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”

  $\Rightarrow$ “To understand this precsiely, 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 ⇨ multiplication of errors
  - Libraries & data structures
  - Infracstructure: memory manager, garbage collectors
- Basic software
  - Operating systems
  - Hypervisors
  - Compilers

Examples of Interactive Verification

- C compiler [4, 15]
- L4 micro-kernel [12, 11, 21]
- Safetly 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]

$\Rightarrow$ This lecture: explore concepts by interactive proof

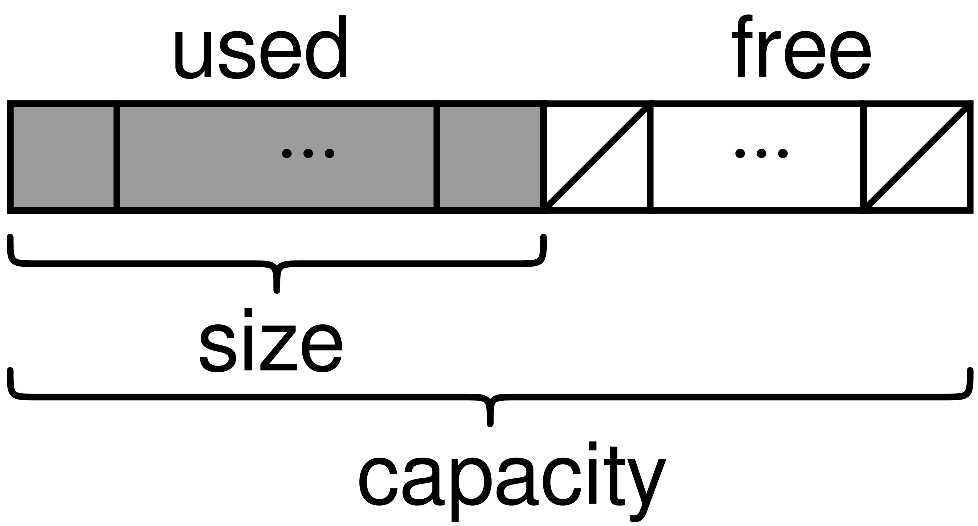
How “Interactive” are Interactive Provers?

- Interative 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

Application to `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;
  }
}
```

(created interactively)

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

The Case for Java

Correctness of Programs
Solve Proof Obligations
Hoare Logic
Pre-/Post-Conditions for if, while, try, catch Pre-/Post-Conditions in connection with inheritance Disjointness of objects and fields
Semantics
Objects, References, Arrays Expressions, Statements, Exceptions Dynamic Dispatch

The Case for C

Correctness of Programs
Solve proof obligations Aliasing of pointers
Hoare Logic
Expressions, . . . as in Java Untyped memory access / casts
Semantics
Byte-addressed memory, bit manipulations Allocation of memory regions Pointer arithmetic and -casts Pointers to local variables / struct fields

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

Correctness of Programs
Proofs about defined functions
Verification Logic
Express statements directly as theorems
Semantics
Recursive functions ($\approx$ ML, Haskell, Scheme) Standard data types (natural numbers, lists, . . .)

Example: Binary Trees

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

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

- Concept: the elements of a tree

```
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

  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 "∨", "∃"

Executing the Code

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 bt-search-full "correct"?
  - It yields "`True`" iff `x` is element of the tree

$\Rightarrow$ Given by bt-contains

- Formulate as lemma ⇨ proof obligation (goals)

  lemma bt-search-full-correct:
  "bt-search-full t x $\longleftrightarrow$ bt-contains t x"

- Proof

  Induction on the tree structure
  apply (induct t)
  Use the definitions
  apply (unfold bt-contains-def bt-elems-def)
  Solve remainder automatically
  apply auto
  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))"

Correctness of Binary Search

- For sorted trees the new function is correct

  lemma bt-search-split-correct:
  $''\text{bt-sorted}\ t \Longrightarrow \text{bt-search-split}\ t\ x \longleftrightarrow \text{bt-contains}\ t\ x''$

  - 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:
    $''\text{bt-sorted}\ (\text{Node}\ l\ v\ r) \Longrightarrow \text{bt-sorted}\ l \wedge \text{bt-sorted}\ r''$
    lemma bt-sorted-elemsD:
    $''\text{bt-sorted}\ (\text{Node}\ l\ v\ r) \Longrightarrow$
    $(\forall x \in \text{bt-elems}\ l.\ x \leq v) \wedge (\forall x \in \text{bt-elems}\ r.\ v \leq x)''$

Data structures & Invariants

- Consider insertion into a binary tree

```
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 bt-insert

- Invariant: sorted trees remain sorted after insertion

  lemma bt-insert-is-sorted:
  "bt-sorted t $\Longrightarrow$ bt-sorted (bt-insert x t)"

- And the inserted element is actually contained in the tree

  lemma bt-insert-result-set:
  "bt-elems (bt-insert x t) $=$ $\{x\}$ $\cup$ bt-elems t"

- Proofs
  - By induction on t
  - Auxiliary facts on sorted trees as before

Verwendung der Korrektheitsaussagen

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

lemma
"bt-sorted t $\Longrightarrow$
let t' = bt-insert x t in
let t'' = bt-insert y t' in
bt-search-split t'' x = True"

- Need auxiliary facts about rows of function applications

lemma bt-contains-bt-insert:
"bt-contains (bt-insert x t) y = (y = x $\vee$ bt-contains t y)"
apply (simp add: bt-contains-def)
apply (simp add: bt-insert-result-set)
done

- Then we can prove the above lemma

by (simp add: Let-def bt-insert-is-sorted
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

  $\Rightarrow$ Proof $\approx$ 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 excercises: 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

[1] Eyad Alkassar, Mark A. Hillebrand, Wolfgang Paul, and Elena Petrova. Automated verification of a small hypervisor. In Peter O'Hearn, Gary T. Leavens, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *LNCS*, pages 40–54, Edinburgh, UK, August 2010. Springer.

[2] Anindya Banerjee, Mike Barnett, and David A. Naumann. Boogie meets regions: a verification experience report. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE'08*, volume 5295 of *LNCS*, pages 177–191. Springer, 2008.

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium (FMCO)*, volume 4111 of *LNCS*. Springer, 2005.

[4] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006.

[5] Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie–An interactive prover-backend for the Verifying C Compiler. *J. Autom. Reason.*, 44:111–144, 2010.

[6] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, 2000.

[7] Matko Botincan, Matthew Parkinson, and Wolfram Schulte. Separation logic verification of c programs with an smt solver. In *4th International Workshop on Systems Software Verification (SSV)*, volume 254. Elsevier, 2009.

[8] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[9] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI 2002)*, 2002.

[10] Chris Hawblitzel and Erez Petrank. Automated verification of practical garbage collectors. *SIGPLAN Not.*, 44(1):441–453, 2009.

[11] Gerwin Klein. Operating system verification — an overview. Technical Report NRL-955, NICTA, Sydney, Australia, June 2008.

[12] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. *Communications of the ACM (CACM)*, 53(6):107–115, June 2010.

[13] K. Rustan Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with chalice. pages 195–222, 2009.

[14] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *16th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-16)*, 2010.

[15] Xavier Leroy. Formal certification of a compiler back-end. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06), Charleston, South Carolina*, January 2006.

[16] Christoph Lüth and Dennis Walter. Certifiable specification and verification of C programs. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress*, volume 5850 of *LNCS*. Springer, 2009.

[17] Andrew McCreight, Zhong Shao, Chunxiao Lin, and Long Li. A general framework for certifying garbage collectors and their mutators. *SIGPLAN Not.*, 42(6):468–479, 2007.

[18] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1–2):200–227, 2005.

[19] MichałMoskal. Programming with triggers. In Bruno Dutertre and Ofer Strichman, editors, *SMT '09: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. ACM, 2009.

[20] Lawrence C. Paulson. Proving security protocols correct. In *LICS '99: Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, page 370, Washington, DC, USA, 1999. IEEE Computer Society.

[21] H. Tuch, G. Klein, and M. Norrish. Verification of the L4 kernel memory allocator. formal proof document. Technical report, NICTA, 2007. http://www.ertos.nicta.com.au/research/l4.verified/kmalloc.pml.

[22] Thomas Tuerk. A formalisation of Smallfoot in HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs)*, volume 5674 of *LNCS*. Springer, 2009.

[23] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. *SIGPLAN Not.*, 43(6):349–361, 2008.