Program verification with Why3, IV

Marc Schoolderman

February 28, 2019

Last exercise

How useful are the counter-examples?

Last exercise

How useful are the counter-examples?

Meh…

Last exercise

Does this function match its specification?

- Formal verification

Is the specification correct?

- Validation

What if the specification itself has functions?

- Informal meta-argumentation
- *Formal validation*?

Aliasing issues

Aliasing and mutability

Compare this:

```
let main (a: array int)
  requires { length a > 0 }
= let b = a in
  a[0] <- 5;
  b[0] <- 42
```

and this:

```
let main (a: array int)
  requires { length a > 0 }
= let b = copy a in
  a[0] <- 5;
  b[0] <- 42
```

Aliasing and function arguments

This is perfectly reasonable:

```
let combine (x y: ref int)
  ensures { !x = old (!x + !y) }
  ensures { !y = 0 }
= x := !x + !y; y := 0
```

Aliasing and function arguments

This is perfectly reasonable:

```
let combine (x y: ref int)
  ensures { !x = old (!x + !y) }
  ensures { !y = 0 }
= x := !x + !y; y := 0
```

What happens now?

```
let main ()
= let x = ref 5 in
  combine x x;
  assert { !x = 10 /\ !x = 0 } (* ?? *)
```

Aliasing and function arguments

This is perfectly reasonable:

```
let combine (x y: ref int)
  ensures { !x = old (!x + !y) }
  ensures { !y = 0 }
= x := !x + !y; y := 0
```

What happens now?

```
let main ()
= let x = ref 5 in
  combine x x;
  assert { !x = 10 /\ !x = 0 } (* ?? *)
```

WhyML forbids this kind of aliassing!

Aliasing and components of types

Suppose we create this type:

```
type mut_list 'a = Nil | Cons 'a (ref (mut_list 'a))
```

Aliasing and components of types

Suppose we create this type:

```
type mut_list 'a = Nil | Cons 'a (ref (mut_list 'a))
```

Then we could do:

```
let circular ()
= let tail = ref Nil in
  let head = Cons 42 tail in
  tail := head
```

What is the length of `circular`?

Aliasing and components of types

Suppose we create this type:

```
type mut_list 'a = Nil | Cons 'a (ref (mut_list 'a))
```

Then we could do:

```
let circular ()
= let tail = ref Nil in
  let head = Cons 42 tail in
  tail := head
```

What is the length of `circular`?

WhyML forbids mutability in certain contexts!

Modelling of data types

`array` and `ref` are not primitive!

See `ref.Ref`:

```
let ref 'a = { mutable contents: 'a }

let function (!) (r: ref 'a)
  = r.contents

let (:=) (r: ref'a) (v: 'a)
  writes { r }
  ensures { !r = v }
= r.contents <- v
```

`array` and `ref` are not primitive!

See `array.Array`:

```
let array 'a = { mutable ghost elts: int -> 'a;
                 length: int }
  invariant { 0 <= length }

function ([]) (a: array 'a) (i: int): 'a = a.elts i
```

`array` and `ref` are not primitive!

See `array.Array`:

```
let array 'a = { mutable ghost elts: int -> 'a;
                 length: int }
  invariant { 0 <= length }

function ([]) (a: array 'a) (i: int): 'a = a.elts i
```

```
val ([]) (a: array 'a) (i: int): 'a
  requires { 0 <= i < length a }
  ensures { result = a[i] }
```

`array` and `ref` are not primitive!

See `array.Array`:

```
let array 'a = { mutable ghost elts: int -> 'a;
                 length: int }
  invariant { 0 <= length }

function ([]) (a: array 'a) (i: int): 'a = a.elts i
```

```
val ([]) (a: array 'a) (i: int): 'a
  requires { 0 <= i < length a }
  ensures { result = a[i] }
```

```
val ([]<-) (a: array 'a) (i: int) (v: 'a)
  requires { 0 <= i < length a }
  ensures { a.elts = Map.set (old a).elts i v }
```

Type invariants

Contracts sometimes better belong to *objects*:

```
type time = { hour: ref int; min: ref int; sec: ref int }

let tickTock (t: time)
  requires { 0 <= !(t.hour) < 24 /\ 0 <= !(t.min) < 60 /\ 0 <= !(t.sec) < 60 }
  ensures  { 0 <= !(t.hour) < 24 /\ 0 <= !(t.min) < 60 /\ 0 <= !(t.sec) < 60 }
  ensures { to_sec t = mod (old (to_sec t+1)) 86400 }
= sec += 1;
  if !sec >= 60 then (sec := 0; min += 1);
  if !min >= 60 then (min := 0; hour += 1);
  if !hour >= 24 then hour := 0
```

Type invariants

A *type invariant* express this more clearly:

```
type time = { hour: ref int; min: ref int; sec: ref int }
  invariant { 0 <= !hour < 24 /\ 0 <= !min < 60 /\ 0 <= !sec < 60 }

let tickTock (t: time)
  ensures { to_sec t = mod (old (to_sec t+1)) 86400 }
= sec += 1;
  if !sec >= 60 then (sec := 0; min += 1);
  if !min >= 60 then (min := 0; hour += 1);
  if !hour >= 24 then hour := 0
```

Type invariants

A *type invariant* express this more clearly:

```
type time = { hour: ref int; min: ref int; sec: ref int }
  invariant { 0 <= !hour < 24 /\ 0 <= !min < 60 /\ 0 <= !sec < 60 }

let tickTock (t: time)
  ensures { to_sec t = mod (old (to_sec t+1)) 86400 }
= sec += 1;
  if !sec >= 60 then (sec := 0; min += 1);
  if !min >= 60 then (min := 0; hour += 1);
  if !hour >= 24 then hour := 0
```

Note: Type invariants can be *momentarily* broken!

Type invariants

Type invariants are also handy in logic:

```
type time = { hour: ref int; min: ref int; sec: ref int }
  invariant { 0 <= !hour < 24 /\ 0 <= !min < 60 /\ 0 <= !sec < 60 }

function to_sec (t: time): int
  = !(t.hour)*3600 + !(t.min)*60 + !(t.sec)

lemma sec_in_day:
  forall t. to_sec t < 86400
```

Bits and bytes

Machine integers

Why3 `int.Int` are unbounded

real world `int` is bounded

How to model this?

Machine integers

Approach 1: *type invariants*

```
type uchar = { value: int }
  invariant { 0 <= value < 256 }

type schar = { value: int }
  invariant { -128 <= value < 128 }
```

- Advantage: easy to understand, understood by all provers
- Disadvantage: need to implement operations as well
  - Otherwise the type invariant is easily broken

Machine integers

Approach 2: *bit vector theories*

```
use bv.BV8

type byte = BV8.t

lemma neg_expr:
  forall x. add (bw_not x) (0x01:byte) = neg x

let function abs (x: byte): byte
  ensures { t'int result = abs (to_int x) }
= let mask = asr x 8 in
  let x = bw_xor x mask in
  return sub x mask
```

- Advantage: good for bit operations
- Disadvantage: signedness not part of type

Machine integers

Approach 3: `mach.int.IntXX` modules

```
use int.Abs
use mach.int.Int32

let abs (x: int32): int32
  ensures { to_int result = abs (to_int x) }
= if x < of_int 0 then -x else x
```

- Advantage: all operations are checked
- Disadvantage: mixing `mach.int` types can be annoying

Machine integers

Approach 3: `mach.int.IntXX` modules

```
use int.Abs
use mach.int.Int32

let abs (x: int32): int32
  ensures { result = abs x }
= if x < 0 then -x else x
```

- Implicit conversions to `int` in many cases

Machine integers

Approach 3: `mach.int.IntXX` modules

```
use int.Abs
use mach.int.Int32

let abs (x: int32): int32
  ensures { result = abs x }
= if x < 0 then -x else x
```

- Implicit conversions to `int` in many cases
- **Note:** *this program contains a bug!*

Floating point

Why3 also supports reasoning about *floating point*.

- `ieee_float.mlw` in the standard library
- Conversions to/from bit vectors
- GAPPA theorem prover

Modelling pointers

Aliassing, revisited

How to model this C program?

```
void combine (int *x, int *y)
{
    *x = *x + *y;
    *y = 0;
}
```

Aliassing, revisited

How to model this C program?

```
void combine (int *x, int *y)
{
    *x = *x + *y;
    *y = 0;
}
```

This does not accurately capture it:

```
let combine (x y: ref int)
  ensures { !x = old (!x + !y) }
  ensures { !y = 0 }
= x := !x + !y; y := 0
```

since `x`, `y` cannot alias in Why3...

Making memory explicit

Solution: model global memory using arrays:

```
type pointer = int

val mem: array int

let combine (x y: pointer)
  requires { 0 <= x < length mem /\ 0 <= y < length mem }
  ensures { x <> y -> mem[x] = old (mem[x]+mem[y]) }
  ensures { mem[y] = 0 }
= mem[x] <- mem[x]+mem[y];
  mem[y] <- 0
```

Making memory explicit

Now we can verify this program:

```
let main()
  requires { length mem > 0x100000 }
= let x: pointer = 0x4234 in
  let y: pointer = 0x4202 in
  combine x x;
  assert { mem[x] = 0 };
  mem[x] <- 2; mem[y] <- 3;
  combine x y;
  assert { mem[x] = 5 };
```

Making memory explicit

Now we can verify this program:

```
let main()
  requires { length mem > 0x100000 }
= let x: pointer = 0x4234 in
  let y: pointer = 0x4202 in
  combine x x;
  assert { mem[x] = 0 };
  mem[x] <- 2; mem[y] <- 3;
  combine x y;
  assert { mem[x] = 5 };
```

This works very well, although:

- Pointers are untyped
- Pointers are integers

Making memory explicit

Working with global memory has drawbacks:

```
let combine (x y: pointer)
  requires { 0 <= x < length mem /\ 0 <= y < length mem }
  ensures { x <> y -> mem[x] = old (mem[x]+mem[y]) }
  ensures { mem[y] = 0 }
= mem[x] <- mem[x]+mem[y];
  mem[y] <- 0

let main ()
  requires { length mem > 0x100000 }
= let x: pointer = 0x4234 in
  let y: pointer = 0x4202 in
  mem[y] <- 5;
  combine x x;
  assert { mem[y] = 5 } (* this doesn't verify!? *)
```

Making memory explicit

Working with global memory has drawbacks:

```
let combine (x y: pointer)
  requires { 0 <= x < length mem /\ 0 <= y < length mem }
  ensures { x <> y -> mem[x] = old (mem[x]+mem[y]) }
  ensures { mem[y] = 0 }
  ensures { forall i. 0 <= i < length mem /\ x<>i /\ y<>i ->
                      mem[i] = old mem[i] }
= mem[x] <- mem[x]+mem[y];
  mem[y] <- 0

let main ()
  requires { length mem > 0x100000 }
= let x: pointer = 0x4234 in
  let y: pointer = 0x4202 in
  mem[y] <- 5;
  combine x x;
  assert { mem[y] = 5 } (* ok! *)
```

Pointer arithmetic

This style can also be used to model C arrays:

```
void clear(int* p, size_t n)
{
    for(int i=0; i<n; i++) p[i] = 0;
}
```

Pointer arithmetic

This style can also be used to model C arrays:

```
void clear(int* p, size_t n)
{
    for(int i=0; i<n; i++) p[i] = 0;
}
```

```
type pointer = int
val mem: array int
let clear (p: pointer) (n: int)
  requires { 0 <= p <= p+n < length mem }
= for i = 0 to (n-1) do mem[p+i] <- 0 done
```

A ‘simple’ pointer model

We can make a custom datatype, based on `array.Array`:

- Index using pointers instead of `int`
- Unlimited size

A ‘simple’ pointer model

We can make a custom datatype, based on `array.Array`:

- Index using pointers instead of `int`
- Unlimited size

```
type pointer 'a

val constant null: pointer 'a

val (==) (p q: pointer 'a): bool
  ensures { result <-> p = q }

type storage 'a = { mutable ghost cell: pointer 'a -> option 'a }
  invariant { cell null = None }

predicate valid (m: storage 'a) (p: pointer 'a)
  = m.cell p <> None
```

A ‘simple’ pointer model: WhyML interface

Dereferencing a pointer:

```
val function ([]) (mem: storage 'a) (p: pointer 'a): 'a
  requires { valid mem p }
  ensures { Some result = mem.cell p }
```

Updating values through a pointer:

```
val ([]<-) (mem: storage 'a) (p: pointer 'a) (v: 'a): unit
  writes { mem }
  requires { valid mem p }
  ensures { mem.cell = Map.set (old mem).cell p (Some v) }
```

A ‘simple’ pointer model: alloc and dealloc

A substitute for `malloc`:

```
val fresh (mem: storage 'a) (v: 'a): pointer 'a
  writes { mem }
  ensures { (old mem).cell result = None }
  ensures { exists v. mem.cell = Map.set (old mem).cell result (Some v) }
```

A ‘simple’ pointer model: alloc and dealloc

A substitute for `malloc`:

```
val fresh (mem: storage 'a) (v: 'a): pointer 'a
  writes { mem }
  ensures { (old mem).cell result = None }
  ensures { exists v. mem.cell = Map.set (old mem).cell result (Some v) }
```

And `free`:

```
val discard (mem: storage 'a) (p: pointer 'a): pointer 'a
  writes { mem }
  ensures { mem.cell = Map.set (old mem).cell p None }
```

A ‘simple’ pointer model: applying

Results:

```
use SimpleMemoryModel

val loc: storage int

let combine (x y: pointer int)
  requires { valid loc x /\ valid loc y }
  ensures { x <> y -> loc[x] = old (loc[x]+loc[y]) }
  ensures { loc[y] = 0 }
  ensures { forall p. valid (old loc) p -> valid loc p }
= loc[x] <- loc[x]+loc[y];
  loc[y] <- 0

let main()
= let x = fresh loc in
  let y = fresh loc in
  combine x x;
  assert { loc[x] = 0 }
```

Modelling pointers

Better is not always *better*:

- More detail in model $\rightarrow$ more accurate

Modelling pointers

Better is not always *better*:

- More detail in model $\rightarrow$ more accurate
- More detail in model $\rightarrow$ more room for errors
- More detail in model $\rightarrow$ more details to prove

Modelling pointers

Better is not always *better*:

- More detail in model $\rightarrow$ more accurate
- More detail in model $\rightarrow$ more room for errors
- More detail in model $\rightarrow$ more details to prove

Einstein principle:

> *Keep things as simple as possible, but no simpler.*

Applications of Why3

Applications of Why3

Why3 is often used as an intermediate tool

- *Modelling* a program written in another language
  - SPARK (Ada), Frama-C (C), Krakatoa (Java) ...

Applications of Why3

Why3 is often used as an intermediate tool

- *Modelling* a program written in another language
  - SPARK (Ada), Frama-C (C), Krakatoa (Java) ...
- An *interface* to various provers
  - EasyCrypt: cryptographic proofs

Applications of Why3

Why3 is often used as an intermediate tool

- *Modelling* a program written in another language
  - SPARK (Ada), Frama-C (C), Krakatoa (Java) ...
- An *interface* to various provers
  - EasyCrypt: cryptographic proofs

Background
Static versus Dynamic Verification in Why3, Frama-C and SPARK 2014 – Kosmatov et al.

`https://hal.inria.fr/hal-01344110`

Two approaches of C verification

C $\rightarrow$ **WhyML** Model C in WhyML

WhyML $\rightarrow$ **C** Write in WhyML, then translate to C

Two approaches of C verification

C $\rightarrow$ **WhyML** Model C in WhyML

WhyML $\rightarrow$ **C** Write in WhyML, then translate to C

Mechanizations to both approaches exist:

C $\rightarrow$ **WhyML** Frama-C/Jessie, Frama-C/WP

WhyML $\rightarrow$ **C** Why3 *code extraction*

Frama-C/Jessie, Frama-C/WP

- Both combine C code with ACSL annotations
- C + ACSL instead of WhyML *programs* + WhyML *logic*
- Both approaches have some limitations:
  - Details of C demand more annotations
  - Exactly *what* are the semantics of C?

Frama-C/Jessie, Frama-C/WP

- Both combine C code with ACSL annotations
- C + ACSL instead of WhyML *programs* + WhyML *logic*
- Both approaches have some limitations:
  - Details of C demand more annotations
  - Exactly *what* are the semantics of C?

```
int x, y;
y = (x=3)+(x=4);
//@assert y == 8;
```

C with ACSL annotations

```
/*@ requires 0 <= x <= 46340*46340-1;
  @ ensures sqr(\result) <= x && sqr(\result+1) > x;
  @ assigns \nothing; */
int isqrt(int x)
{
        int z = 0, odd = 1, sum = 1;
        /*@ loop invariant sum == sqr(z+1);
          @ loop invariant odd == 2*z+1;
          @ loop invariant sqr(z) <= x;
          @ loop assigns z, sum, odd;
          @ loop variant x - z; */
        while(sum <= x) {
                z++;
                sum += odd += 2;
        }
        return z;
}
```

Program extraction

The other direction: *generate* code instead of *modelling* it!

- Targets: OCaml, CakeML, C
- `why3 extract -D` *`language file.mlw`*

Limitations:

- Not every Why3 construct can be translated to C
  - E.g., `int.Int`, `array.Array`
  - Very much a *work in progress*

In short

It is (still) easiest to *model by hand*.

- Can leave out unnecessary details
- Can use all WhyML constructs

Downside:

- You have to *document* and *justify* your model!

Wrap up

Is it sometimes the case that 1 = 2?

Sometimes things can go wrong!

- Typos in Why3 *axioms*
- Soundness bugs in provers happen
  - In the past: Alt-Ergo 1.30, Z3 4.3.2
  - If you find any, report these!

Is it sometimes the case that 1 = 2?

Sometimes things can go wrong!

- Typos in Why3 *axioms*
- Soundness bugs in provers happen
  - In the past: Alt-Ergo 1.30, Z3 4.3.2
  - If you find any, report these!

You could improve things:

- Why3 is open source
- Invent new verification techniques using Why3
- Find interesting bugs in software using Why3

Final remarks

- Carnival, so light homework.
- Deadline: Tuesday 18:00
- Coming weeks:
  - Why3 project assignment details available on Tuesday
  - Wednesday: *discussion hour* MERC 1.00.28 at 10:30
  - Thursday: no more lectures!
  - Questions? My office: M1.3.03, or `mschool@cs.ru.nl`
- After the break: *Coverity*
  - Thursday 11 April, 8:30 – guest lecture by Rody Kersten