Constrained Horn Clause Solvers in Stainless

Sankalp Gambhir, Viktor Kunčak

Laboratory For Automated Reasoning And Analysis

EPFL

Stainless

- Verifier for Scala programs
- Completely automatic; target complex properties
- Case study: invertability of lossless image encoding [1]
- Case study: verifying the Scala library's HashMap [2]
- Others: logical systems, data structures, algorithms[1]

[1] `https://github.com/epfl-lara/bolts/`

Stainless

- Verifier for Scala programs
- Completely automatic; target complex properties
- Case study: invertability of lossless image encoding [1]
- Case study: verifying the Scala library's HashMap [2]
- Others: logical systems, data structures, algorithms[1]
- Reduces programs to a core functional language
- Implements a type checking procedure generating verification conditions
- Uses SMT solvers to discharge verification conditions

[1] `https://github.com/epfl-lara/bolts/`

Verification with Stainless

```
def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)
  if n <= 1 then
    1
  else
    fibonacci(n - 1) + fibonacci(n - 2)
}.ensuring { res => res >= n }
```

Verification with Stainless

```
def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)
  if n <= 1 then
    1
  else
    fibonacci(n - 1) + fibonacci(n - 2)
}.ensuring { res => res >= n }
```

Verification with Stainless

```scala
def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)                  // dynamic precondition
  if n <= 1 then
    1
  else
    fibonacci(n - 1) + fibonacci(n - 2)
}.ensuring { res => res >= n }  // dynamic postcondition
```

Verification with Stainless

```
import stainless.lang.*

def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)                 // *static* precondition
  if n <= 1 then
    1
  else
    fibonacci(n - 1) + fibonacci(n - 2)
}.ensuring { res => res >= n }  // *static* postcondition
```

Verification with Stainless

```
import stainless.lang.*

def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)                 // *static* precondition
  if n <= 1 then
    1
  else
    assume(f1 >= n - 1 && f2 >= n - 2)
    f1 + f2
}.ensuring { res => res >= n }  // *static* postcondition
```

```scala
import stainless.lang.*

def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)                   // *static* precondition
  if n <= 1 then
    1
  else
    assume(f1 >= n - 1 && f2 >= n - 2)
    f1 + f2                          // <-- simple induction
}.ensuring { res => res >= n }  // *static* postcondition
```

Verification with Stainless

```
import stainless.lang.*

def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)
  if n <= 1 then
    1
  else
    assume(f1 != -1 && f2 != -1)
    f1 + f2                         // <-- simple induction...?  (maybe -2 + 1?)
}.ensuring { res => res != -1 }
```

Verification with Stainless

```scala
import stainless.lang.*

def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)
  if n <= 1 then
    1
  else
    f1 + f2                                  // insufficient; unroll!
}.ensuring { res => res != -1 }

f1 = if (n - 1) <= 1 then 1 else f3 + f4 // + assume(f3 != -1 && f4 != -1)
f2 = if (n - 2) <= 1 then 1 else f5 + f6 // + assume(f5 != -1 && f6 != -1)
```

Verification with Stainless

```
import stainless.lang.*

def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)
  if n <= 1 then
    1
  else
    f1 + f2                                      // insufficient; unroll!
}.ensuring { res => res != -1 }

f1 = if (n - 1) <= 1 then 1 else f3 + f4 // + assume(f3 != -1 && f4 != -1)
f2 = if (n - 2) <= 1 then 1 else f5 + f6 // + assume(f5 != -1 && f6 != -1)
```

Ad infinitum.

Verification with Stainless

```
import stainless.lang.*

def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)
  if n <= 1 then
    1
  else
    f1 + f2                                  // insufficient; unroll!
}.ensuring { res => res != -1 }

f1 = if (n - 1) <= 1 then 1 else f3 + f4 // + assume(f3 != -1 && f4 != -1)
f2 = if (n - 2) <= 1 then 1 else f5 + f6 // + assume(f5 != -1 && f6 != -1)
```

Ad infinitum. Lucky? Timeout. Unlucky? Stack Overflow.

Inductive verification

`fibonacci(n) != -1` *is* an invariant.

Inductive verification

`fibonacci(n) != -1` *is* an invariant.

But not an *inductive* invariant.

Inductive verification

`fibonacci(n) != -1` *is* an invariant.

But not an *inductive* invariant.

How did we get here?

- We tried to encode our VC to suit the solver
- The problem is inductive, but the solver is not!
- But, we can verify very complex programs

Constrained Horn Clauses: alternative to Stainless' unfolding

`fibonacci(x) == y` $\mapsto$ `fibonacci_(x, y)`

Constrained Horn Clauses: alternative to Stainless’ unfolding

$$\texttt{fibonacci(x) == y} \mapsto \texttt{fibonacci\_(x, y)}$$

```
forall n. n >= 0 && n <= 1
            ==> fibonacci_(n, 1)                              // then branch

forall n f1 f2. n >= 0 && n > 1
       && fibonacci_(n-1, f1)
       && fibonacci_(n-2, f2)
            ==> fibonacci_(n, f1 + f2)                        // else branch

forall n res. fibonacci_(n, res) ==> res != -1                // postcondition
```

Constrained Horn Clauses: alternative to Stainless' unfolding

$$\texttt{fibonacci(x) == y} \mapsto \texttt{fibonacci\_(x, y)}$$

```
forall n. n >= 0 && n <= 1
          ==> fibonacci_(n, 1)                          // then branch

forall n f1 f2. n >= 0 && n > 1
      && fibonacci_(n-1, f1)
      && fibonacci_(n-2, f2)
          ==> fibonacci_(n, f1 + f2)                    // else branch

forall n res. fibonacci_(n, res) ==> res != -1          // postcondition
```

Eldarica:

```
( define-fun fibonacci_ (( A Int ) ( B Int ) ) Bool ( >= B 1) )
```

Stainless Verification Pipeline

Stainless

Scala Compiler

↓

Inlining

↓

Imperative Elim.

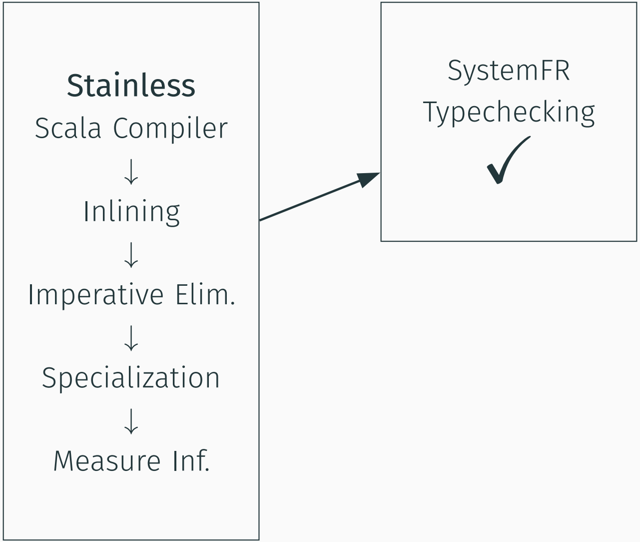
↓

Specialization

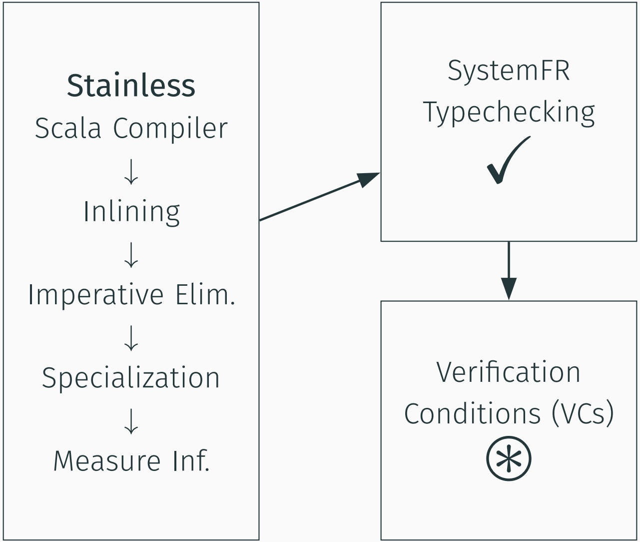
↓

Measure Inf.

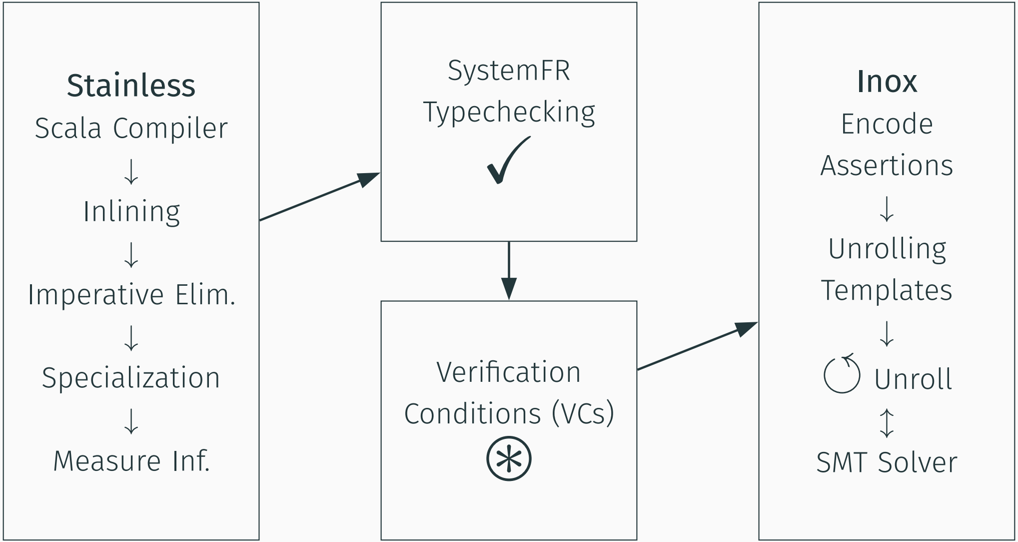
Stainless Verification Pipeline



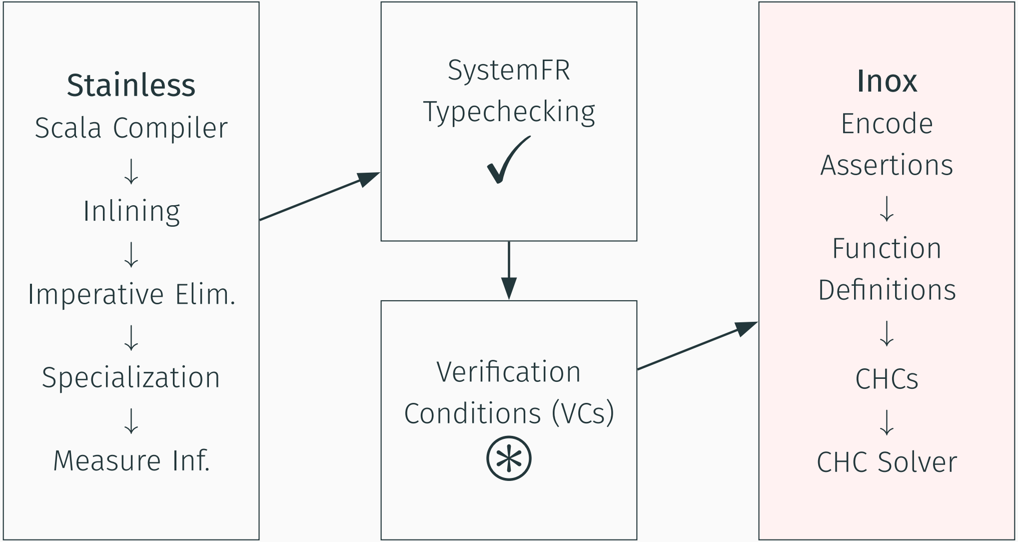
Stainless Verification Pipeline



Stainless Verification Pipeline



Stainless Verification Pipeline



Demo

Demo

- Support for z3/Spacer [3] and Eldarica [4]
- Invariant strengthening for first-order recursive integer programs
- Numerical properties of algebraic data types

```scala
/**
  * Fibonacci, with inductive and non-inductive invariants
  */
def fibonacci(n: BigInt): BigInt = {
  require(n >= 0)
  decreases(n)
  if n < 2 then
    BigInt(1)
  else
    fibonacci(n - 1) + fibonacci(n - 2)
}.ensuring(res => res != -1)
```

cvc5	z3/Spacer	Eldarica
✘	✔	✔

```
/**
  * McCarthy's 91 Function, nested recursive version
  */
def mccarthy91(n: BigInt): BigInt = {
  if n > 100 then
    n - 10
  else
    mccarthy91(mccarthy91(n + 11))
}.ensuring(res => res != -1)
```

cvc5	z3/Spacer	Eldarica
✔	✔	✔

```scala
/**
  * McCarthy's 91 Function, tail recursive version
  *
  * Postcondition on wrapper only
  */
def mccarthy91Tail(n: BigInt): BigInt = {
  mcRec(n, 1)
}.ensuring(res => res != -1)

@tailrec
def mcRec(n: BigInt, c: BigInt): BigInt = {
  if c == 0 then
    n
  else
    if n > 100 then
      mcRec(n - 10, c - 1)
    else
      mcRec(n + 11, c + 1)
}
```

cvc5	z3/Spacer	Eldarica
✘	✔	✔

```scala
def lengthLemma(l: List[BigInt]): Boolean

def heightLemma[A](t: Tree[A]): Boolean

def heightSizeLemma[A](t: Tree[A]): Boolean
```

cvc5	z3/Spacer	Eldarica
✘	✘	✔
✘	✘	✔
✘	✘	✔

```scala
trait Tree[A] {
  def leafSize: BigInt =
    self match
      case Leaf(_) => 1
      case Node(left, right) => left.leafSize + right.leafSize

  def mirrored: Tree[A] =
    self match
      case Leaf(_) => self
      case Node(left, right) => Node(right, left)
}

def leafSizeLemma[A](tree: Tree[A]): Boolean = {
  tree.mirrored.leafSize == tree.leafSize
}.ensuring(res => res)
// would need to learn
// mirrored(input, result) :- input.leafSize == result.leafSize
```

cvc5	z3/Spacer	Eldarica
✔	✘	✘

Constrained Horn Clauses

- Better inductive reasoning
- Much more *predictable* interprocedural analysis
- Better platform for representing and exchanging problems
- *Not* a replacement for the SMT backend

Constrained Horn Clauses

- Better inductive reasoning
- Much more *predictable* interprocedural analysis
- Better platform for representing and exchanging problems
- *Not* a replacement for the SMT backend
- Counterexample-complete, but not strictly more or less expressive

Ongoing work: Dealing with Algebraic Data Types

- Some support from solvers directly
- Recursively defined functions complicate things
- Unrolling is capable of checking properties we cannot check/infer with CHCs/ATPs right now

Conclusion

- Support for using CHC solvers in Stainless
- Invariant strengthening for functional programs
- Ongoing: supporting RDFs over ADTs
- Next:
    - Leveraging existing unrolling + CHCs together
    - Higher-order functions

Conclusion

- Support for using CHC solvers in Stainless
- Invariant strengthening for functional programs
- Ongoing: supporting RDFs over ADTs
- Next:
  - Leveraging existing unrolling + CHCs together
  - Higher-order functions

Thanks! Questions?

References

[1] Mario Bucev and Viktor Kunčak. **“Formally verified quite OK image format.”** In: *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design–FMCAD 2022*. TU Wien. 2022, pp. 343–348.

[2] Samuel Chassot and Viktor Kunčak. **“Verifying a Realistic Mutable Hash Table: Case Study (Short Paper).”** In: *International Joint Conference on Automated Reasoning*. Springer. 2024, pp. 304–314.

[3] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. **“SMT-based model checking for recursive programs.”** In: *Formal Methods in System Design* 48 (2016), pp. 175–205.

[4] Hossein Hojjat and Philipp Rümmer. **“The ELDARICA horn solver.”** In: *2018 Formal Methods in Computer Aided Design (FMCAD)*. IEEE. 2018, pp. 1–7.