Interactive Software Verification

Spring Term 2013

Holger Gast
gasth@in.tum.de

16.4.2013
Goals of the Lecture

• Formulate precise statements and arguments about programs

• Develop a language for talking about correctness

• Understand fundamental concepts of software verification

• Explain your thought to a computer / program
  • “To understand this properly, we implement it”
  ⇒ “To understand this precisely, we explain it to Isabelle”

• Online demo: double/even; add
Motivation for Verification

- Debugging & maintenance are time-consuming & expensive
- Embedded systems
  - Control safety-critical devices (air bag, planes)
  - Replacement only by physical updates
- Foundational routines \(\Rightarrow\) multiplication of errors
  - Libraries & data structures
  - Infrastructure: memory manager, garbage collectors
- Basic software
  - Operating systems
  - Hypervisors
  - Compilers
Examples of Interactive Verification

- C compiler [4, 15]
- L4 micro-kernel [12, 11, 21]
- Safety of robot movements [16]
- JavaCard API [8]
- Garbage collectors [17]
- Security protocols [20]
- Algorithmic questions [6, 18]
Examples of Automatic Verification

- Libraries of data structures [23]
- Garbage collectors [10, 14]
- MS Hypervisor (VerifsoftXT project) [1]
- Sanity checks for Java programs [9]
- Concurrent programs [13]
- Algorithms on lists, trees, ... [7, 22, 3]
Why get involved at all?

- “Automatic” methods still require algorithmic insights
  - Annotating programs with assertions about state
  - Maintaining ghost state with logical/abstract state
  - Stating lemmas / auxiliary assertions

- The proof process needs to be understood in some detail
  - Control the proof search of SMT solvers [19]
  - Provability of given goals often not immediately clear [5]
  - Beginners need support by experts [2]

⇒ This lecture: explore concepts by interactive proof
How “Interactive” are Interactive Provers?

• Interactive prover: Isabelle

• User
  • Defines notions / terms of interest
  • States theorems about definitions
  • Guides Isabelle towards finding proofs

• Extensive automatic proof support
  • Term rewriting (proofs about equality)
  • Resolution in first order predicate logic (FOL)
  • Tableau prover for higher order logic (HOL)

• Proof automation will be discussed as necessary
What does “correct” mean anyway?

```java
public class ArrayList<E> {
    private Object[] data;
    private int size;

    public void add(E e) {
        ensureCapacity(size + 1);
        data[size++] = e;
    }

    public void ensureCapacity(int minCapacity) {
        int oldCap = data.length;
        if (minCapacity > oldCap) {
            Object oldData[] = data;
            int newCap = (oldCap * 3) / 2 + 1;
            if (newCap < minCapacity)
                newCap = minCapacity;
            data = Arrays.copyOf(data, newCap);
        }
    }
}
```
ArrayList – Inner Structure

- The first `size` slots contain object references (or null)
- The remaining slots in `elementData` are null (GC!)
Arguments about Correctness

Question: how can we convince a very sceptical person that our program is correct? About what do we argue?

- Content of object fields at specific points of program
- Result of method calls
- Relation between parameters and return value of methods
- Information gained by case distinctions
- Result / effect of assignment
- Consistency of objects
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;
    }
}

(created interactively)
### 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
## The Case for Java

<table>
<thead>
<tr>
<th>Correctness of Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve Proof Obligations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hoare Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-/Post-Conditions for if, while, try, catch</td>
</tr>
<tr>
<td>Pre-/Post-Conditions in connection with inheritance</td>
</tr>
<tr>
<td>Disjointness of objects and fields</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects, References, Arrays</td>
</tr>
<tr>
<td>Expressions, Statements, Exceptions</td>
</tr>
<tr>
<td>Dynamic Dispatch</td>
</tr>
</tbody>
</table>
The Case for C

<table>
<thead>
<tr>
<th>Correctness of Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve proof obligations</td>
</tr>
<tr>
<td>Aliasing of pointers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hoare Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions, . . . as in Java</td>
</tr>
<tr>
<td>Untyped memory access / casts</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte-addressed memory, bit manipulations</td>
</tr>
<tr>
<td>Allocation of memory regions</td>
</tr>
<tr>
<td>Pointer arithmetic and -casts</td>
</tr>
<tr>
<td>Pointers to local variables / struct fields</td>
</tr>
</tbody>
</table>
Demo: Multiplication by Addition

```
verify-statement mult-by-add
vars: {*
    int i;
    int j;
    int r;
  *}
pre: "i = I ∧ j = J ∧ i ≥ 0 ∧ j ≥ 0"
post: "r = I * J"
{*
  r = 0;
 /*@ r = (I - i) * j ∧ j = J ∧ i ≥ 0*/
while (i > 0) {
    r = r + j;
    i = i - 1;
}
*}
```
The Plan for this Lecture

• Start small, then advance to the more complex
  • Look at simple (functional) languages
  • Learn interactive proving
  • Define imperative language with variables, heap, etc.
  • Prove properties about heap manipulating programs

• Demonstrate / experience the concepts in the small
  • Lecture presents concepts
  • Reduction to main aspects
  • Implementation & use of core aspects in Isabelle
Start with Isabelle’s Language

<table>
<thead>
<tr>
<th>Correctness of Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proofs about defined functions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verification Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Express statements directly as theorems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive functions (≈ ML, Haskell, Scheme)</td>
</tr>
<tr>
<td>Standard data types (natural numbers, lists, ... )</td>
</tr>
</tbody>
</table>
Example: Binary Trees

- Definition: a tree is either
  - A leaf *or*
  - An inner node with two sub-trees

```plaintext
datatype bt =
  Leaf
| Node bt int bt
```

- Concept: the elements of a tree

```plaintext
fun in-order :: "bt ⇒ int list"
where
"in-order Leaf = []"
| "in-order (Node l v r) = in-order l @ [v] @ in-order r"
definition
"bt-elems t ≡ set (in-order t)"
definition
"bt-contains t x ≡ x ∈ bt-elems t"
```
Algorithm: Search in Tree

- Naive implementation

  ```plaintext
  fun bt-search-full :: “bt ⇒ int ⇒ bool”
  where
  ”bt-search-full Leaf x = False”
  | ”bt-search-full (Node l v r) x =
    (v = x ∨ bt-search-full l x ∨ bt-search-full r x)”
  ```

- Observations

  - We can **program** in Isabelle
  - Semantics: just as in functional languages
  - We can use mathematical concepts, like “∨”, “∃”
export-code bt-search-full in OCaml
module-name BTSearchFull
file "code/btsearchfull.ML"

Generates an executable (OCaml-) program:

type bt = Leaf | Node of bt * int * bt;;

let rec bt_search_full
xa0 x = match xa0, x with Leaf, x -> false
| Node (l, v, r), x ->
    eq_int v x || (bt_search_full l x || bt_search_full r x);;

One can write entire C compilers in this way [4, 15]
Correcteness of Isabelle Functions

• Why is \texttt{bt-search-full} “correct”?
  • It yields “\texttt{True}” iff \(x\) is element of the tree
    \(\Rightarrow\) Given by \texttt{bt-contains}

• Formulate as lemma \(\Rightarrow\) proof obligation (**goals**)
  \texttt{lemma} \texttt{bt-search-full-correct:}
  "\texttt{bt-search-full} \(t\ x\) \(\iff\) \texttt{bt-contains} \(t\ x\)"

• Proof
  \textit{Induction on the tree structure}
  \texttt{apply} (\texttt{induct} \(t\))
  \textit{Use the definitions}
  \texttt{apply} (\texttt{unfold} \texttt{bt-contains-def} \texttt{bt-elems-def})
  \textit{Solve remainder automatically}
  \texttt{apply} \texttt{auto}
  \texttt{done}
Binary Search in Trees

- A tree is **sorted** if its elements by in-order traversal are sorted
  
  **definition**
  
  ”bt-sorted t ≡ sorted (in-order t)”

- Now we can search by divide-and-conquer

  **fun** bt-search-split :: ”bt ⇒ int ⇒ bool”

  **where**
  
  ”bt-search-split Leaf x = False”
  
  | ”bt-search-split (Node l v r) x =
  
  (v = x ∨ (if x < v
  
   then bt-search-split l x
  
   else bt-search-split r x))”

---

Interactive Software Verification, 16.4.2013
Correctness of Binary Search

- For sorted trees the new function is correct

  **Lemma** bt-search-split-correct:
  \[ \text{bt-sorted } t \implies \text{bt-search-split } t \times \iff \text{bt-contains } t \times \]

- Correctness condition as before
- Now requires assumption / pre-condition on sortedness

- Proof
  - Structural induction on \( t \)
  - Auxiliary facts (immediate by definitions)

    **Lemma** bt-sorted-branchesD:
    \[ \text{bt-sorted } (\text{Node } l \; v \; r) \implies \text{bt-sorted } l \land \text{bt-sorted } r \]

    **Lemma** bt-sorted-elemsD:
    \[ \text{bt-sorted } (\text{Node } l \; v \; r) \implies \\
    (\forall x \in \text{bt-elems } l. \; x \leq v) \land (\forall x \in \text{bt-elems } r. \; v \leq x) \]
Consider insertion into a binary tree

```haskell
fun bt-insert :: "[ int, bt ] ⇒ bt"
where
  "bt-insert x Leaf = Node Leaf x Leaf"
| "bt-insert x (Node l v r) =
    (if x = v
      then (Node l v r)
      else if x < v
        then Node (bt-insert x l) v r
        else Node l v (bt-insert x r))"
```

Descend to the “correct” place recursively

Then add the element
Correctness of \texttt{bt\text{-}insert}

- Invariant: sorted trees remain sorted after insertion
  \begin{itemize}
  \item \underline{lemma} \texttt{bt-insert-is-sorted:}
  \begin{quote}
  ”bt\text{-}sorted \, t \implies \text{bt\text{-}sorted (bt\text{-}insert \, x \, t)}”
  \end{quote}
  \end{itemize}

- And the inserted element is actually contained in the tree
  \begin{itemize}
  \item \underline{lemma} \texttt{bt-insert-result-set:}
  \begin{quote}
  ”bt\text{-}elems (bt\text{-}insert \, x \, t) = \{x\} \cup \text{bt\text{-}elems} \, t”
  \end{quote}
  \end{itemize}

- Proofs
  \begin{itemize}
  \item By induction on \( t \)
  \item Auxiliary facts on sorted trees as before
  \end{itemize}
Verwendung der Korrektheitsaussagen

- Now: serveral function applications in a row
- What can we know about the result of:

  \[
  \text{lemma} \quad " \text{bt-sorted } t \implies \text{let } t' = \text{bt-insert } x \ t \ \text{in} \ \\
  \text{let } t'' = \text{bt-insert } y \ t' \ \text{in} \ \\
  \text{bt-search-split } t'' \ x = True"
  \]

- Need auxiliary facts about rows of function applications

  \[
  \text{lemma bt-contains-bt-insert:} \quad " \text{bt-contains (bt-insert } x \ t \text{) } y = (y = x \lor \text{bt-contains } t \ y)"
  \]

  \[
  \text{apply (simp add: bt-contains-def)}
  \quad \text{apply (simp add: bt-insert-result-set)}
  \quad \text{done}
  \]

- Then we can prove the above lemma

  \[
  \text{by (simp add: Let-def bt-insert-is-sorted} \ \\
  \text{bt-search-split-correct bt-contains-bt-insert)}
  \]
Conclusion: Isabelle as a Proof Assistant

- Define concepts & functions
- Formulate statements about these
- Proof these statements
  - Obvious parts done automatically
  - Can structure proof along informal arguments
  \[ \Rightarrow \text{Proof} \approx \text{very precise version of explanation} \]
- The „shoulders of a giant“
  - Extensive libraries of standard concepts available
  - Frequent goals about concepts automated
Content of the Lecture

• Interactive proofs with Isabelle

• Semantics of programming languages

• Hoare-Logics for imperative languages

• Techniques & heuristics for verification

• Heap manipulating programs
  • Burstall’s memory model
  • Separation Logic
What You can Learn

1. How to argue about the correctness of (imperative) software
   - Datastructures, state, memory modification
   - Making informal arguments precise

2. Concepts underlying formal correctness arguments
   - Semantics (meaning) of programming languages
   - Hoare logic & verification conditions
   - Heap models

3. Lab exercises: small exemplary proofs
   - Get a detailed grasp of concepts
   - Introduction to Isabelle
Organisation

- Mode: teams (of 2) work together during lab
- If need be: prove remainder at home
- Goal: define concepts & find example proofs along lecture
- We will now find a common time for the exercises
- Examns: oral or written, depending on number of participants
Final remarks

- I hope I have been able to . . .
  - inform you
  - get you interested
  - motivated you
  - challenged you

- . . . such that we meet again next week!
Literatur


