Refinement Types for Stainless

Katarzyna Marek, Samuel Chassot, Matt Bovel

Outline

1. Dotty(fork) has refinement types!

But a limited solver.

2. Stainless has a powerful solver!

But no refinements types.

3. Let's integrate Dotty(fork) with Stainless!

Refinement Types in Dotty(fork)

Refinement Types

Refinement types are types qualified with logical predicates. For example,

```
{x: Int with x > 0}
```

denotes the type of all integers `x` such that `x > 0`.

Implemented in many languages (Liquid Haskell, Boolean refinement types in F*, Subset types in Dafny, etc.).

Prior art in Scala: SMT-based checking of predicate-qualified types for Scala (Schmid and Kunčak, Scala Symposium 2016), Refined, Iron.

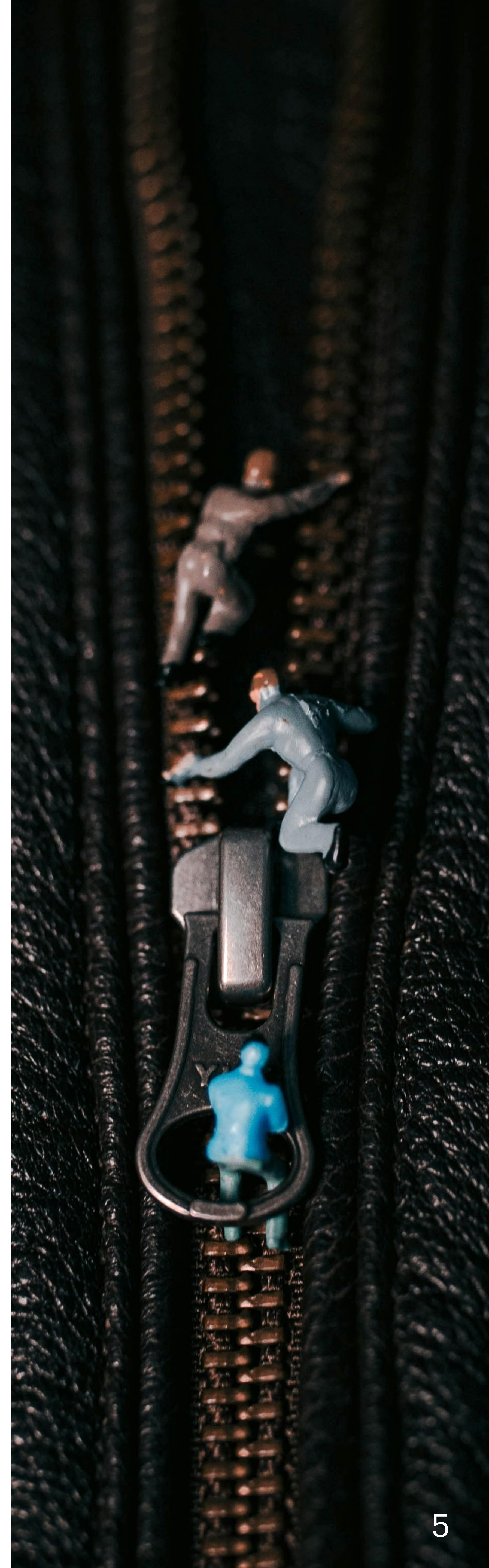
Example: Safe Lists Zipping

Consider the zip function:

```scala
def zip[A, B](xs: List[A], ys: List[B]): List[(A, B)]
```

Scala doesn't require both lists to have the same size:

```scala
val pairs = List(1, 2).zip(List("a"))
// List((1, "a"))
```



Example: Safe Lists Zipping

Requiring same sizes,

```
def zip[A, B](
  xs: List[A],
  ys: List[B] with ys.size == xs.size,
): {res: List[A, B] with res.size = xs.size}
```

Usage:

```
val l1: List[Int] with l1.size == 3 = …
val l2: List[String] with l2.size == 3 = …
zip(l1, l2): List[(Int, String)] with l3.size == 3
```

Need to show:

```
    {l2: List[String] with l2.size == 3}
 <: {l2: List[String] with l2.size == l1.size}
```

Limited Solver

Refinement types in Dotty ships with a minimal solver. It can:

- Reason about equality (like on the previous slide)
- Constant-fold and normalize: `1 + 2 <-> 3` and `y + z <-> z + y`
- Order: `x > 2 && y > x -> y > 2,` but not `x < 10 -> x + 2 < 8`
- Unfolding:

```
def fact(n: Int):
  {r: Int with r == (if n == 0 then 1 else n * fact(n - 1))}
```

`fact(1) = 1`, but not `fact(2) = 2`

Stainless Verifier

Stainless: Automated Proof

Verification Framework for Scala

```scala
def zip[A, B](xs: List[A], ys: List[B]): List[(A, B)] = {
  require(xs.size == ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))

    case _ => Nil()
}.ensuring (res => res.size == xs.size && res.map[A](p => p._1) == xs)
```

```
stainless summary

zip.scala:5:5:     zip  non-negative measure                                        valid  U:smt-cvc5  0.0
zip.scala:9:11:    zip  postcondition                                               valid  U:smt-cvc5  0.1
zip.scala:9:24:    zip  measure decreases                                           valid  U:smt-cvc5  0.0
zip.scala:9:24:    zip  precond. (call zip((scrut._1.t): @DropVCs , (scrut._...)    valid  U:smt-cvc5  0.1
zip.scala:11:15:   zip  postcondition                                               valid  U:smt-cvc5  0.0
----------------------------------------------------------------------------------------------------------
total: 5    valid: 5    (0 from cache, 0 trivial) invalid: 0    unknown: 0    time:    0.29
```

```
[Warning ] Found counter-example:
[Warning ]   xs: List[A] -> Cons[A](A#0, Nil[A]())
[Warning ]   ys: List[B] -> Nil[B]()
```

Supported Scala Subset

Algebraic Data Types

Pure functional

Imperative variables

While loops

Local classes

Pattern matching

Type classes

Inner functions

Traits

Polymorphism

Type parameters

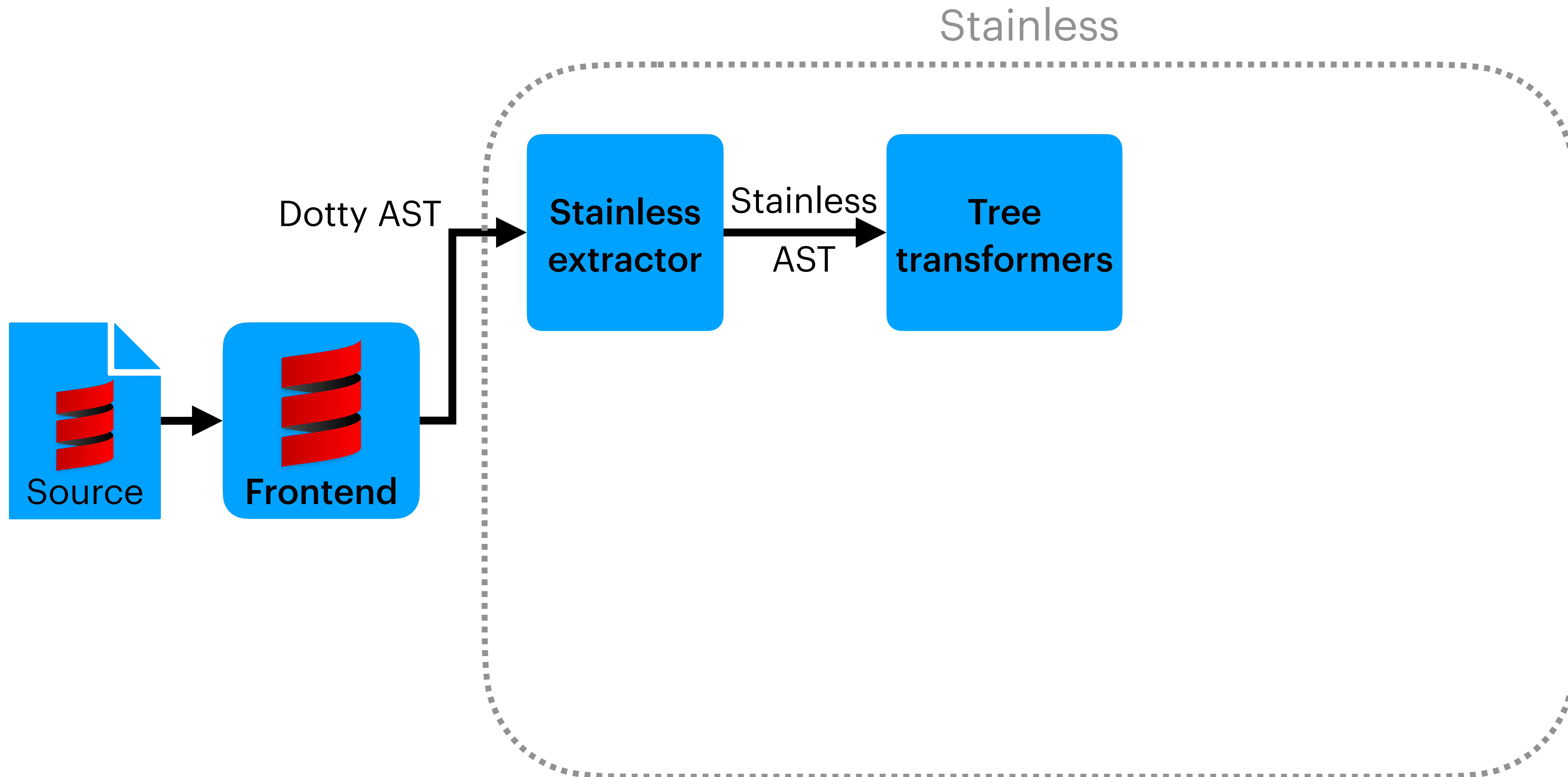
(Unreachable) exceptions

Type alias

Type members

Stainless

Verification framework for Scala



Stainless

Verification framework for Scala

[Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang, (OOPSLA), November 2019.]

[Philippe Suter, et al. Satisfiability modulo recursive programs. In Static Analysis Symposium (SAS), 2011.]
[Nicolas Voirol, et al. Counter-example complete verification for higher-order functions. In Scala Symposium, 2015.]

Internal trees

Before VC generation phase

```
def zip(xs: Cons[Int], ys: Cons[Boolean]): Cons[(Int, Boolean)] = {
  require(xs.size == ys.size)
  decreases(xs.size)
  (xs.t, ys.t) match
    case (xs0 @ Cons(_, _), ys0 @ Cons(_, _)) => Cons((xs.h, ys.h), zip(xs0, ys0))
    case _                                    => Cons((xs.h, ys.h), Nil())
}.ensuring(res => res.size == xs.size)
```

Internal trees

Before VC generation phase

```
def zip(xs: { v: List[Int] | v.isInstanceOf[Cons] },
        ys: { v: List[Boolean] | v.isInstanceOf[Cons] }
       ): { v: List[(Int, Boolean)] | v.isInstanceOf[Cons] } = {
  require(size[Int](xs) <= size[Boolean](ys))
  decreases(size[Int](xs))
  val targetBound: (List[Int], List[Boolean]) = (xs.t, ys.t)
  targetBound match {
    case (xs0 @ Cons(_, _), ys0 @ Cons(_, _)) =>
      Cons[(Int, Boolean)]((xs.h, ys.h), zip(xs0, ys0))
    case _ =>
      Cons[(Int, Boolean)]((xs.h, ys.h), Nil[(Int, Boolean)]())
  }
}.ensuring {
  (res: { v: List[(Int, Boolean)] | v.isInstanceOf[Cons] }) =>
    size[(Int, Boolean)](res) == size[Int](xs)
}
```

Stainless internal trees already rely on refinement types

Refinement Types
+
Stainless Verifier

Zip

```
def zip[A, B](a: List[A], b: List[B] with b.size == a.size ):
  { l: List[(A, B)] with l.size == a.size
                      && l.map[A](_._1) == a
                      && l.map[B](_._2) == b }=
   (a, b) match
    case (Nil(), Nil()) => Nil[(A, B)]()
    case (Cons(h1, t1), Cons(h2, t2)) =>
      Cons((h1, h2), zip(t1, t2))
```

```
stainless summary
/…/Zip.scala:4:5:     zip  non-negative measure                                                 valid from cache            0.0
/…/Zip.scala:8:3:     zip  body assertion: match exhaustiveness                                 valid from cache            0.0
/…/Zip.scala:8:3:     zip  type-checking                                                        valid            U:smt-z3   0.1
/…/Zip.scala:9:28:    zip  cast correctness                                                     valid            U:smt-z3   0.1
/…/Zip.scala:9:28:    zip  type-checking                                                        valid            U:smt-z3   0.1
/…/Zip.scala:11:7:    zip  cast correctness                                                     valid            U:smt-z3   0.2
/…/Zip.scala:11:7:    zip  type-checking                                                        valid            U:smt-z3   0.0
/…/Zip.scala:11:22:   zip  measure decreases                                                    valid from cache            0.0
/…/Zip.scala:11:30:   zip  cast correctness                                                     valid from cache            0.0
/…/Zip.scala:11:30:   zip  precond. (argument types: call zip[A, B]((scrut._1.t): @DropVCs , { ...)  valid from cache            0.0
----------------------------------------------------------------------------------------------------------------------------------
total: 10   valid: 10   (5 from cache, 0 trivial) invalid: 0    unknown: 0    time:    0.59
```

Scala Refinement Types
+ Stainless Verifier

Powerful solver for refinements (for a subset of Scala),

- e.g., verifies the zip example

Additional features:

- opt-out termination checking
- advanced exhaustivity checks

Stainless Verifier
+ Refinement Types

Specification in the signature

New syntax?

Interface

```
trait Sorting:
  def sort[T : Ordering](xs: List[T]): List[T] = {
    (??? : List[T])th l.isSorted }
  }.ensuring(res => res.isSorted)
```

```
class MergeSort extends Sorting { … }

class InstertionSort extends Sorting { … }
```

Higher-order functions

```scala
case class Pos(i: Int):
  require(i > 0)

def toRoots(xs: List[Pos]): List[Int] =
  xs.map(e => squareRoot(e.i))

def squareRoot(i: Int): Int =
  require(i > 0)
  ???
```

```scala
def toRoots(xs: List[Int]): List[Int] =
  xs.map(squareRoot)

def squareRoot(i: Int): Int =
  require(i > 0)
  ???
```

stainless summary				
/…/HigherOrderExample.scala:11:10:	toRoots	precond. (call squareRoot(i))	valid from cache	0.0
total: 1	valid: 1 (1 from cache, 0 trivial)	invalid: 0	unknown: 0	time: 0.04

stainless summary				
/…/HigherOrderExample.scala:11:10:	toRoots	precond. (call squareRoot(i))	invalid U:smt-z3	0.3
total: 1	valid: 0 (0 from cache, 0 trivial)	invalid: 1	unknown: 0	time: 0.34

Mutation

Intuitive semantics

```
def f(...): {res with T: p(res)}
```

Refinements on result types of **functions** are **postconditions**

```
val x: T with p(x)
```

Refinements on **values** are **assertions**

```
var x: T with p(x)
```

Refinements on **variables** are **invariants**

```
sort[T : Ordering](xs: List[T]):
```

```
val ls: List[Int] with ls.isSorted =
 List(1, 2, 4)
```

```
var ls: List[Int]
  with ls.isSorted =
   List(1, 2, 4)

  ls = List(5, 4) //error
```

```
r ls: List[Int] with ls.is
ist(1, 2, 4)

ls = List(5, 4) //error
```

```
val ls0: List[Int]
  with ls0.isSorted =
    List(1, 2, 4)

val ls1: List[Int]
  with ls1.isSorted =
    List(5, 4) //error
```

References to mutable state

```
case class Box(var v: Int):
  def setV(y: Int):
    {r: Unit with v == y} =
    v = y

val b = Box(3)
val p1: Unit with b.v^0 == 1
     = b.setV(1)
val p2: Unit with b.v^1 == 5
     = b.setV(5)

var p1: Unit with b.v^0 == 1
      = b.setV(1)
val p2: Unit with b.v^1 == 5
      = b.setV(5)

(p1: Unit with b.v^0 == 1) = p1 // ok
(p1: Unit with b.v^1 == 1) = p1 // error
```

Refinements on result types of **functions** are **postconditions**

Refinements on **values** are **assertions**

```
val b0: Box = Box(3)
val res1:
 (b0: Box, r: Unit with b0.v == 1)
  = setV(b0, 1)
val p1 = res1._2
val b1 = res1._1
val res2:
  (b1: Box, r: Unit with b1.v == 5)
  = setV(b1, 5)
```

0: Box, y0: Int):

b1.v == y0)

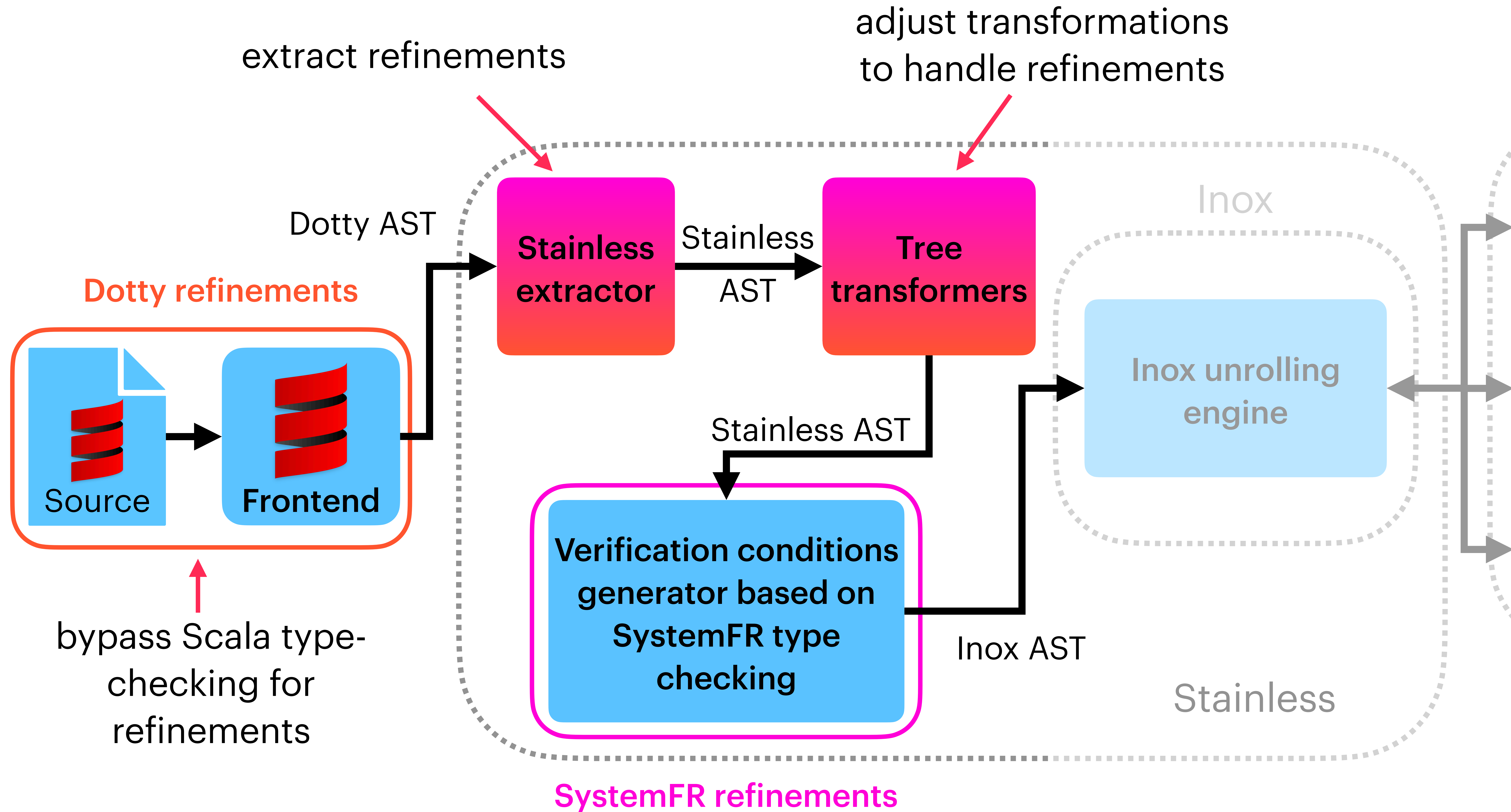
able states from

variables is **disallowed**

Intuitive semantics

	Pure refinements	**References** to **mutable states**	**Side-effects** in refinements
`def f(...): {res with T: p(res)}`	assertion/invariant	assertion	❌
`val x: T with p(x)`	assertion/invariant	assertion	❌
`var x: T with p(x)`	invariant	❌	❌

Implementation


extract refinements
adjust transformations to handle refinements
Dotty AST
Stainless extractor
Stainless AST
Tree transformers
Inox
Dotty refinements
Source
Frontend
Inox unrolling engine
Stainless AST
Verification conditions generator based on SystemFR type checking
Inox AST
Stainless
bypass Scala type-checking for refinements
SystemFR refinements

Stainless
+
Refinement Types
=
Powerful Verifier with Expressive Specification

- github.com/epfl-lara/dotty
- github.com/epfl-lara/stainless
- github.com/epfl-lara/inox

Thank you. Questions?