Interactive Software Verification

Spring Term 2013

Holger Gast

gasth@in.tum.de

30.4.2013

Isabelle So Far

- Basic use
- Definitions create new . . . (possibly not all)
  - New types
  - New constants
  - New theorems about constants
- Basic proof techniques
- Introduction to recursion and induction

Recap: Proofs in Isabelle

- Goals: $\llbracket\, knowledge \rrbracket \Longrightarrow to\ be\ derived$
- Simplifier `simp`
  - Proofs by term rewriting with equalities
  - Matching: replaces unknowns in rules by terms
  - Example use: compute result of defined HOL functions
- Elementary proof steps
  - `rule`: introduction rules $\llbracket ?P; ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$
  - `erule`: elimination rules $\llbracket ?P \wedge ?Q; \llbracket ?P; ?Q \rrbracket \Longrightarrow ?R \rrbracket \Longrightarrow ?R$
  - `drule`/`frule`: Vorwärts-Schlüsse $?P \wedge ?Q \Longrightarrow ?P$
- Automatic proof search
  - `auto`: elementary steps + `simp` (using heuristics)
  - `force`, `fastforce`: like `auto`, but fails if does not solve
  - `fast`, `blast`: $\approx$ only elementary steps, no simp

Example: In-order Traversal of Trees

fun in-order :: "bt ⇒ int list"
where
"in-order Leaf = []"
| "in-order (Node l v r) = in-order l @ [v] @ in-order r"

- Rewrite rules f`.simps`

  in-order Leaf = []
  in-order (Node ?l ?v ?r) = in-order ?l @ [?v] @ in-order ?r

- Induction (over call depth) f`.induct`

  $[\![\text{?P Leaf}; \bigwedge l\, v\, r.\ [\![\text{?P } l; \text{?P } r]\!] \Longrightarrow \text{?P (Node } l\ v\ r)]\!] \Longrightarrow \text{?P ?a}$

Note: analogous to `bt.induct` only by accident (because function is structural recursion)

- Using the induction rule

  `apply (induct` *args* `rule:` *regel*`)`

⇒ Replaces `?a` (by *args*) and then `?P`

Example: Induction over the Call Depth

lemma length-of-in-order-by-fun-induct:
"int (length (in-order t)) = ninner t"
apply (induct rule: in-order.induct)

Yields proof obligations

proof (prove): step 1
goal (2 subgoals):
1. int (length (in-order Leaf)) = ninner Leaf
2. $\bigwedge$l v r. $[\![$int (length (in-order l)) = ninner l;
int (length (in-order r)) = ninner r$]\!]$
$\Longrightarrow$ int (length (in-order (Node l v r))) = ninner (Node l v r)

Idea: prove an assertion about behaviour of `in_order`

apply (simp-all only: in-order.simps)
2. $\bigwedge$l v r. $[\![$int (length (in-order l)) = ninner l;
int (length (in-order r)) = ninner r$]\!]$
$\Longrightarrow$ int (length (in-order l @ [v] @ in-order r)) = ninner (Node l v r)

Today: Further Basics & Applications

- Basics
  - Types: polymorphism and type classes
- Functions
  - Tail recursion & loops
  - Termination proofs
- Induction
  - Induction over the call depth
  - Tail recursion & (loop-like) invariants
  - Nested induction
- Examples in `Demo03.thy`
- Overall goal: prepare for reasoning about language semantics

Basics: Polymorphism and Type Classes

- Theories are often independent of the type of data
  - Equalities in rings: $\mathbb{Z}$, $\mathbb{R}$, ...
  - Basis: axioms of rings
  - Then show for special cases that axioms are fulfilled

$\Rightarrow$ Theorems are specialized automatically

- Idea: polymorphism
  - Allow type variables α, β, ...
  - Isabelle syntax: `'a`, `'b`, ... as types
  - As for terms: free type variables and type unknowns

$\Rightarrow$ Isabelle replaces type variables as necessary
(Hindley/Milner type inference, similar to ML, Haskell, ...)

$\Rightarrow$ Terms (definitions, theorems, ...) have many types

Basics: Polymorphism and Type Classes

- Idea: type classes (also: sorts)
  - Type classes restrict replacements for type variables
  - Isabelle syntax: `'a :: linorder`
    As in type restrictions `i :: int`
    Can be combined: `x :: 'a :: linorder`
  - Type classes can introduce syntax for operators, . . .
  - They can give axioms that types must obey

  ⇒ Instances of type classes
- Show sorts: `declare[[show_sorts]]` at theory level, `using [[show_sorts]]` in proofs
- In jEdit: hover over type variable

Example: Orders

- Class `ord`
  - Notation $\leq$, $<$
  - No axioms about behaviour / meaning of these
- Class `preorder`
  - `less_le_not_le`: $x < y \longleftrightarrow x \leq y \wedge \neg (y \leq x)$
  - `order_refl`: $x \leq x$
  - `order_trans`: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
- Class `linorder` (linear / total orders)
  - `linear`: $x \leq y \vee y \leq x$

Example: Orders

- If a type is an instance of an order class
  - We have basic theorems about orders (`Orderings.thy`)
  - Automatic provers about orders (transitivity!) kick in

$\Rightarrow$ Enables re-use of theories & automated provers

$\Rightarrow$ Worth considering for one's own types

- Example: `word` for machine-level words / bit representations

Polymorphism

```
fun bt-sum :: "bt ⇒ int"
where "bt-sum Leaf = 0"
| "bt-sum (Node l u r) = bt-sum l + u + bt-sum r"
```

Same for: in-order, nleafs, ninner, . . . — *there must be a better way*

```
fun bt-fold :: "'a ⇒ ('a ⇒ int ⇒ 'a ⇒ 'a) ⇒ bt ⇒ 'a"
where "bt-fold u f Leaf = u"
| "bt-fold u f (Node l x r) = f (bt-fold u f l) x (bt-fold u f r)"

lemma bt-sum-by-bt-fold:
 "bt-sum t = bt-fold 0 (λa b c. a + b + c) t"
 by (induct t) auto

definition
 "bt-forall P t ≡ bt-fold True (λa b c. a ∧ P b ∧ c) t"

lemma bt-forall-result:
"bt-forall P t = (∀ x ∈ bt-elems t. P x)"
 apply (induct t)
 apply (unfold bt-forall-def)
 apply auto
 done
```

Induction on Call Depth

- Example: `merge` (note: “sequential” implied)

  fun merge :: "('a :: linorder) list ⇒ 'a list ⇒ 'a list"
  where
  "merge [] ys = ys"
  | "merge xs [] = xs"
  | "merge (x # xs) (y # ys) =
  (if x ≤ y
  then x # merge xs (y # ys)
  else y # merge (x # xs) ys)"

- Merge is correct

  lemma merge-sorted:
  "⟦ sorted xs; sorted ys ⟧ ⟹ sorted (merge xs ys)"

Direct Attempts – Rather Inelegant

- Problem with `induct xs`: cases reduce one argument each

  apply (induct xs)
  apply (auto simp add: sorted-Cons)
  txt {* Problem: second argument is not decreased *}

$\Rightarrow$ Induction hypothesis does not apply in one of the cases

- We must perform induction to make arguments "smaller"

  apply (induct "length xs + length ys" arbitrary: xs ys)
  apply (case-tac xs)
  apply (case-tac ys)
  apply (auto simp add: sorted-Cons set-of-merge)
  ...*more cases and auto*...
  done

Correctness proof of `merge`

apply (induct xs ys rule: merge.induct)
apply (simp-all add: sorted-Cons)
apply (auto simp add: set-of-merge)
done

- Merge works with with both arguments symmetrically

$\Rightarrow$ Not defined along the structure of one list

$\Rightarrow$ Cannot argue by induction on list structure

- Approach: we wish to prove a statement about the result

$\Rightarrow$ Induction on the computation leading to the result

- Later: extension to inductively defined relations

$\Rightarrow$ Will be central in talking about semantics of languages

Tail recursion

- Recursion uses space on the stack

⇒ Overflows on large/deep data structures

- Tail recursion
  - Recursive call only as "final step"

  ⇒ Compiler can optimize into loop
- Example of naive recursion

  fun addup :: "int list ⇒ int"
  where "addup [] = 0"
  | "addup (x # xs) = x + addup xs"

  - Execute + after recursion

  ⇒ Must use a stack

  ⇒ Re-write to use tail recursion

Compare: Imperative Solution

- Direct relation to usual practice
  - Tail recursion = loop
  - Accumulator = auxiliary local variable
  - Induction hypothesis ≈ loop invariant
- Example `sumup`

```
public static int sumup(int a[]) {
  int res = 0;
  for (int i = 0; i != a.length; i++) {
    // Invariante: res = \Sum j=0..i-1: a[i]
    // Äquiv.: \Sum j=0..n-1. a[i]
    //              = res + \Sum j=i..n-1. a[i]
    res = res + a[i];
  }
  return res;
}
```

Example of Tail Recursion

- Tail-recursive variant of summation

  fun addup-tr :: "int ⇒ int list ⇒ int"
  where "addup-tr accu [] = accu"
  | "addup-tr accu (x # xs) = addup-tr (x + accu) xs"

  - Keep partial sums in accumulator
  - Recursive call is final step

  ⇒ Stack space for current call can be discarded

- Is this variant "correct"? ⇨ Have to prove

  lemma addup-correct:
  "addup-tr 0 xs = addup xs"
  apply (induct xs)

- Typical: induction hypothesis too weak ⇨ goal unsolvable

  $\text{addup-tr}\ 0\ \text{xs} = \text{addup}\ \text{xs} \Longrightarrow \text{addup-tr}\ \text{x}\ \text{xs} = \text{x} + \text{addup}\ \text{xs}$

Solution: Generalize over Accumulator

- Statement must take into account arbitrary acucmulators

  lemma addup-correct-aux:
  "$\forall$ accu. addup-tr accu xs = addup xs + accu"
  by (induct xs) auto

  - `accu` appears in the result / right-hand-side
  - Is forall quantified ⇨ quantified in IH
- Conclusion

  lemma addup-correct:
  "addup-tr 0 xs = addup xs"
  by (simp add: addup-correct-aux)
- Method `induct` supports this common case

  lemma addup-correct-aux':
  "addup-tr accu xs = addup xs + accu"
  by (induct xs arbitrary: accu) auto

Generalization to Trees

- Sum of values in the tree

  definition
  "btsum bt $\equiv$ listsum (in-order bt)"

- Recursive solution straightforward $\approx$ `in-order`

- Tail recursion: explicit stack (next slide)

- Relation to previous ideas
  - Accu $=$ Stack $+$ partial sum
  - Function call $\approx$ execution of loop body

$\Rightarrow$ $\approx$ state transitions in language semantics

Application: Loops

```
public static int compute(TreeNode t) {
  int res = 0;
  Stack<TreeNode> stack = new Stack<TreeNode>();
  stack.push(t);
  while (!stack.isEmpty()) {
    TreeNode x = stack.pop();
    if (x instanceof Node) {
      Node n = (Node) x;
      res += n.val;
      stack.push(n.right);
      stack.push(n.left);
    }
  }
  return res;
}
```

Tail-recursive Sum over Trees

function btsum-tr :: "(bt list × int) ⇒ (bt list × int)"
where
"btsum-tr ([], res) = ([], res)"
| "btsum-tr (Leaf # stack, res) = btsum-tr (stack, res)"
| "btsum-tr (Node l v r # stack, res) = btsum-tr (l # r # stack, v + res)"
by pat-completeness auto

- Use list as stack
- State = (`stack`,`res`)
- Mirrors loop of previous slide
- Correctness

  lemma btsum-tr-correct:
  "snd (btsum-tr ([t],0)) = btsum t"

- First task: prove termination of the function

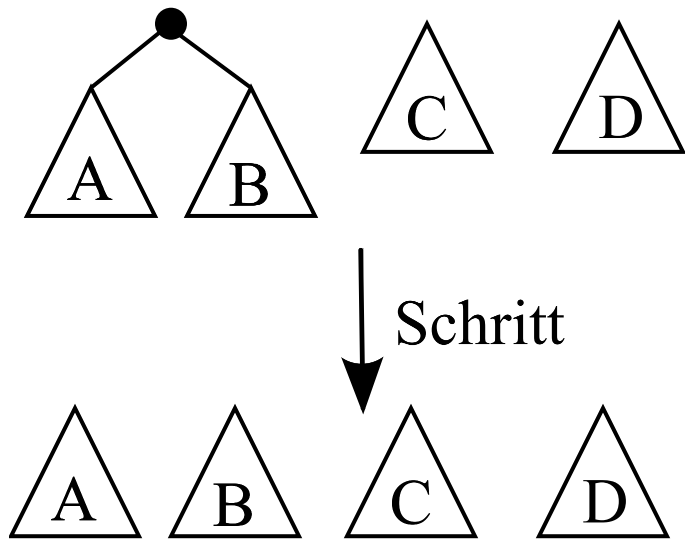
Termination Proof

- HOL functions must terminate (otherwise: inconsistencies)
- How to the steps of the function help?

  "btsum-tr (Leaf # stack, res) = btsum-tr (stack, res)"
  "btsum-tr (Node l v r # stack, res) = btsum-tr (l # r # stack, v + res)"

- Idea: after step less work remains
  - Stack is "list of tasks"
  - `Leaf`s are no tasks at all
  - `Node`s are processed, their children remain
- Technical argument
  - Give well-founded order such that
  - The function arguments are smaller in recursive call

Idea Termination Proof



"btsum-tr (Node l v r # stack, res) = btsum-tr (l # r # stack, v + res)"

Proof of Termination

- Idea: count the "number of tasks" (could also use built-in `size` function)

  fun node-size :: "bt ⇒ nat"
  where "node-size Leaf = 1"
  | "node-size (Node l - r) = 1 + node-size l + node-size r"
  definition
  "stack-size stack ≡ listsum (map node-size stack)"

- For a function $'a \rightarrow$ `nat` order $\{(x, y).\, f\, x < f\, y\}$ is well-founded

$$\mathrm{wf}\,(\mathrm{measure}\, f) \qquad (\texttt{wf-measure})$$

- Use this with the `relation` proof method

  termination btsum-tr
  apply (relation "measure (λ(stack,-). stack-size stack)")
  apply (auto simp add: stack-size-def)
  done

Aside: Multiset Orders

- Insight: the recursion step either
  - Removes a tree
  - Or removes a tree and replaces it with two smaller trees

$\Rightarrow$ New argument is "less" in the multiset order [1, §21]

- Multiset orders over a well-founded order are well-founded

$\Rightarrow$ Prove termination by comparing by that order!

apply (relation "inv-image (mult (measure size)) (multiset-of ∘ fst)")
apply (auto intro!: wf-mult)

  - Look only at first argument of function (i.e. the tree)
  - Take the multiset of
  - Compare multisets in recursion call be multiset order

Now: Correctness of `btsum-tr`

- Need to prove snd (btsum-tr ([t],0)) = btsum t
- Again: generalize over "current state"
  - Relation between partial result and final result
  - Analogous to loop invariant in iterative solution

⇒ snd (btsum-tr (stack,res)) = res + listsum (map btsum stack)

- Idea: induction over call depth (again)

  apply (induct "(stack,res)" arbitrary: stack res rule: btsum-tr.induct)

  - Induction step changes the state ⇨ on `(stack,res)`
  - Its contents is "arbitrary"
- Tip: look at goals without arbitrary, just to know what such unsolvable goals look like.

Nested Induction

- Common scenario
  - Function on several inductively-defined arguments
  - Induction only on one of them (or call depth)
  - Result: IH & proof require induction over other argument
- Example: addition on natural numbers (Z=zero, S=successor)

  datatype $\mathcal{N} = \mathsf{Z} \mid \mathsf{S}\ \mathcal{N}$
  fun addN :: "$\mathcal{N} \Rightarrow \mathcal{N} \Rightarrow \mathcal{N}$" (infixl "$\oplus$" 65)
  where
  "addN Z b = b"
  | "addN (S a) b = S (addN a b)"

- We seek to prove commutativity

  lemma add-comm:
  "$a \oplus b = b \oplus a$"

Nested Induction

- Direct `induct a` yields goals

  goal (2 subgoals):
  1. $b = b \oplus Z$
  2. $\bigwedge a.\ a \oplus b = b \oplus a \Longrightarrow S\,(b \oplus a) = b \oplus S\,a$

$\Rightarrow$ Must prove for arbitrary `b`

$\Rightarrow$ Prove these as auxiliary lemmata by induction, too

lemma add-comm-Z:
"$b = b \oplus Z$"
by (induct b) auto
lemma add-comm-S:
"$b \oplus S\,c = S\,(b \oplus c)$"
by (induct b) auto
lemma add-comm:
"$a \oplus b = b \oplus a$"
by (induct a) (auto simp add: add-comm-Z add-comm-S)

Standard Constructions

- Motivation: OO patterns
  - Get a solution quicker
  - Use established solutions
  - Get known nice properties for free
- The same exists for functional programming
  - Standard solutions for common problems
  - Get pre-defined functions and constants
  - Access readily available theorems
- Examples
  - Option (encode success / failure)
  - Maps (partial functions, valuations for variables)
  - Lists (map, fold, filter, . . .)

Type `option`

- Common phenomenon: partial functions
  - Lookup of value of undefined variable
  - Non-terminating function
  - Illegal function arguments
  - Like exceptions in programming languages
- Solution: result of type `option`
- Example: search for an element with given properties

fun get-first :: "('a ⇒ bool) ⇒ 'a list ⇒ 'a option"
where "get-first - [] = None"
| "get-first P (x # xs) =
(if P x then Some x else get-first P xs)"
lemma get-finds-existing:
"(∃ x ∈ set xs. P x) ⟹ get-first P xs ≠ None"
by (induct xs) auto

Type `map`

- Abbreviation: $'a \rightharpoonup 'b$ is really only $'a \Rightarrow 'b\,\mathsf{option}$
- Introduces notation for concrete maps & their updates

  $[''x'' \mapsto 1, ''y'' \mapsto 2]\,''x'' = \mathsf{Some}\,1$
  $(f\,(''x'' \mapsto 1))\,''y'' = f\,''y''$

- `dom` and `ran` give domain and range of a map

  $\mathsf{dom}\,[''x'' \mapsto 1, ''y'' \mapsto 2] = \{''x'', ''y''\}$
  $\mathsf{ran}\,[''x'' \mapsto 1, ''y'' \mapsto 2] = \{1, 2\}$

- Example

  types vals = "string ⇀ int"
  datatype exp =
  Var string
  | Const int
  | Plus exp exp

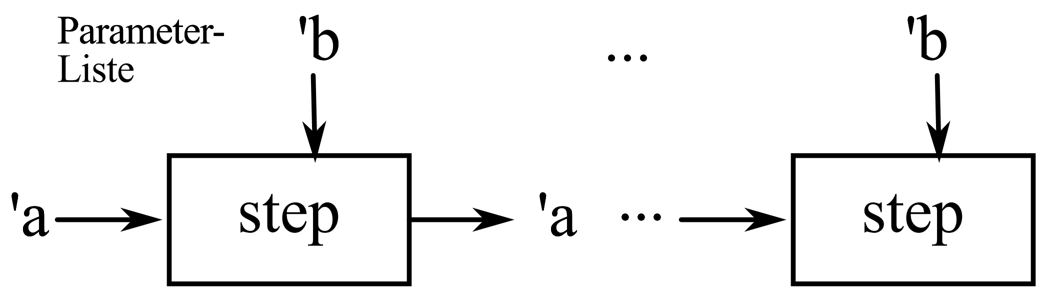
Evaluating Expressions

```
fun eval :: "vals ⇒ exp ⇒ int option"
where "eval v (Var x) = v x"
| "eval v (Const n) = Some n"
| "eval v (Plus a b) =
  (case eval v a of
    None ⇒ None
   | Some x ⇒
    (case eval v b of
     None ⇒ None
    | Some y ⇒ Some (x+y)))"

fun exp-vars :: "exp ⇒ string set"
where "exp-vars (Var x) = {x}"
 | "exp-vars (Const -) = {}"
 | "exp-vars (Plus a b) = exp-vars a ∪ exp-vars b"
```

Type of Lists

- Structure: well-known with `hd`, `tl`, ...
- Standard operations
  - `map`: lift operation to list argument-wise
  - `fold`: primitive / structural recursion over list
- Special case: `foldl` is lifting of state transitions
  - Type $('a \Rightarrow 'b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\,\mathrm{list} \Rightarrow 'a$
  - State is $'a$
  - Transition $('a \Rightarrow 'b \Rightarrow 'a)$



Example: Execution of Instructions

datatype instr = Init int | Add int | Mul int
types state = int

fun exec-instr :: "state ⇒ instr ⇒ state"
where "exec-instr - (Init i) = i"
| "exec-instr s (Add i) = s + i"
| "exec-instr s (Mul i) = s * i"

types block = "instr list"

definition
"exec-block state b ≡ foldl exec-instr state b"

lemma
"exec-block 0 [Init 1, Add 2, Mul 5] = 15"
by (simp add: exec-block-def)

Today

- Additional material on functions
  - Termination proofs
  - Induction over the call depth
  - Generalization of inductive statements
- Tail recursion
  - Relation to imperative programming
  - Techniques for inductive correctness proofs
- Standard constructions in Isabelle
  - Partial functions: `option` & `map`
  - States and transitions `fold`

References

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, Cambridge, 1998.