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

```java
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 $\iff$ “usual” predicate logic s.u.
- Basis: typed terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>$T$</td>
<td>Constants with given type $T$</td>
</tr>
<tr>
<td>$x$</td>
<td>$T$</td>
<td>Variables of given type $T$</td>
</tr>
<tr>
<td>$\lambda x.t$</td>
<td>$S \Rightarrow T$</td>
<td>Anonymous function ($t: T$ for $x: S$)</td>
</tr>
<tr>
<td>$f\ a$</td>
<td>$S$</td>
<td>Function application ($f: S \Rightarrow T$, $a: S$)</td>
</tr>
</tbody>
</table>

- Types can be made explicit with ::
- Type inference computes type ($\approx$ 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: \((i + 4) \times j\)
- Predicates
  - Atomic statements about values/terms
  - Examples: even\((i + 1)\), divisible\((i, j)\)
- Formulae (or: formulas)
  - Connectives (\(\land, \lor, \neg, \rightarrow\), . . .
  - Quantifiers (\(\forall, \exists\))
• Statements in HOL are just terms of type bool
• Predicates are just functions ... ⇒ bool
  \[
even : \text{int} \Rightarrow \text{bool} \\
divisible : \text{int} \Rightarrow \text{int} \Rightarrow \text{bool}
\]
• Connectives are just functions bool ⇒ ... ⇒ bool
  \[
\wedge : \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool} \\
\rightarrow : \text{bool} \Rightarrow \text{bool} \Rightarrow \text{bool}
\]
• Only nicer infix notation
• Example: \(A \rightarrow B\) is really (\text{op--->} A B)
• Quantifiers — hmm, just a sec
Meta-Logic & Goals

- Isabelle provides built-in connectives and quantifiers

  \[ \rightarrow \text{ Implication (read as "'provable''") } \]
  \[ \land \text{ "for all" } \]
  \[ \equiv \text{ equality } \]

- Nested implications & currying \([P_1; \cdots; P_n] \rightarrow Q\)
  Read: “Prove \(Q\) from \(P_1 \ldots P_n\)”’
  Or: \([\text{Information } ] \rightarrow \text{to be derived}\)

- Type for “statement” here is prop

- Application: statements about bool terms

- **Note:** use isabelle jedit -m brackets to obtain display
Examples

- $\forall \varepsilon. \exists \delta. \text{abs}(f(x + \delta) - f(x)) \leq \varepsilon$

- $[\ x \in A; \forall x \in A. x > 0 \] \implies x > 0$

- $\text{bt-contains}(\text{bt-insert} \times t) y = (y = x \lor \text{bt-contains} t y)$

- $\text{bt-sorted} t \implies \text{bt-sorted} (\text{bt-insert} \times t)$
Free & bound Variables

- A variable is **bound** if it occurs as a $\lambda$-parameter further outward in the term (underlined in the examples).

- A variable is **free** if it is not bound (example: overstrike)

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

- More precisely: must talk about **occurrences of variables**

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

- 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-quantifier

  **Lemma** divisible-transitive:
  
  \[
  \text{divisible a b; divisible b c} \implies \text{divisible a c}
  \]

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

  \[
  \text{divisible ?a ?b; divisible ?b ?c} \implies \text{divisible ?a ?c}
  \]

  - \( \Rightarrow \) Lemma becomes a **rule** in Isabelle

  - Application of rules substitutes unknowns
Quantifiers as Constants

• Recall: $A \rightarrow B$ internally is $(\text{op---> A B})$

• Analogously for quantifiers
  
  $\forall x. \, P \, x$ internally becomes $\text{All} \, (\lambda x. \, P \, x)$

• $P \, x$ 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 $\text{All}$ that defines the overall result

• Nice: Binding by quantifier is exactly binding by $\lambda$

  $\Rightarrow$ Higher-order Abstract Syntax (HOAS) [4]
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:
"\[ \text{divisible} \ a \ b; \text{divisible} \ b \ c \implies \text{divisible} \ a \ c \]"

- **Goal**

  proof (prove): step 0
  goal (1 subgoal):
  1. \[ \text{divisible} \ a \ b; \text{divisible} \ b \ c \implies \text{divisible} \ a \ c \]

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

  1. \[ \exists \ k. \ a = k \cdot b; \exists \ k. \ b = k \cdot c \implies \exists \ k. \ a = k \cdot c \]

  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 substitution \( \sigma \) for unknowns in \( l \)
    - Such that \( t' = \sigma l \)
    - Replace \( t' \) by \( \sigma r \)
      - If no such \( \sigma \) exists, do nothing
  - Repeat exhaustively
- Detail: treats \( \equiv \) as \( = \)
- Extension: conditional rewriting

\[
[P_1 \cdots P_n] \Rightarrow l = r
\]

Applies rule only if \( \sigma P_i = \text{True} \) 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
" [ divisible a b; divisible b c ] \implies 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:
[ divisible a b; divisible b c ] \implies 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:
\land k. [ a = k * b; \exists k. b = k * c ] \implies \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:
\land ka. b = ka * c \implies \exists ka. k * b = ka * c

. . .

[1] Applying instance of rewrite rule "HOL.simp-thms-38":
\exists x. ?t1 = x \equiv True
Elementary Proof Steps

- Suppose a goal is given

\[ [P_1 \cdots P_n] \rightarrow Q \]

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

\[ [\hat{P}, Q'_1 \cdots Q'_n] \rightarrow \hat{P}' \]

  - erule (elim)

\[ [\hat{P}, [P'_1 \cdots P'_n] \rightarrow Q] \rightarrow Q \]

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

\[ [Q'_1 \cdots Q'_n] \rightarrow \hat{Q} \]

- In each case: substitute unknowns on-the-fly
Example: Conjunction/Disjunction

- **conjI** says what must be proven to introduce $\land$
  
  $$[?P; ?Q] \implies ?P \land ?Q$$

- **conjE**: what information do we gain by eliminating/using $\land$
  
  $$[?P \land ?Q; [?P; ?Q] \implies ?R] \implies ?R$$

- **conjunct1, conjunct2** characterize logical implications

  $$?P \land ?Q \implies ?P$$

  $$?P \land ?Q \implies ?Q$$

- Analogously for $\lor$: **disjI1, disjI2, disjE**

  $$?P \implies ?P \lor ?Q$$

  $$?Q \implies ?P \lor ?Q$$

  $$[?P \lor ?Q; ?P \implies ?R; ?Q \implies ?R] \implies ?R$$
Example: Quantifiers

- \( \forall \)-quantifier: allI, allE, spec
  
  \[(\bigwedge x. \ ?P\ x) \implies \forall x. \ ?P\ x\]

  \[\forall x. \ ?P\ x; \ ?P\ ?x \implies \ ?R] \implies \ ?R\]

  \[\forall x. \ ?P\ x \implies \ ?P\ ?x\]

- \( \exists \)-quantifier: exI, exE

  \[?P\ ?x \implies \exists x. \ ?P\ x\]

  \[\exists x. \ ?P\ x; \bigwedge x. \ ?P\ x \implies \ ?Q] \implies \ ?Q\]
Example: Implication

- Note: Connection between \( \rightarrow \) and \( \Rightarrow \)

- \text{impI}: assume left-hand side in the proof annehmen
  \[ (?P \Rightarrow ?Q) \Rightarrow ?P \rightarrow ?Q \]

- \text{impE}: when using an implication, we need to show the
  left-hand side
  \[ [?P \rightarrow ?Q; ?P; ?Q \Rightarrow ?R] \Rightarrow ?R \]

- \text{mp} (modus ponens): classical/natural use of implications
  \[ [?P \rightarrow ?Q; ?P] \Rightarrow ?Q \]
**Example-Proofs**

**Lemma rules-example:**

"$[\exists x. \ x \in A; \ \forall x \in B. \ P x; \ A \subseteq B ] \implies \exists x. \ P x$"

**apply** (erule exE)
**apply** (drule subsetD)
**apply** assumption
**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:
  "[ \exists x. x \in A; \forall x \in B. P x; A \subseteq B ] \implies \exists x. P x"

by auto

lemma rules-example-auto2:
  "[ \exists x. x \in A; \forall x \in B. P x; A \subseteq B ] \implies \exists x. P x"

by fast

lemma rules-example-auto3:
  "[ \exists x. x \in A; \forall x \in B. P x; A \subseteq B ] \implies \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/definition 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 $\forall n. P n \rightarrow P(n + 1)$
  • Structural induction: trees, lists, . . .
    • First leafs/empty lists
    • Then along construction steps
      $\forall l r. P l \land P r \Rightarrow P(\text{Node } l r)$
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
    ```
    [?P Leaf; \l v r. [?P l; ?P r] ⇒ ?P (Node l v r)] ⇒ ?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
  
  "\text{divisible}\ a\ b \equiv \exists\ k.\ a = k \times b"

  or equivalently (almost)

  "\text{divisible}' \equiv \lambda a\ b.\ \exists\ k.\ a = k \times b"

- Of course we have

  \textbf{lemma divisible-eqv:}

  "\text{divisible}\ a\ b \iff \text{divisible}'\ a\ b"

  \textbf{by (simp add: divisible-def divisible'-def)}

- By technical differences in definition theorems

  \( \text{divisible}\ ?a\ ?b \equiv \exists\ k.\ ?a = k \times ?b \)

  \( \text{divisible}' \equiv \lambda a\ b.\ \exists\ k.\ a = k \times b \)

  \( \Rightarrow \) divisible-def can be unfolded only with two arguments
Recursive Functions ($\texttt{fun}$)

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

$\Rightarrow$ Isabelle requires a termination proof

- $\texttt{fun}$ tries to find proof automatically
- $\texttt{function}$ allows user to make proof explicit

- $\texttt{fun}$ works reliably in case of
  - Structural recursion of data types
  - Natural number arguments getting “obviously” smaller
Theorems about Functions

\[ \text{fun in-order :: } \text{"bt } \Rightarrow \text{ int list"} \]
\[ \text{where} \]
\[ \text{"in-order Leaf = []"} \]
\[ \text{| " in-order (Node l v r) = in-order l @ [v] @ in-order r"} \]

• Rewrite-rules \$f\$.simps
  \[ \text{in-order Leaf = []} \]
  \[ \text{in-order (Node ?l ?v ?r) = in-order ?l @ [?v] @ in-order ?r} \]

• Induction (idea: over call depth) \$f\$.induct
  \[ \text{[?P Leaf; } \land \text{l v r. [?P l; ?P r] } \Rightarrow \text{ ?P (Node l v r)] } \Rightarrow \text{ ?P ?a} \]

Hinweis: zufällig analog zu \$bt\$.induct

• Technical: application of rules in induct
  apply (induct rule: \textit{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. ∀ l v r. [[int (length (in-order l)) = ninner l;
   int (length (in-order r)) = ninner r]
   ⇒ 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. ∀ l v r. [[int (length (in-order l)) = ninner l;
   int (length (in-order r)) = ninner r]
   ⇒ 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 ≈ imperative loops
References


