Formal verification of Scala programs with Stainless

Romain Ruetschi, EPFL LARA

June 14th, 2019

Who am I?

What is Stainless?

What is Stainless?

Stainless is a verification framework for higher-order programs written in a (now fairly substantive) subset of Scala.

We currently support the following features:

- TODO

What Stainless verifies

What Stainless verifies

Stainless supports verifying:

- **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 immutable parameters of constructors (which remain true for all constructed values)

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

- completeness of pattern matching
- division by zero, array bounds checks
- map domain checks

Moreover, Stainless prevents *PureScala* programs from:

- creating null values or unininitalized local variables or fields
- explicitly throwing an exception
- overflows and underflows on sized integer types

Verifying typeclasses

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_identity(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)
}
```

```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))
        }
      }
  }
```

```
def lemma_optionCombineAssoc(x: Option[A], y: Option[A], z:
  // TODO
}
```

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

    override def law_assoc(x: Option[A], y: Option[A], z: O
      super.law_assoc(x, y, z) because lemma_optionCombineA
  }
```

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

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

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

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

Termination checking

Termination checking

A *verified* function in stainless is guaranteed to never crash, however, it can still lead to an infinite evaluation. Stainless therefore provides a termination checker that complements the verification of safety properties.

Under the hood

Case studies

ConcRope

Parellel Map-Reduce pipeline

Actor systems

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

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

    case _ => this
  }
}

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

Smart contracts verification

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

- Writing smart contracts in Scala
- Specifiybg 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 and dependent function types

Refinement types

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

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
```

Coming up

Coming up

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

Further work

Further work

- Reasoning about I/O and concurrency (via ZIO?)
- Support for exceptions
- Scala 2.13 / latest Dotty / TASTY support
- Standalone front-end for a verification friendly input language
- Eta / Frege front-end
- GraalVM/Truffle back-end

Getting started

Acknowledgments

References I