Teaching deductive verification to undergraduate students

Sandrine Blazy
University of Rennes 1, CNRS IRISA, Inria

sandrine.blazy@irisa.fr

IFIP WG 1.9 / 2.15 Oxford 2018.07.13

The students

About 100 undergraduate students, 3rd year (2nd semester)

Expected prior experience:

- introduction to functional programming (Racket, 1st year)
- introduction to team programming : modules and interfaces, test driven development, version control, contract programming (Scala, 2nd year)
- initiation to logic (propositional and predicate calculus, 3rd year)
- basic data structures (Java, 3rd year, 1st semester)

Course organisation

- Lectures: (7-2) * 2 hours
  - presentation of ideas and concepts
  - interactive demos
- Exercises: 8 * 2 hours
  - practice in group settings
  - prepare labs
- Labs: 10 * 2 hours
  - work in pair in small-group settings
  - submit a Why file at the end of the session (can be improved until the end of the week)
- Written exam (2 hours)

Total : 52h for each student (from January to April), mandatory course

Deductive verification in Why3

WhyML programming language: a subset of OCaml with imperative features

Several provers in our Linux distribution (AltErgo, cvc4, Eprover, Z3)

Many examples borrowed from the Why3 gallery of verified programs

Syllabus

1. First specifications
   - **Test of specifications**
   - First programs operating only over integers
     - loops, loop invariants and variants
     - immutability / mutable variables, let constructs
2. Type invariant
   - **Arrays, first sorts, matrices**
3. Algebraic data types
4. Ghost code
   - **Recursive** data types (incl. lists and trees) and **programs**
6. Weakest precondition calculus

Writing formulas : hints

Many recipes are given to the students.

- to avoid bad practices
  - verbose and difficult to read formulas
    - too many variables, quantifiers,
    - big formula that should be split (e.g. a post-condition)
    - more than minimalistic formulas (e.g., the loop invariant is 0<i<N, so that it becomes easier to prove)
- to help understand why a proof failed
  - try other provers,
  - then split the current goal until the formula becomes simple,
  - then look at the goal and its hypotheses;
  - if needed, add some assertions

Why3 is very useful !

Testing a specification
What is a precise specification ?

```
module Max

val max (i j : int) : int
ensures { result = i \/ result = j
}

end
```

```
module Max

val max (i j : int) : int
ensures { i<=j -> result = j }
ensures { i>j -> result = i }

end
```

```
module Test

let test () =
    let tmp = max 3 4 in
        assert { tmp = 4 }
end
```

Testing a specification

- Easily accepted by students
- but, may be difficult to assert by provers

```
let test () =
    let a = make 3 0 in
        a[0] <- 7; a[1] <- 3; a[2] <- 1;
        selection_sort a;
        assert { a [0] = 1 };
        assert { a [1] = 3 };
        assert { a [2] = 7 }
```

ensures { sorted a /\ permut a (old a) }

Arrays, sorts, matrices : practising loop invariants

Basic examples where (part of) the loop invariant « looks like » the post-condition (e.g. array search)

Many examples were studied so that the students managed to understand loop invariants.

- Write the loop invariant first (e.g. Dutch flag)

Advanced examples with nested loops (e.g. insertion sort) and harder to guess invariants (e.g. selection sort, bubble sort)

Encouraging results : Why3 is very useful to find the errors of the students

Dealing with assertions and loop invariants

Difficult case : when an assertion is required in the program

Bad student practise :

- write an imprecise invariant and/or post-condition,
- then add the unproved formula into an assertion,
- and add more assertions

Recursive programs

Programs manipulating lists and trees

- comparison between recursive and iterative programs

Well-known recursive programs (towers of Hanoi, a backtracking program)

Use of a ghost variable and a type invariant to handle a better suited data structure

- ring buffer (array and sequence)
- trie (array and tree)

Axiomatisation of a recursive program, that is implemented using a loop

```
function fibonacci int : int
…
axiom fibn: forall n:int. n>1 ->
        fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

Assessment

progression level	students tasks	methodology	technicalities
Basic	click and see until all the POs become green : easy and fun	Learn a new methodology and a new syntax Understand what is **a precise specification**	The specification is the only source of error. • basic programs • assertions only used in tests
Intermediate	Understand the 1st failures Motto : **take time to think** • long process • almost black magic for some students	Remember the methodology Understand what does « **the code satisfies its spec** » mean I observed many inconsistencies !	• (all kinds of) loop invariants • type invariants • use of assertions ➤ **don't forget to think** • many sources of errors
Advanced	more **thinking** : ~~define and~~ use a data structure that facilitates the proof	Understand the different POs	• ghost variable / code • linking a recursive spec with an imperative program • add axioms (lemma functions)

Questions ?