Program verification with Why3, III

Marc Schoolderman

February 21, 2019

Sorting - statistics

Very promising results: 15 submissions

- 8 – proof complete
- 5 – proof half complete
- 2 – needs work

Don't forget `report.txt`!

Sorting

- Selection sort – relatively easy
- Insertion sort – relatively easy
- Bubblesort – relatively hard

Sorting

- Selection sort – relatively easy
- Insertion sort – relatively easy
- Bubblesort – relatively hard

What does it mean to be sorted?

```
not exists i j. 0 <= i < j < length result /\ result[i] > result[j]

forall k. 0 < k < length result -> result[k-1] <= result[k]

forall i j. 0 <= i <= j < length result -> result[i] <= result[j]

IntArraySorted.sorted result
```

Sorting

Proving that your algorithm permutes elements

- Using `Array.swap` – trivial?
- Direct array manipulation – very hard

Sorting

Proving that your algorithm permutes elements

- Using `Array.swap` – trivial?
- Direct array manipulation – very hard

Note: my solutions can be found at
`http://cs.ru.nl/~mschool/swan2019/`

(Will be uploaded to BrightSpace soon.)

Proof techniques

Why can't I prove something?

Reasons a proof might fail:

1. What you want to prove is not true!
   - Fix your specifications!
2. Not enough information to support a proof
   - E.g., missing invariant
3. Automatic provers don't find the proof
   - Special case: proof by induction

Why can't I prove something?

Reasons a proof might fail:

1. What you want to prove is not true!
   - Fix your specifications!
2. Not enough information to support a proof
   - E.g., missing invariant
3. Automatic provers don't find the proof
   - Special case: proof by induction

Why can't I prove something?

Reasons a proof might fail:

1. What you want to prove is not true!
   - Fix your specifications!
2. Not enough information to support a proof
   - E.g., missing invariant
3. Automatic provers don't find the proof
   - Special case: proof by induction

Why can't I prove something?

Reasons a proof might fail:

1. What you want to prove is not true!
   - Fix your specifications!
2. Not enough information to support a proof
   - E.g., missing invariant
3. Automatic provers don't find the proof
   - Special case: proof by induction

Proof remedies I

Use more power:

- Increase time/memory limits for provers
- Try different provers

Use human intelligence:

- Add lemmas or `assert`'s to state *intermediate steps*

Proof remedies I

Use more power:

- Increase time/memory limits for provers
- Try different provers

Use human intelligence:

- Add lemmas or `assert`'s to state *intermediate steps*

  Hard "49 is not prime"

Proof remedies I

Use more power:

- Increase time/memory limits for provers
- Try different provers

Use human intelligence:

- Add lemmas or `assert`'s to state *intermediate steps*

  Hard ”49 is not prime”

  Easy ”49 is 7*7; so 49 is not prime”

Proof remedies I

Use more power:

- Increase time/memory limits for provers
- Try different provers

Use human intelligence:

- Add lemmas or `assert`'s to state *intermediate steps*

  Hard ''49 is not prime''

  Easy ''49 is 7*7; so 49 is not prime''

- Try to discover a general principle to add as a lemma!

Proof remedies II

The dark arts:

- Use *proof transformations*

`split_*`	break formula into smaller components
`inline_*`	expand function calls
`compute_in_goal`	more aggressive rewrite
`eliminate_*`	replace high level concepts by simpler ones
`induction_ty_lex`	induction over algebraic data

Proof remedies II

The dark arts:

- Use *proof transformations*

`split_*`	break formula into smaller components
`inline_*`	expand function calls
`compute_in_goal`	more aggressive rewrite
`eliminate_*`	replace high level concepts by simpler ones
`induction_ty_lex`	induction over algebraic data
`induction` *var* `from` *start*	mathematical induction
`exists` *object*	solving existential goals

Proof remedies II

The dark arts:

- Use *proof transformations*

`split_*`	break formula into smaller components
`inline_*`	expand function calls
`compute_in_goal`	more aggressive rewrite
`eliminate_*`	replace high level concepts by simpler ones
`induction_ty_lex`	induction over algebraic data
`induction` *var* `from` *start*	mathematical induction
`exists` *object*	solving existential goals
`smoke_detector`	finds inconsistencies, should not be provable!

Proof remedies II

The dark arts:

- Use *proof transformations*

`split_*`	break formula into smaller components
`inline_*`	expand function calls
`compute_in_goal`	more aggressive rewrite
`eliminate_*`	replace high level concepts by simpler ones
`induction_ty_lex`	induction over algebraic data
`induction` *var* `from` *start*	mathematical induction
`exists` *object*	solving existential goals
`smoke_detector`	finds inconsistencies, should not be provable! 

Not entirely predictable beforehand what works and what doesn't.

Proof remedy: last resort

Use an interactive theorem prover such as Coq

- Slow, painful, expertise needed
- Most often will show you why the proof cannot work
- No extra ‘safety guarantee’

More systematic remedy

Which do you have more experience with?

- Writing a program for a particular problem
- Proving a mathematical theorem

More systematic remedy

Which do you have more experience with?

- Writing a program for a particular problem
- Proving a mathematical theorem

So, write programs instead of proofs!

Ghost code and Let lemmas

The Aristotelian Universe

The Why3 Universe

1. *Logical layer*
2. *Program layer*

The Why3 Universe

1. *Logical layer*
2. *Program layer*

```
let foo (x: int) = prime x
```

This will give the error message:

- ‘‘Logical symbol prime is used in a non-ghost context’’

The Why3 Universe

1. *Logical layer*
2. *Ghost layer*
3. *Program layer*

The Why3 Universe

1. *Logical layer*
2. *Ghost layer*
3. *Program layer*

```
let ghost foo (x: int) = prime x
```

This is fine!

The spirit of ghost code

- Functions, variables, and expressions can be marked as *ghost*.
- Ghosts can observe, but not affect the ‘real world’

The spirit of ghost code

- Functions, variables, and expressions can be marked as *ghost*.
- Ghosts can observe, but not affect the ‘real world’

Therefore:

- Ghost code only modify ghost data.
    - Why3 will deduce the *ghostness* of an expression.

The spirit of ghost code

- Functions, variables, and expressions can be marked as *ghost*.
- Ghosts can observe, but not affect the 'real world'

Therefore:

- Ghost code only modify ghost data.
  - Why3 will deduce the *ghostness* of an expression.
- Ghost code **can** use *purely logical* functions/types

The spirit of ghost code

- Functions, variables, and expressions can be marked as *ghost*.
- Ghosts can observe, but not affect the 'real world'

Therefore:

- Ghost code only modify ghost data.
    - Why3 will deduce the *ghostness* of an expression.
- Ghost code **can** use *purely logical* functions/types
- Ghost code **must** always terminate!

The spirit of ghost code

- Functions, variables, and expressions can be marked as *ghost*.
- Ghosts can observe, but not affect the ‘real world’

Therefore:

- Ghost code only modify ghost data.
  - Why3 will deduce the *ghostness* of an expression.
- Ghost code **can** use *purely logical* functions/types
- Ghost code **must** always terminate!
- Ghost code can be safely **erased** from programs.

Example: shifting an array

For insertion sort, we could use this to insert x at the right spot:

- `find` a l h x finds a position $i \in [l, h)$ to insert x into a

```
let x = a[pos] in
let i = find a 0 pos x in
Array.self_blit a i (i+1) (pos-i);
a[i] <- x;

assert { permut_all (old a) a };
```

Problem: *extremely hard to prove that a stays a permutation!*

Example: shifting an array

We can also shift the array using just swaps:

```
let shift (a: array int) (i j: int)
= for k = j downto i+1 do
    swap a k (k-1);
  done
```

Example: shifting an array

Which needs to be proven correct:

```
let shift (a: array int) (i j: int)
  requires { 0 <= i <= j < length a }
  ensures { forall k. i+1 <= k <= j -> a[k] = old a[k-1] }
  ensures { a[i] = old a[j] }
  ensures { permut_sub (old a) a i (j+1) }
= for k = j downto i+1 do
    invariant { permut_sub (old a) a i (j+1) }
    invariant { forall k'. i <= k' < k -> a[k'] = old a[k'] }
    invariant { forall k'. k < k' <= j -> a[k'] = old a[k'-1] }
    invariant { a[k] = old a[j] }
    swap a k (k-1);
  done
```

Example: shifting an array

Now we *can* easily prove that:

1. `shift` returns a permutation
2. `shift` does the same thing as the original code

Example: shifting an array

Now we *can* easily prove that:

1. `shift` returns a permutation
2. `shift` does the same thing as the original code

```
let b = ghost copy a in
assert { permut_all (old a) b };
shift b i pos;
assert { permut_all (old a) b };

let x = a[pos] in
let i = find a 0 pos x in
Array.self_blit a i (i+1) (pos-i);
a[i] <- x;

assert { array_eq a b };
assert { permut_all (old a) a };
```

Consider a ghost function without *external* side-effects:

```
let ghost foo (x: some_type)
  requires { p x }
  ensures { q x }
= (* ... *)
```

let lemma

Consider a ghost function without *external* side-effects:

```
let ghost foo (x: some_type)
  requires { p x }
  ensures { q x }
= (* ... *)
```

Observe: `foo` can be called *at any time* to get $Q(x)$ from $P(x)$.

Consider a ghost function without *external* side-effects:

```
let ghost foo (x: some_type)
  requires { p x }
  ensures { q x }
= (* ... *)
```

Observe: `foo` can be called *at any time* to get $Q(x)$ from $P(x)$.

- If `foo` is correct, this means $P(x) \rightarrow Q(x)$ for all x!

let lemma

Ghost functions like these be turned into *lemma functions*:

```
let lemma foo (x: some_type)
  requires { p x }
  ensures { q x }
= (* ... *)
```

let lemma

Ghost functions like these be turned into *lemma functions*:

```
let lemma foo (x: some_type)
  requires { p x }
  ensures { q x }
= (* ... *)
```

After this definition, Why3 behaves as if you had proved:

```
lemma foo:
  forall x: some_type. p x -> q x
```

Induction using `let rec lemma`

Sometimes a proof needs induction:

```
lemma fib_property:
  real_fib 0 = 0.0 /\
  real_fib 1 = 1.0 /\
  forall k. k >= 2 -> real_fib (k-2) +. real_fib (k-1) = real_fib k

lemma functional_equivalence:
  forall k. k >= 0 -> real_fib k = from_int (fib k)
```

Induction using `let rec lemma`

This can be done using a recursive lemma function:

```
let rec lemma fib_equivalence (k: int)
  requires { k >= 0 }
  ensures { real_fib k = from_int (fib k) }
  variant { k }
= if k >= 2 then begin
    assert { real_fib (k-2) +. real_fib (k-1) = real_fib k };
    fib_equivalence (k-1);
    fib_equivalence (k-2);
  end
```

Another example: `fact` is positive

Typically:

- Lemma functions look rather strange
- Lemma functions mimic the *structure* of an induction

```
let rec lemma fact_positive (n: int)
  requires { n >= 0 }
  ensures { fact n > 0 }
  variant { n }
= if n = 0 then ()
  else fact_positive (n-1)
```

Proof minimization

Why does context size matter?*

- The only people in the cereal cafe are from Stoke.
- Every person would make a great Uber driver, if he or she is not allergic to gluten.
- When I love someone, I avoid them.
- No one is a werewolf, unless they have orange skin and blond hair.
- No one from Stoke fails to Instagram their breakfast.
- No one ever asks me whether I prefer Wills to Harry, except people in the cereal cafe.
- People from Thanet wouldn't make great Uber drivers.
- None but werewolves Instagram their breakfast.
- The people I love are the ones who do not ask me whether I prefer Wills to Harry.
- People with orange skin and blond hair are not allergic to gluten.

* Source of puzzle: https://www.theguardian.com/science/2017/jan/30/can-you-solve-it-lewis-carroll-on-brexit-britain

Why does context size matter?[*]

- The only people in the cereal cafe are from Stoke.
- Every person would make a great Uber driver, if he or she is not allergic to gluten.
- When I love someone, I avoid them.
- No one is a werewolf, unless they have orange skin and blond hair.
- No one from Stoke fails to Instagram their breakfast.
- No one ever asks me whether I prefer Wills to Harry, except people in the cereal cafe.
- People from Thanet wouldn't make great Uber drivers.
- None but werewolves Instagram their breakfast.
- The people I love are the ones who do not ask me whether I prefer Wills to Harry.
- People with orange skin and blond hair are not allergic to gluten.

Claim: **People from Thanet are allergic to gluten**

[*] Source of puzzle:
https://www.theguardian.com/science/2017/jan/30/can-you-solve-it-lewis-carroll-on-brexit-britain

Why does context size matter?*

- Every person would make a great Uber driver, if he or she is not allergic to gluten.

- People from Thanet wouldn't make great Uber drivers.

Claim: **People from Thanet are allergic to gluten**

* Source of puzzle:
https://www.theguardian.com/science/2017/jan/30/can-you-solve-it-lewis-carroll-on-brexit-britain

Ways to minimize context

Common sense:

- When a proof is done, get rid of unnecessary lemmas/asserts
- Replace specific lemmas with general ones
- Don't `use` unnecessary or large modules

Ways to minimize context

Common sense:

- When a proof is done, get rid of unnecessary lemmas/asserts
- Replace specific lemmas with general ones
- Don't `use` unnecessary or large modules

Why3 tools:

- Ghost code
- Context manipulation
- Abstract blocks

Context manipulation: minimizing formulas

logic formula	*vc_split*	*added to context*
`a /\ b`		`a /\ b`
	❶ `a`	
	❷ `b`	

Context manipulation: minimizing formulas

logic formula	*vc_split*	*added to context*
`a /\ b`	❶ `a` ❷ `b`	`a /\ b`
`a && b`	❶ `a` ❷ `a -> b`	`a /\ b`

Context manipulation: minimizing formulas

logic formula	*vc_split*	*added to context*
`a /\ b`	1. `a` 2. `b`	`a /\ b`
`a && b`	1. `a` 2. `a -> b`	`a /\ b`
`b by a`	1. `a` 2. `a -> b`	`b`

Context manipulation: abstract blocks

Complex blocks of code can be summarized:

```
if !b = 1 then
  if !a = 1 then a := 0
  else begin a := 1; b := 0 end
```

Context manipulation: abstract blocks

Complex blocks of code can be summarized:

```
begin
  requires { 0 <= !a <= 1 /\ 0 <= !b <= 1 }
  ensures { !b*2 + !a = old (!a + !b) }
  if !b = 1 then
    if !a = 1 then a := 0
    else begin a := 1; b := 0 end
end
```

- Like a function contract, but without the function
- *What is said in the abstract block, stays in the abstract block*

Summary

- Proof transformations
- Ghost code
- Let lemmas
  - Can do all kinds of induction!
- Minimizing proof context
- Abstract blocks