# Formal verification of Scala programs with Stainless

Romain Ruetschi

LARA, EPFL

Typelevel Summit Lausanne 2019

# About me

- Romain Ruetschi (Romac)
- MSc in Computer Science from EPFL
- ~2 years as an engineer at LARA

# Outline

- Stainless: Verification framework for Scala
- What Stainless verifies
- Termination checker
- Case study: Verifying typeclasses
- More case studies
- Bonus
- Coming soon / further work

# Stainless: Verification framework for Scala

Stainless is a verification framework for higher-order programs written in a subset of Scala, named *PureScala*:

- Traits, abstract classes, case classes, implicit classes, methods
- Higher-order functions, lambdas
- Any, Nothing, co-/contra-variant type parameters
- Single inheritance
- Anonymous and local classes, inner functions

- Type members, type aliases
- GADTs
- `PartialFunctions`
- Set, Bag, List, Map, Array, Byte, Short, Int, Long, BigInt
- Local state, `while`, traits/classes with `vars`, and more...

Currently supports Scala 2.12.x, 2.13 coming up!

Some Dotty-specific features:

- Intersection and union types
- Dependent function types
- Extension methods
- Opaque types

Currently only supports Dotty 0.12.0, will try to catch up.

# What Stainless verifies

- **Assertions** which should hold at the place where they are stated, but are checked statically
- **Postconditions** using `ensuring` function: assertions for return values of functions
- **Preconditions** using `require` function: assertions on function parameters
- **Loop invariants**: inductive assertions that hold in each loop iteration after the while condition check passes
- **ADT/Class invariants**: assertions on constructors parameters (which remain true for all constructed values)

Stainless also automatically performs **automatic checks for the absence of runtime failures**:

- Exhaustiveness of pattern matching (taking guards into account)
- Division by zero, array bounds checks
- Map domain checks

Moreover, Stainless also checks *PureScala* programs from:

- Creating null values or unininitalized local variables or fields
- Cxplicitly throwing an exception
- Cverflows and underflows on sized integer types

# Termination checker

A *verified* function in stainless is guaranteed to never crash, however, it can still lead to an infinite evaluation.

Curry-Howard correspondance tells us that non-terminating functions allows us to prove any proposition.

Stainless therefore provides a termination checker that complements the verification of safety properties.

# Pipeline

TODO: Image

- Scala/Dotty compiler
- Extraction
- Lowering
- Inox
- SMT solver

# Tutorial: Insertion sort

```scala
def isSorted(l: List[BigInt]) : Boolean = l match {
  case Nil              => true
  case _ :: Nil         => true
  case x1 :: x2 :: rest =>
    x1 < x2 && isSorted(x2 :: rest)
}
```

```scala
def sInsert(x: BigInt, l: List[BigInt]) : List[BigInt] = {
  l match {
    case Nil => x :: Nil
    case e :: rest if (x == e) => l
    case e :: rest if (x < e)  => x :: e ::rest
    case e :: rest if (x > e)  => e :: sInsert(x, rest)
  }
}
```

```scala
def sInsert(x: BigInt, l: List[BigInt]) : List[BigInt] = {
  require(isSorted(l))
  // same as before
} ensuring { res =>
  isSorted(res) &&
  res.size == l.size + 1 &&
  res.content == l.content ++ Set(x)
}
```

```scala
def sort(l: List[BigInt]): List[BigInt] = l match {
  case Nil     => Nil
  case x :: xs => sInsert(x, sort(xs))
} ensuring { res =>
  isSorted(res) &&
  res.size == l.size &&
  res.content == l.content
}
```

```
 stainless summary

sInsert   postcondition   valid   nativez3   0.081
sort      postcondition   valid   nativez3   0.931
sort      precondition    valid   nativez3   0.429

total: 3  valid: 3  invalid: 0  unknown: 0  time: 1.441
```

# Comparison

- **Stainless:** 27 LOC
- **Coq:** 140 LOC

# Case study: Verifying typeclasses

```scala
Seq(1, 2, 3, 4).par.fold(10)(_ - _)

// ((((10 - 1) - 2) - 3) - 4) => 0
// (10 - 1) - (2 - (3 - 4))   => 6
```

```scala
Seq(1, 2, 3, 4).par.fold(0)(_ + _)

// ((((10 + 1) + 2) + 3) + 4) => 10
// (10 + 1) + (2 + (3 + 4))   => 10
```

```scala
abstract class Semigroup[A] {
  def combine(x: A, y: A): A

  @law def law_assoc(x: A, y: A, z: A) =
    combine(x, combine(y, z)) == combine(combine(x, y), z)
}
```

```scala
abstract class Monoid[A]
  extends Semigroup[A] {

  def empty: A

  @law def law_leftIdentity(x: A) =
    combine(empty, x) == x

  @law def law_rightIdentity(x: A) =
    combine(x, empty) == x
}
```

```scala
case class Sum(get: BigInt)

implicit def sumMonoid = new Monoid[Sum] {
  def empty = 0
  def combine(x: Sum, y: Sum) = Sum(x.get + y.get)
}
```

```
 stainless summary

law_leftIdentity    law   valid   nativez3   0.223
law_rightIdentity   law   valid   nativez3   0.407
law_assoc           law   valid   nativez3   0.944

total: 3  valid: 3  invalid: 0  unknown: 0  time: 1.574
```

```scala
implicit def optionMonoid[A](implicit val S: Semigroup[A]) =
  new Monoid[Option[A]] {
    def empty: Option[A] = None()

    def combine(x: Option[A], y: Option[A]) =
      x match {
        case None()   => y
        case Some(xv) => y match {
          case None()   => x
          case Some(yv) => Some(S.combine(xv, yv))
        }
      }
  }
```

```scala
implicit def optionMonoid[A](implicit val S: Semigroup[A]) =
  new Monoid[Option[A]] {
    // ...

    override def law_assoc(@induct x: Option[A], y: Option[A],
      super.law_assoc(x, y, z)
  }
```

```scala
def foldMap[M, A](xs: List[A])(f: A => M)(implicit M: Monoid[A
  xs.map(f).fold(M.empty)(M.append)

@extern
def parFoldMap[M, A](xs: List[A])(f: A => M)(implicit M: Monoi
  xs.toScala.par.map(f).fold(M.empty)(M.append)
} ensuring { res =>
  res == foldMap(xs, f)
}
```

# More case studies

# Conc-Rope

Verified data-structure which provides

- Worst-case $O(logn)$ time lookup, update, split and concatenation operations
- Amortized O(1) time append and prepend operations

Very useful for efficient data-parellel operations!

[ConcRope] TODO: Ref

# Parellel Map-Reduce pipeline

Fully verified implementation of the previous running example, using a Conc-Rope under the hood instead of Scala's ‘par’ operator.

Built by Lucien Iseli, BSc student, as a semester project.

# Actor systems

```
case class Primary(backup: ActorRef, counter: Counter) extends
  require(backup.name == "backup")

  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavi
    msg match {
      case Inc =>
        backup ! Inc
        PrimBehav(backup, counter.increment)

      case _ => this
    }
}
```

```scala
case class Backup(counter: Counter) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavi
    case Inc => BackBehav(counter.increment)
    case _ => this
  }
}
```

```scala
def invariant(s: ActorSystem): Boolean =
  (s.behaviors(PrimaryRef), s.behaviors(BackupRef)) match {
    case (Primary(bRef, p), Backup(b)) if bRef == BackupRef =>
      val pending = s.inboxes(PrimaryRef -> BackupRef).length
      p.value == b.value + pending
    case _ => false
  }
```

```scala
def preserveInv(s: ActorSystem, n: ActorRef, m: ActorRef) = {
  require(invariant(s))
  val next = s.step(n, m)
  invariant(next)
}.holds
```

# Smart contracts

We also maintain a fork of Stainless, called *Smart* which supports:

- Writing smart contracts in Scala
- Specifying and proving properties of such programs, including precise reasoning about the `Uint256` data type
- Generating Solidity source code from Scala, which can then be compiled and deployed using the usual tools for the Ethereum software ecosystem

For example, we have modeled and verified a voting smart contract developed by SwissBorg.

[0] https://github.com/epfl-lara/smart

# Bonus: Refinement types

```scala
type Nat = { n: BigInt => n >= BigInt(0) }
```

```scala
def sortedInsert(
  xs: { List[Int] => xs.nonEmpty },
  x:  { Int => x <= xs.head }
): { res: List[Int] => isSorted(res) } = {
  x :: xs // VALID
}
```

# Bonus: Dependent function types

```
trait Entry {
  type Key
  val key: Key
}

def extractKey(e: Entry): e.Key = e.key

def extractor: (e: Entry) => e.Key = extractKey(_)
```

```scala
case class IntEntry() extends Entry {
  type Key = Int
  val key: Int = 42
}

assert(extractor(entry) == 42) // VALID
```

# Other features

- sbt plugin + metals integration
- Ghost context
- Partial evaluation

# Coming soon(ish)

- VC generator via bidirectional typechecker for *System FR* (TODO: ref)
- Indexed recursive types
- Higher-kinded types
- Better support for GADTs
- WebAssembly backend
- Better metals/IDE integration

# Further work

- Port synthesis and resource analysis frameworks over from Leon predecessor
- Reasoning about I/O and concurrency (via ZIO?)
- Support for exceptions
- Scala 2.13 / latest Dotty / TASTY support
- Standalone front-end for a custom input language
- Eta / Frege front-end
- GraalVM/Truffle back-end

# Learn more

- Installation
- Tutorial
- Ghost context
- Imperative features
- Working with existing code
- Proving theorems
- Stainless library
- and more...

=> stainless.epfl.ch

# Acknowledgments

Stainless is the latest iteration of our verification system for Scala, which was built and improved over time by many EPFL PhD students: Nicolas Voirol, Régis Blanc, Eva Darulova, Etienne Kneuss, Ravichandhran Kandhadai Madhavan, Mikaël Mayer, Emmanouil Koukoutos, Ruzica Piskac, Philippe Suter, as well as Marco Antognini, Ivan Kuraj, Lars Hupel, Samuel Grütter, and myself.

Many thanks as well to our friends at TripleQuote for their help with the compiler and sbt plugins.

# References I

TODO

# References I