Formal verification of Scala programs with Stainless

Romain Ruetschi

Laboratory for Automated Reasoning and Analysis, EPFL

Scala Romandie Meetup

October 2019

About me

Romain Ruetschi

@_romac

MSc in Computer Science, EPFL

Engineer at LARA, EPFL

Stainless

Stainless is a *formal verification* tool for Scala* programs

Formal Verification

Goal: *Prove* that a *program* satisfies a *specification*

Specification

- “The size of list is a positive integer”
- “List concatenation is associative”
- “This Monoid instance respects the Monoid laws”
- “The program does not divide by zero”
- “This actor system correctly performs leader election via the *Chang and Roberts algorithm*”

Verification with Stainless

Assertions: checked statically where they are defined

Postconditions: assertions for return values of functions

Preconditions: assertions on function parameters

Class invariants: assertions on constructors parameters

+ Loop invariants

```scala
def f(x: A): B = {
  require(prec)
  body
} ensuring (res ⇒ post)
```

$$\forall x \,.\, prec[x] \implies post[x](body[x])$$

```scala
def size: BigInt = this match {
  case Nil      => 0
  case x :: xs => 1 + xs.size
} ensuring (res => res >= 0)
```

```scala
def isSorted(l: List[BigInt]): Boolean = l match {
  case x :: (y :: ys) => x <= y && isSorted(y :: ys)
  case _              => true
}

def insert(e: BigInt, l: List[BigInt]): List[BigInt] = {
  require(isSorted(l))

  l match {
    case Nil                => e :: Nil
    case x :: xs if e <= x => e :: l
    case x :: xs           => x :: insert(e, xs)
  }
} ensuring (res => isSorted(res))

def sort(l: List[BigInt]): List[BigInt] = (l match {
  case Nil     => Nil
  case x :: xs => insert(x, sort(xs))
}) ensuring (res => isSorted(res))
```

Static checks

Stainless also automatically performs automatic checks for the absence of runtime failures, such as:

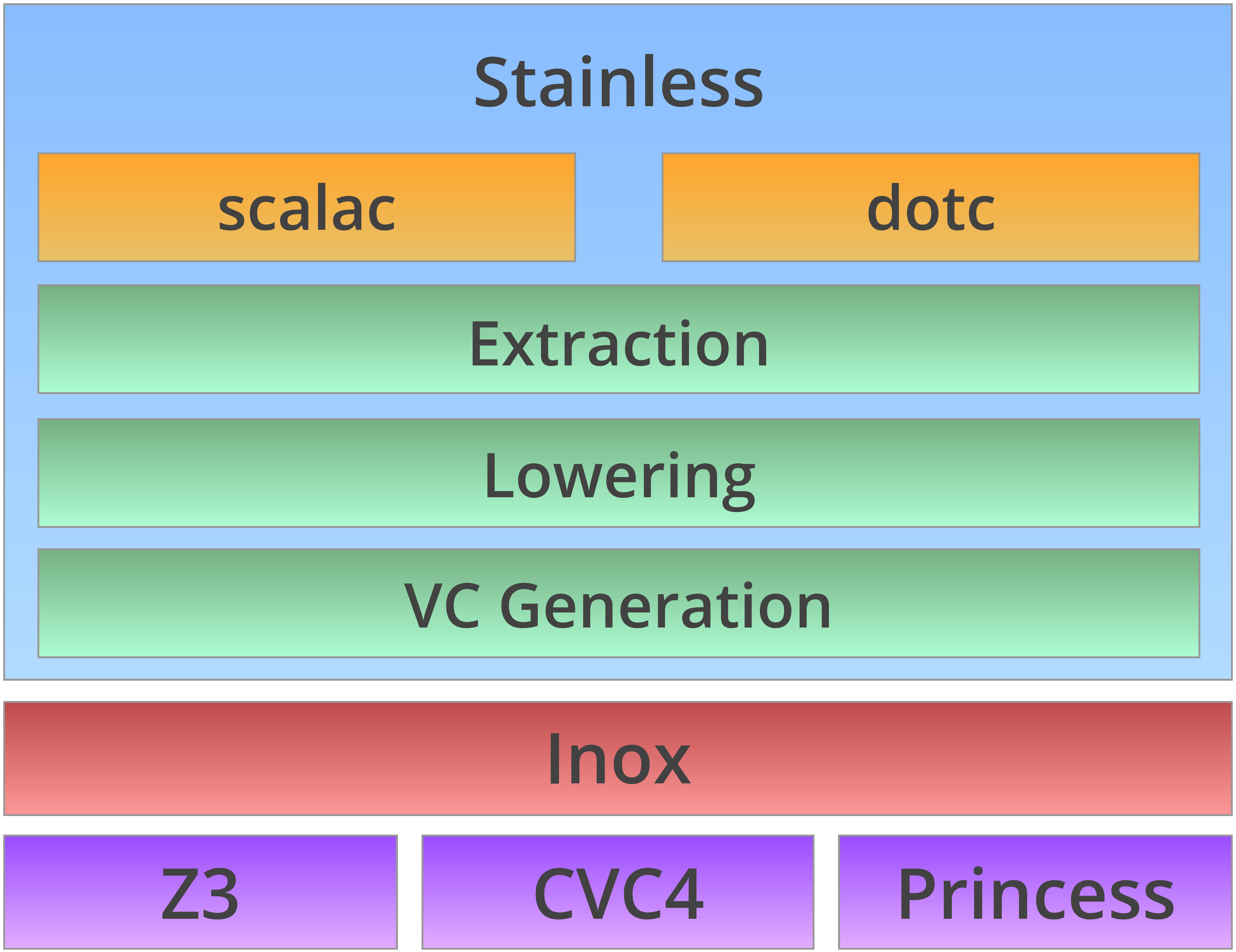
- **Exhaustiveness of pattern matching (w/ guards)**
- **Division by zero, array bounds checks**
- **Map domain checks**

Static checks (2)

Moreover, Stainless also prevents PureScala programs from:

- **Creating null values**
- **Creating uninitialised local variables or fields**
- **Explicitly throwing an exception**
- **Overflows and underflows on sized integer types**

Pipeline


Stainless
scalac
dotc
Extraction
Lowering
VC Generation
Inox
Z3
CVC4
Princess

SMT Solvers

- SMT stands for *Satisfiability Modulo Theories*
- Think: SAT solver on steroids!
- Can reason not only about boolean algebra, but also integer arithmetic, real numbers, lists, arrays, sets, bitvectors, etc.

SMT Solvers

Very good at answering questions like:

*Is there an assignment for **x, y, z** such that **formula** is true?*

$$\exists x,y,z.\ x > 1\ \&\&\ (\neg f(x\ y) \Longrightarrow size(z) == 0)$$

- If yes, will say **SAT**, and output the assignment (*model*)
- If no, will say **UNSAT**

SMT Solvers

We can use this to ask questions like:

Is ***formula*** true for all values of ***x, y, z***?

SMT Solvers

We can use this to ask questions like:

Is ***formula*** true for all values of ***x, y, z****?*

We ask this equivalent question instead:

Is there an assignment for ***x, y, z*** such that **¬** ***formula*** is true?

SMT Solvers

Is there an assignment for ***x, y, z*** such that **¬** ***formula*** is true?

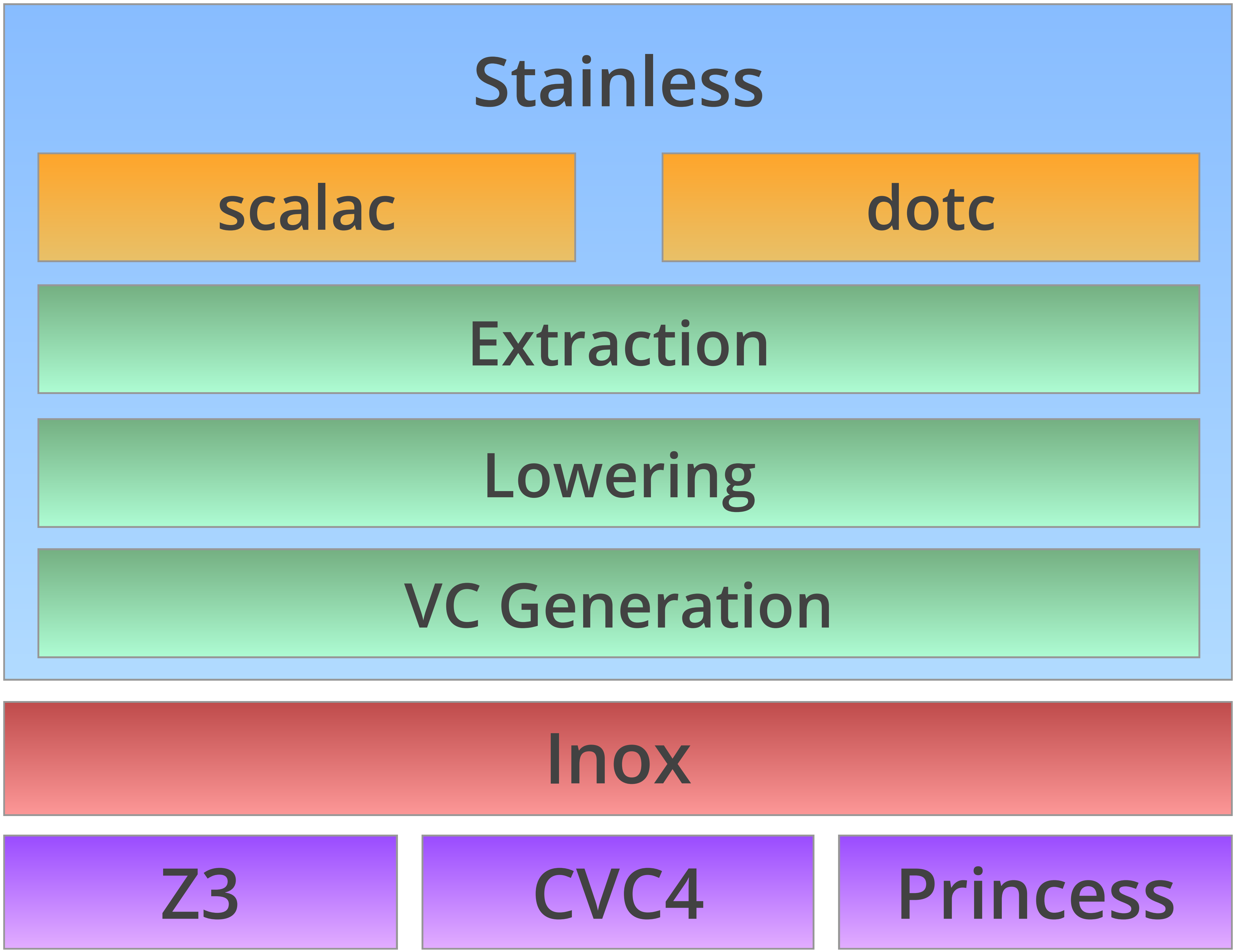
- If solver says **UNSAT**, it means that ***formula*** is true for all ***x, y, z.***
- If solver says **SAT**, then the model it ouputs is a *counter-example* to our formula, ie. specific values of ***x, y, z*** such that ***formula*** is false.

SMT Solvers

Problem:

SMT solvers do not support recursive functions, lambdas, polymorphism, etc.

Pipeline


Stainless
scalac
dotc
Extraction
Lowering
VC Generation
Inox
Z3
CVC4
Princess

Inox

Provides first-class support for:

- Polymorphism
- Recursive functions
- Lambdas
- ADTs, integers, bit vectors, strings, sets, multisets, map
- Quantifiers
- ADT invariants
- Dependent and refinement types

Inox

Uniform Decision Procedure

```
(φ, B) = unroll(φ, _)
while(true) {
  checkSat(φ ∧ B) match {
    case "SAT" ⇒ return "SAT"
    case "UNSAT" ⇒ solve(φ) match {
      case "UNSAT" ⇒ return "UNSAT"
      case "SAT" ⇒ (φ, B) = unroll(φ, B)
    }
  }
}
```

underapproximation

Over-approximation
(leaf calls uninterpreted)

fair unrolling of
- body
- postcondition

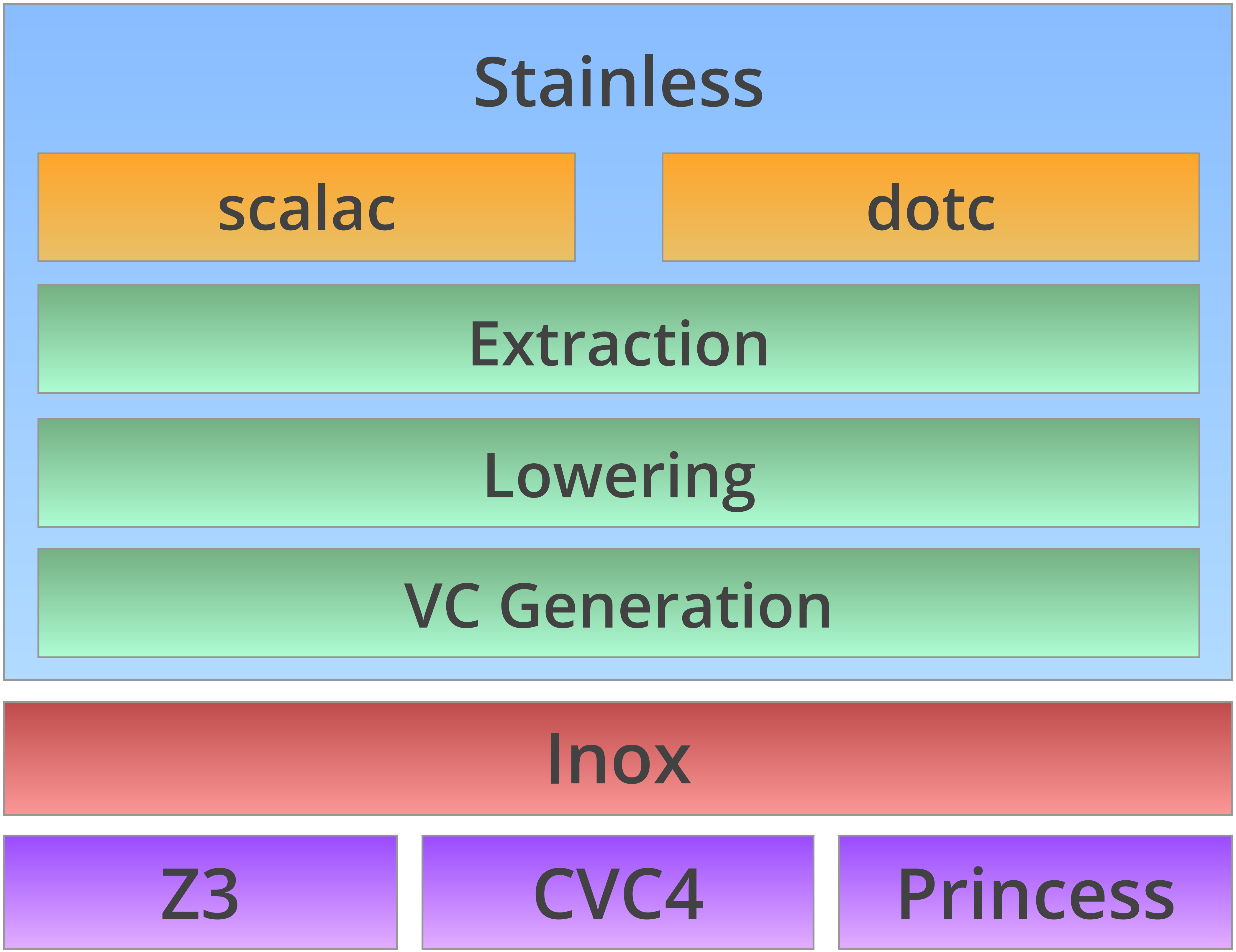
Procedure need know nothing about the values that show up in the sufficient surjectivity criterion.
Complexity in most cases: NP (suf. surjectivity gives depth)

Inox

Problem:

Inox does not support classes, subtyping, pattern matching, variance, methods, loops, mutation, etc.

Pipeline


Stainless
scalac
dotc
Extraction
Lowering
VC Generation
Inox
Z3
CVC4
Princess

Stainless

That's where Stainless comes in:

- Parse and type-check the Scala program using scalac
- Check that the extracted program fits into our fragment, PureScala
- Lower the extracted program down to Inox's input language
- Generate verification conditions to be checked by Inox
- Report the results to the user

PureScala

- Set, Bag, List, Map, Array, Byte, Short, Int, Long, BigInt, ...
- Traits, abstract/case/implicit classes, methods
- Higher-order functions
- Any, Nothing, variance, subtyping
- Anonymous classes, local classes, inner functions
- Partial support for GADTs
- Type members, type aliases
- Limited mutation, while, traits/classes with vars, partial functions, ...

Lowering phases

`InnerClasses`, **`Laws`**, `SuperCalls`, `Sealing`, **`MethodLifting`**, `FieldAccessors`, `ValueClasses`, `AntiAliasing`, `ImperativeCodeElimination`, `ImperativeCleanup`, `AdtSpecialization`, `RefinementLifting`, **`TypeEncoding`**, `FunctionClosure`, `FunctionInlining`, **`InductElimination`**, `SizeInjection`, `PartialEvaluation`

Verification Condition Generation

```scala
def f(x: A): B = {
  require(prec)
  body
} ensuring (res => post)
```

```scala
prec ==> { val res = body; post }
```

Verification Condition Generation

```
def insert(e: BigInt, l: List[BigInt]) = {
  require(isSorted(l))

  l match { /* ... */ }

} ensuring (res => isSorted(res))
```

```
isSorted(l) ⟹ {
  val res = l match { /* … */ }
  isSorted(res)
}
```

Verification of type classes

Type classes

```scala
import stainless.annotation.law

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)
}
```

Algebraic structure as an abstract class

Operations as abstract methods

Laws as concrete methods annotated with @law

Type classes

```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
}
```

Stronger structures expressed via subclassing

Can define additional operations and laws

Instances

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

implicit def sumMonoid = new Monoid[Sum] {

  def empty = Sum(0)

  def combine(x: Sum, y: Sum) = Sum(x.get + y.get)
}
```

Type class instance as an object

Only needs to provide concrete implementation for the operations

Stainless automatically verifies that the laws hold

Result

```
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 S: Semigroup[A]) =
  new Monoid[Option[A]] {

    def empty: Option[A] = None()

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

    // ...
```

Here we need to provide Stainless with a hint:

```scala
  // ...
  override def law_assoc(x: Option[A], y: Option[A], z: Option[A]) =
    super.law_assoc(x, y, z) because {
      (x, y, z) match {
        case (Some(xv), Some(yv), Some(zv)) =>
          S.law_assoc(xv, yv, zv)

        case _ => true
      }
    }
}
```

When combining three `Some[A]`, use the fact that the `combine` operation on A is associative, which we know because of the `Semigroup[A]` instance in scope.

Working with existing code

```scala
case class TrieMapWrapper[K, V](
  @extern theMap: TrieMap[K, V]
) {
  @extern @pure
  def contains(k: K): Boolean = {
    theMap contains k
  }

  @extern
  def insert(k: K, v: V): Unit = {
    theMap.update(k, v)
  } ensuring {
    this.contains(k) && this.apply(k) == v
  }

  @extern @pure
  def apply(k: K): V = {
    require(contains(k))
    theMap(k)
  }
}
```

Actor systems

Counter

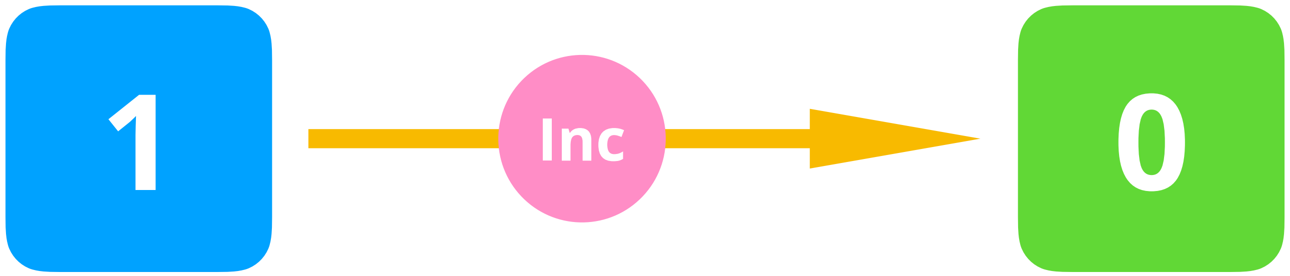
0

0


Inc
0
0

1

0


1
Inc
0

1

1

```scala
case class Primary(backup: ActorRef, counter: BigInt)
  extends Behavior {

  def processMsg(msg: Msg): Behavior = msg match {
    case Inc =>
      backup ! Inc
      Primary(backup, counter + 1)

    case _ => this
  }
}
```

```scala
case class Backup(counter: BigInt) extends Behavior {

  def processMsg(msg: Msg): Behavior = msg match {
    case Inc =>
      Backup(counter + 1)

    case _   => this
  }
}
```

```scala
def invariant(s: ActorSystem): Boolean = {
  val primary = s.behaviors(PrimaryRef)
  val backup  = s.behaviors(BackupRef)

  val pending = s.inboxes(PrimaryRef -> BackupRef)
                 .filter(_ == Inc)
                 .length

  primary.counter == backup.counter + pending
}
```

```scala
def preserve(s: ActorSystem, n: ActorRef, m: ActorRef) = {
  require(invariant(s))

  val next = s.step(n, m)

  invariant(next)
}.holds
```

Demo (?)

Case studies

Conc-Rope

We ship a verified implementation of this data-structure, which provides:

- Worst-case *O(log n)* time lookup, update, split and concatenation operations
- Amortized *O(1)* time append and prepend operations

Very useful for efficient data-parallel operations!

cf. A. Prokopec, M. Odersky. Conc-trees for functional and parallel programming

Leader election

- Fully verified implementation of *Chang and Roberts algorithm for leader election* as an actor system
- Runs on top of Akka
- ~100 lines of code for the implementation
- ~2000 lines of code for specification + proofs

`github.com/epfl-lara/stainless-actors`

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 re-entrancy, and 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

`github.com/epfl-lara/smart`

Other examples

You can find more verified code in our test suite, and in our *Bolts* repository:

- Huffman coding
- Reachability checker
- Left-Pad!
- and more…

`github.com/epfl-lara/bolts`

Give it a spin!

```
$ sbt new epfl-lara/stainless-project.g8
```

Learn more

stainless.epfl.ch

github.com/epfl-lara/stainless

gitter.im/epfl-lara/stainless

Learn more (2)

- Installation (standalone, sbt, docker)
- Tutorial
- Ghost context
- Imperative features
- Wrapping existing/external code
- Proving theorems
- Stainless library

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, Jad Hamza, Régis Blanc, Eva Darulova, Etienne Kneuss, Ravichandhran Kandhadai Madhavan, Georg Schmid, Mikaël Mayer, Emmanouil Koukoutos, Ruzica Piskac, Philippe Suter, as well as Marco Antognini, Ivan Kuraj, Lars Hupel, Samuel Grütter, Romain Jufer, and myself.

Many thanks as well to our friends at LAMP, ScalaCenter, and TripleQuote for their help and advice.

Thank you!