Program verification with Why3, II

Marc Schoolderman

February 14, 2019

Assessment

- Marks for exercise are purely *subjective*
  - *excellent, good, average, fair, poor*
  - Don't count towards final grade
- Grading for first part: small (but real) case study
  - Verification products: models, proofs
  - Short report on your formalization
  - Evaluation of Why3

  No written exam.

For both: working in *pairs* allowed.

Recap

Last week:

- Intro Why3
- WhyML consists of *two layers*:
  Logical formulas + Program code
- Function contracts
- `while` loops: *invariants* and *variants*

Finding good invariants

A loop `invariant` must hold:

1. **Before** the loop even starts
2. **During** the loop
3. **After** the loop ends

It's okay to guess invariants, but make *educated* guesses.

Why3 news

Why3 1.2.0 is out since 11 february...

1. Syntactic sugar for references
2. GTK3 support
3. Z3 4.8.x support

This week

No tutorial hour on wednesday!

Deadline for exercise 3: 19 february, 18:00

Composite data in Why3

Arrays

WhyML is not limited to just `ref` and `int`:

```
use int.Int;
use array.Array;

let main ()
= let a: array int = Array.make 3 0 in
  a[1] <- 1;
  a[2] <- 1;
  let x = a[0] + a[1] + a[2] in
  assert { x = 2 }
```

Arrays

Shorthand:

```
use int.Int;
use array.Array;

let main ()
= let a = make 3 0 in
  a[1] <- 1;
  a[2] <- 1;
  let x = a[0] + a[1] + a[2] in
  assert { x = 2 }
```

Reasoning about arrays

We can talk about the *sum* of an array:

```
use int.Int
use array.Array
use array.ArraySum

let main ()
= let a = make 3 0 in
  a[1] <- 1;
  a[2] <- 1;
  (* sum a l h: the sum of all a[i], where l <= i < h *)
  assert { sum a 0 3 = 2 }
```

More reasoning about arrays

... or count elements:

```
use int.Int
use array.Array
use array.ArraySwap
use array.NumOf

let main ()
= let a = make 3 0 in
  a[1] <- 1;
  a[2] <- 1;
  swap a 0 1;
  assert { numof (fun i x -> x > 0) a 0 3 = 2 }
```

Note: limited support for *higher order functions*.

Basic operations on arrays: array.Array

`a[i], a[i] <- x`	elements access, update
`length` a	get array size
`make` n $init$	creation
`append` a b	appending
`sub` a i len	slicing
`copy` a	cloning
`fill` a i len	writing
`blit` a_1 i_1 a_2 i_2 len	copying elements
`self_blit` a i_1 i_2 len	copying elements

More array functions: see `stdlib/array.mlw`.

Equality on arrays

This will not verify:

```
use int.Int
use array.Array

let main ()
= let a = make 3 0 in
  a[1] <- 1;
  a[2] <- 1;
  let b = copy a in
  assert { a = b }
```

Any idea why not?

Equality on arrays

Instead:

```
use int.Int
use array.Array

let main ()
= let a = make 3 0 in
  a[1] <- 1;
  a[2] <- 1;
  let b = copy a in
  assert { forall i. 0 <= i < length b -> a[i] = b[i] }
```

Equality on arrays

There is a predicate for this:

```
use int.Int
use array.Array
use array.ArrayEq

let main ()
= let a = make 3 0 in
  a[1] <- 1;
  a[2] <- 1;
  let b = copy a in
  assert { array_eq a b }
```

Demo: Kadane's algorithm

The maximum subarray problem

What is the largest contiguous sum in an array?

The maximum subarray problem

What is the largest contiguous sum in an array?

```
let maxSum (a: array int): int
= let max_so_far = ref 0 in
  let max_ending_here = ref 0 in
  for cur = 0 to length a - 1 do
    max_ending_here := !max_ending_here + a[cur];
    if !max_ending_here < 0 then max_ending_here := 0;
    if !max_ending_here > !max_so_far then max_so_far := !max_ending_here;
  done;
  return !max_so_far
```

Side effects in function contracts

Reasoning about program state

You now know two types that are *mutable*:

- `ref 'a`, `array 'a`

Consider this function:

```
let increment (x: ref int)
= x := !x + 1
```

Reasoning about program state

You now know two types that are *mutable*:

- `ref 'a`, `array 'a`

Consider this function:

```
let increment (x: ref int)
= x := !x + 1
```

How to write a contract for `increment`?

Reasoning about program state

This doesn't work:

```
let increment (x: ref int)
  ensures { !x = !x + 1 }
= x := !x + 1
```

Reasoning about program state

This doesn't work:

```
let increment (x: ref int)
  ensures { !x = !x + 1 }
= x := !x + 1
```

Everything in the postcondition always refers to *final state*

The `old` pseudo-function

But we can do this:

```
let increment (x: ref int)
  writes { x }
  ensures { !x = old !x + 1 }
= x := !x + 1
```

The `old` pseudo-function

But we can do this:

```
let increment (x: ref int)
  writes { x }
  ensures { !x = old !x + 1 }
= x := !x + 1
```

`writes`: This function can modify `x`

`old !x`: "`!x` in the *initial state*"

The `old` pseudo-function

But we can do this:

```
let increment (x: ref int)
  writes { x }
  ensures { !x = old !x + 1 }
= x := !x + 1
```

`writes`: This function can modify `x`

`old !x`: “`!x` in the *initial state*”

Note: Why3 can deduce `writes` here by itself.

Another example:

```
use array.Array
use array.ArraySwap

let sort3 (a: ref int)
  requires { length a = 3 }
  ensures { a[0] <= a[1] <= a[2] }
= if a[0] > a[1] then swap a 0 1;
  if a[1] > a[2] then swap a 1 2;
  if a[0] > a[1] then swap a 0 1;
```

label and at

We can also refer to intermediate states:

```
use array.Array
use array.ArraySwap

let sort3 (a: ref int)
  requires { length a = 3 }
  ensures { a[0] <= a[1] <= a[2] }
= if a[0] > a[1] then swap a 0 1;
  label Swap in
  if a[1] > a[2] then swap a 1 2;
  if a[0] > a[1] then swap a 0 1;
  assert { a[0] <= a[0] at Swap <= old a[0] }
```

Partially defined functions

Out-of-bounds array access

What should happen now?

```
use int.Int
use array.Array

let main ()
= let a = make 3 0 in
  a[1] <- 1;
  a[2] <- 1;
  let x = a[42] in
  assert { x = a[42] }
```

Safety conditions

Certain WhyML operations generate *safety conditions*:

- Simply part of the contract: `requires`

```
let ([]) (a: array 't) (i: int)
  requires { 0 <= i < length a }
  ensures { result = a[i] }
= (* ... *)

let div (x y: int): int
  requires { y <> 0 }
  ensures { result = div x y }
= (* ... *)
```

(Distinguish *logical* `div` from *program* `div`!)

Quiz

What is the difference between this:

```
let div (x y: int): int
  requires { y <> 0 }
  ensures { result = div x y }
= (* ... *)
```

and this:

```
let myDiv (x y: int): int
  ensures { y <> 0 -> result = div x y }
= (* ... *)
```

Results of program execution

When running a program, one of these can happen:

1. Normal termination: postcondition holds
2. It doesn't terminate: prevented by `variant`
3. Undefined behaviour: prevented by checking preconditions

Results of program execution

When running a program, one of these can happen:

1. Normal termination: postcondition holds
2. It doesn't terminate: prevented by `variant`
3. Undefined behaviour: prevented by checking preconditions
4. *Exceptional termination*: an exception is raised

Exceptions in Why3

All exceptions are *checked*: specify the *exceptional postcondition*.

```
exception OutOfBounds

let safe_get (a: array 't) (i: int)
  ensures { result = a[i] }
  ensures { 0 <= i < length a }
  raises { OutOfBounds -> i < 0 \/ i >= length a }
= if i < 0 || i >= length a then raise OutOfBounds
  else return a[i]
```

Handling an exception

To catch exceptions, use `try ... with`:

```
let firstElement (a: array int)
= try
    safe_get a 0
  with
    OutOfBounds -> 0
  end
```

Partial functions in logic

WhyML *logical layer* has no contracts or exceptions!

```
function div (x y: int): int
  = (* ... *)

let div (x y: int): int
  requires { y <> 0 }
  ensures { result = div x y }
= (* ... *)
```

Partial functions in logic

WhyML *logical layer* has no contracts or exceptions!

```
function div (x y: int): int
  = (* ... *)

let div (x y: int): int
  requires { y <> 0 }
  ensures { result = div x y }
= (* ... *)
```

What is `div x 0` in the *purely logical layer*?

Undefinedness

```
function div (x y: int): int
```

All functions in the *logical layer* must be *pure* and *total*

Pure No side-effects

Total Always produce a result for every input

Undefinedness

```
function div (x y: int): int
```

All functions in the *logical layer* must be *pure* and *total*

Pure No side-effects

Total Always produce a result for every input

Partial functions are "made total" by assuming an *unknown* output

```
lemma div_1: exists x: int. div 42 0 = x (* provable *)
lemma div_2: div 42 0 * 0 = 0            (* provable *)
lemma div_3: div 42 0 = 5                (* not provable, not disprovable *)
```

Abstract definitions

Why3 allows declaring functions *without* a definition

Abstract definitions

Why3 allows declaring functions *without* a definition:

Logic:

```
function next_prime(n: int): int
```

Program:

```
val next_prime (n:int): int
```

Abstract definitions

Why3 allows declaring functions *without* a definition:

Logic:

```
function next_prime(n: int): int

axiom next_prime_def1:
  forall n. next_prime n > n

axiom next_prime_def2:
  forall n. prime (next_prime n)
```

Program:

```
val next_prime (n:int): int
  ensures { result > n }
  ensures { prime n }
```

Ex falso sequitur quodlibet!

How to shoot yourself in the foot:

```
constant max_int: int

axiom max_int_def:
  forall n. n <= max_int

lemma woops: 1 = 2
```

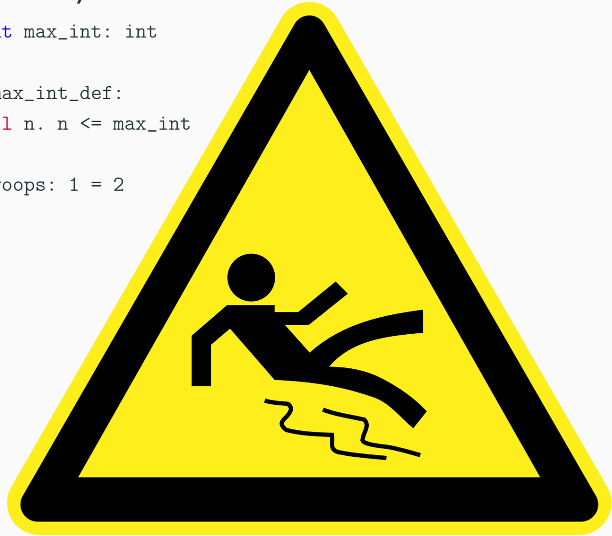
Ex falso sequitur quodlibet!

How to shoot yourself in the foot:

```
constant max_int: int

axiom max_int_def:
  forall n. n <= max_int

lemma woops: 1 = 2
```



Functional data types

Constructed & polymorphic types

Sum types: algebraic types

```
type list 'a = Nil | Cons 'a (list 'a)
```

Product types: tuples and records

```
type numbered_pair 'a = (int, 'a)
type vector = { x: real; y: real }
```

Working with composite types

Creating, accessing, updating:

```
function up (len: real): vector      = { x=0.0; y=len }
function size (v: vector): real      = sqrt (v.x*v.x + v.y*v.y)
function flatten (v: vector): vector = { v with y = 0.0 }
```

Pattern matching:

```
function append (xs ys: list 'a): int =
  match xs with
  | Cons x xs' -> Cons x (append xs' ys)
  | Nil -> ys
  end
```

```
function sum (pair: (int,int)): int
  = let (a,b) = pair in a+b
```

Recursion in Why3

For *recursive types*, we also want *recursive* functions.

How to prevent an *infinite recursion*?

Recursion in Why3

For *recursive types*, we also want *recursive* functions.

How to prevent an *infinite recursion*? **Variants!**

```
let rec append (xs ys: list 'a): int
  variant { length xs }
= match xs with
  | Cons x xs' -> Cons x (append xs' ys)
  | Nil -> ys
  end
```

(Similar to a *"measure"* in PVS, Coq)

Structural recursion in Why3

Algebraic types support *structural recursion*:

```
let rec append (xs ys: list 'a): int
  variant { xs }
= match xs with
  | Cons x xs' -> Cons x (append xs' ys)
  | Nil -> ys
  end
```

Recursion in logic vs programs

Pure logic:

- Why3 tries to "guess"

```
function length (xs: list 't): int
  = match xs with
    | Nil -> 0
    | Cons _ xs -> 1+length xs
    end


function fac (n: int): int
= if n <= 0 then 1 else n*fac (n-1)
(* why3 cannot prove termination! *)
```

Program code:

- explicit `variant`

```
let rec length (xs: list 't): int
  variant { xs }
= match xs with
  | Nil -> 0
  | Cons _ xs -> 1+length xs
  end

let rec fac (n: int): int
  variant { n }
= if n <= 0 then 1 else n*fac (n-1)
```

Recursion in logic vs programs

Often we can avoid repeating ourselves:

Both logic and program code:

```
let rec function length (xs: list 't): int
  variant { xs }
= match xs with
  | Nil -> 0
  | Cons _ xs -> 1+length xs
  end

let rec function fac (n: int): int
  variant { n }
= if n <= 0 then 1 else n*fac (n-1)
```

Lexigraphical variants

You can have more than one variant:

```
let rec function ackermann (m n: int): int
  variant { m, n }
= if m <= 0 then n+1
  else if n <= 0 then ackermann (m-1) 1
  else ackermann (m-1) (ackermann m (n-1))
```

Common types in the Why3 standard library

The most common types are already implemented:

Maybe: `option 'a`

- `option.Option`

Linked lists: `list 'a`

- `list.ListRich`

Binary trees: `tree 'a`

- `bintree.Tree`

Set theory: `set 'a`

- `set.Set`

Abstract types

Like functions and predicates, *types* can be abstract:

```
type set 'a

constant empty: set 'a

function add 'a (set 'a): set 'a

predicate mem 'a (set 'a)

axiom empty_def:
  forall x. not mem x empty
axiom add_def:
  forall x y: 'a, s: set 'a. mem x (add y s) <-> x = y \/ mem x s
```

Abstract types

Like functions and predicates, *types* can be abstract:

```
type set 'a

constant empty: set 'a

function add 'a (set 'a): set 'a

predicate mem 'a (set 'a)

axiom empty_def:
  forall x. not mem x empty
axiom add_def:
  forall x y: 'a, s: set 'a. mem x (add y s) <-> x = y \/ mem x s
```

Equality of objects

1. In *logic*, we can test all objects for equality, as if:

```
predicate (=) 'a 'a
```

And so, for example:

```
lemma singleton_not_nil: Cons 5 Nil <> Nil
```

Equality of objects

1. In *logic*, we can test all objects for equality, as if:

```
predicate (=) 'a 'a
```

   And so, for example:

```
lemma singleton_not_nil: Cons 5 Nil <> Nil
```

2. In *programs*, you only get this for `int`, in int.Int:

```
val (=) (x y: int): bool
  ensures { result <-> x = y }
```

Equality of objects

1. In *logic*, we can test all objects for equality, as if:

```
predicate (=) 'a 'a
```

And so, for example:

```
lemma singleton_not_nil: Cons 5 Nil <> Nil
```

2. In *programs*, you only get this for `int`, in int.Int:

```
val (=) (x y: int): bool
  ensures { result <-> x = y }
```

Annoying, but this is done for good reasons!

User-defined equality

Two solutions:

- Implement it!

```
let rec (==) (x y: list int)
  ensures { result <-> x = y }
  variant { x }
= match x, y with
  | Nil, Nil -> true
  | Cons x xs, Cons y ys -> x = y && xs == ys
  | _, _ -> false
  end
```

User-defined equality

Two solutions:

- Implement it!

```
let rec (==) (x y: list int)
  ensures { result <-> x = y }
  variant { x }
= match x, y with
  | Nil, Nil -> true
  | Cons x xs, Cons y ys -> x = y && xs == ys
  | _, _ -> false
  end
```

- Pretend to have implemented it!

```
val (==) (x y: list int)
  ensures { result <-> x = y }
```

Summary

- WhyML data types (mutable, functional)
- Verification of array programs
- Subtleties of *logical definitions*
- Reasoning about state updates