Interactive Software Verification

Spring Term 2013

Holger Gast

gasth@in.tum.de

23.4.2013

Organisational Remarks

- Every worksheet will have 10 points that are distributed among the exercises according to difficulty or effort required.
- Work in teams of 2.
- Send the solution to Lars Noschinski (see course web page)
- **Clearly indicate the team in all submissions**
- If you reach 40% of the maximal points, your grade in the exam will be improved by 0.3 to obtain the final grade.

Recap: Correctness of `ArrayList`

```
public class ArrayList<E> {
  private Object[] elementData;
  private int size;
  public boolean add(E e) {
   ensureCapacity(size + 1);
   // Can insert the new element since enough space is 'free',
   // i.e. the element 'size' is not occupied since
   // elementData.length > size, i.e. there will not be
   // ArrayOutOfBoundsException
   // Reason: ensureCap guarantees .length >= size + 1
   elementData[size++] = e;
   return true;
  }
  public void ensureCapacity(int minCapacity) {
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
      // minCapacity > elementData.length
      Object oldData[] = elementData;
      int newCapacity = (oldCapacity * 3)/2 + 1;
      if (newCapacity < minCapacity)
        newCapacity = minCapacity;
      elementData = Arrays.copyOf(elementData, newCapacity);
      // elementData.length = newCapacity && newCapacity >= minCapacity.
    } else {
      // elementData.length >= minCapacity
    }
    // at this point *in any case* elementData.length >= minCapacity;
  }
}
```

Recap: The Big Picture

Correctness of Programs
Solve verification conditions Arguments about application domain
Hoare Logic
Verification rules for language constructs Generator for verification conditions
Semantics
Define meaning of programs Describe behaviour of programs

Today: Basics of Isabelle

- Syntax of Isabelle/HOL
- Definitions
  - Constants
  - Functions
  - Data types
- Proofs
  - Unfolding definitions
  - Applying equalities and function definitions
  - Quantifiers and variables
  - Induction

HOL Syntax

Syntax of Isabelle/HOL

- HOL: higher order logic ⇨ Δ "usual" predicate logic s.u.
- Basis: typed terms

Term	Type	Meaning
c	T	Constants with given type T
x	T	Variables of giveb type T
$\lambda x.t$	$S \Rightarrow T$	Anonymous function ($t: T$ for $x: S$)
$f\ a$	S	Function application (für $f: S \Rightarrow T$, $a: S$)

- Types can be made explicit with ::
- Type inference computes type (≈ from those of constants)

Functions with Several Arguments

- Java/C/. . . : arguments in parentheses
- HOL functions appear to have only one argument
- Solution: instead of $(s_1 \ldots s_n) \Rightarrow t$ write

$$(s_1 \Rightarrow (s_2 \Rightarrow \ldots (s_n \Rightarrow t)))$$

- Instead of $f(a_1 \ldots a_n)$, just write

$$f\,a_1\,a_2 \cdots a_n$$

⇒ "‘Currying"’ (german "‘Schönfinkeln"’)

- Benefit: simpler definitions and recursion steps
- Loss: needs a bit of getting used to

"Usual" First-order Predicate Logic

- "Logical statements" built at three levels
- Terms
  - Variables, constants
  - Primitive arithmetic on values ($+$, $-$, etc.)
  - Example: $(i + 4) * j$
- Predicates
  - Atomic statements about values/terms
  - Examples: $\texttt{even}(i + 1)$, $\texttt{divisible}(i, j)$
- Formulae (or: formulas)
  - Connectives ($\wedge$, $\vee$, $\neg$, $\longrightarrow$, $\ldots$)
  - Quantifiers ($\forall$, $\exists$)

Isabelle: Terms & Logic

- Statements in HOL are just terms of type `bool`
- Predicates are just functions $\ldots \Rightarrow$ `bool`

$$\texttt{even}\colon \texttt{int} \Rightarrow \texttt{bool}$$

$$\texttt{divisible}\colon \texttt{int} \Rightarrow \texttt{int} \Rightarrow \texttt{bool}$$

- Connectives are just functions $\texttt{bool} \Rightarrow \ldots \Rightarrow \texttt{bool}$

$$\wedge\colon \texttt{bool} \Rightarrow \texttt{bool} \Rightarrow \texttt{bool}$$

$$\longrightarrow\colon \texttt{bool} \Rightarrow \texttt{bool} \Rightarrow \texttt{bool}$$

- Only nicer infix notation
- Example: $A \longrightarrow B$ is really (`op-->` A B)
- Quantifiers — hmm, just a sec

Meta-Logic & Goals

- Isabelle provides built-in connectives and qantifiers

$$\begin{array}{cl} \Longrightarrow & \text{Implication (read as ''`provable''')} \\ \bigwedge & \text{``for all''} \\ \equiv & \text{equality} \end{array}$$

- Nested implications & currying $[\![\mathsf{P}_1;\cdots;\mathsf{P}_n]\!]\Longrightarrow\mathsf{Q}$
  Read: "Prove Q from $P_1 \ldots P_n$"'
  Or: $[\![$Information $]\!]\Longrightarrow$to be derived
- Type for "statement" here is `prop`
- Application: statements about `bool` terms
- **Note:** use `isabelle jedit -m brackets` to obtain display

Examples

- $\forall \varepsilon.\, \exists \delta.\, \mathrm{abs}\,(f\,(x + \delta) - f\,x) \le \varepsilon$
- $[\![x \in A; \forall x \in A.\, x > 0]\!] \Longrightarrow x > 0$
- $\text{bt-contains}\,(\text{bt-insert}\,x\,t)\,y = (y = x \vee \text{bt-contains}\,t\,y)$
- $\text{bt-sorted}\,t \Longrightarrow \text{bt-sorted}\,(\text{bt-insert}\,x\,t)$

Free & bound Variables

- A variable is bound if it occurs as a λ-parameter further outward in the term (underlined in the examples).
- A variable is free if it is not bound (example: overstrike)

$$(\lambda x.\ \underline{x} + y)\overline{a} * b$$

- More precisely: must talk about occurrences of variables

$$(\lambda x.\ \underline{x} + y)\overline{x} * b$$

- Isabelle output: different markup for free/bound
- Quantifiers constitute binding (see below)

Free Variables and Logic

- Free variables in theorems denote arbitrary values

$\Rightarrow$ Implicit forall-quantifierr

lemma divisible-transitive:
" $\llbracket$ divisible a b; divisible b c $\rrbracket \Longrightarrow$ divisible a c"

- Technical variant: unknowns
  - Marked by prefix ?
  - Will be replaced automatically if necessary
- Free variables in proven theorems become unkowns

$\llbracket$divisible ?a ?b; divisible ?b ?c$\rrbracket \Longrightarrow$ divisible ?a ?c

$\Rightarrow$ Lemma becomes a rule in Isabelle

- Application of rules substitues unknowns

Quantifiers as Constants

- Recall: $A \longrightarrow B$ internally is `(op--> A B)`
- Analogously for quantifiers

$$\forall x.\ P x \qquad \text{internally becomes} \qquad \texttt{All}\ (\lambda x.\ P\, x)$$

  - $P\, x$ is a statements about any x

  $\Rightarrow$ Intuition: obtain a boolean value for each x

  $\Rightarrow$ Have boolean function $\lambda x.\, P\, x$

  - Quantifier is constant `All` that defines the overall result

- Nice: Binding by quantifier is exactly binding by λ

$\Rightarrow$ Higher-order Abstract Syntax (HOAS) [4]

Constants & Definitions

- Any kind of definition introduces
  - One or more constants
  - Corresponding theorems to reason about the constant

- Example: divisibility as existence of divisor

  definition
  divisible :: "int ⇒ int ⇒ bool"
  where
  "divisible a b ≡ ∃ k. a = k * b"

  - New constant divisible :: "int ⇒ int ⇒ bool"
  - Theorem `divisible_def`: divisible ?a ?b ≡ ∃ k. ?a = k * ?b

Example: Transitivity of `divisible`

lemma divisible-transitive:
" $\llbracket$ divisible a b; divisible b c $\rrbracket \Longrightarrow$ divisible a c"

- Goal

  proof (prove): step 0
  goal (1 subgoal):
  1. $\llbracket$divisible a b; divisible b c$\rrbracket \Longrightarrow$ divisible a c

- **apply** (unfold divisible-def): use definition

  1. $\llbracket \exists\, k.\ a = k * b; \exists\, k.\ b = k * c \rrbracket \Longrightarrow \exists\, k.\ a = k * c$

  Remark: unknowns in definition get replaced

- **apply** auto : try automated proofs first
- **done** : yes, done!
- Alternative: `auto` uses the simplifiers for equalities

  apply (auto simp add: divisible-def)

Proofs

Proofs in Isabelle

- Isabelle maintains open goals
- `lemma`/`theorem` generates initial goal from lemma
- `apply` method
  - Applies method
  - Leaves remaining goals
  - Methods: automated provers, single proof steps, . . .
- If no further goals left: `done`
- Alternative: Isar [3, 5]
  - Resembles / formalizes mathematical proofs
  - Sub-proofs executes in parallel
  - May be more human-readable
  - Usually not used in software verification

Prover: `simp`

- Does term rewriting [1, 2]
  - Uses equalities $l = r$ as rewrite rules
  - Visits a term t (bottom up)
  - At each possible redex t'
    - Find a possible subtitution σ for unknowns in l
    - Such that $t' = \sigma\, l$
    - Replace t' by $\sigma\, r$

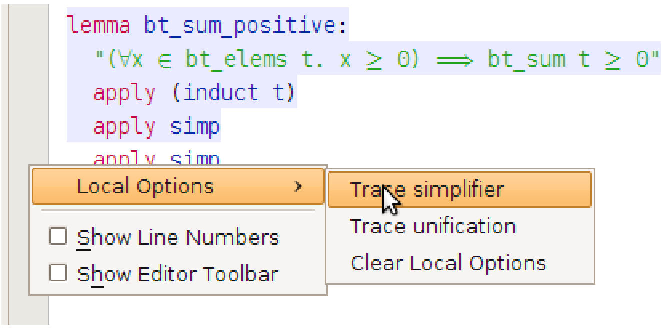
    If no such σ exists, do nothing
  - Repeat exhaustively
- Detail: treats $\equiv$ as =
- Extension: conditional rewriting

$$[\![P_1 \cdots P_n]\!] \Longrightarrow l = r$$

  Applies rule only if $\sigma\, P_i = \texttt{True}$ can be proven by rewriting

The Rule-Set for `simp`

- Rewriting rules given as simpset
- Actual `simpset` is composed from
  - Context of the application (`simp` declarations at theorems)
  - Modifications by `add:`/`del:`/`only:`
  - Local rules extracted from goal's premises
- To show/trace application of rules: trace simplifier



- Alternative: `using [[simp_trace]]` in proof

Example: Transitivity of Divisibility

lemma
"$\llbracket$ divisible a b; divisible b c $\rrbracket \Longrightarrow$ divisible a c"
by (auto simp add: divisible-def)

[0]Adding rewrite rule "HOLBasics.divisible-def":
divisible ?a ?b $\equiv \exists$ k. ?a = k * ?b
[1]SIMPLIFIER INVOKED ON THE FOLLOWING TERM:
$\llbracket$divisible a b; divisible b c$\rrbracket \Longrightarrow$ divisible a c
[1]Applying instance of rewrite rule "HOLBasics.divisible-def":
divisible ?a ?b $\equiv \exists$ k. ?a = k * ?b
[1]Rewriting:
divisible a b $\equiv \exists$ k. a = k * b
. . .

[1]SIMPLIFIER INVOKED ON THE FOLLOWING TERM:
$\bigwedge$k. $\llbracket$a = k * b; $\exists$ k. b = k * c$\rrbracket \Longrightarrow \exists$ k. a = k * c
[1]Adding rewrite rule "??.unknown":
a $\equiv$ k * b
[1]Applying instance of rewrite rule "??.unknown":
a $\equiv$ k * b
[1]SIMPLIFIER INVOKED ON THE FOLLOWING TERM:
$\bigwedge$k ka. b = ka * c $\Longrightarrow \exists$ ka. k * b = ka * c
. . .
[1]Applying instance of rewrite rule "HOL.simp-thms-38":
$\exists$ x. ?t1 = x $\equiv$ True

Bitte ausprobieren

Elementary Proof Steps

- Suppose a goal is given

$$\llbracket P_1 \cdots P_n \rrbracket \Longrightarrow Q$$

- Deduction from P_i
  - `frule`/`drule` (forward/destruct)

$$\llbracket \hat{P}, Q_1' \cdots Q_n' \rrbracket \Longrightarrow \hat{P}'$$

  - `erule` (elim)

$$\llbracket \hat{P},\ \llbracket P_1' \cdots P_n' \rrbracket \Longrightarrow Q \rrbracket \Longrightarrow Q$$

- Refining Q
  - `rule` (apply logical rule/theorem)

$$\llbracket Q_1' \cdots Q_n' \rrbracket \Longrightarrow \hat{Q}$$

- In each case: substitute unknowns on-the-fly

Example: Conjunction/Disjunction

- `conjI` says what must be proven to introduce $\wedge$

$$\llbracket ?P; ?Q \rrbracket \Longrightarrow ?P \wedge ?Q$$

- `conjE`: what information do we gain by eliminating/using $\wedge$

$$\llbracket ?P \wedge ?Q; \llbracket ?P; ?Q \rrbracket \Longrightarrow ?R \rrbracket \Longrightarrow ?R$$

- `conjunct1`, `conjunct2` characterize logical implications

$$?P \wedge ?Q \Longrightarrow ?P$$

$$?P \wedge ?Q \Longrightarrow ?Q$$

- Analogously for $\vee$: `disjI1`, `disjI2`, `disjE`

$$?P \Longrightarrow ?P \vee ?Q \qquad ?Q \Longrightarrow ?P \vee ?Q$$

$$\llbracket ?P \vee ?Q; ?P \Longrightarrow ?R; ?Q \Longrightarrow ?R \rrbracket \Longrightarrow ?R$$

Example: Quantifiers

- $\forall$-quantifier: `allI`, `allE`, `spec`

$$(\bigwedge x.\ ?P\ x) \Longrightarrow \forall x.\ ?P\ x$$

$$[\![\forall x.\ ?P\ x;\ ?P\ ?x \Longrightarrow ?R]\!] \Longrightarrow ?R$$

$$\forall x.\ ?P\ x \Longrightarrow ?P\ ?x$$

- $\exists$-quantifier: `exI`, `exE`

$$?P\ ?x \Longrightarrow \exists x.\ ?P\ x$$

$$[\![\exists x.\ ?P\ x;\ \bigwedge x.\ ?P\ x \Longrightarrow ?Q]\!] \Longrightarrow ?Q$$

Example: Implication

- Note: Connection between $\longrightarrow$ and $\Longrightarrow$
- `impI`: assume left-hand side in the proof annehmen

$$(?P \Longrightarrow ?Q) \Longrightarrow ?P \longrightarrow ?Q$$

- `impE`: when using an implication, we need to show the left-hand side

$$[\![?P \longrightarrow ?Q; ?P; ?Q \Longrightarrow ?R]\!] \Longrightarrow ?R$$

- `mp` (modus ponens): classical/natural use of implications

$$[\![?P \longrightarrow ?Q; ?P]\!] \Longrightarrow ?Q$$

Example-Proofs

lemma rules-example:
"$\llbracket\, \exists x.\ x \in A; \forall x \in B.\ P\,x; A \subseteq B \,\rrbracket \Longrightarrow \exists x.\ P\,x$"
apply (erule exE)
apply (drule subsetD)
apply assumption
apply (drule bspec)
apply assumption
apply (rule exI)
apply assumption
done

Please try out and examine goals

Provers: `auto` et al.

- Combine
  - Proof search with intro/elim/dest rules
  - Rewriting by `simp`
- Rule set given in claset (classical rules)
- Heuristics
  - Solve "obvious" goals directly
  - Leave non-obvious goals for the user
- Similarly
  - `force`: like `auto`, but tries to solve all goals
  - `fastforce`: another variant, different heuristics
  - `fast`: no simplifier

Examples of Automated Proofs

lemma rules-example-auto1:
$''\llbracket \exists x.\ x \in A; \forall x \in B.\ P\,x; A \subseteq B \rrbracket \Longrightarrow \exists x.\ P\,x''$
by auto

lemma rules-example-auto2:
$''\llbracket \exists x.\ x \in A; \forall x \in B.\ P\,x; A \subseteq B \rrbracket \Longrightarrow \exists x.\ P\,x''$
by fast

lemma rules-example-auto3:
$''\llbracket \exists x.\ x \in A; \forall x \in B.\ P\,x; A \subseteq B \rrbracket \Longrightarrow \exists x.\ P\,x''$
by blast

Functions

Remark: Foundation of Induction& Recursion

- Basic idea
  - Choose a set A with a well-founded order $\preceq$
  - Statement/defintion for "smallest" elements first
  - Statement/definition about "successively larger" elements

  $\Rightarrow$ Cover the entire set A
- Well-known examples
  - Natural induction (over $\mathbb{N}$):
    First 0, then $\forall n.\, P\, n \longrightarrow P\,(n+1)$
  - Structual induction: trees, lists, ...
    - First leafs/empty lists
    - Then along construction steps
      $\forall l\, r.\, P\, l \wedge P\, r \implies P\,(\mathtt{Node}\; l\; r)$

Inductive Data Types

- Command `datatype` introduces
  - New type
  - Constructors for elements of the type

    datatype bt =
      Leaf
    | Node bt int bt

- Constants / constructors

  "Leaf" :: "bt
  "Node" :: "bt ⇒ int ⇒ bt ⇒ bt"

- And theorems
  - Induction

$$\llbracket ?P\ \mathrm{Leaf};\ \bigwedge l\ v\ r.\ \llbracket ?P\ l;\ ?P\ r \rrbracket \Longrightarrow ?P\ (\mathrm{Node}\ l\ v\ r) \rrbracket \Longrightarrow ?P\ ?bt$$

  - Case-distinction by the `case` method
  - Injectivity & inequality of constructors for `simp`

Functions in HOL

- Anonymous functions: $\lambda x.\ e$ (where x can occur in e)
- Constants

  ''divisible a b $\equiv$ $\exists$ k. a = k * b''

  or equivalently (almost)

  ''divisible' $\equiv$ λa b. $\exists$ k. a = k * b''

- Of course we have

  lemma divisible-eqv:
  ''divisible a b $\longleftrightarrow$ divisible' a b''
  by (simp add: divisible-def divisible'-def)

- By technical differences in definition theorems

  divisible ?a ?b $\equiv$ $\exists$ k. ?a = k * ?b
  divisible' $\equiv$ λa b. $\exists$ k. a = k * b

$\Rightarrow$ divisible-def can be unfolded only with two arguments

Recursive Functions (`fun`)

- Follows functional languages (ML/Haskell/Scheme/. . .)
- Main difference: functions must terminate

⇒ Isabelle requires a termination proof

  - `fun` tries to find proof automatically
  - `function` allows user to make proof explicit

- `fun` works reliably in case of
  - Structural recursion of data types
  - Natural number arguments getting "obviously" smaller

Theorems about Functions

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 (idea: over call depth) f`.induct`

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

Hinweis: zufällig analog zu `bt.induct`

- Technical: application of rules in `induct`

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

➪ Substitutes unknowns ?P, ?a

Example: Induction on 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 a statement about the 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)

What happens next . . .

- You know the basics
  - Syntax Isabelle/HOL
  - Definitions
  - Proofs: elementary steps, simplifier, . . .
  - Data types & recursive functions
- Next up: examples & techniques for proofs by induction
  - Basic examples: structural induction
  - Termination proofs
  - Tail recursion $\approx$ imperative loops

References

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

[2] Reinhard Bündgen. *Termersetzungssysteme—Theorie, Implementierung, Anwendung*. Vieweg, Braunschweig/Wiesbaden, 1998.

[3] Tobias Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 259–278. Springer, 2003.

[4] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation (PLDI)*, volume 23 (7) of *SIGPLAN Notices*, pages 199–208, Atlanta, Georgia, June 22-24 1988. ACM Press.

[5] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.