universität innsbruck

Summer Term 2024

Interactive Theorem Proving using Isabelle/HOL

Session 4

René Thiemann

Department of Computer Science

Outline

Calculational Reasoning

Aim: Support Proofs with Chains of (In)Equalities

$$a = b \leq c = d < e = f \qquad \hookrightarrow \qquad a < f$$

Solution: Combination of (In)Equalities by Transitivity

`also` – first occurrence in chain initializes auxiliary fact `calculation` to `this`; further occurrences combine `calculation` and `this` via transitivity and update `calculation` accordingly

Concluding a Chain of Transitive Combinations

`finally` – combine `calculation` and `this` via transitivity and update `this` accordingly

Also Useful for Calculational Reasoning

- implicit term abbreviation “ `...` ” refers to previous right-hand side of (in)equality
- method “ `.` ” tries to prove current subgoal by assumption

Example

```
fun sum :: "nat ⇒ nat" where
  "sum 0 = 0"
| "sum (Suc n) = Suc n + sum n"

lemma "sum n = n * (n + 1) div 2"
proof (induction n)
  case IH: (Suc n)
  have "sum (Suc n) = (n + 1) + sum n" by auto
  also have "... = (n + 1) + (n * (n + 1)) div 2" using IH by auto
  also have "... = (2 * (n + 1) + (n * (n + 1))) div 2" by auto
  also have "... = ((2 + n) * (n + 1)) div 2" by auto
  also have "... = (Suc n * (Suc n + 1)) div 2" by auto
  finally show ?case .
qed simp
```


Further Remarks

- calculational reasoning works with several relations, e.g., (=), (≤), (<), (⊆) and (⊂)
- calculational reasoning does not work with flipped relations such as (>); (>) is just an abbreviation of λ x y. y < x

```
have "a > b" ⟨proof⟩                have "b < a" ⟨proof⟩
also have "... > c" ⟨proof⟩         also have "c < ..." ⟨proof⟩
finally (* fails *)                 finally (* here you see why *)
```

- calculational reasoning with equality supports contexts

```
have "a = f b" ⟨proof⟩              have "a ≤ b + c" ⟨proof⟩
also have "b = c" ⟨proof⟩           also have "c ≤ d" ⟨proof⟩
also have "f ... = d" ⟨proof⟩       finally have "a ≤ b + d" .
finally have "a = d" .              (* fails *)
```


Proofs by Induction Revisited

Example Induction Proof of Last Week – Reversing a List Twice

```
lemma rev_rev[simp]: "reverse (reverse xs) = xs"
proof (induction xs)
  case (Cons x xs)
  then show ?case
    by (auto simp: rev_app)
qed auto
```

Approach

- state variable on which induction should be applied
- choose own variable names for each case
- identify and add auxiliary lemmas on demand
- solve trivial cases via `qed auto`
- not everything explained: usage of arbitrary variables and preconditions

Motivation – Fast Implementation of List Reversal

```
fun rev_it :: "'a list ⇒ 'a list ⇒ 'a list" where
  "rev_it [] ys = ys"
| "rev_it (x # xs) ys = rev_it xs (x # ys)"

fun fast_rev :: "'a list ⇒ 'a list" where
  "fast_rev xs = rev_it xs []"

lemma fast_rev: "fast_rev xs = reverse xs"
```

First Problem

- core property is `rev_it xs [] = reverse xs`
- induction on `xs` yields problematic subgoal: 2nd arguments of `rev_it` differ!
  `rev_it xs [] = reverse xs ⟹ rev_it xs [x] = reverse xs @ [x]`
  (minor non-relevant change: in the definition of `reverse` we replaced append by Isabelle's predefined append function (`@`))

Solving First Problem

- core property is `rev_it xs [] = reverse xs`
- proving this property by induction leads to an IH which is too weak:
  2nd argument of `rev_it` is no longer `[]` in subgoal
- solution: generalize property

  `rev_it xs ys = reverse xs @ ys` (creativity required)

Second Problem

- still the induction proofs fails on (simplified) subgoal

  ```
      rev_it xs ys = reverse xs @ ys
  ⟹ rev_it xs (x # ys) = reverse xs @ x # ys
  ```
- the 2nd arguments of `rev_it` still differ
  (in particular the 2nd argument of `rev_it` in the IH is still the original `ys`)
- aim: perform induction on `xs`, but permit change of variable `ys` in IH

Solving Second Problem – Arbitrary Variables

- solution: tell induction method which variables should be arbitrary

  perform induction on x for arbitrary y and z
- effect
  - y and z can be freely instantiated in the IH
  - y and z within induction proof have no connection to y and z outside induction proof

Finalizing Proof of Previous Slide

```
have "rev_it xs ys = reverse xs @ ys"
proof (induction xs arbitrary: ys)
  case (Cons x xs ys)    (* IH is: rev_it xs ?ys = reverse xs @ ?ys *)
  thus ?case by auto
qed auto
```

- for each case one chooses names of arguments of constructor and arbitrary variables
- after "`arbitrary:`" there can be several variable names

Premises in Induction Proofs

- the `induction` method can also deal with goals containing premises, e.g.,
  `A x ⟹ B y ⟹ C x y`
- whenever we are within `case (CName ...)`:
  - `CName.IH` refers to IH
  - `CName.prems` refers to premises
- since premises weaken IHs, or make IHs more complex to apply, it sometimes is preferable to omit premises from property that is proven by induction

Premises in Induction Proofs – Examples

```
have "A (x :: nat) ⟹ B y ⟹ C x y" proof (induction x)
  case (Suc x)                      (* annoying, "B y" is contained in IH *)
  thm Suc.prems — ‹A (Suc x), B y›
  thm Suc.IH    — ‹A x ⟹ B y ⟹ C x y›

assume "B y"      (* if y is not changed, move properties of y outside *)
have "A (x :: nat) ⟹ C x y" proof (induction x)
  case (Suc x)
  thm Suc.prems — ‹A (Suc x)›
  thm Suc.IH    — ‹A x ⟹ C x y›

have "A (x :: nat) ⟹ B y ⟹ C x y" proof (induction x arbitrary: y)
  case (Suc x y)     (* since y is changed, cannot move "B y" outside *)
  thm Suc.prems — ‹A (Suc x), B y›
  thm Suc.IH    — ‹A x ⟹ B ?y ⟹ C x ?y›
```


Controlling the Proof State and Isabelle's Simplifier

The Simplifier

- applies (conditional) equations exhaustively; these equations are also called simp rules
- equations are always oriented left-to-right: given equation $c \Longrightarrow l = r$ and goal
  - try to find subterm $l\sigma$ in goal and replace it by $r\sigma$ provided that $c\sigma$ simplifies to `True`
  - consequence: equation should satisfy that both c and r are somehow smaller than l
  - examples
    - `n < m ⟹ (n < Suc m) = True` might be used as simp rule
    - `Suc n < m ⟹ (n < m) = True` will lead to non-termination
- boolean proposition `A` is implicitly considered as equation `A = True`
- equations taken from implicit simpset
- certain commands (like `datatype` and `fun`) implicitly extend simpset

Globally Modifying the Simpset

- globally add equation to simpset: `declare` *fact* `[simp]` or `lemma` *name* `[simp]: ...`
- globally delete simp rule from simpset: `declare` *fact* `[simp del]`

Locally Modifying the Simpset within a Proof

- `note [simp] =` *facts*
- `note [simp del] =` *facts*

Predefined Simpsets and Notable Simp Rules

- depending on proof goal, several standard simpsets and simp rules might be useful
- these are not used by default, since they can drastically change or blow-up your proof goal (exponential increase)
- `numeral_eq_Suc`: convert number literals into `Suc`-representation: `1000 = Suc(...)`
- `Let_def`: expand `let`s
- `ac_simps`: use commutativity and associativity of operators
- `algebra_simps`, `field_simps`: add distributivity laws, etc.

The `simp` Method – Using Simp Rules Automatically

- `simp` – apply simplifier to first subgoal
- `simp_all` – apply simplifier to all subgoals
- modifier `add:` *fact** – locally add equation as simp rule or activate predefined simpset
- modifier `del:` *fact** – locally delete simp rules from simpset
- modifier `only:` *fact** – only use specified simp rules
- modifier `flip:` *fact** – locally delete simp rules and add their symmetric versions

Comparing `simp` and `auto`

- `auto` includes `simp` and `simp_all`, but also does classical reasoning
  - advantage: more powerful than `simp` (modifiers: `auto simp add: ...`)
  - disadvantages occur if `auto` does not completely solve a goal
    - might turn provable goal into unprovable one
    - new proof obligation might be unreadable (too many changes)
    - starting a structured proof after `auto` is brittle, since result of `auto` will easily change
- use `simp` to have more control over proof state

Controlling Proof State – Unfolding Equations Explicitly

- `unfold` *fact*$^+$ – method that unfolds equations (similar to `simp only:` *fact*$^+$)
- `unfolding` *fact*$^+$ – exhaustively use equations for simplification

Controlling Proof State – Applying Single Equation

- `subst` *fact* – method that applies conditional equation and adds conditions as new goals

A More Complete Grammar of Proofs

proof	::=	*prefix** `sorry`	
	|	*prefix** `by` *method method*$^?$	
	|	*prefix** `proof` *method*$^?$ *statement** `qed` *method*$^?$	
	|	*prefix** `done`	final step, if no goals left
prefix	::=	`apply` *method*	
	|	`unfolding` *fact*$^+$	
	|	`using` *fact*$^+$	

- `apply` and `unfolding` are used for step-wise proof exploration

Styles of Proofs

- structured proofs (Isar-proofs)
  - Isabelle/Isar
    - proof-language that was introduced here in this lecture
    - Isar: Intelligible semi-automated reasoning
    - PhD thesis of Makarius Wenzel
  - intermediate goals are explicitly stated
  - readable without inspecting proof state
- apply-style proofs (of form `apply`* `done`)
  - traditional style of proofs (used in Coq, HOL-Light, ...)
  - sequence of proof methods (apply this method, then that, then ...)
  - readable if one inspects intermediate proof goals
- both styles have their own advantages; mixture is possible
- often: proof exploration via apply-style, then rewrite into Isar-style

Demo

- soundness of insertion sort