Proving Algebraic Laws in Scala using Stainless and System FR

Speaker: Viktor Kunčak, Lambda Days, 14 February 2020

https://stainless.epfl.ch

Getting Simple Things Right

Not **all** of our code can be perfect.

Getting Simple Things Right

Not **all** of our code can be perfect.

We should know how to make
some parts of our code perfect.

Getting Simple Things Right

Not **all** of our code can be perfect.

We should know how to make
some parts of our code perfect.

formal verification

Getting Simple Things Right

Not **all** of our code can be perfect.

We should know how to make
some parts of our code perfect.

formal verification
with **stainless.epfl.ch**

Software to Achieve a Goal

goal

Software to Achieve a Goal

software

goal

Software to Achieve a Goal

software

after manual tests

goal

Software to Achieve a Goal

software

after manual tests

after QuickCheck

goal

Software to Achieve a Goal

software

after manual tests

after QuickCheck

stainless

goal

Example: Lists of Differences

$$\begin{tikzcd} \begin{array}{|c|c|c|}\hline 100 & 101 & 95 \\ \hline\end{array} \arrow[r, shift left, "\mathit{diffs}"] & \begin{array}{|c|c|c|}\hline 100 & +1 & -6 \\ \hline\end{array} \arrow[l, shift left, "\mathit{undiff}"] \end{tikzcd}$$

monthly account balances

initial value, monthly gains

Lists of Differences in Scala

```scala
def undiff(l: List[Int]): List[Int] =
  l.scanLeft(0)(_ + _).tail
  // List(a,b).scanLeft(z)(f) = List(z, f(z,a), f(f(z,a),b))

val testUndiff = undiff(List(100, 1, -6)) // 100, 101, 95
```

Lists of Differences in Scala

```
def undiff(l: List[Int]): List[Int] =
  l.scanLeft(0)(_ + _).tail
  // List(a,b).scanLeft(z)(f) = List(z, f(z,a), f(f(z,a),b))

val testUndiff = undiff(List(100, 1, -6)) // 100, 101, 95
```

Goal: define `diffs` such that:

```
val testDiff = diffs(List(100, 101, 95)) // 100, 1, -6

// more generally:
def check(l: List[Int]): Boolean =
  undiff(diffs(l))==l  // should always evalute to true
```

diffs Implementation

```
def diffs(l: List[Int]): List[Int] = {
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 =>    h :: (h1 - h) :: t1

      }
  }
}
```

diffs Implementation

```
// 100 :: 101 :: 95 :: Nil  ==>  100 :: 1 :: -6 :: Nil
def diffs(l: List[Int]): List[Int] = {
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>     //  100 :: 101 :: 95 :: Nil
      diffs(t) match { //         101 :: -6 :: Nil
        case h1 :: t1 =>    h :: (h1 - h) :: t1
                       //  100 ::     1     :: -6 :: Nil
      }
  }
}
```

diffs Implementation

```
// 100 :: 101 :: 95 :: Nil  ==>  100 :: 1 :: -6 :: Nil
def diffs(l: List[Int]): List[Int] = {
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>      //  100 :: 101 :: 95 :: Nil
      diffs(t) match { //         101 :: -6 :: Nil
        case h1 :: t1 =>    h :: (h1 - h) :: t1
                       //  100 ::      1     :: -6 :: Nil
      }
  }
} ensuring (undiff(_) == l) // in Scala: runtime assertion
```

Testing diffs

```
scala> :load Test.scala
Test.scala:17: warning: match may not be exhaustive.
It would fail on the following input: Nil

scala> diffs(List(3, 13, 23))  // each time ensuring runs, too
res1: List[Int] = List(3, 10, 10)

scala> diffs(List())
res2: List[Int] = List()

scala> diffs(List(4,3,20,100,23,5))
res3: List[Int] = List(4, -1, 17, 80, -77, -18)
...
```

Write more tests.

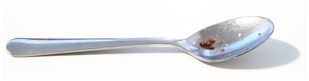
Testing diffs

```
scala> :load Test.scala
Test.scala:17: warning: match may not be exhaustive.
It would fail on the following input: Nil

scala> diffs(List(3, 13, 23))  // each time ensuring runs, too
res1: List[Int] = List(3, 10, 10)

scala> diffs(List())
res2: List[Int] = List()

scala> diffs(List(4,3,20,100,23,5))
res3: List[Int] = List(4, -1, 17, 80, -77, -18)
...
```

Write more tests. And QuickCheck! Great talk yesterday by John Hughes.

But...

But...

Does it ever end? Will we ever be done, or  ?

But...

Does it ever end? Will we ever be done, or ?
Run Stainless on the file:

But...

Does it ever end? Will we ever be done, or ?
Run Stainless on the file:

```
$ stainless-scalac ListDiffsInt.scala --solvers=smt-cvc4
  => Found measure for scanLeft.
  => Found measure for diffs.
  Generating VCs for those functions: undiff, diffs
  - Now solving 'postcondition' VC for diffs @12:30...
  ...
  total: 23 | valid: 23 (5 from cache) | invalid: 0 | unknown: 0
```

But...

Does it ever end? Will we ever be done, or ?
Run Stainless on the file:

```
$ stainless-scalac ListDiffsInt.scala --solvers=smt-cvc4
  => Found measure for scanLeft.
  => Found measure for diffs.
  Generating VCs for those functions: undiff, diffs
  - Now solving 'postcondition' VC for diffs @12:30...
  ...
  total: 23 | valid: 23 (5 from cache) | invalid: 0 | unknown: 0
```

Done.

Stainless generated 23 formulas that imply correctness (for **all** inputs).
It then automatically proved that these formulas are logically valid.

It Was All Automatic – This Was The Entire Input

```
import stainless.lang._
import stainless.collection._
object Diffs {
  def undiff(l: List[Int]): List[Int] =
    l.scanLeft(0)(_ + _).tail
  def diffs(l: List[Int]): List[Int] = {
    l match {
      case Nil() => l
      case _ :: Nil() => l
      case h :: t =>
        diffs(t) match {
          case h1 :: t1 => h :: (h1 - h) :: t1
        }
    }
  } ensuring (undiff(_) == l)
}
```

It Was All Automatic – This Was The Entire Input

```
import stainless.lang._
import stainless.collection._
object Diffs {
  def undiff(l: List[Int]): List[Int] =
    l.scanLeft(0)(_ + _).tail
  def diffs(l: List[Int]): List[Int] = {
    l match {
      case Nil() => l
      case _ :: Nil() => l
      case h :: t =>
        diffs(t) match {
          case h1 :: t1 => h :: (h1 - h) :: t1
        }
    }
  } ensuring (undiff(_) == l)
}
```

Proof guarantees
like Coq, Isabelle/HOL
+ automation
+ specs and code executable
(like for QuickCheck)

What Did Stainless Prove? Why does it hold?

```
def diffs(l: List[Int]): List[Int] = {
  // function terminates for all inputs
  l match { // match is exhaustive
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match { // match is exhaustive
        case h1 :: t1 =>  h :: (h1 - h) :: t1
      }
  }
} ensuring (undiff(_) == l) // spec holds for all inputs
```

Rest of The Talk

Two questions:

Rest of The Talk

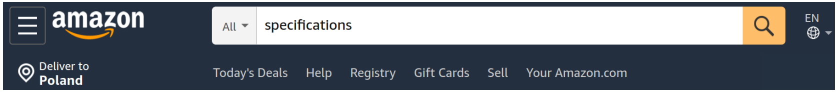
Two questions:

1. How to get specifications?

Rest of The Talk

Two questions:

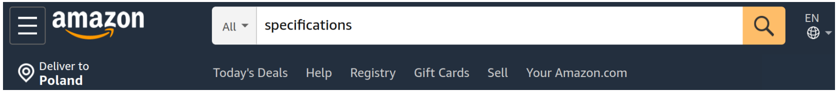
1. How to get specifications?



Rest of The Talk

Two questions:

1. How to get specifications?

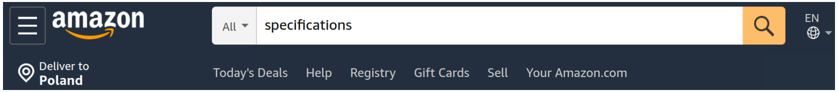


2. Will this ever stop?

Rest of The Talk

Two questions:

1. How to get specifications?



2. Will this ever stop? (termination of functions)

```
f => (a,b) => (f(a), f(b))          (x => x(x))(x => x(x))

def map(f, l) = f(l.head) :: (() => map(l.tail ()))
```

Question 1: How to get specifications?

Source #1 of **Free Specifications**: Semantics

```
def diffs(l: List[Int]): List[Int] = {
  l match { // match is exhaustive
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match { // match is exhaustive
        case h1 :: t1 =>  h :: (h1 - h) :: t1
      }
  }
}
```

We never want: match to fail, Nil.head, x/0, list(-1)

Stainless generates these from the code alone (nothing to write)!

Source #2 of **Free Specifications**: Semantics#

```
def diffs(l: List[Int]): List[Int] = {
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 =>  h :: (h1 - h) :: t1
      }
  }
}
```

Scala: Int is a signed 32-bit integer, $[-2^{31}, 2^{31}-1]$. Ops are modulo 2^{32}

Do we want to allow overflow and underflow for “-” in our program?

Overflows: An Example of Sharper Semantics

Stainless correctly models Int as 32-bit signed integer of Scala.
Our program so far was correct: diffs is really inverse of undiff.
Stainless also supports BigInt, which maps to $\mathbb{Z}$ (runs 100x slower).
But stainless can do overflow checking for all Int operations:

```
strict-arithmetic = true
```

```
  - Result for 'body assertion: Subtraction overflow' VC for diffs:
  => INVALID
  Found counter-example:
    l: List[Int] -> -2147483648 :: 0 :: Nil()
```

which is due to $0 - (-2^{31})$ computation in:

```
    case h1 :: t1 => h :: (h1 - h) :: t1
```

Diffs of Increasing Sequence

```
def increasing(l: List[Int]): Boolean =
  l match {
    case Nil() => true
    case _ :: Nil() => true
    case x1 :: x2 :: xs =>
      x1 <= x2 && increasing(x2::xs)
  }
def diffs(l: List[Int]): List[Int] = {
  require(increasing(l))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Diffs of Increasing Sequence

Again counterexample!
-536870909 :: 1644167168 :: Nil()

```
def increasing(l: List[Int]): Boolean =
  l match {
    case Nil() => true
    case _ :: Nil() => true
    case x1 :: x2 :: xs =>
      x1 <= x2 && increasing(x2::xs)
  }
def diffs(l: List[Int]): List[Int] = {
  require(increasing(l))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Diffs of Increasing Sequence

```
def increasing(l: List[Int]): Boolean =
  l match {
    case Nil() => true
    case _ :: Nil() => true
    case x1 :: x2 :: xs =>
      x1 <= x2 && increasing(x2::xs)
  }
def diffs(l: List[Int]): List[Int] = {
  require(increasing(l))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Again counterexample!

-536870909 :: 1644167168 :: Nil()

```
Prelude> 1644167168 - (-536870909)
2181038077
Prelude> 2^31
2147483648
```

Diffs of Increasing Sequence

```
def increasing(l: List[Int]): Boolean =
  l match {
    case Nil() => true
    case _ :: Nil() => true
    case x1 :: x2 :: xs =>
      x1 <= x2 && increasing(x2::xs)
  }
def diffs(l: List[Int]): List[Int] = {
  require(increasing(l))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Again counterexample!

-536870909 :: 1644167168 :: Nil()

```
Prelude> 1644167168 - (-536870909)
2181038077
Prelude> 2^31
2147483648
```

```
scala> 1644167168 - (-536870909)
res0: Int = -2113929219
```

Diffs of Increasing Sequence

```
def increasing(l: List[Int]): Boolean =
  l match {
    case Nil() => true
    case _ :: Nil() => true
    case x1 :: x2 :: xs =>
      x1 <= x2 && increasing(x2::xs)
  }
def diffs(l: List[Int]): List[Int] = {
  require(increasing(l))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Again counterexample!

-536870909 :: 1644167168 :: Nil()

```
Prelude> 1644167168 - (-536870909)
2181038077
Prelude> 2^31
2147483648
```

```
scala> 1644167168 - (-536870909)
res0: Int = -2113929219

scala> 2^31
res1: Int = 29
```

Diffs of Increasing **Non-Negative** Sequence

```
def diffs(l: List[Int]): List[Int] = {
  require(l == Nil() || (l.head >= 0 && increasing(l)))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Diffs of Increasing **Non-Negative** Sequence

```
def diffs(l: List[Int]): List[Int] = {
  require(l == Nil() || (l.head >= 0 && increasing(l)))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

All verifies!

Stainless also proves that **require** holds for all recursive calls.

Diffs of Increasing **Non-Negative** Sequence

```
def diffs(l: List[Int]): List[Int] = {
  require(l == Nil() || (l.head >= 0 && increasing(l)))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

All verifies!

Stainless also proves that **require** holds for all recursive calls.

Why does it hold in the example? **Unfold functions** to see.

Aside: What If Specifications Are Wrong?

Aside: What If Specifications Are Wrong?

You are used to programming but not specifying, your specifications are initially wrong much more often than code!

And that's okay.

Strive to write specs that are as *independently wrong* compared to code as possible.

Chance of writing code that meets the spec but where *both* of them are wrong is much lower than the chance of code alone being wrong.

My Mistake: Thought This Would Prevent Overflows

```
def increasingS(l: List[Int]): Boolean =
  l match {
    case Nil() => true
    case x :: Nil() => 0 <= x
    case x1 :: x2 :: xs =>
      x1 <= x2 && increasingS(x2::xs)
  }
def diffs(l: List[Int]): List[Int] = {
  require(increasingS(l))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

My Mistake: Thought This Would Prevent Overflows

```
def increasingS(l: List[Int]): Boolean =
  l match {
    case Nil() => true
    case x :: Nil() => 0 <= x
    case x1 :: x2 :: xs =>
      x1 <= x2 && increasingS(x2::xs)
  }
def diffs(l: List[Int]): List[Int] = {
  require(increasingS(l))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Again overflow found!

This Variant Does Verify

```
def increasing(l: List[Int]): Boolean =
  l match {
    case Nil() => true
    case x :: Nil() => 0 <= x
    case x1 :: x2 :: xs =>
      0 <= x1 && x1 <= x2 && increasing(x2::xs)
  }
def diffs(l: List[Int]): List[Int] = {
  require(increasing(l))
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Source #3 of Specifications: Symbolic Tests

Like tests, but with symbolic values for elements.

```
def symTest1(x1: BigInt, x2: BigInt,
             x3: BigInt, x4: BigInt): Boolean = {
  diffs(List(x1,x2,x3,x4)) ==
        List(x1, x2 - x1, x3 - x2, x4 - x3)
}.holds
```

Stainless proves it in 1.5 seconds on a laptop, for all (infinitely many) values x1,x2,x3,x4. Also works for Int.
Such tests prove or give a counterexample once recursion gets unfolded by the depth of the data structure.

Source #4 of Specifications: Inverses

diffs and undiff are inverse of each other:

```
def diffs(l: List[Int]): List[Int] = {
  ...
} ensuring (undiff(_) == l) // spec holds for all inputs
```

Further examples:

- (lossless) compression and decompression (Huffman, LZW)
- printing and parsing
- serialization: storing to files, network, file formats

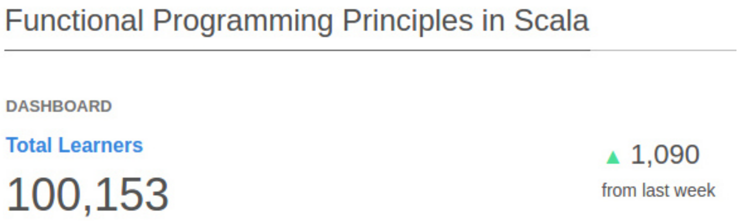
Source #5 of Specifications: Another Implementation

Regression verification: past versions as a reference.

```
def diffs2(l: List[Int]): List[Int] = {
  l match {
    case Nil() => l
    case h :: t =>
      h :: l.zip(t).map({ case (h1,h2) => h2 - h1})
  }
}
@induct
def equiv(l: List[Int]): Boolean = {
  diffs(l) == diffs2(l)
}.holds
```

Source #6 of Specifications: MOOCS

Popular course and specialization on Coursera, e.g. *first* course



Functional Programming Principles in Scala

DASHBOARD

Total Learners

100,153

▲ 1,090

from last week

Consider *second* course, **Functional Program Design in Scala**

Lecture 2.1 - **Structural Induction on Trees**

Correctness of binary search tree storing integers - IntSet.scala

Complete IntSet.scala Example from the MOOC

```scala
case class Empty() extends IntSet
case class Node(left: IntSet, elem: Int, right: IntSet) extends IntSet
abstract class IntSet {
  def contains(x: Int): Boolean = this match {
    case Empty() => false
    case Node(left, elem, right) ⇒
      if (x < elem) left.contains(x)
      else if (x > elem) right.contains(x)
      else true }
  def incl(x: Int): IntSet = this match {
    case Empty() => Node(Empty(),x,Empty())
    case Node(left, elem, right) =>
      if (x < elem) Node(left.incl(x), elem, right)
      else if (x > elem) Node(left, elem, right.incl(x))
      else this }
}
```

What the lecture segment proves

Algebraic properties: relate multiple operations

- unlike simple ensuring that had no user defined functions

For all s:IntSet, x:Int, y:Int

```
P1:  ! Empty().contains(x)
P2:  s.incl(x).contains(x)
P3:  x != y ==>  (s.incl(x).contains(y) == s.contains(y))
```

How does Martin Odersky prove these properties in the lecture?

- induction on tree structure (assume for subtrees)
- equational reasoning (substitute equals for equals)
- case analysis (e.g., ordering between integer elements)

Manual Proof vs Stainless

Coursera lecture segment with manual proof: **15 minutes**
Type and implementation itself: 24 lines of code

Proving using stainless:

- **20 lines** of properties and the statement they should be shown by induction
- **1 second** of waiting for stainless to finish

Source #7 of Specifications: Models

seL4 microkernel effort in Nicta: Haskell OS as a model for C version

Lighter versions:

- sizes: gives us idea when things are growing, shrinking
- sets: what is lost and what not (and we can automate it!)
- lists: great specs, even if automation more challenging

Size as a Model

Size abstraction on diffs:

```
def diffs(l: List[Int]): List[Int] = {
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
} ensuring (_.size == l.size)
```

Set as a Model for Trees: Set Content

```scala
sealed abstract class Tree
case class Leaf() extends Tree
case class Node(left: Tree, value: BigInt, right: Tree) extends Tree

def content(tree: Tree): Set[BigInt] = tree match {
  case Leaf() => Set.empty[BigInt]
  case Node(l, v, r) => content(l) ++ Set(v) ++ content(r)
}
def isBST(tree: Tree) : Boolean = tree match {
  case Leaf() => true
  case Node(left, v, right) => {
    isBST(left) && isBST(right) &&
    forall((x:BigInt) => (content(left).contains(x) ==> x < v)) &&
    forall((x:BigInt) => (content(right).contains(x) ==> v < x))
  }
}
```

Set as a Model for Trees: Spec Using Content

```
def insert(tree: Tree, value: BigInt): Node = {
  require(isBST(tree))
  tree match {
    case Leaf() => Node(Leaf(), value, Leaf())
    case Node(l, v, r) => (if (v < value) {
      Node(l, v, insert(r, value))
    } else if (v > value) {
      Node(insert(l, value), v, r)
    } else {
      Node(l, v, r)
    })
  }
} ensuring(res => isBST(res) &&
                  content(res) == content(tree) ++ Set(value))
```

Source #8 of Specifications: Abstract Laws

```
abstract class Monoid[A] {
  def empty: A
  def append(x: A, y: A): A
  // Put laws into type class definitions:
  @law def law_leftIdentity(x: A) =
    append(empty, x) == x

  @law def law_rightIdentity(x: A) =
    append(x, empty) == x

  @law def law_associativity(x: A, y: A, z: A) =
    append(x, append(y, z)) == append(append(x, y), z)
}
def bigIntAdditiveMonoid: Monoid[BigInt] = new Monoid[BigInt] {
  def empty = 0
  def append(x: BigInt, y: BigInt) = x + y
  // Stainless inserts and proves monoid laws automatically
}
```

Parallelism: Talk of Gabriele Keller Yesterday

```
def fold[A](xs: Collection[A])(m: Monoid[A]): A = {
  // divide and conquer
  // m ensures that result does not depend on how we divide
  ...
}

def scan[A](xs: Collection[A])(m: Monoid[A]): Collection[A] =
  // clever divide and conquer parallel scan using m.append
  // m ensures that result does not depend on how we divide
  ...
}
```

Conc Tree and Conc Rope

Aleksandar Prokopec, Martin Odersky: Conc-Trees for Functional and Parallel Programming. LCPC 2015: 254-268

New data structures, building blocks for parallel collections

- ConcTree: $O(\log n)$ lookup, update, split, concat
- ConcRope: additionally $O(1)$ amortized prepend and append

Designed and proved (manually) in the paper above

Formally proved correct by Ravichandhran Kandhadai Madhavan in Leon and stainless (450 lines; 30sec); Ravi also formalized proofs of running time bounds.

R. Madhavan, S. Kulal, V. Kuncak: Contract-based resource verification for higher-order functions with memoization. POPL 2017: 330-343

Question 2: Will this every stop?

Stainless Tries to Prove Every Function Terminating

```
def diffs(l: List[Int]): List[Int] = {
  decreases(l) // inferred automatically
  l match {
    case Nil() => l
    case _ :: Nil() => l
    case h :: t =>
      diffs(t) match {
        case h1 :: t1 => h :: (h1 - h) :: t1
      }
  }
}
```

Inference of lexicographic measures works in many cases.

Why We Want Termination

It's a desired property for many internal functions!
Equation associated with, e.g., $f(x) = 1 + f(x)$ is a false statement.
If we do not check termination we must do things like:

- tell users that defining such functions means assuming false (minefield for users)
- give up equations associated with the function (makes manual and automated reasoning uglier)
- extend all types (e.g. Int, Boolean, ...) with $\perp$
  (was in fashion, but no longer: HOL, Isabelle/HOL, Coq, NuPRL)

Some functions that interact with environment should not be terminating, but you can transform them into terminating ones automatically by adding "fuel" parameters and an option type.

When do Higher-Order Functions Terminate

We say a function terminates if it terminates on terminating inputs:

```
  def pairMap[A,B](f: A => B)(p: (A,A)): (B, B) =
    (f(p._1), f(p._2))
```

parMap will diverge given certain diverging *f*.
But if *f* always terminates, so will *pairMap*.
If we have well-founded set of types, we can define such property by recursion on the type structure:

$[\![A \Rightarrow B]\!] = \{f \mid \forall a \in [\![A]\!].\, (f\, a)$ evaluates in finitely many steps to $v \in [\![B]\!]\}$

System FR

A type system foundation for Stainless verifier:

- Extends System F with refinements and recursive types
- Dependently typed system with Π, Σ, $\bigcap, \bigcup, \{\_\}$
- Type refinements capture preconditions, postconditions
- Support for contravariant data types and streams: indices and $\bigcap$
- Type checking ensures termination using measures (not restricted to structural recursion for function definitions)

Proven correct by interpreting types as sets of untyped terms.

Proofs formalized in (20k lines of) Coq by Jad Hamza.

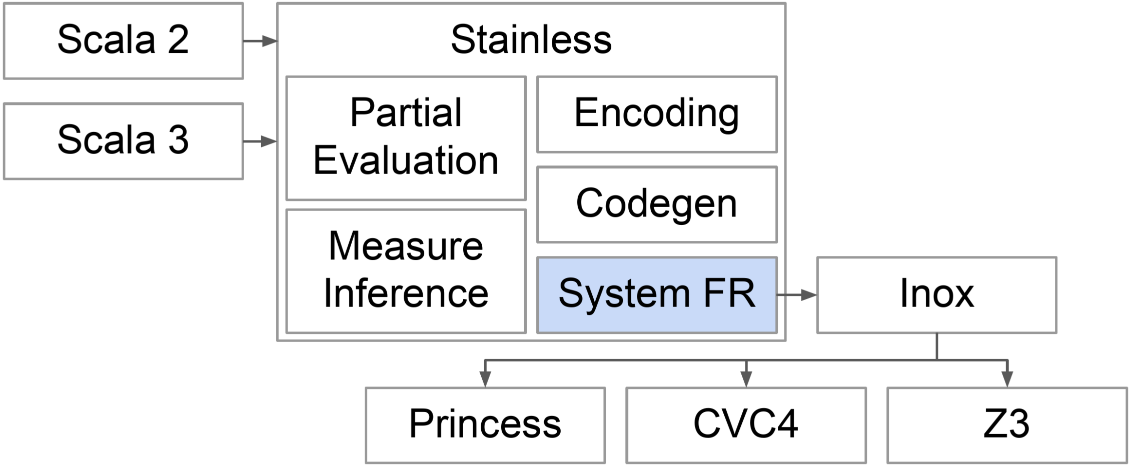
Verification condition generator adapted to follow the type system.

J. Hamza, N. Voirol, V. Kuncak

System FR: Formalized Foundations for the Stainless Verifier

Proc. ACM Program. Lang. OOPSLA. 2019.

Under the Hood of Stainless



Case Studies

Verified 14k lines of Scala code

- 5.8k verification conditions
- 6.5 minutes

Verified examples include

- Monad laws. Sorting algorithms. Graph reachability. Dynamic programming.
- Lazy and concurrent data-structures. Simple distributed algorithms.
- LZW Compression. Model of key server. Smart contracts.
- Number theory properties (Gödel numbering)

Uses

Teaching “Formal Verification” course at EPFL

Collaboration with Interchain Foundation spinoff (distributed algorithms)

Discussions with IOHK and others in the past

Stainless for Haskell or Closure, anyone?

Stainless currently supports Scala 2 and an early version of Scala 3.

We are open to collaboration on adding front ends for other languages!

We have a cool **parsing combinator library** that uses zippers and derivatives for efficient parsing and tree building:

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

Developed by Romain Edelmann, a PhD student in my group

Formalized Mathematics

System FR is about **computable** functions.
We have:

```
(Any ⇒ Any) <: Any
```

and subtyping for us is set inclusion. But the set of **total** functions on non-trivial set cannot be included in itself.
For reasoning about mathematical functions, we need something else.
Solution: use this type system inside a general-purpose logic:

- We have soundness and embedding of System FR in Coq
- We can alternatively embed System FR in set theory

Among the most sophisticated systems for formalized set theory and mathematics is **Mizar** by Andrzej W. Trybulec, born 1941 in Kraków.

Acknowledgments

Stainless is our new version of a verification system, primarily developed by Nicolas Voirol (PhD student, EPFL), with many contributions from Romain Ruetschi and from Jad Hamza.

Stainless extends the expressive power of Leon (leon.epfl.ch/doc), which does verification but also synthesis, constraint programming, repair, and programming by demonstration.
Leon itself was built through contributions of EPFL PhD students: Regis Blanc, Eva Darulova, Etienne Kneuss, Ravichandhran Kandhadai Madhavan, Mikaël Mayer, Emmanouil Koukoutos, Ruzica Piskac, Philippe Suter, as well as Romain Ruetschi, Marco Antognini, Ivan Kuraj, Lars Hupel, Samuel Grütter.

diffs Comes from Leon's Synthesis Benchmarks

Video and/or demo from `http://leon.epfl.ch`

```
def diffs(l : List[BigInt]): List[BigInt] =  {
  ???[List[BigInt]]
} ensuring { (res : List[BigInt]) =>
  res.size == l.size && undiff(res) == l
}

def undiff(l: List[BigInt]) = {
  l.scanLeft(BigInt(0))(_ + _).tail
}
```

Conclusions

Automated verification can be easier than hand proofs
Key design choice: use purely functional language for both the code and the properties

- **convenience**: developers can use a familiar language, reuse properties from QuickCheck-like testing
- **feasibility**: functional constraints can be handled using SMT solvers and iterative function unrolling, enabling the discovery of both *counterexamples* and *proofs*
- **expressive power**: many invariants, but also other language features (e.g. imperative) and non-functional properties (time, memory) can be encoded into purely functional constraints

Stainless can make your code shine.

`https://stainless.epfl.ch`