universität innsbruck

Summer Term 2024

Interactive Theorem Proving using Isabelle/HOL

Session 5

René Thiemann

Department of Computer Science

Outline

Function Definitions Revisited

Overlapping Equations

- when declaring a new function via `fun`, the equations may be overlapping
- internally, the equations are preprocessed to become non-overlapping; patterns are instantiated on demand
- effect of preprocessing becomes visible in various places, e.g., the simplification rules

Example

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

is translated into function without overlap, which then determines simp rules

```
fun drop_last :: "'a list ⇒ 'a list" where
  "drop_last (x # y # ys) = x # drop_last (y # ys)"
| "drop_last [] = []"
| "drop_last [v] = []"
```

Underspecification

- `fun` accepts function definitions where not all of the cases have been covered
  `fun head1 where "head1 (x # xs) = x"`
- `case` expressions do not enforce that all cases are covered
  `fun head2 where "head2 xs = (case xs of x # _ ⇒ x)"`
- however, HOL is a logic of total functions; what is the value of `head1 []` or `head2 []`?
- to model underspecification, Isabelle/HOL has a special constant `undefined :: 'a`
- `undefined :: 'a` is an ordinary value of type `'a` and not some kind of error
  - `undefined :: nat` is a natural number (but we don't know which one)
  - `undefined :: bool` is either `True` or `False` (but we don't know the alternative)
- `undefined` is used to fill in missing cases during preprocessing
  `"head1 [] = undefined"`
  `"head2 xs = (case xs of x # _ ⇒ x | [] ⇒ undefined)"`
- the missing cases are usually not revealed to the user, e.g., `head1.simps` only consists of original equation

Computation Induction

- consider again

```
fun drop_last :: "'a list ⇒ 'a list" where
  "drop_last (x # y # ys) = x # drop_last (y # ys)"
| "drop_last [] = []"
| "drop_last [v] = []"
```

- aim: prove `lemma "length (drop_last xs) = length xs - 1"`
- "natural" induction scheme (computation induction) follows structure of algorithm
  - consider all cases of function, i.e., `x # y # ys`, `[]` and `[v]` for `drop_last`
  - provide IH for recursive calls, i.e., for `y # ys` in first case of `drop_last`
  - computation induction is sound, since termination has been proven by `fun`
  - computation induction rule is automatically generated by `fun`, e.g., `drop_last.induct` is:

    $(\bigwedge x\ y\ ys.\ P\ (y\ \#\ ys) \Longrightarrow P\ (x\ \#\ y\ \#\ ys)) \Longrightarrow P\ [] \Longrightarrow (\bigwedge v.\ P\ [v])$
    $\Longrightarrow P\ xs$
- `induction`-method can use custom induction rule via `rule:` *induct_thm*
  `lemma ... by (induction xs rule: drop_last.induct) auto`
- case names when using computation induction are just numbers (1, 2, ...)

Computation Induction and Underspecification

- computation induction considers all cases of function
- what if function is underspecified?
- example
  `fun head where "head (x # xs) = x"`
- potential computation induction rule is incorrect
  $(\bigwedge x\ xs.\ P\ (x\ \#\ xs)) \Longrightarrow P\ xs$
- obviously, also the missing cases have to covered, these become visible in induction rule
  `thm head.induct:` $(\bigwedge x\ xs.\ P\ (x\ \#\ xs)) \Longrightarrow P\ [] \Longrightarrow P\ xs$

Manual Termination Proofs

Failing Termination Proofs

- consider Isabelle functions

```
fun gen_list :: "nat ⇒ nat ⇒ nat list" where   (* gen_list n m = [n .. m] *)
  "gen_list n m = (if n ≤ m then n # gen_list (Suc n) m else [])"
fun split :: "_ ⇒ _ list ⇒ _ list × _ list" where  ...
fun qsort :: "'a :: linorder list ⇒ 'a list" where
  "qsort [] = []"
| "qsort (x # xs) = (case split x xs of
     (low, high) ⇒ qsort low @ [x] @ qsort high)"
```

- problem: `fun` fails for `qsort` and `gen_list`, since it cannot find termination proof
- there are several reasons why a termination proof cannot be found
  1. the internal heuristic is too weak (here: neither `n` nor `m` decrease in `gen_list`)
  2. the heuristic is able to find the right terminating argument, but auxiliary facts are missing (here: splitting a list into `low` and `high` does not increase the length)
  3. in case of higher-order recursion unprovable termination conditions might be generated
  4. the function does not terminate
- solution in cases 1 – 3: perform termination proofs manually

The `function` Command

- via `function` one can separate a function definition from its termination proof
- outer syntax:

  `function (sequential)`? *name* `::` *ty* `where` *eqns* ⟨*proof*⟩
  `termination` ⟨*proof*⟩
- explanations
  - in the proof after `function` one has to show that all cases have been covered and that no conflicting results may occur in case of overlapping equations
    - for underspecified or overlapping equations, use `(sequential)` to trigger preprocessing
    - then resulting proof is always the same: `by pat_completeness auto`
  - only after successful termination proof, simp rules and induction scheme become available
- `fun` is just a wrapper around `function`:

  `fun` *name* `where` *eqns*

  is the same as

  `function (sequential)` *name* `where` *eqns* `by pat_completeness auto`
  `termination by lexicographic_order`

Manual Termination Proofs

- termination proofs of function `f` are usually of the following shape
  - provide a well-founded relation <
  - show *args_rec* < *args_lhs* for each equation `f` *args_lhs* `= ... f` *args_rec* `...`, taking into account if-then-else and case-expressions in the context indicated by ... ...
  - if `f` has multiple arguments, then these are automatically converted into tuples
- termination proofs are started in Isabelle via
  - the standard proof method (where the relation becomes a schematic variable)
  - or via the method `relation` *less_than* where the relation is directly fixed
- important well-founded relations are
  - `measure (m :: _ ⇒ nat)`
    - compare elements by mapping them to natural numbers
    - examples for `m`
      `length`, `count :: tree ⇒ nat`, `height :: tree ⇒ nat`, `id :: nat ⇒ nat`
  - `measures (ms :: (_ ⇒ nat) list)`
    - lexicographic combination of multiple measures from left to right
    - this is what is internally used by method `lexicographic_order`
  - well-foundedness of both `measure m` and `measures ms` is by simp

Example Termination Proof

```
function gen_list :: "nat ⇒ nat ⇒ nat list" where
  "gen_list n m = (if n ≤ m then n # gen_list (Suc n) m else [])"
  by pat_completeness auto

termination
proof
 1. wf ?R
 2. ⋀n m. n ≤ m ⟹ ((Suc n, m), (n, m)) ∈ ?R
oops

termination by (relation "measure (λ (n,m). Suc m - n)") auto
(* after relation command and discharging trivial wf-requirement,
   the goal is equivalent to: *)
 1. ⋀n m. n ≤ m ⟹ Suc m - Suc n < Suc m - n
```

Example Termination Proof

```
function qsort :: "'a :: linorder list ⇒ 'a list" where
  "qsort [] = []"
| "qsort (x # xs) = (case split x xs of
     (low, high) ⇒ qsort low @ [x] @ qsort high)"
  by pat_completeness auto

termination
proof (relation "measure length")
(* after simplification, the goals are: *)
 1. ⋀ ... (low, high) = split x xs ⟹ length low < Suc (length xs)
 2. ⋀ ... (low, high) = split x xs ⟹ length high < Suc (length xs)
```


A Simpset for Termination Proofs

- simp lemmas that are particularly useful for termination proofs can be stored in a dedicated simpset: `termination_simp`
- method `lexicographic_order` in particular tries to finish termination proof obligations by `auto simp: termination_simp`
- having adjusted this simpset accordingly, proofs might become automatic again

An Automatic Termination Proof for Quicksort

```
(* show that split is just two applications of filter;
   advantage: many facts about filter are already known *)
lemma split: "split a xs = (filter (λ x. x ≤ a) xs, filter (λ x. ¬ x ≤ a) xs)"
  by (induction xs) auto

declare split[termination_simp]

fun qsort :: "'a :: linorder list ⇒ 'a list" where
  "qsort [] = []"
| "qsort (x # xs) = (case split x xs of
     (low, high) ⇒ qsort low @ [x] @ qsort high)"
```


Termination versus Termination

- two notions of termination
  1. `function` definitions require termination proof
  2. application of simp rules should terminate
- 1 does not imply 2!
  - reason: evaluation strategy of if-then-else is ignored by simplifier
  - example: lhs of `gen_list.simps` is always applicable and introduces recursive call

    `gen_list ?n ?m = (if ?n ≤ ?m then ?n # gen_list (Suc ?n) ?m else [])`
  - in these cases it is advisable to
    - globally delete simp rules from simpset

      `declare gen_list.simps[simp del]`
    - locally add simp rules in proof for specific arguments via attribute `of`

      ```
      case (1 n m)
      note [simp] = gen_list.simps[of n m]

      (* instantiated simp rule *)
      gen_list n m = (if n ≤ m then n # gen_list (Suc n) m else [])
      ```


Example Proof

```
declare gen_list.simps[simp del]

lemma "length (gen_list n m) = Suc m - n"
proof (induction n m rule: gen_list.induct)
  case (1 n m)
  note [simp] = gen_list.simps[of n m]
  from 1 show ?case by auto
qed
```

- since `gen_list` takes two arguments, induction is performed simultaneously on both variables (`induction n m rule: gen_list.induct`)
- after activating simp rules locally, proof is automatic thanks to suitable shape of computation induction rule

  `(⋀n m. (n ≤ m ⟹ P (Suc n) m) ⟹ P n m) ⟹ P x y`

  (note that IH is only accessible if we are in the correct if-then-else branch)

Attributes

Attributes

- attributes can be used to change a fact
- these changes are usually made to help the automation
  - instantiate variables
    - choice of existential witness or of universal elimination
    - non-terminating simp rules
  - discharge assumptions
  - obtain an equation in the other direction
- syntax: *fact* [$attr_1$, ..., $attr_n$]

Some Useful Attributes

- `of` – instantiation of schematic variables (by position from left to right)
  `‹?x = ?y ⟹ ?y = ?z ⟹ ?x = ?z›[of _ 5 x]` ⇝
  `‹?x = 5 ⟹ 5 = x ⟹ ?x = x›`
- `where` – instantiation of schematic variables (by name)
  `‹?x = ?y ⟹ ?y = ?z ⟹ ?x = ?z›[where y = 5 and z = x]` ⇝
  `‹?x = 5 ⟹ 5 = x ⟹ ?x = x›`
- `OF` – discharge assumptions using existing facts (by position)
  `‹?P ⟶ ?Q ⟹ ?P ⟹ ?Q›[OF ‹A ⟶ B x›]` ⇝ `‹A ⟹ B x›`
- `symmetric` – get symmetric version of equation
  `‹?P ⟹ ?a = ?b›[symmetric]` ⇝ `‹?P ⟹ ?b = ?a›`
- `rule_format` – replace HOL connectives by Pure connectives
  `‹∀x. ?P x ⟶ ?Q›[rule_format]` ⇝ `‹?P ?x ⟹ ?Q›`
- `simplified` – view result after simplification, e.g.,
  `case (Cons x xs) thm Cons.IH[simplified]`
- combined example: `‹∀x. A x ⟶ B x›[rule_format, of 5]` ⇝ `‹A 5 ⟹ B 5›`

Attributes versus Isar-Style

- most of the attributes can easily be simulated by standard Isar proofs
- example
  - instead of writing

```
from Cons.IH(2)[of 3] other_fact show ?case by auto
```

  - one could also write

```
from Cons.IH
have ‹(* spelled out version of second IH with value 3 inserted *)›
  by auto
with other_fact show ?case by auto
```

- advantage of attributes: generate required facts on the fly, without having to type a (large) statement
- advantage of Isar style: proof is more readable without looking at Isabelle output

Demo

soundness of quicksort (covers computation induction, termination proof, attributes)