Lecture 13: Formal Verification
Software Construction (CS-214)

Viktor Kuncak, EPFL

Specifying and Checking Properties

In previous weeks we have seen how to specify properties using:

- assert, require, ensuring
- equations between functions (associativity, abstraction functions)

We have also seen techniques to:

- debug programs to localize sources of errors
- check assertions during execution (monitoring)
- test programs on many automatically generate inputs using ScalaCheck

The above techniques are very useful.
But they check program behavior only on a tiny fraction of possible executions.

mSort

```
sealed abstract class List[T]
case class Nil[T]() extends List[T]
case class Cons[T](h: T, t: List[T]) extends List[T]

  def mSort(list: List[Int]): List[Int] = {
    list match
      case Cons(h1, Cons(h2, rest)) =>
        val (s1, s2) = split(rest)
        merge(mSort(s1), mSort(s2))
      case _ => list
  }
```

Checking mSort using ScalaCheck - False Confidence!

```
val genNil: Gen[List[Int]] = const(Nil[Int]())
val genCons: Gen[List[Int]] =
  for
    h <- arbitrary[Int]
    tail <- genList
  yield Cons[Int](h, tail)

def genList: Gen[List[Int]] = oneOf(genNil, lzy(genCons))
lazy given arbList: Arbitrary[List[Int]] = Arbitrary(genList)
def tests(n: Int) = Test.Parameters.default.withMinSuccessfulTests(n)

def mSortOK = Prop.forAll { (l: List[Int]) =>
  val res = mSort(l)
  res != Cons(123456, Nil())
}.check(tests(5_000_000))
// + OK, passed 5000000 tests.
```

Checking mSort using Stainless Verifier

```
def mSortProperty(lst: List[Int]): Unit = {
}.ensuring(_ => mSort(lst) != Cons(123456, Nil()))
// [Warning ] Found counter-example:
// [Warning ] lst: List[Int] -> Cons[Int](123456, Nil[Int]())
```

Stainless examines the code and the property, uses symbolic reasoning (theorem proving, constraint solving)

- can prove property for *all inputs*
- can find bugs that are hard to find using testing/fuzzing

Outputs of ScalaCheck: 1) program is wrong, or 2) we don't know (checked 5M tests).
Outputs of verifier: 1) program is wrong, 2) we don't know, or 3) **program is correct**.

Installing Stainless

Stainless is written in Scala and has packaged versions for Linux, MacOS, and Windows (WSL recommended).

Run it by invoking

```
stainless fileName.scala ...
```

You can build Stainless from Scala sources:

- `https://github.com/epfl-lara/stainless/`

From the above link you can find release packages and documentation links:

- `https://epfl-lara.github.io/stainless/installation.html`

The Challenge of (Black Box) Testing

Suppose we want to **test** that addition of two **Long** integer values is commutative by trying all possible values.

assert$(x + y == y + x)$

(The values x and y are arbitrary, they may come, e.g., from input.)
Suppose we can run 10 tests every **nanosecond**. How long to test **all** cases?

Number of tests:	$2^{64} \cdot 2^{64} = 2^{128} > 10^{38}$
Seconds:	10^{28}
Days:	$1.15 \cdot 10^{23}$
Years:	$3.15 \cdot 10^{20}$

Ten billion times since “big bang”.

Moreover, for unbounded integers (BigInt), there are infinitely many values.

Rescue: Proofs of Programs

We have seen (Week 4) how to:

- prove using induction and a chain of equations that certain properties hold (properties of lists, tree set, balanced trees, monad laws)
- translate imperative programs to functional ones so we can use equations for them, too
- check properties of imperative programs with loops using Hoare logic rules

The above approaches gives rigorous mathematical proofs, but:

- we can make mistakes then applying these techniques on paper
- it can be tedious to apply the rules

In Formal Verification (of software), we use *theorem proving* and *program transformation* to construct formal proofs of program correctness.

- Beyond today's lecture, see CS-550: Formal Verification (MSc, Autumn)

Formal Verification

Goal: rigorously *prove* that computer systems “do what they should do”

“do what they should do” = satisfy a specification

How?

1. Define a mathematically rigorous notion of a system satisfying a specification (easiest for functional programs)
   - cover *all* behaviors and not just a small sample of them
2. Use combination of automated tools and human effort to construct the proof
   - eliminate errors in human proofs
   - automate some proof steps to make verification easier (it is still hard)

Revisiting Commutativity Example

assert(x + y == y + x)

A modern software verifier has **built-in** knowledge of commutativity.

- it's a mathematical **theorem** about integers modulo 2^{64}
- it can also prove theorems about unbounded integers
  (this would need infinitely many tests!)

A verifier uses **logical rules**, studied in **formal (mathematical) logic**, to take existing theorems, like $x + y = y + x$, and derive new ones, e.g., $x + (y + z) = (z + y) + x$.

Verifiers also use **automated theorem proving procedures**, which can do proof search and apply constraint solving algorithms to discover an unbounded number of facts using known theorems and rules.

Compiling to Formulas

How do we go from specified programs to mathematical theorems?

Compile programs and specifications to *formulas* (≈ pure expressions of type Boolean)

- this includes translating programs with, e.g., state to mathematical functions

We call these formulas **verification conditions**.

If verification condition is valid formula, then program satisfies specification.

Analogy:

language compiler (see CS-320)	verification-condition generator (see CS-550)
program $\mapsto$ machine code	(program $+$ specification) $\mapsto$ formula

$$\boxed{\text{program verifier}} = \boxed{\text{verification condition generator}} \bigoplus \boxed{\text{theorem prover}}$$

Tools for Formal Verification

Proof assistants: check mathematical theorems (including those about programs)

- Coq proof assistant: `https://coq.inria.fr/`
- Lisa proof framework: `https://github.com/epfl-lara/lisa`
- Isabelle proof assistant: `https://isabelle.in.tum.de/`
- Lean proof assistant: `https://lean-lang.org/`

Program verifiers focus on *programs* instead of arbitrary proofs:

- Stainless for Scala: `https://github.com/epfl-lara/stainless`
- Dafny: `https://dafny.org/`
- Why3: `https://www.why3.org/`
- $F*$: `https://www.fstar-lang.org/`

Many rely on theorem provers to automate proof (e.g. z3, cvc5). Not using LLMs!
Tools may still need specifications and proof hints. We explore this using Stainless.

Example: list with map and size

```
enum List[T]:
  case Nil()
  case Cons(head: T, tail: List[T])

  def map[U](f: T => U): List[U] =
    this match
      case Nil() => Nil()
      case Cons(head, tail) => Cons(f(head), tail.map(f))

  def size: BigInt = {
    this match
      case Nil() => BigInt(0)
      case Cons(_, tail) => BigInt(1) + tail.size
  }.ensuring(_ >= 0)
```

Example: zip

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {

  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))

    case _ => nil
}.ensuring (_.map(_._1) == xs)
```

```
warning: Found counter−example:
   xs: List[Int] −> Cons[Int](0, Nil[Int]())
   ys: List[Boolean] −> Nil[Boolean]()
```

zip with precondition (require)

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {
  require(xs.size <= ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))

    case _ => nil
}.ensuring (_.map(_._1) == xs)

// Verification succeeds.
```

Stainless performs inductive reasoning: when proving **ensuring** for zip(xs,ys), it assumes that **ensuring** holds for the result of recursive call, zip(xs0,ys0).

Observations about zip

Multiple **recursive** functions in *code* and *specifications* (zip, size, map)

- we can use one function (e.g., map) to specify another (e.g. zip)
- the definitions of List, size, map, zip are in Stainless library

The system was able to **prove** properties of such code (establish that they hold for *all* possible values)

The system was able to **find counterexamples** when code or specification were incorrect.

All functions were shown terminating (this is needed for sound reasoning).

require Restricts How We Can Call a Function

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {
  require(xs.size <= ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))

    case _ => nil
}.ensuring (_.map(_._1) == xs)

val exampleCall = zip(Cons(1, Nil()), Nil())

// [Warning ] size[Int](Cons[Int](1, Nil[Int]())) <= size[Boolean](Nil[Boolean]())
// [Warning ] zip.scala:32:19: => INVALID
```

We cannot call this version of zip when xs is longer, even though the function is well-defined (and symmetrical in xs vs ys).

A More Generous Specification of zip

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))
    case _ => nil
}.ensuring: res =>
  (!(xs.size <= ys.size) || res.map(_._1) == xs) &&
  (!(ys.size <= xs.size) || res.map(_._2) == ys)
```

Instead of **require**, we use implication.

- we write p ==> q as its equivalent !p || q
- we combined two specifications using conjunction (&&)

Now we can always call zip, and we can conclude different things depending on how the lengths of lists compare.

When We Want require: head and apply - Using Stainless

```
extension[T] (lst: List[T])
  def head: T = {
    require(lst != Nil())
    lst match // no warning for Nil case!
      case Cons(h,t) => h
  }

  def apply(n: BigInt): T = {
    require(0 <= n && n < lst.size)
    lst match // no warning for Nil case! Stainless concludes 0 < lst.size, so lst != Nil
      case Cons(h,t) =>
        if n == 0 then h else t.apply(n − 1)
  }

val testApplyOK = Cons(1, Cons(2, Cons(3, Nil()))).apply(2) // accepted
// val testApplyNo = Cons(1, Cons(2, Cons(3, Nil()))).apply(3) // rejected
```

Same Code Using Only Scala Compiler

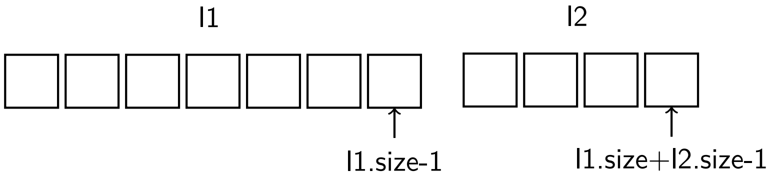
```
def apply(n: BigInt): T = {
  require(0 <= n && n < lst.size)
  lst match
    case Cons(h,t) =>
      if n == 0 then h else t.apply(n − 1)
}

val testApplyOK = Cons(1, Cons(2, Cons(3, Nil()))).apply(2) // accepted
val testApplyNo = Cons(1, Cons(2, Cons(3, Nil()))).apply(3) // accepted, but will crash!

// [warn] match may not be exhaustive. // but it is, in fact, exhaustive!
// [warn]
// [warn] It would fail on pattern case: List.Nil()
// [warn] lst match
```

Compiler warned for case where cannot be crash, but not when it crashes for sure

- Note, however, that Scala compiler is much faster than Stainless

Proof by Induction (Insight: Reduce both l1 and i)

```
def appendIndex[T](l1: List[T], l2: List[T], i: BigInt): Unit = {
  require(0 <= i && i < (l1 ++ l2).size) // assumptions for theorem to be well−defined
    l1 match // proof
      case Cons(x,xs) if (i > 0) => appendIndex[T](xs, l2, i − 1) // use I.H. (recursive call)
      case _ => () // base case and i=0 − follow from axioms and function unfolding
}.ensuring: _ =>
   (l1 ++ l2)(i) == (if i < l1.size then l1(i) else l2(i − l1.size)) // theorem
```



To Verify One Function, We May Need to Verify Another

The previous proof fails unless we also prove the following **ensuring** of ++:

```
extension[T] (xs: List[T])
  def ++(ys: List[T]): List[T] = {
    xs match
      case Nil() => ys
      case Cons(h, t) => Cons(h, t ++ ys)
  }.ensuring: res =>
      res.size == xs.size + ys.size
```

Indeed, in the theorem statement:

```
  (l1 ++ l2)(i) == (if i < l1.size then l1(i) else l2(i - l1.size)) // theorem
```

to know that l2(i − l1.size)) is safe to call, we need to know: i − l1.size < l2.size but what we know is i < (l1 ++ l2).size
Also, it was important to specify that .size returns non-negative integer.

Proving lst.map(single).flatten == lst: say "use induction"

```
extension[T] (ll: List[List[T]])
  def flatten: List[T] =
    ll match
      case Cons(h, t) => h ++ t.flatten
      case Nil() => Nil[T]()

def single[T](x: T) = Cons(x, Nil[T]())

import stainless.annotation.*
def examQuestion[T](@induct lst: List[T]): Unit = {
}.ensuring( _ => {
    val ll: List[List[T]] = lst.map(single)
    ll.flatten == lst })
```

@induct: asks Stainless to do proof by induction (generate **match** and recursive calls).

Equivalence Checking

Do functions isSortedR and isSortedB return same result for all lists?

```
def isSortedR(l: List[Int]): Boolean = {
  def loop(p: Int, l: List[Int]): Boolean = l match {
    case Nil() => true
    case Cons(x, xs) if (p <= x) => loop(x, xs)
    case _ => false
  }
  if (l.isEmpty) true
  else loop(l.head, l.tail)
}

def isSortedB(l: List[Int]): Boolean = {
  if l.isEmpty then true
  else if !l.tail.isEmpty && l.head > l.tail.head then false
  else isSortedB(l.tail)
}
```

Equivalence Checking: Use ensuring to Say that Results Are Equal

Do functions isSortedR and isSortedB return same result for all lists?

```
def isSortedR(l: List[Int]): Boolean = {
  def loop(p: Int, l: List[Int]): Boolean = l match {
    case Nil() => true
    case Cons(x, xs) if (p <= x) => loop(x, xs)
    case _ => false
  }
  if (l.isEmpty) true
  else loop(l.head, l.tail)
}

def isSortedB(l: List[Int]): Boolean = {
  if l.isEmpty then true
  else if !l.tail.isEmpty && l.head > l.tail.head then false
  else isSortedB(l.tail)
}
//.ensuring(_ == isSortedR(l)) // Verifies, so functions always give same result
```

Equivalence Checking Mode of Stainless

Instead of writing **ensuring** clause in one function or another, we invoke

```
stainless equiv-sorted.scala --equivchk=true --timeout=3 \
  --comparefuns=isSortedB --models=isSortedR

# ...
# List of functions that are equivalent to model EquivSorted.isSortedR: EquivSorted.isSortedB
# ...
```

Larger example: remove duplicates in a list (unique): demo.

Example: Sum Elements of Array

```
def sum(a: Array[Int], from: Int, to: Int): Int = {
  if from >= to then 0
  else a(from) + sum(a, from + 1, to)
}
```

Stainless cannot see that the loop terminates

Demo to verify:

- specify what decreases
- specify bounds for variables
- turn off strict-arithmetic

Example: Sum Elements of Array

```
import stainless.annotation.*
import stainless.lang.*

def sum(a: Array[Int], from: Int, to: Int): Int = {
  require(0 <= from && from <= to && to <= a.length)
  decreases(to - from)
  if from >= to then 0
  else a(from) + sum(a, from + 1, to)
}
```

Soundness and Termination

We can prove any postcondition for a non-terminating function f:

```
def f(x: BigInt): BigInt = {
  f(x)
}.ensuring(_ =>
  1 == 2)
```

- a key reasoning principle in Stainless is *function induction*
- this principle gives meaningful properties (for total functions) only when the functions terminates
- it is especially important if we use specifications that are not executed (so we do not even notice non-termination)

Termination is also a good property to have in its own: if the function does not terminate, it often indicates an error

- checking function termination is a form of specification for free, programmers do not need to write properties to say what they want

Caution: Accidental Non-termination via Postcondition

```
def f(x: BigInt): BigInt = {
  x + 1
}.ensuring(_ => f(x) == 42 && false)

f(42)
```

There is infinite recursion in this program, because evaluating ensuring leads to evaluating its postcondition that invokes the function with the same argument.

This program would be accepted without termination checks: the ensuring of function f is unfolding inside the ensuring.

Instead, we should use 'res => ...' to refer to the result of the function.

decrease: the way to specify termination

decreases(e) when e is integer expression

- precondition must imply $0 \le e$
- in each recursive call the value must be less than e

```
import stainless.annotation.*
import stainless.lang.*

def sum(a: Array[Int], from: Int, to: Int): Int = {
  require(0 <= from && from <= to && to <= a.length)
  decreases(to − from)
  if from >= to then 0
  else a(from) + sum(a, from + 1, to)
}
```

Note that:

- 0 <= to − from thanks to the precondition
- to − (from + 1) − to < to − from, so measure decreases

Lexicographic measures

In general, the argument of is an n-tuple called a *measure*
Example of pairs of numbers:
decreases(p,q)
The numbers need to be positive and the ordering is defined as follows: $(p,q) > (p',q')$ iff:

$$p > p' \vee (p = p' \wedge q > q')$$

Example: $(100,2) > (100,1) > (99,123456) > (99,123455) > \dots$

A more general requirement would be that the argument of decreases can be ordered by some relation $>$ such that there are no infinite chains $t_1 > t_2 > ...$

- well-founded relation in mathematics (e.g. set theory)
- Stainless does not support directly measures beyond lexicographic ones

decreases on structures

When we say that a structure decreases, it means its *size* decreases

- Stainless synthesizes internal size functions used only for termination

Examples:

- Length of a list
- Number of nodes in a tree

We can define our own functions and use them as measures, but they need to be proven to terminate themselves.

Measure Checking and Inference

By default, Stainless checks that the measure decreases in recursive calls and that they are non-negative.

- to turn off this check, use −−check−measures=false

Stainless can find some decrease clauses automatically when they are not specified, but often it needs help.

- in the presence of mutual recursion, this requires looking at multiple functions at once, including higher-order functions

If Stainless it does not find measure and you do not know how to find it, you can turn termination inference and termination checking using −−infer−measures=false −−check−measures=false

Catching Non-Terminating Code (Rarely Works)

```
def map[U](f: T => U): List[U] =
  this match
    case Nil() => Nil()
    case Cons(head, tail) => Cons(f(head), this.map(f))
```

Catching Non-Terminating Code (Rarely Works)

```
def map[U](f: T => U): List[U] =
  this match
    case Nil() => Nil()
    case Cons(head, tail) => Cons(f(head), this.map(f))
```

```
zip.scala:5:7: warning: Function map loops given inputs:
thiss: List[T] -> Cons[T](T#2, Nil[T]())
     f: (T) => U -> (x158: T) => U#0

            def map[U](f: T => U): List[U] =
                ^
```

Callback: Specifying Balanced Trees

```
    def ++(ys: Conc[T]): Conc[T] = {
      require(xs.isBalanced && ys.isBalanced)
      decreases(abs(xs.height − ys.height))

...

    }.ensuring(res =>
        appendAssocInst(xs, ys) && // lemma instantiation for rebalancing
        res.isBalanced &&
        res.height <= max(xs.height, ys.height) + 1 &&
        res.height >= max(xs.height, ys.height) &&
        res.toList == xs.toList ++ ys.toList)
```

Our main goal was to prove res.toList and res.isBalanced properties, but we needed to *strengthen* the specification to make proof work.

import stainless.lang.StaticChecks.*

Doing this import in Stainless makes sure that compiled code does not do runtime checks for **require**, **ensuring** and **assert**.

def require(pred: **=>** Boolean): Unit = ()

def assert(pred: **=>** Boolean): Unit = ()

extension[A](x: A) **def ensuring**(cond: A **=>** Boolean): A = x

Without such import, running a tree would make all operations, instead of $O(\log(n))$, worse than linear time, due to .toList and ++

Demo: Constant Folding Expression Simplifier

```
def constfold1(e: Expr)(using anyCtx: Env) = {
  e match
    case Add(Number(n1), Number(n2)) => Number(n1 + n2)
    case Minus(Number(n1), Number(n2)) => Number(n1 − n2)
    case e => e
}.ensuring(evaluate(_) == evaluate(e))

val constFold1Simp = new SoundSimplifier:
  override def apply(e: Expr, anyCtx: Env) = constfold1(e)(using anyCtx)

def mapExpr(e: Expr, f: SoundSimplifier)(using anyCtx: Env): Expr = {
  val mapped: Expr = e match
    case Number(_) => e
    case Var(_) => e
    case Add(e1, e2) => Add(mapExpr(e1, f), mapExpr(e2, f))
    case Minus(e1, e2) => Minus(mapExpr(e1, f), mapExpr(e2, f))
  f(mapped, anyCtx)
}.ensuring(evaluate(_) == evaluate(e))
```

Simple Imperative Code: Search in an Array

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int = {
  var i = from
  while i < to && a(i) != x do
    i = i + 1
  if i < to then i
  else −1
}
```

Let's verify it!

- bounds on variables
- termination using decreases
- does it do what it should do?

find Does Not Crash

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int = {
  require(0 <= from && from <= to && to <= a.size)
  var i = from
  (while i < to && a(i) != x do
    decreases(to − i)
    i = i + 1
  ).invariant(from <= i && i <= to)
  if i < to then i
  else −1
}
```

find Either Gives Correct Index or -1. Amazing Specification Proven!

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int = {
  require(0 <= from && from <= to && to <= a.size)
  var i = from
  (while i < to && a(i) != x do
    decreases(to − i)
    i = i + 1
  ).invariant(from <= i && i <= to)
  if i < to then i
  else −1
}.ensuring(res =>
  (from <= res && res < to && a(res) == x) ||
  res == −1)
```

-1 is Also an Amazing (Constant) Function

```
def find(a: Array[Int], from: Int, to: Int, x: Int): Int = {
  require(0 <= from && from <= to && to <= a.size)






  −1 // always return −1
}.ensuring(res =>
  (from <= res && res < to && a(res) == x) ||
  res == −1)
```

Full Functional Specification: Characterizes Output

```
def existsIn(a: Array[Int], from: Int, to: Int, x: Int): Boolean =
  require(0 <= from && from <= to && to <= a.size)
  decreases(to − from)
  !(from == to) &&
  ((a(from) == x) || existsIn(a, from, to − 1, x))

def find(a: Array[Int], from: Int, to: Int, x: Int): Int = {
  require(0 <= from && from <= to && to <= a.size)
  var i = from
  (while i < to && a(i) != x do
    decreases(to − i)
    i = i + 1
  ).invariant(from <= i && i <= to && !existsIn(a, from, i, x))
  if i < to then i
  else −1
}.ensuring(res =>
  (from <= res && res < to && a(res) == x) ||
  (res == −1 && !existsIn(a, from, to, x)))
```

Some Limitations of Stainless

Does not support Scala standard library; Stainless library small, invariant lists

- co-variant data structures can work, but verification is slower

No sharing of mutable structures (more restrictive than Rust).

Function values cannot refer to mutable state

Difficulties with nested arrays: restrictions on aliasing, solver performance

Equality of functions is not extensional, and also not the one in Scala

Measure inference does not work on instantiated generic types, e.g. List[List[Int]]

Not all aspects of Scala's type system supported

- e.g. type members, intersections, unions do not work

Automatic transformation to functional code can lead to confusing error messages

Case Studies: Programs Verified Using Stainless

Verification of code used by European Space Agency (with Ateleris GmbH):

- ASN.1/ACN Decoders and Encoders declaratively specified and generated as specified Scala code, proven correct by Stainless (VMCAI 2025)
- From Verified Scala to STIX File System Embedded Code using Stainless (NFM 2022), generated efficient C code (with preallocated data)

Verification of complex data structures:

- Hash tables (LongMap) shown behaviorally equivalent to lists (IJCAR 2024)
- Quite Okay Image Format `https://qoiformat.org/` (FMCAD 2022)
  decode(encode(img))=img
- Balanced trees, ConcTrees (Purely Functional Data Structure lecture, week 7)
- structures such as Fibonacci heaps

Tendermint blockchain client; algorithms used by Digital Asset
Soundness of System F (typed λ calculus, first-class polymorphism)
More examples: `https://github.com/epfl-lara/bolts/`