Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Formal Verification with Lean

Presentation for the Functional Programming Seminar

Ștefan-Octavian Radu

Advisor: Felix Herrmann

08.02.2024

https://www.informatik.uni-wuerzburg.de/is/

Formal Verification

What is Formal Verification?

- Proving through *formal methods* that a program behaves „*as expected*".

Formal Methods?

- the usage of mathematically rigorous techniques (i.e. proofs)

As Expected?

- execution strictly follows a formal description of its behavior (specification)

Formal Verification

- in the context of Programming Languages
- formal modelling of the semantics (meaning) of the programming language
- 3 ways:
  - operational semantics
  - axiomatic semantics
  - denotational semantics

Axiomatic Semantics (Floyd-Hoare Logic)

- useful for reasoning about concrete programs
- framework for deducing valid formulas in a mechanical way

Hoare Triple:

(precondition) **{P} C {Q}** *(postcondition)*

(command)

“Whenever P holds before the execution of C, Q will hold after the execution, or C does not terminate”

- thinking in terms of preconditions and postconditions
- P, Q are formulas in first order logic
- can only prove partial correctness

Axiomatic Semantics (Floyd-Hoare Logic)

- based on derivation rules for an imperative programming language

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ \textsc{Skip}$$

$$\frac{\{P\}\ S\ \{R\} \qquad \{R\}\ T\ \{Q\}}{\{P\}\ S;\ T\ \{Q\}}\ \textsc{Seq}$$

$$\frac{\{P \land B\}\ S\ \{Q\} \qquad \{P \land \neg B\}\ T\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T\ \{Q\}}\ \textsc{If}$$

etc.

Dafny
Programming Language

Denotational Semantics

- “denotation” = the class of objects that a statement refers to (dict. def.)
- describe meaning of programs (*what a program means*)
- mapping to mathematical objects / functions

“idealised compiler”: source code to mathematics

- compositionality
- S, T on the right hand side of the equality only as arguments to $[\![\,]\!]$

$$
\begin{aligned}
[\![\texttt{S ; T}]\!] &= \ldots [\![\texttt{S}]\!] \ldots [\![\texttt{T}]\!] \ldots \\
[\![\texttt{if B then S else T}]\!] &= \ldots [\![\texttt{S}]\!] \ldots [\![\texttt{T}]\!] \ldots \\
[\![\texttt{while B do S}]\!] &= \ldots [\![\texttt{S}]\!] \ldots
\end{aligned}
$$

Denotational Semantics

For arithmetics

```
eval : Stx → (String → ℤ ) → ℤ
-- expressions such as
eval (Stx.add e1 e2) st = (eval e1 st) + (eval e2 st)
```

For a programming language

```
Statement → { State × State }
-- e.g.
nop     → { (s, s) | s ∈  States }
x = a   → { (s, t) | s, t ∈  States, t = s[x ← a] }
S ; T   → r1 ∘ r2 = { (a, c) | ∃ b, (a, b) ∈  r1 ∧ (b, c) ∈  r2 }
```

etc.

Operational Semantics

- describes how a program is executed (*what a program does*)
- big-step operational semantics (natural semantics)
  - transition from the initial state **directly** to the final state

$$(S, s) \Rightarrow t$$

Starting in a state s, executing S terminates in the state t

- small-step operational semantics
  - transitions accounting for **one step** of execution

$$(S, s) \Rightarrow (T, t)$$

Starting in a state s and executing one step of S, leaves the program T to be executed in state t

Proof Assistants / Interactive Theorem Provers

- aid in the development of formal proofs
- software tools:
  - Isabelle/HOL
  - Coq
  - Lean

```
example: p → q → (q ∧ p) :=
by
  intros hp hq
  apply And.intro
  . assumption
  . assumption
```

▶ goals accomplished 🎉

About Lean

- Interactive Theorem Prover
- Developed at Microsoft Research
  - by Leonardo de Moura
  - also known for Z3 (smt solver), among others

- Based on **Dependent Type Theory**
- **Also** a standalone powerful **general-purpose** functional programming language
  - Lean is actually implemented in …

LEAN 🙀

Functions and Definitions

- simple definitions

```
def a_nat: Nat := 42
def hello: String := "hello"
def world := " world!"
def hello_world := hello ++ world

#check a_nat          ■ a_nat : Nat
#check hello_world        ■ hello_world : String
#eval hello_world       ■ "hello world!"
```

- functions in two ways

```
def add_1 (k: Nat): Nat := k + 1
def add_1': Nat → Nat := λ k =>  k + 1
def add_2 (a b: Nat) := a + b

#check add_1           ■ add_1 (k : Nat) : Nat
#check add_1'          ■ add_1' (a✝ : Nat) : Nat
#check add_1 3         ■ add_1 3 : Nat
```

Pattern Matching

```
#print Nat
-- inductive Nat : Type
-- number of parameters: 0
-- constructors:
-- Nat.zero : Nat
-- Nat.succ : Nat → Nat

def factorial n :=
  match n with
  | 0      => 1
--| zero => 1
  | k + 1   => (k + 1) * factorial k
--| succ k => (k + 1) * factorial k
```

Functional programming in Lean

List operations

```
#eval [1, 2, 3]                   ▪ [1, 2, 3]
#eval 5 :: [1, 2]                 ▪ [5, 1, 2]
#eval [1, 2] ++ [3, 4]            ▪ [1, 2, 3, 4]
#eval (List.range 5)              ▪ [0, 1, 2, 3, 4]
```

Special commands

```
#check -- gives the type
#eval  -- evaluates
#print -- prints definition
```

Higher order functions

```
#eval (List.range 5).map (λ x => x ^ 2)         ▪ [0, 1, 4, 9, 16]
#eval  (λ x => x ^ 2) <$> (List.range 5)        ▪ [0, 1, 4, 9, 16]

#eval  (List.range 5).foldl (λ x y => x + y) 0  ▪ 10
#eval  (List.range 5).foldl (. + .) 0           ▪ 10

#eval  (List.range 5).filter (λ x => x ≤ 2)     ▪ [0, 1, 2]
#eval  (List.range 5).filter (. ≤ 2)            ▪ [0, 1, 2]
```

- structures
- type classes
- inductive types
- monads
- …
- etc.

Dependent Types

- Type Theory ⇒ Bertrand Russell
- Types can depend of parameters
- “Dependently typed programs are, by their nature, proof carrying code” - Altenkirch, McBride, McKinna, Why Dependent Types Matter

```
def k: Nat := 3

def p_eq_p {p: Prop}: p = p := by rfl

#check p_eq_p      ■ p_eq_p {p : Prop} : p = p
#check p = p       ■ p = p : Prop
#check Prop        ■ Prop : Type
#check Type        ■ Type : Type 1
#check Type 1      ■ Type 1 : Type 2
#check Type 2      ■ Type 2 : Type 3
#check Type 3      ■ Type 3 : Type 4
-- ...
```

```
def strange_foo (b: Bool)
  : if b then String else Nat :=
    match b with
    | true => "lean"
    | false => (42: Nat)

#eval strange_foo true       ■ "lean"
#eval strange_foo false      ■ 42
```

Tactic Mode

```
theorem test₁: p → q → (q ∧ p) :=
  λ hp hq => And.intro hq hp
```

```
#check And.intro
```

```
And.intro {a b : Prop} (left : a)
(right : b) : a ∧ b
```

```
theorem test₂: p → q → (q ∧ p) :=
by
```

```
▶ 1 goal
p q : Prop
⊢ p → q → q ∧ p
```

Tactic Mode

```
theorem test₁: p → q → (q ∧ p) :=
  λ hp hq => And.intro hq hp
```

```
#check And.intro
```

```
And.intro {a b : Prop} (left : a)
(right : b) : a ∧ b
```

```
theorem test₂: p → q → (q ∧ p) :=
by
  intros hp hq
```

```
▶ 1 goal
p q : Prop
hp : p
hq : q
⊢ q ∧ p
```

Tactic Mode

```lean
theorem test₁: p → q → (q ∧ p) :=
  λ hp hq => And.intro hq hp
```

```lean
#check And.intro
```

```
And.intro {a b : Prop} (left : a)
(right : b) : a ∧ b
```

```lean
theorem test₂: p → q → (q ∧ p) :=
by
  intros hp hq
  apply And.intro
```

```
▶ 2 goals
case left
p q : Prop
hp : p
hq : q
⊢ q
```

```
case right
p q : Prop
hp : p
hq : q
⊢ p
```

Tactic Mode

```
theorem test₁: p → q → (q ∧ p) :=
  λ hp hq => And.intro hq hp
```

```
#check And.intro
```

```
And.intro {a b : Prop} (left : a)
(right : b) : a ∧ b
```

```
theorem test₂: p → q → (q ∧ p) :=
by
  intros hp hq
  apply And.intro
  . assumption
```

```
case left
p q : Prop
hp : p
hq : q
⊢ q
```

▶ goals accomplished 🎉

Tactic Mode

```
theorem test₁: p → q → (q ∧ p) :=
  λ hp hq => And.intro hq hp
```

```
#check And.intro
```

```
And.intro {a b : Prop} (left : a)
(right : b) : a ∧ b
```

```
theorem test₂: p → q → (q ∧ p) :=
by
  intros hp hq
  apply And.intro
  . assumption
  . assumption
```

```
case left
p q : Prop
hp : p
hq : q
⊢ q
```

▶ goals accomplished 🎉

```
case right
p q : Prop
hp : p
hq : q
⊢ p
```

Big-Step Semantics

(S, s) ⇒ t

Starting in a state s, executing S terminates in the state t

…

what is S?

we need a programming language

Brainfuck

BF	Meaning
>	Increment the pointer.
<	Decrement the pointer.
+	Increment the byte at the pointer.
-	Decrement the byte at the pointer.
.	Output the byte at the pointer.
,	Input a byte and store it in the byte at the pointer.
[	Jump forward past the matching] if the byte at the pointer is zero.
]	Jump backward to the matching [unless the byte at the pointer is zero.

BF	C equivalent
>	++p;
<	--p;
+	++*p;
-	--*p;
.	putchar(*p);
,	*p = getchar();
[	while (*p) {
]	}

```
+[-->-[>>+>-----<<]<--<---]>-.>>>+.>>..+++[.>]<<<<.+++.------.<<-.>>>>+.
```

Define the Syntax

```
inductive Op : Type where
  | nop       : Op                -- nop
  | pInc      : Op                -- >
  | pDec      : Op                -- <
  | vInc      : Op                -- +
  | vDec      : Op                -- -
  | input     : Op                -- ,
  | output    : Op                -- .
  | brakPair  : Op -> Op          -- [ s ]
  | seq       : Op -> Op -> Op    -- s t
```

```
def toString op :=
    match op with
    | Op.nop => ""
    | Op.pInc => ">"
    | Op.pDec => "<"
    | Op.vInc => "+"
    | Op.vDec => "-"
    | Op.output => "."
    | Op.input => ","
    | Op.brakPair op' => "[" ++ toString op' ++ "]"
    | Op.seq op1 op2 => (toString op1) ++ (toString op2)

instance: ToString Op where
  toString op := op.toString
```

Define the Syntax

```
-- [->+<]
#eval (brakPair (seq vDec (seq pInc (seq vInc pDec))): Op)      ■ [->+<]
```

```
notation "#"    => Op.nop
notation ">"    => Op.pInc
notation "<"    => Op.pDec
notation "+"    => Op.vInc
notation "~"    => Op.vDec
notation "^"    => Op.output
notation ","    => Op.input
notation "[" ops "]" => (Op.brakPair ops)
notation a:50 "_" b:51  => Op.seq a b
```

```
#eval ([~_>_+_<]: Op)      ■ [->+<]
```

Big-Step Semantics

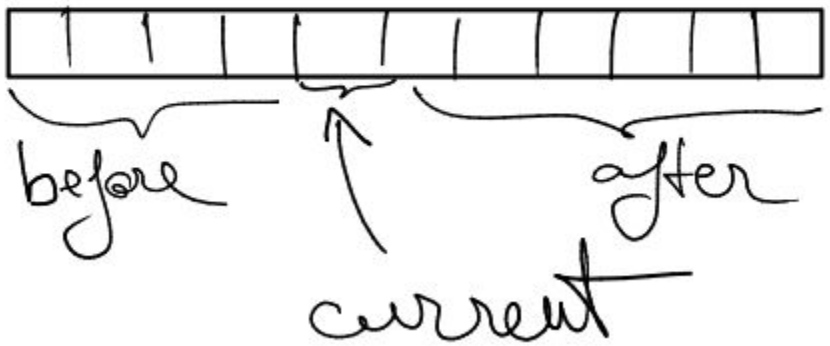
(S, s) ⇒ t

Starting in a state s, executing S terminates in the state t

…

what are <u>s</u> and <u>t</u>?

Define the state




```
#check State.mk [1, 2, 3] "" [] 0 [0, 0, 0]
```

```
structure State : Type where
  inp: List Nat
  out: String
  before: List Nat
  current: Nat
  after: List Nat
  deriving Repr

notation "*" s:100 => State.current s
```

Define State Operations

>

```
def applyPInc (s: State): State :=
    match s.after with
      | [] => s -- if the end of the band is reached, do nothing
      | h :: t => ⟨s.inp, s.out, *s :: s.before, h, t⟩
```

–

```
def applyVDec (s: State): State :=
  match *s with
  | 0 => ⟨s.inp, s.out, s.before, 0, s.after⟩
  | k + 1 => ⟨s.inp, s.out, s.before, k, s.after⟩
```

etc.

;

```
def applyInput (s: State): State :=
  match s.inp with
    | [] => ⟨[], s.out, s.before, *s, s.after⟩
    | h :: t => ⟨t, s.out, s.before, h, s.after⟩
```

Big-Step Semantics

(S, s) ⇒ t

Starting in a state s, executing S terminates in the state t

Derivation rules!

Warmup Theorem!

```
(--, [2] ⟹ [0])  -- (*s -= 2, [2] ⟹ [0])

theorem dec_2: (~_~, (State.mk [] "" [] 2 [])) ⟹ State.mk [] "" [] 0 [] :=
```

Warmup Theorem!

```
(--, [2] ⟹ [0])  -- (*s -= 2, [2] ⟹ [0])

theorem dec_2: (~_~, (State.mk [] "" [] 2 [])) ⟹ State.mk [] "" [] 0 [] :=
```

```
inductive BigStep: Op × State → State → Prop where
 | vDec s: BigStep (Op.vDec, s) s.applyVDec
 | seq (S s T t u)
   (h:  BigStep (S, s) t)
   (h': BigStep (T, t) u):
     BigStep ((Op.seq S T), s) u
```

$$\frac{(S, s) \Rightarrow t \qquad (T, t) \Rightarrow u}{(S; T, s) \Rightarrow u}$$

```
infix:110 " ⟹" => BigStep
```

Proving Theorems

Warmup Theorem!

```
theorem dec_2: (~_~, (State.mk [] "" [] 2 [])) ⟹ State.mk [] "" [] 0 [] :=
by
  apply BigStep.seq
  case h =>
    apply BigStep.vDec
  case h' =>
    apply BigStep.vDec
```

▶ goals accomplished 🎉

```
inductive BigStep: Op × State → State → Prop where
```

```
| vDec s: BigStep (Op.vDec, s) s.applyVDec
```

```
| seq (S s T t u)
  (h:  BigStep (S, s) t)
  (h': BigStep (T, t) u):
   BigStep ((Op.seq S T), s) u
```

```
-- ([-], [n] ⟹ [0]) -- (while(*s) { *s -= 1}, [n] ⟹ [0])

theorem dec_n {n: Nat}: ([~], (State.mk [] "" [] n []))
  ⟹ State.mk [] "" [] 0 [] :=
```

Proving Theorems

```
-- ([-], [n] ⟹ [0]) -- (while(*s) { *s -= 1}, [n] ⟹ [0])
```

```
| brakPairTrue {ops} {s t u: State}
  (c: *s ≠ 0)
  (body: BigStep (ops, s) t)
  (rest: BigStep ((Op.brakPair ops), t) u):
    BigStep (Op.brakPair ops, s) u
| brakPairFalse ops (s: State) (c: *s = 0):
    BigStep (Op.brakPair ops, s) s
```

$$\frac{(S, s) \Rightarrow t \qquad ([S], t) \Rightarrow u}{([S], s) \Rightarrow u}\ [\text{TRUE}]$$

$$\frac{}{([S], s) \Rightarrow s}\ [\text{FALSE}]$$

Proving Theorems

```
-- ([-], [n] ⟹ [0]) -- (while(*s) { *s -= 1}, [n] ⟹ [0])

theorem dec_n {n: Nat}: ([~], (State.mk [] "" [] n []))
  ⟹ State.mk [] "" [] 0 [] :=
  by
    induction n
    case zero =>
      . apply BigStep.brakPairFalse
        . simp
    case succ d hd =>
      . apply BigStep.brakPairTrue
        . simp
        . apply BigStep.vDec
        . rw [State.applyVDec]
          simp
          assumption
```

```
| brakPairTrue {ops} {s t u: State}
  (c: *s ≠ 0)
  (body: BigStep (ops, s) t)
  (rest: BigStep ((Op.brakPair ops), t) u):
    BigStep (Op.brakPair ops, s) u
| brakPairFalse ops (s: State) (c: *s = 0):
    BigStep (Op.brakPair ops, s) s
```

Complete Big-Step Semantics

```
inductive BigStep: Op × State → State → Prop where
  | nop  (s: State): BigStep (Op.nop, s)  s
  | pInc (s: State): BigStep (Op.pInc, s) s.applyPInc
  | pDec (s: State): BigStep (Op.pDec, s) s.applyPDec
  | vInc s: BigStep (Op.vInc, s) s.applyVInc
  | vDec s: BigStep (Op.vDec, s) s.applyVDec
  | brakPairTrue {ops} {s t u: State}
    (c: *s ≠ 0)
    (body: BigStep (ops, s) t)
    (rest: BigStep ((Op.brakPair ops), t) u):
      BigStep (Op.brakPair ops, s) u
  | brakPairFalse ops (s: State) (c: *s = 0):
      BigStep (Op.brakPair ops, s) s
  | seq (S s T t u)
    (h:  BigStep (S, s) t)
    (h': BigStep (T, t) u):
      BigStep ((Op.seq S T), s) u
  | input s: BigStep (Op.input, s) s.applyInput
  | output s: BigStep (Op.output, s) s.applyOutput
```

Other Theorems?

```
def bfSum_in: Op := ,_>_,_<_(bfAddition)
#eval bfSum_in     ■ ,>,<[->+<]

-- sum a b = a + b

theorem bfSum: (bfSum_in, (State.mk (a :: b :: i) o l x (y :: r)))
  ⟹State.mk i o l 0 ((a + b) :: r) :=
```

```
#eval bfSwap'     ■ [<+>-]
def bfSwapTX: Op := >_(bfSwap')          -- x[t+x-]
def bfSwapXY: Op := >_(bfSwap')          -- y[x+y-]
def bfSwapYT: Op := <_<_(bfSwap'')       -- t[y+t-]

def bfSwap: Op := (bfSwapTX)_(bfSwapXY)_(bfSwapYT)
#eval bfSwap     ■ >[<+>-]>[<+>-]<<[>>+<<-]

theorem swap: (bfSwap, State.mk [] "" l 0 (x :: y :: r))
  ⟹State.mk [] "" l 0 (y :: x :: r) :=
```

Termination

```
#eval (Op.fromString "[<>.,+-]><")    ■ [<>.,+-]><
```

Termination

```
def fromString (s: String): Op :=
  parse (s.toUTF8.toList)
where
  parse (chrl: List UInt8): Op :=
    match chrl with
    | [] => nop
    | h :: t =>
      match h with
      | 62 => Op.seq Op.pInc   (parse t)          -- 62 >
      | 60 => Op.seq Op.pDec   (parse t)          -- 60 <
      | 43 => Op.seq Op.vInc   (parse t)          -- 43 +
      | 45 => Op.seq Op.vDec   (parse t)          -- 45 -
      | 46 => Op.seq Op.output (parse t)          -- 46 .
      | 44 => Op.seq Op.input  (parse t)          -- 44 ,
      | 91 =>                                     -- 91 [
        let head := t.takeWhile (λ x => x ≠ 93) -- 93 ]
        let tail := t.dropWhile (λ x => x ≠ 93) -- 93 ]
        let body := parse head
        let rest := parse tail
        Op.seq (Op.brakPair body) rest
      | _ => parse t -- anything else skip
```

Termination

```
def fromString (s: String): Op :=
  parse (s.toUTF8.toList)
where
  parse (chrl: List UInt8): Op :=
    match chrl with
    | [] => nop
    | h :: t =>
      match h with
      | 62 => Op.seq Op.pInc    (parse t)           -- 62 >
      | 60 => Op.seq Op.pDec    (parse t)           -- 60 <
      | 43 => Op.seq Op.vInc    (parse t)           -- 43 +
      | 45 => Op.seq Op.vDec    (parse t)           -- 45 -
      | 46 => Op.seq Op.output (parse t)           -- 46 .
      | 44 => Op.seq Op.input  (parse t)           -- 44 ,
      | 91 =>                                       -- 91 [
        let head := t.takeWhile (λ x => x ≠ 93) -- 93 ]
        let tail := t.dropWhile (λ x => x ≠ 93) -- 93 ]
        let body := parse head     ■ fail to show termination for   Op.fromString.parse
        let rest := parse tail
        Op.seq (Op.brakPair body) rest
      | _ => parse t -- anything else skip
```

Termination

```
def fromString (s: String): Op :=
  parse (s.toUTF8.toList)
where
  parse (chrl: List UInt8): Op :=
    match chrl with
    | [] => nop
    | h :: t =>
      match h with
      | 62 => Op.seq Op.pInc   (parse t)        -- 62 >
      | 60 => Op.seq Op.pDec   (parse t)        -- 60 <
      | 43 => Op.seq Op.vInc   (parse t)        -- 43 +
      | 45 => Op.seq Op.vDec   (parse t)        -- 45 -
      | 46 => Op.seq Op.output (parse t)        -- 46 .
      | 44 => Op.seq Op.input  (parse t)        -- 44 ,
      | 91 =>                                   -- 91 [
        let head := t.takeWhile (λ x => x ≠ 93) -- 93 ]
        have _ : head.length < t.length.succ :=
          by
            have h' : head = List.takeWhile (λ x => x ≠ 93) t := by simp
            rw [h']
            exact ln_take_l_lt_len_l t (λ x => x ≠ 93)

        let tail := t.dropWhile (λ x => x ≠ 93) -- 93 ]
        have _ : tail.length < t.length.succ :=
          by
            have h' : tail = List.dropWhile (λ x => x ≠ 93) t := by simp
            rw [h']
            exact ln_drop_l_lt_len_l t (λ x => x ≠ 93)

        let body := parse head
        let rest := parse tail
        Op.seq (Op.brakPair body) rest
      | _ => parse t -- anything else skip
```

```
theorem ln_take_l_lt_len_l {α : Type} (l: List α) (f: α → Bool):
  (l.takeWhile f).length < l.length.succ :=
  by
    induction l with
    | nil =>
      simp
      rw [List.takeWhile]
      simp
      exact Nat.zero_lt_succ 0
    | cons head tail h =>
      rw [List.takeWhile]
      cases f head with
      | false =>
        simp
        apply Nat.zero_lt_succ
      | true =>
        simp
        apply Nat.succ_lt_succ
        rw [Nat.succ_eq_add_one] at h
        assumption

theorem ln_drop_l_lt_len_l {α : Type} (l: List α) (f: α → Bool):
  (l.dropWhile f).length < l.length.succ :=
  by
    induction l with
    | nil =>
      rw [List.dropWhile]
      simp
      exact Nat.zero_lt_succ 0
    | cons head tail h =>
      rw [List.dropWhile]
      cases f head with
      | false =>
        simp
        apply Nat.succ_lt_succ
        exact Nat.lt_succ_self (List.length tail)
      | true =>
        simp
        have h' := Nat.lt_succ_self (Nat.succ (List.length tail))
        exact Nat.lt_trans h h'
```

Termination

IT WORKS!!!

```
#eval (Op.fromString "[<>.,+-]><")        ■ [<>.,+-]><
```

Only that it doesn't
🙁

```
#eval (Op.fromString "+[->-[>+>-<]<-<-]")        ■ +[->-[>+>-<]]<-<-
```

Resources

https://lean-lang.org/
https://adam.math.hhu.de/#/g/leanprover-community/nng4
https://leanprover-community.github.io/
https://github.com/leanprover-community/mathlib4
https://leanprover.zulipchat.com/