Formal verification of Scala programs with Stainless

Romain Ruetschi

Laboratory for Automated Reasoning and Analysis

Typelevel Summit Lausanne
June 2019

About me

Romain Ruetschi

@_romac

MSc in Computer Science, EPFL

Engineer at LARA, under the supervision of Prof. Viktor Kunčak

Outline

Stainless

Verification

Termination

Verifying type classes

Case studies

Bonus

Coming soon / further work

Stainless

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

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 and local classes, inner functions
- Partial support for GADTs
- Type members, type aliases
- Limited mutation, while, traits/classes with vars, partial functions, …

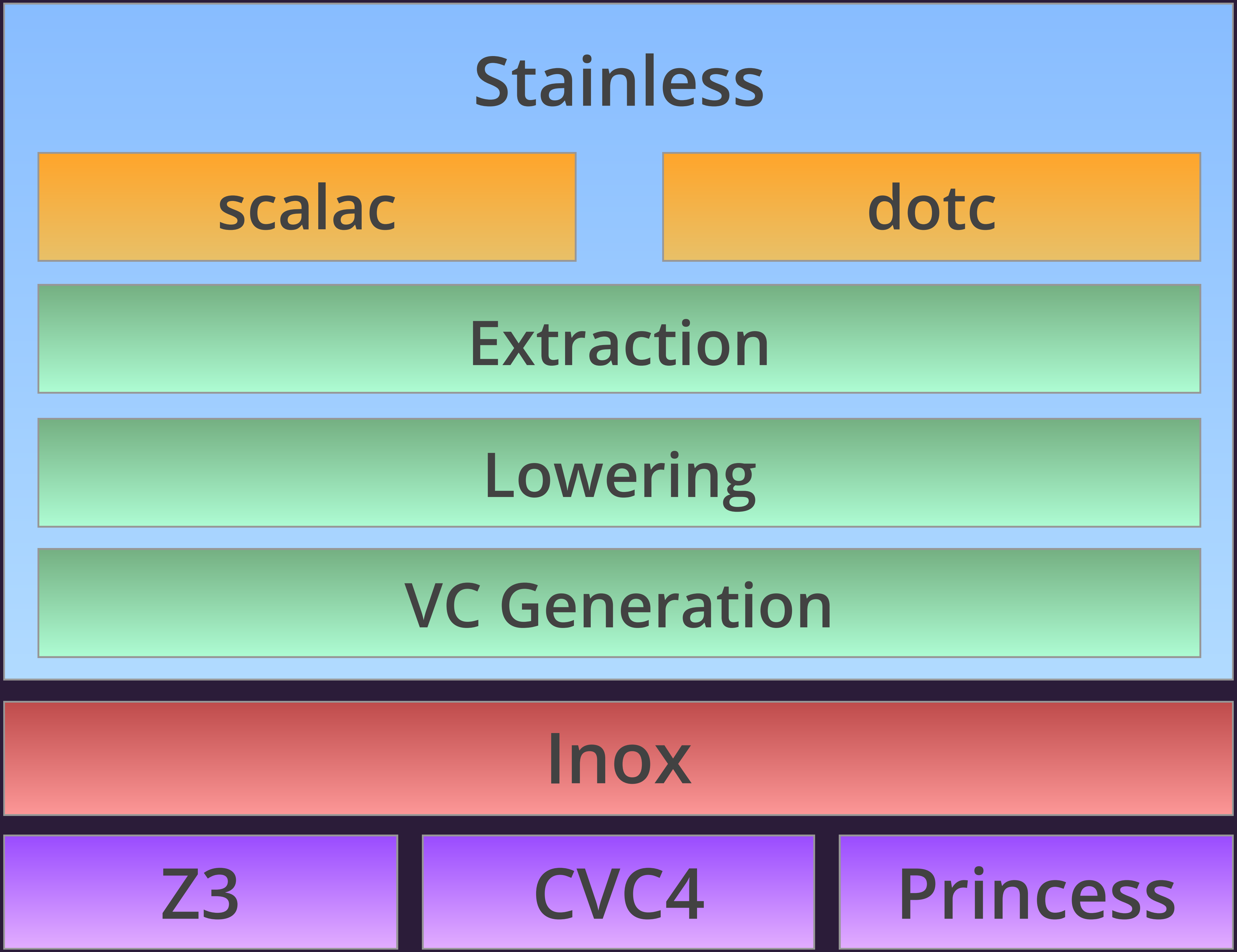
Currently backed by scalac 2.12

+ some Dotty features

- Dependent function types
- Extension methods
- Opaque types

Dotty 0.12 support only

Pipeline


Stainless
scalac
dotc
Extraction
Lowering
VC Generation
Inox
Z3
CVC4
Princess

Lowering phases

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

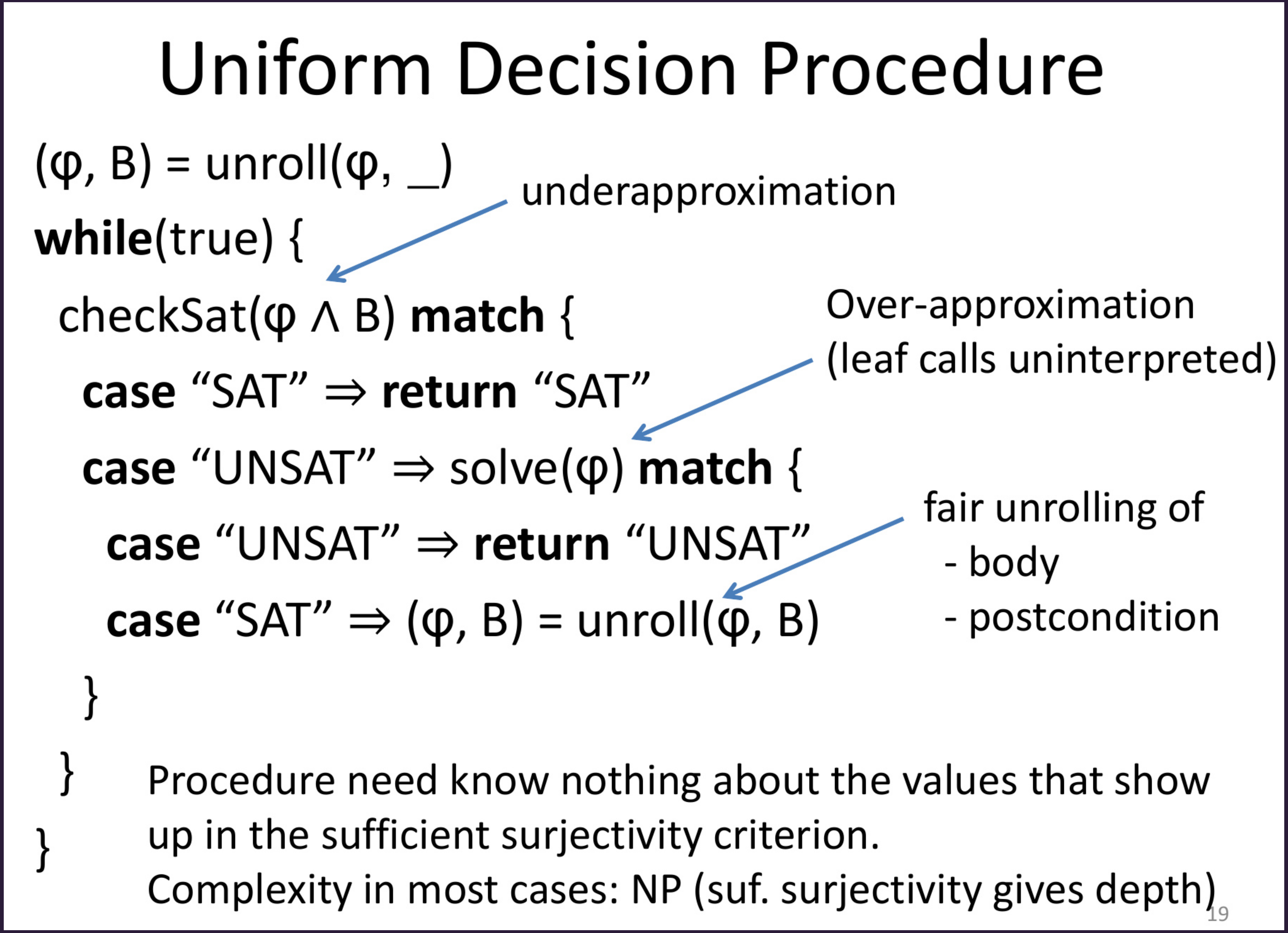
Inox

Inox is a solver for higher-order functional programs which provides first-class support for features such as:

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

Backed by SMT solvers such as Z3 or CVC4

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)

Verification

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

Verification

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

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

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

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**

Termination

- Stainless comes with a powerful termination checker
- User-defined measures, CFA, structural sizes, etc.
- Important for correctness and soundness of proofs

Termination

```
def append[A](l: List[A], r: List[A]): List[A] = l match {
  case Nil()    ⇒ r
  case x :: xs ⇒ x :: append(l, r)
}                              ^
```

```
[Warning] Result for append
[Warning]  => BROKEN (Loop Processor)
[Warning]   Function append loops given inputs:
[Warning]   l: List[A] -> Cons[A](A#0, Nil[A]())
[Warning]   r: List[A] -> Nil[A]()
```

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

```
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

Generated VC

Stainless then yields the following *verification condition*:

```
val isSum: (Object) ⇒ Boolean = (x: Object) ⇒ x is Sum
val xObj: { x: Object | isSum(x) } = SumObject(x)
val yObj: { x: Object | isSum(x) } = SumObject(y)
val zObj: { x: Object | isSum(x) } = SumObject(z)

{
  combine(isSum, thiss, x, combine(A, thiss, y, z)) ==
  combine(isSum, thiss, combine(A, thiss, x, y), z)
}
```

This program/formula is then passed down to the *Inox* solver

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.

Induction

```
implicit def listSemi[A] = new Semigroup[List[A]] {
  def empty = Nil
  def combine(x: List[A], y: List[A]) = x ++ y

  override def law_assoc(@induct x: List[A], y: List[A], z: List[A]) = {
                         ^^^^^^^
    super.law_assoc(x, y, z)
  }
}
```

Base case: `x == Nil`

```
    law_assoc(Nil, y, z)
⟺  Nil ++ (y ++ z) == (Nil ++ y) ++ z
⟺  y ++ z == y ++ z
⟺  true
```

```
Inductive step: x == xh :: xt

law_assoc(xt, y, z) ==> law_assoc(xh :: xt, y, z)
```

Inox automatically finds the following proof:

```
  (xh :: xt) ++ (y ++ z)
= xh :: (xt ++ (y ++ z)) // def ++
= xh :: ((xt ++ y) ++ z) // law_assoc(xt, y, z)
= (xh :: (xt ++ y)) ++ z // def ++
= ((xh :: xt) ++ y) ++ z // def ++
```

Sequential fold

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

foldMap(List(1, 2, 3, 4))(Sum(_)).get
// ⇒ 10
```

Parallel fold

```scala
@extern
def parFoldMap[M, A](xs: List[A])(f: A ⇒ M)
  (implicit M: Monoid[A]): M = {

  xs.toScala.par.map(f).fold(M.empty)(M.append)

} ensuring { res ⇒
  res == foldMap(xs, f)
}

parFoldMap(List(1, 2, 3, 4))(Sum(_)).get
// ⇒ 10
```

Other 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-parellel operations!

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

Parallel Map-Reduce pipeline

Fully verified implementation of a parallel Map-Reduce pipeline, using the aforementioned verified Conc-Rope implementation under the hood + various type classes.

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

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

`github.com/epfl-lara/smart`

Actor systems

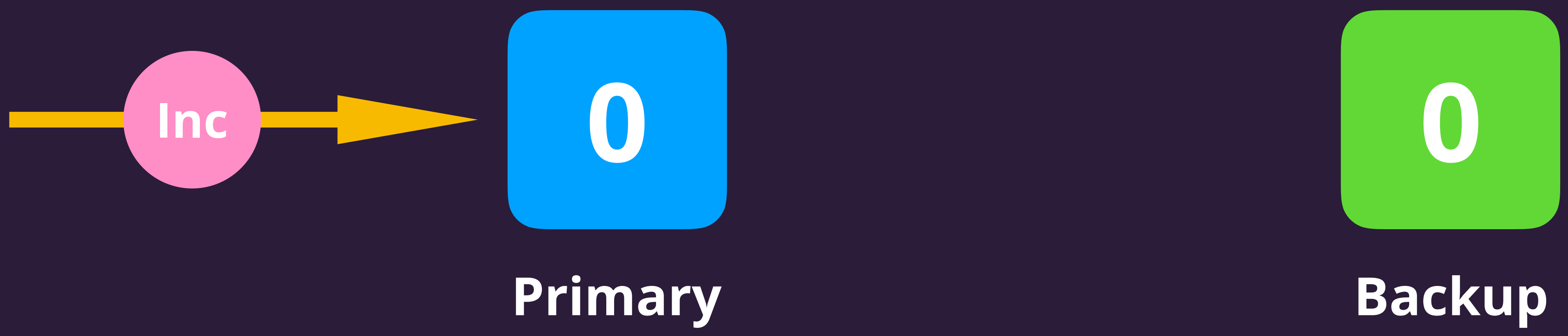
Counter

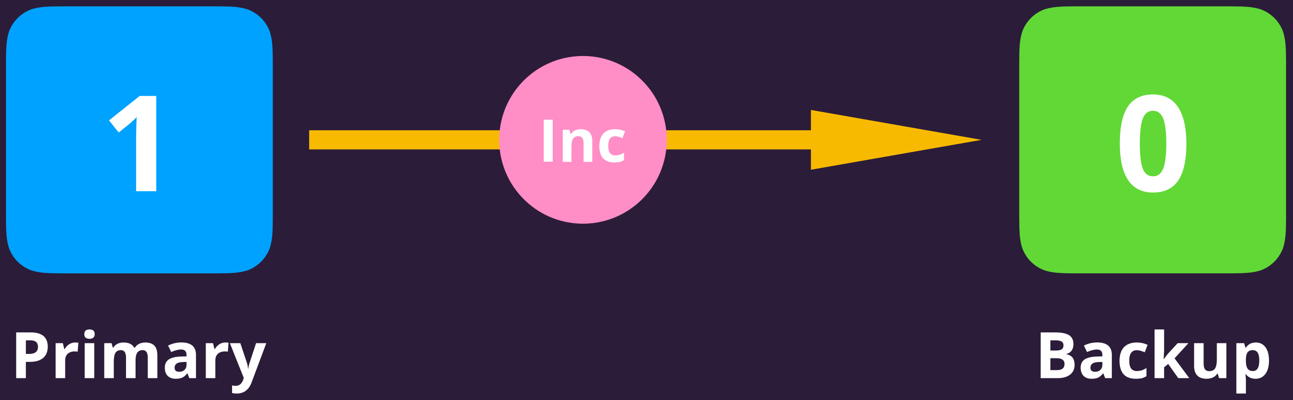
0

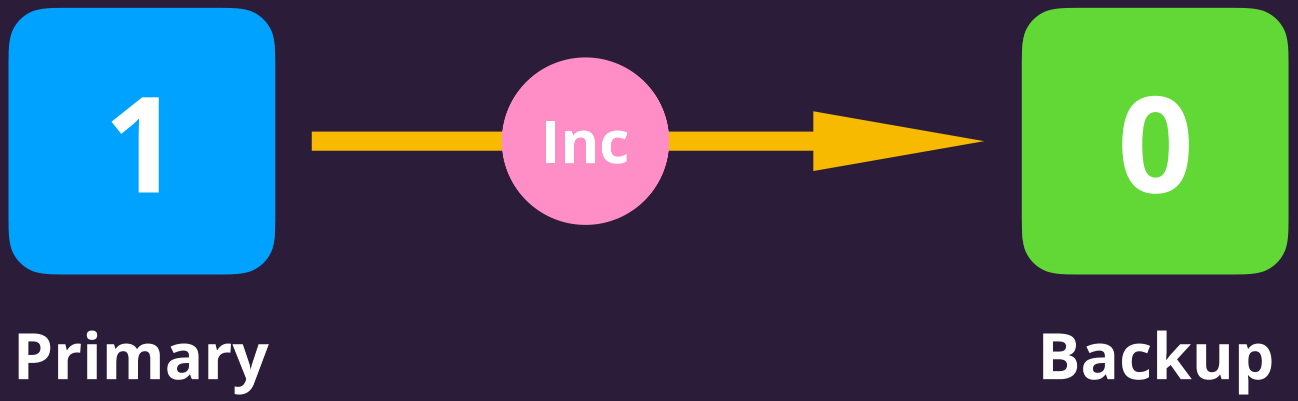
Primary

0

Backup


Inc
0
Primary
0
Backup


1
Inc
0
Primary
Backup


1
Inc
0
Primary
Backup

1
Primary
1
Backup

```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
```

Akka

Such systems can then run on top of Akka via a thin wrapper

… and more!

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

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

`github.com/epfl-lara/bolts`

Bonus

Refinement types

```
type Nat = { n: BigInt => n >= 0 }
```

Available via our Dotty front-end

Makes use of our fork of Dotty 0.12, which adds syntax support for refinement types (but notype checking)

`github.com/epfl-lara/dotty`

Refinement types

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

```
def sortedInsert(xs: List[Int]): List[Int] = {
  require {
    isSorted(xs) && (xs.isEmpty || x <= xs.head)
  }

  x :: xs

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

There is more!

- sbt/compiler plugin + metals integration (currently broken, fix coming up)
- Ghost context
- Partial evaluation
- DSL for writing proofs

Coming soon(ish)

- **Higher-kinded types**
- Better support for refinement types
- VC generation via type-checking algorithm

Coming later

- Scala 2.13
- Latest Dotty
- Linearity

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.

Learn more

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

`stainless.epfl.ch`

`github.com/epfl-lara/stainless`

Thank you!