

Rewriting

The Simplifier

Supports Rewriting, in particular:

- Regular rewriting
- Rewriting of HO-Patterns,
- Ordered Rewriting
- Conditional Rewriting
- Context - Rewriting
- Automatic Case-Splitting

INSTRUMENTATION NECESSARY, so it is necessary to tell which rule should be used HOW.
Simplification is quite predictable,
using[[simp_trace]] shuts on tracing of the rewriter

The Simplifier

Regular Rewriting:

- Left-right of rewriting of rules of the form:

$$c\ t_1\ \ldots\ t_n = e$$

where $c\ t_1\ \ldots\ t_n$ is the pattern ($c \in C$), which linear (all free variables distinct) and

$$FV(t_1) \cup \ldots FV(t_n) \supseteq FV(e)$$

```
apply(simp add: <thm>)
```

The Simplifier

Regular Rewriting: Examples.

Suc(x + y) = x + Suc(y)

(a # A) @ B) = a # (A @ B)

… (many computational rules
resulting from “fun” or “primrec”)

True $\wedge$ X = X

(a + b) + c = a + (b + c)

if True then b else c = b

…

The Simplifier

Higher-order Patterns:

- constant head, i.e. of the form $c\ t_1 \ldots t_n$
- linear in free variables, $FV(t_1) \cup \ldots FV(t_n) \supseteq FV(e)$
- λ-expressions !
- All Higher-Order Variables occur only in the form:

$$F(x_1 \ldots x_n) \text{ for distinct } x_i$$

Example:

$$\forall(\lambda\ x.\ P(x) \wedge Q(x)) = \forall(\lambda\ x.\ P(x)) \wedge (\forall(\lambda\ x.Q(x))$$

The Simplifier

Supports Ordered Rewriting:

- There is an implicit wf-ordering on terms. Rewriting is only done if the re-written term is smaller.

  Example commutativity: a+b = b+a

  With a little trickery, one can have ACI rewriting:

disj_comms(2):	$(P \vee Q \vee R) = (Q \vee P \vee R)$
disj_comms(1):	$(P \vee Q) = (Q \vee P)$
disj_ac(3):	$((P \vee Q) \vee R) = (P \vee Q \vee R)$
disj_ac(2):	$(P \vee Q \vee R) = (Q \vee P \vee R)$
disj_ac(1):	$(P \vee Q) = (Q \vee P)$
disj_absorb:	$(A \vee A) = A$
disj_left_absorb:	$(A \vee A \vee B) = (A \vee B)$

The Simplifier

Supports Rewriting, in particular:

- Conditional Rewriting

if_P: $P \Longrightarrow (\text{if } P \text{ then } x \text{ else } y) = x$

if_not_P: $\neg P \Longrightarrow (\text{if } P \text{ then } x \text{ else } y) = y$

```
apply(simp add: if_P if_not_P)
```

(Not necessary, somewhere in the library it is stated:

```
declare if_P [simp] if_not_P [simp] )
```

...)

The Simplifier

Supports Rewriting, in particular:

- Context - Rewriting

HOL.if_cong:

$$b = c \Longrightarrow$$

$$(c \Longrightarrow x = u) \Longrightarrow$$

$$(\neg\, c \Longrightarrow y = v) \Longrightarrow$$

$$(\text{if } b \text{ then } x \text{ else } y) = (\text{if } c \text{ then } u \text{ else } v)$$

HOL.conj_cong:

$$P = P' \Longrightarrow (P' \Longrightarrow Q = Q') \Longrightarrow (P \wedge Q) = (P' \wedge Q')$$

```
apply(simp  cong: if_cong)
```

The Simplifier

Supports Rewriting, in particular:

- Automatic Case-Splitting

  (by a new type of rule which is NOT constant head)

  split_if_asm: P (if Q then x else y) = (¬ (Q ∧ ¬ P x ∨ ¬ Q ∧ ¬ P y))

  split_if: P (if Q then x else y) = ((Q ⟶ P x) ∧ (¬ Q ⟶ P y))

For any data type (example: Option):

Option.option.split_asm:
P (case x of None ⇒ f1 | Some x ⇒ f2 x) =
(¬ (x = None ∧ ¬ P f1 ∨ (∃a. x = Some a ∧ ¬ P (f2 a))))

Option.option.split:
P (case x of None ⇒ f1 | Some x ⇒ f2 x) =
((x = None ⟶ P f1) ∧ (∀a. x = Some a ⟶ P (f2 a)))

apply(simp split: split_if_asm split_if)

Tableaux Prover

fast, blast and auto

Tableaux Provers going back to LeanTAP

- For Logic terms and Set terms
- Uses all rules classified as
  - introduction rule   (keyword: intro)
    works on conclusion of a goal
  - elimination rule   (keyword: elim)
    works on assumptions of a goal
  - destruction drule (keyword:: dest)
    works on assumptions of a goal
    applies destructively (eg. modus ponens)
  - frule works on assumptions of a goal,
    applies non-destructively

fast, blast and auto

fast

- will apply safe intro/elim/drule's blindly
  (these are rules like conjI, conjE, disjE, ... allI, exE, ...
  Rules that will transform a subgoal into an equivalent
  one, without loosing "logical content")
- with backtrack on unsafe rules
  (refines a subgoal into a logically stronger one,
  can lead into a dead end).
  fast works for HO-Terms, but is fairly slow slow

blast

- dito, but resticted to first-order reasoning

auto

- intertwines simp and blast

fast, blast and auto

blast

- works similarly like fast, but is resticted to first-order reasoning
- Substantially faster than fast, can treat transitivity rules.

auto

- intertwines simp, blast, and fast

A Summary of Proof Methods

- advanced automated procedures:
  - simp **[**add: <thmname>+**]** **[**del: <thmname>+**]**
    [split: <thmname>+**]** **[**cong: <thmname>+**]**
  - auto **[**simp: <thmname>+**]**
    [intro: <thmname>+**]** **[**intro [!]: <thmname>+**]**
    [dest: <thmname>+**]** **[**dest [!]: <thmname>+**]**
    [elim: <thmname>+**]** **[**elim[!]: <thmname>+**]**

Paramodulation Prover

A Summary of Proof Methods

- another automated procedures based on ordered paramodulation calculus
  (Canonical ref: http://www.gilith.com/papers/metis.pdf)

  - metis <thmname>+

$$\frac{}{A_1 \vee \cdots \vee A_n}\text{AXIOM } [A_1, \ldots, A_n] \qquad \frac{}{L \vee \neg L}\text{ASSUME } L$$

$$\frac{A_1 \vee \cdots \vee A_n}{A_1[\sigma] \vee \cdots \vee A_n[\sigma]}\text{INST } \sigma \qquad \frac{A_1 \vee \cdots \vee A_n}{A_{i_1} \vee \cdots \vee A_{i_m}}\text{FACTOR}$$

$$\frac{A_1 \vee \cdots \vee L \vee \cdots \vee A_m \qquad B_1 \vee \cdots \vee \neg L \vee \cdots \vee B_n}{A_1 \vee \cdots \vee A_m \vee B_1 \vee \cdots \vee B_n}\text{RESOLVE } L$$

Linear Arithmetic Prover

A Summary of Proof Methods

- advanced automated procedures based on Coopers Algorithm for linear Presburger Arithmetics.

(Chaieb, Nipkow. Proof Synthesis and Reflection for Linear Arithmetic. J. Automated Reasoning, 41:33-59, 2008)

- arith

The Sledgehammer Interface (external provers)

Magic Device:

- sledgehammer - command.
  - asks well-known automatic first-order theorem provers such as
    - Vampire (binary resolution and superposition)
    - E (FOL-Eq saturation prover)
    - CVC4 (SMT prover)
    - Z3 (SMT prover)

  … if they can construct a proof based on all Isabelle theorems existing at this point, reconstructs an Isabelle proof.

  - does not work for proofs involving (deep) HO-Reasoning and/or induction.

Conclusion

- Isabelle focusses on interactive proofs (enabling presentation of intermediate steps, and structuring of proofs and prover instrumentations)
- … but this does not mean that there are no automatic proof techniques available and that classical ATP's are "better" in that sense …
- Highly-tuned (=competition) ATPs can be faster, though, due to more aggressive compilations