universität innsbruck

Summer Term 2024

Interactive Theorem Proving using Isabelle/HOL

Session 6

René Thiemann

Department of Computer Science

Outline

Projects

Projects

- 1–3 person projects
- for many person projects individual contributions have to be clarified
- all projects can be started quite soonish
  (lacking knowledge for some projects: inductive definitions and sets)
- evaluation rules: website
- project topics (details: website)
  - Congruence Closure (2–3 persons)
  - A Compiler for the Register Machine from Hell (2 persons)
  - Propositional Logic (2 persons)
  - Tseitin Transformation (2 persons)
  - BIGNAT - Natural Numbers of Arbitrary Size (1 person)
  - The Euclidean Algorithm - Inductively (1 person)
- project assignment after break

Proof Methods

Last Session: Attributes

- attributes can modify facts: `of`, `OF`, `symmetric`, `rule_format`, `simplified`, …
- attributes can also specify usage of facts; examples
  - how to declare that rule should be used in specific method, e.g., simplification
    - `lemma fact[simp]: …` — when stating lemma
    - `declare fact[simp]` — outside proof
    - `note [simp] = fact` — locally within proof
  - what to declare
    - `declare fact[simp]` — add to standard simpset
    - `declare fact[simp del]` — delete from standard simpset
    - `declare fact[termination_simp]` — add to termination simpset
    - `declare fact[intro]` — declare as introduction rule
    - `declare fact[elim]` — declare as elimination rule
    - `declare fact[dest]` — declare as destruction rule

Kinds of Rules

- simplification rules – (conditional) equations used from left to right
- introduction rules – if conclusion of rule matches conclusion of subgoal, replace it by premises of rule (generating one new subgoal per premise)
- destruction rules – replace first premise of subgoal matching major premise of rule by conclusion (together with remaining premises) of rule
- elimination rules – like destruction rules, but rule is supposed to not loose (destruct) information (compare `conjunct1` with `conjE`)

Examples

- `have "∀x. P x" apply (rule allI)` $\rightsquigarrow$ `⋀x. P x`
- `have "A ∧ B ⟹ C" apply (drule conjunct2)` $\rightsquigarrow$ `B ⟹ C`
- `have "A ∨ B ⟹ C" apply (erule disjE)` $\rightsquigarrow$ 1. `A ⟹ C` 2. `B ⟹ C`

  (`drule` and `erule` are designed to apply dest-rules and elim-rules, respectively)

Equational Proof Methods

- `unfold` *fact*$^+$ – exhaustively apply equational facts (replacing left-hand sides by right-hand sides); usually as initial method
- `simp`/`simp_all` – exhaustively apply simp rules to first/all subgoal(s)

Proof Methods for Classical Reasoning

- (`intro | elim`) *fact*$^+$ – exhaustively apply intro/elim rules; usually as initial method
- `blast` (`best`, `fast`) – solve first subgoal by exhaustive proof search (up to certain bound) using all known intro/dest/elim rules (using best-first search, depth-first search)

Combined Proof Methods

- `force` (`fastforce`, `bestsimp`) – solve first subgoal by combination of equational and classical reasoning
- `auto` – apply combination of equational and classical reasoning to all subgoals and leave result as new subgoals

Selection of Methods

- distinction between
  - initial methods (predictable outcome, used at start of proof, e.g. `rule`, `intro`, `dest`, `unfold`, …)
  - final methods (solve some proof goals, e.g., `fast`, `best`, `auto`, `blast`, `linarith`, `presburger`, `algebra`, `metis`, `smt`, …)
- problem: how to know all the methods?
- solution
  - learn initial methods
  - use try0 to find suitable final method, it will try out several known methods and then inform about success
  - example

```
lemma "∀x. ∃y. P x y ⟹ ∃f. ∀x. P x (f x)"
  try0
(* output window shows successful method, e.g., by metis;
   after insertion of method, try0-invocation should be eliminated *)
```


Modifiers of Methods

success of methods can be increased by manual adaptions, e.g., addition of simp rules

Modifiers for Classical Methods

classical methods (like `blast` and `auto`) take following modifiers:

- `intro:` *fact*$^+$ – add additional intro rules
- `dest:` *fact*$^+$ – add additional dest rules
- `elim:` *fact*$^+$ – add additional elim rules
- `del:` *fact*$^+$ – delete classical rules

Note

when used with combined methods (like `force` and `auto`), modifiers for simplifier use prefix `simp` (like `simp add:`, `simp del:`, …)

The Split-Modifier

- consider goal that requires a case-analysis because of a case-expression, e.g. on lists

```
sorted (case g x of [] ⇒ [5] | y # ys ⇒ ys @ zs @ [y])
```

- for each datatype split rules are created that support such a case-analysis (`nat.splits`, `prod.splits`, `list.splits`, `bool.splits`, …)
- split rules are equalities that can be used by the simplifier, e.g., for lists:

```
P (case xs of [] ⇒ c | y # ys ⇒ f y ys) =
((xs = [] ⟶ P c) ∧ (∀ y ys. xs = y # ys ⟶ P (f y ys)))
```

- split rules have to be activated manually via `split`-modifier, syntax is `split:` *fact*$^+$
- split-modifier works in methods that use the simplifier: `simp`, `auto`, `force`, …
- example

```
have "sorted (case g x of [] ⇒ [5] | y # ys ⇒ ys @ zs @ [y])"
apply (simp only: split: list.splits)
1. (g x = [] ⟶ sorted [5]) ∧
   (∀y ys. g x = y # ys ⟶ sorted (ys @ zs @ [y]))
```

remark: `only`-modifier changes simpset so that only specified facts are used (here: none)

Demo

soundness of mergesort via modifiers

Composition of Methods

- sometimes, it is useful to apply several methods sequentially, e.g.,

```
lemma "∀ x :: nat. x < 30 ⟶ (∃ y z. y + x ≤ z ∧ odd y ∧ odd z)"
  apply (intro allI impI)
  apply (rule exI[of _ 5])
  apply (rule exI[of _ 35])
  by auto
```

- instead of using several `apply`s, one can combine methods sequentially via `,` or `;`

```
lemma "∀ x :: nat. x < 30 ⟶ (∃ y z. y + x ≤ z ∧ odd y ∧ odd z)"
  by (intro allI impI, rule exI[of _ 5], rule exI[of _ 35], auto)
```

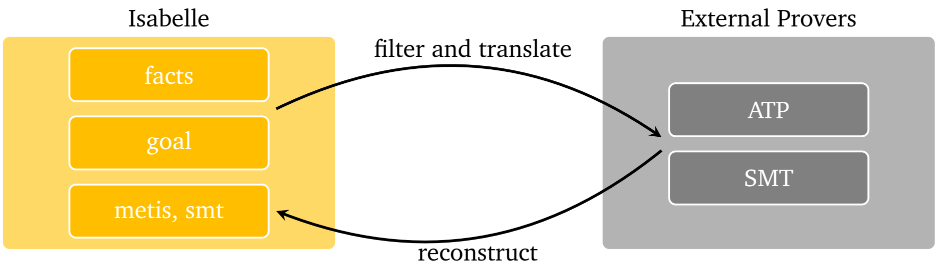
  - `apply (`*method1*`, `*method2*`)` is the same as `apply` *method1* `apply` *method2*
  - `apply (`*method1*`; `*method2*`)` first apply *method1* and apply *method2* on all new subgoals that are produced by *method1*

- (dis)advantages of sequential composition of methods
  - + fast to type; supports nested cases, e.g., `by (cases xs; cases ys; auto)` triggers case-analysis on all four combinations of whether lists `xs` and `ys` are (non)empty
  - − excessive use is hard to maintain and read, since no intermediate proof goals are visible

Sledgehammer

Sledgehammer

tool that applies automated theorem provers (ATPs) and satisfiability-modulo-theory (SMT) solvers to current subgoal



Phase 1: From Isabelle to External Provers

aim: prove $\Phi \models \psi$ where Φ is collection of all available facts and ψ is current goal

- selection problem
  - find-theorems after loading `Main` shows 22200 theorems ($\leq |\Phi|$)
  - current ATPs are not performing well when using all available facts
  - relevance filter: select top N facts that might be relevant for current goal
  - choice of N depends on target ATP
  - different relevance filters available, e.g., syntax guided or trained via machine learning
- language problem
  - untyped FOL (ATP) $\neq$ typed HOL (Isabelle) $\neq$ SMT languages
  - solution: encoding (e.g., encode type-information into terms, etc.)
  - adds a certain amount of imprecision
- overall workflow: for each external prover P (in parallel)
  - select $\{\varphi_1, \dots, \varphi_{N_P}\} \subseteq \Phi$ by relevance filter
  - ask P to prove $encode_P(\varphi_1 \longrightarrow \dots \longrightarrow \varphi_{N_P} \longrightarrow \psi)$
  - collect successful proofs

Phase 2: From External Provers to Isabelle

aim: prove $\Phi \models \psi$ where Φ is collection of all available fact and ψ is current goal
phase 1: obtain proof of $encode_P(\varphi_1 \longrightarrow \ldots \longrightarrow \varphi_{N_P} \longrightarrow \psi)$

- reconstruction problem
  - external proof is unreliable (buggy external provers)
  - external proof is non-trivial to replay in Isabelle (e.g., imprecision of encoding)
  - solution
    - analyze external proof: which φ_i have been used when proving ψ?
    - reconstruction of proof by finding HOL-proof using Isabelle inferences, where search is started from scratch, but restricted to used φ_i
- `metis`
  - metis is Isabelle built-in ATP (first-order ordered resolution and paramodulation)
  - its inferences go through Isabelle's proof kernel (correct by construction)
  - `metis` *fact** – apply metis using some auxiliary facts, e.g., the used φ_i's
- `smt`
  - alternative reconstruction mechanism to `metis`
  - main conceptual difference: for finding suitable inferences, again SMT solvers are invoked

Sledgehammer in Action

- standalone: via command `sledgehammer` (available in proof-mode)
  - `have` *statement* `sledgehammer` or `apply` *method* `sledgehammer` but not `have` *statement* `proof simp sledgehammer`
  - after invocation wait some seconds on answer in output panel (or abort by erasing `sledgehammer` command)
  - copy successful proof from output panel; erase `sledgehammer` command
- in combination: `try` combines `try0` with `sledgehammer`
  - note: in `try`, `sledgehammer` has a rather short time-limit, unlike in standalone version
- separate user manual for sledgehammer is available: `isabelle doc sledgehammer`

Strategies for Sledgehammer and Find-Theorems

- sledgehammer is only applicable if it completely solves a goal (all or nothing)
  - strategy: if sledgehammer cannot solve a goal in one step, add intermediate goals manually
- find-theorems helps you more in exploring possibilities and getting names
  - what kind of theorems are there to prove $\sum \ldots = \sum \ldots$?
  - what is the name of the distributivity law between addition and multiplication?

Demo

$\sqrt{2}$ is irrational