The Why and How of Deductive Program Verification

Jean-Christophe Filliâtre

Marktoberdorf Summer School 2026

lecture 3

today: programs to the rescue of logic

1. axioms and definitions
2. lemma functions
3. ghost code

we need logical symbols (constants, functions, and predicates) for the purpose of specification and/or pure code

we have seen several examples already, e.g.

- 0, +, >, etc.
- `sum`
- `odd`, `half`, `div`

sometimes the symbol is known to the prover

for instance, SMT solvers include integer arithmetic as one of their theories

so we simply declare such symbols on the Why3 side

```
function (+) int int : int
function (-) int int : int
function (*) int int : int
...
```

and map them directly to the SMT symbols

no axiom or definition is needed

we have to be careful anyway

a good example is division

SMT solvers know about division, so we can declare

```
function div int int : int
function mod int int : int
```

and map these to functions `div` and `mod` of SMT

is this Euclidean division?
or truncated division?
or floored division?
or even something else?

we have to carefully

- read the SMT documentation, make some tests
- document the division on the Why3 side

(see "Division and Modulus for Computer Scientists" by Dan Leijen)

the logic of SMT solvers is total, and so is that of Why3

thus `div 1 0` is a valid term of type `int`

```
goal G0: div 1 0 = div 1 0  (* provable *)
```

yet we know nothing about it

```
goal G1: div 1 0 = div 2 0  (* not provable *)
```

there are other options: partial logic, total logic with $x/0 \stackrel{\text{def}}{=} 0$, etc.

sometimes the logical symbols can be defined,
for instance as mere shortcuts

examples:

```
function half (n: int) : int =
  div n 2

predicate odd (n: int) =
  mod n 2 = 1
```

you have to read it,
understand it, and accept it

this is directly mapped to a similar definition in the SMT input

when the symbol is not predefined, and not easily definable,
we can resort to an axiomatization

```
function fact (n: int) : int
axiom Fact0: fact 0 = 1
axiom Factn: forall n. n > 0 -> fact n = n * fact (n-1)
```

symbol `fact` is only axiomatized on nonnegative integers

we know nothing about `fact (-1)` or `fact (-2)`

Why3 source

```
function fact...
axiom Fact0: ...
axiom Factn: ...

goal G: fact 3 = 6
```

SMT input

```
(declare-fun fact (Int) Int)

;; "Fact0"
(assert (= (fact 0) 1))

;; "Factn"
(assert (forall ((n Int))
  (=> (< 0 n)
      (= (fact n)
         (* n (fact (- n 1)))))))

;; Goal "G"
(assert (not (= (fact 3) 6)))
(check-sat)
```

axioms are dangerous; a tiny mistake can lead to an inconsistency

example:

```
function fact (n: int) : int
axiom Fact0: fact 0 = 1
axiom Factn: forall n. n >= 0 -> fact n = n * fact (n-1)

goal G: fact 3 = 5  (* proved! *)
```

SMT solvers are good at finding inconsistencies,
as in the previous example,

yet checking the consistency of a set of axioms remains undecidable

the declaration of a program function,
such as

```
val bertrand (n: int) : int
  requires { n > 0 }
  ensures  { prime result }
  ensures  { n < result < 2*n }
```

this is wrong!

is equivalent to an axiom

indeed, WP is going to introduce facts in the logical context, which we only claimed in the function contract

what can we do to prevent the introduction of inconsistencies?

- proofread axioms carefully
- try to derive `false` (this is called smoke detector)
- replace axioms with a definition

it would be much better to define function `fact` instead

problem: this is a recursive definition, and it is important to ensure termination
(think about `function f (n: int) : int = 1 + f n`)

solution: we have everything we need on the program side (variant, vcgen)
so let us define `fact` as a program

this is a program

and also a logical symbol

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

termination required

we get a VC, as for any other program function

Why3 source

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

goal G: fact 3 = 6
```

SMT input

```
(declare-fun fact (Int) Int)

;; "fact'def"
(assert (forall ((n Int))
  (=>
    (<= 0 n)   ← precondition
    (ite (= n 0)
      (= (fact n) 1)
      (= (fact n)
           (* n (fact (- n 1)))))
  )))

;; Goal "G"
(assert (not (= (fact 3) 6)))
(check-sat)
```

for a program to be accepted as a logical symbol,

- it must terminate
- it must be pure:
  no mutation of a global variable, no exception raised, etc.

Why3 checks all that

$$\sum_{a \leqslant i < b} f(i)$$

```
let rec function sum (f: int -> int) (a b: int) : int
  variant { b - a }
= if b <= a then 0 else sum f a (b - 1) + f (b - 1)
```

lemma functions

say we want to prove that $n! \geqslant 1$ for all $n \geqslant 0$

```
let rec function fact ...

lemma fact_pos: forall n. n >= 0 -> fact n >= 1
```

this requires induction, and SMT solvers do not perform induction

we can prove this using a function; this is called a lemma function

this is a program

and also a lemma

no return value

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

termination required

we get a VC, as for any program function

$$
\begin{array}{l}
\forall n. \\
\quad n \geqslant 0 \rightarrow \qquad \longleftarrow \textit{ precondition} \\
\quad\quad n > 0 \rightarrow \\
\quad\quad\quad n - 1 < n \wedge 0 \leqslant n \wedge \qquad \longleftarrow \textit{ variant decreases} \\
\quad\quad\quad n - 1 \geqslant 0 \wedge \qquad \longleftarrow \textit{ precondition recursive call} \\
\quad\quad\quad (\texttt{fact}\ (n-1) \geqslant 1 \rightarrow \texttt{fact}\ n \geqslant 1) \wedge \\
\quad\quad \neg(n > 0) \rightarrow \\
\quad\quad\quad \texttt{fact}\ n \geqslant 1
\end{array}
$$

postcondition recursive call = induction hypothesis $\longrightarrow$

generally speaking, a definition

$$\texttt{let lemma } f\ \vec{v}\ \texttt{: unit requires } P \texttt{ ensures } Q\ =\ e$$

generates

- a verification condition for f
- a ghost function f
- the axiom $\forall \vec{v}.\, P \to Q$, added to the context

Why3 source

```
let rec function
  fact...

let rec lemma
  fact_pos...

goal T:
  fact 987 > 0
```

SMT input

```
(declare-fun fact (Int) Int)

;; "fact'def"
(assert (forall ((n Int))
  ...

;; "fact_pos"
(assert (forall ((n Int))
  (=> (<= 0 n) (<= 1 (fact n)))))

;; Goal "T"
(assert (not (< 0 (fact 987))))
(check-sat)
```

like any other axiom

a lemma function is a ghost function, that we can call anywhere in the middle of a program

```
...
fact_pos 42;
...
```

now we have `fact 42` $\geqslant 1$ *in the context*

a lemma function does not have to be a recursive function

a loop is perfectly fine

```
let lemma fact_pos_loop (n: int) : unit
  requires { n >= 0 } ensures { fact n >= 1 }
= for i = 1 to n do invariant { fact (i - 1) >= 1 } () done
```

only the function contract matters; the function body can be anything
(provided it is observationally pure and terminating)

(do Fibonacci numbers really need an introduction?)

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \quad \text{for } n \geqslant 2 \end{aligned}$$

1. define Fibonacci numbers with `let rec function`
2. show that $\forall n.\, F_n \geqslant 0$ using a recursive lemma function
3. show that $\forall n.\, F_n \geqslant 0$ using a `for` loop
4. verify a function that computes F_n using an array (code is given)
5. verify a function that computes F_n using two variables (code is given)

ghost code

data and code added to the program
to make the specification and/or the proof simpler

it is not going to be executed

we search the smallest Fibonacci number greater than n

```
let first_fib (n: int) : (r: int)
  requires { n >= 0 }
  ensures  { exists i. i >= 0 /\ fib i <= n < fib (i+1) = r }
= let ref a = 0 in
  let ref b = 1 in

  while b <= n do


    a, b <- b, a+b
  done;
  return b
```

```
let first_fib (n: int) : (r: int)
  requires { n >= 0 }
  ensures  { exists i. i >= 0 /\ fib i <= n < fib (i+1) = r }
= let ref a = 0 in
  let ref b = 1 in

  while b <= n do
    invariant { exists i. i >= 0 /\
                  a = fib i <= n /\ b = fib (i+1) }
    a, b <- b, a+b
  done;
  return b
```

keep track of i with a ghost variable

```
let first_fib (n: int) : (r: int)
  requires { n >= 0 }
  ensures  { exists i. i >= 0 /\ fib i <= n < fib (i+1) = r }
= let ref a = 0 in
  let ref b = 1 in
  let ghost ref i = 0 in    ← here
  while b <= n do
    invariant { i >= 0 /\ a = fib i <= n /\ b = fib (i+1) }
    a, b <- b, a+b;
    i <- i+1 ←——— this is ghost code
  done;
  return b
```

return *i* as a ghost output


```
let first_fib (n: int) : (r: int, ghost i: int)
  requires { n >= 0 }
  ensures  { i >= 0 /\ fib i <= n < fib (i+1) = r }
= let ref a = 0 in
  let ref b = 1 in
  let ghost ref i = 0 in
  while b <= n do
    invariant { i >= 0 /\ a = fib i <= n /\ b = fib (i+1) }
    a, b <- b, a+b;
    i <- i+1
  done;
  return b, i
```

```
...
let f, i = first_fib 1000 in
...
```

this is ghost

to model data structures

```
type hash_table = {
  buckets: array (list (key, value));
  ghost m: key -> list value;
}
invariant { forall k v. ... }
```

implementation

model

gluing invariant

you cannot build values

```
type int32 = abstract {
  ghost value: int;
}
invariant { -0x8000_0000 <= value <= 0x7fff_ffff }
```

```
val (+) (a: int32) (b: int32) : int32
  requires { -0x8000_0000 <= a.value + b.value <= 0x7fff_ffff }
  ensures  { result.value = a.value + b.value }
```

this way, SMT solvers only handle mathematical integers

the code is verified with its ghost code,
but executed without it

consequences:

- ghost code may read regular data but can't modify it
- regular code does not see ghost data
- ghost code cannot modify the control flow of regular code

this is the principle of non interference

```
let f x =
  ...
  ghost while true do () done;   ← would change the control flow
  ...   ← everything now verified
```

Why3 checks the non interference of ghost code with regular code

for any program expression, Why3 keeps track of

- its possible non termination
- exceptions possibly raised
- variables possibly read and modified
  (memory aliases are statically known)

see The Spirit of Ghost Code
JCF, Léon Gondelman, Andrei Paskevich
FMSD 2016

takeaways

axioms are dangerous

no mechanical way to chase inconsistencies

a definition is always preferable

a proof can be a program

and not even necessarily a pure program

thank you