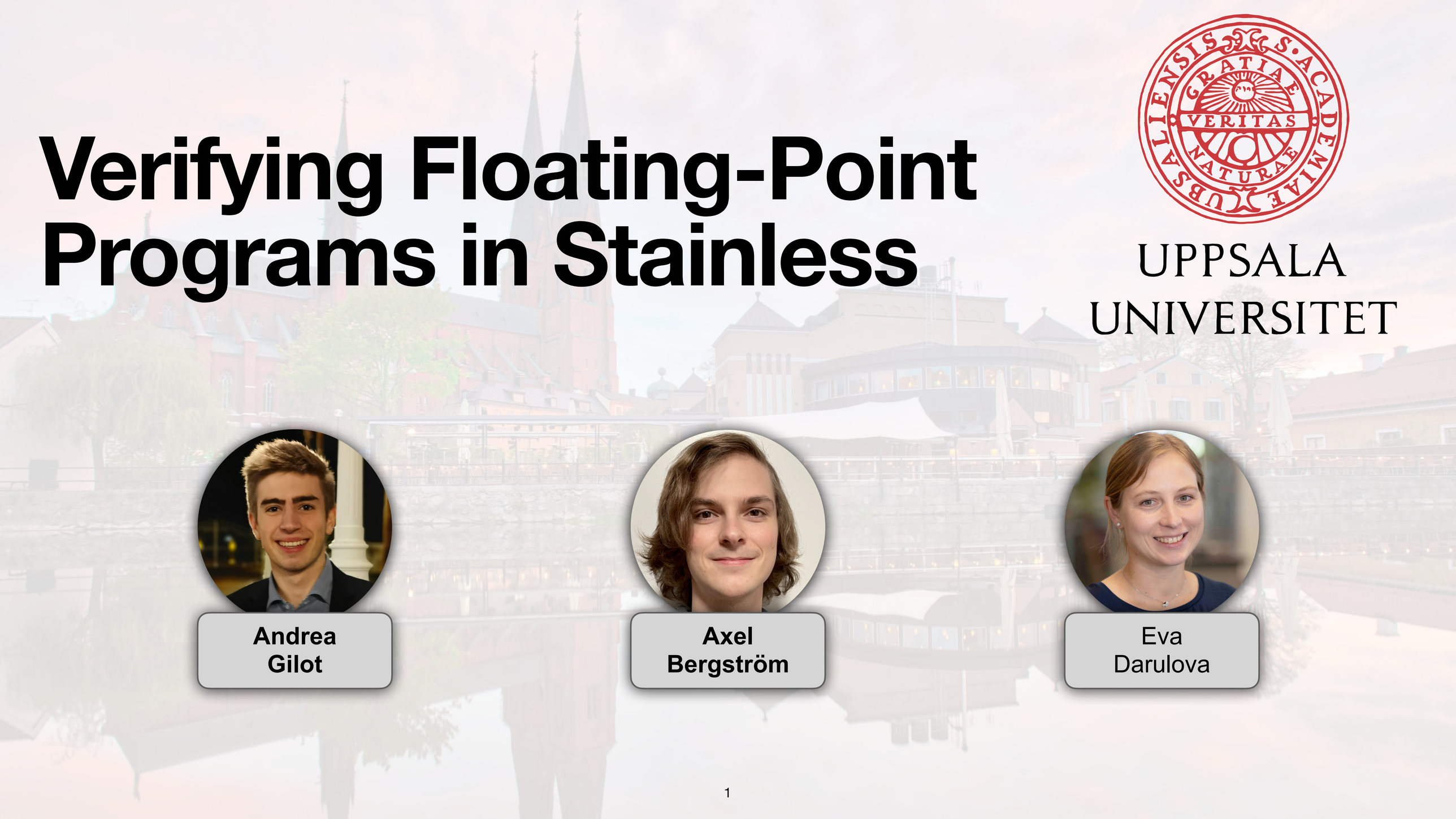
Verifying Floating-Point Programs in Stainless



Andrea Gilot

Axel Bergström

Eva Darulova

Floating-point support in deductive verifiers

Floating-point support in deductive verifiers

		
Language	C	Java
Transcendental function properties	Axiomatised	Axiomatised
Automatic assertions	✅	❌
Evaluation on floating-point user code	Case studies	❌

Floating-point support in deductive verifiers

			**Stainless**
Language	C	Java	Scala (incl. higher-order functions, polymorphism, recursion)
Transcendental function properties	Axiomatised	Axiomatised	
Automatic assertions	✅	❌	
Evaluation on floating-point user code	Case studies	❌	

Floating-point support in deductive verifiers

			**Stainless**
Language	C	Java	Scala (incl. higher-order functions, polymorphism, recursion)
Transcendental function properties	Axiomatised	Axiomatised	Verified
Automatic assertions	✅	❌	
Evaluation on floating-point user code	Case studies	❌	

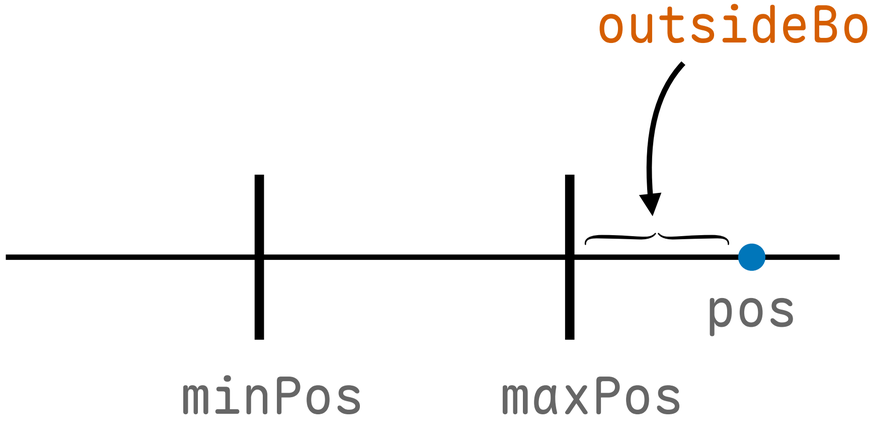
Floating-point support in deductive verifiers

			**Stainless**
Language	C	Java	Scala (incl. higher-order functions, polymorphism, recursion)
Transcendental function properties	Axiomatised	Axiomatised	Verified
Automatic assertions	✅	❌	✅
Evaluation on floating-point user code	Case studies	❌	

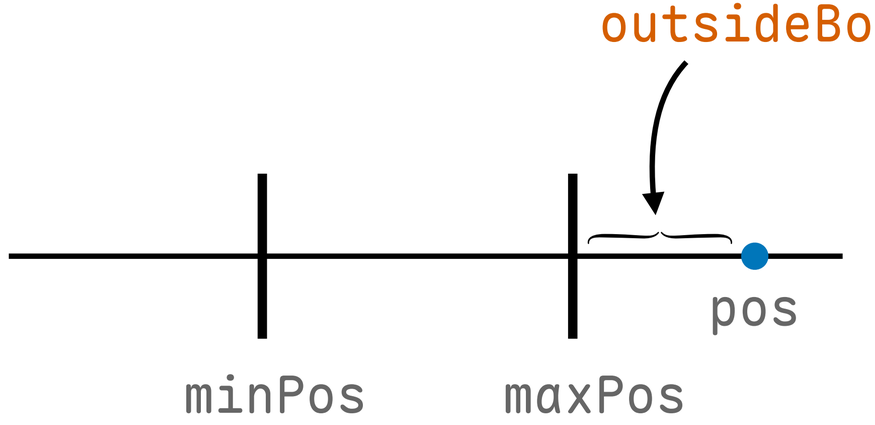
Floating-point support in deductive verifiers

			**Stainless**
Language	C	Java	Scala (incl. higher-order functions, polymorphism, recursion)
Transcendental function properties	Axiomatised	Axiomatised	Verified
Automatic assertions	✅	❌	✅
Evaluation on floating-point user code	Case studies	❌	GitHub mining

```
def outsideBorder(pos: Double, minPos: Double, maxPos: Double): Double =
    if pos < minPos then minPos - pos
    else if pos > maxPos then pos - maxPos
    else 0 // the entity is inside the map edge
```

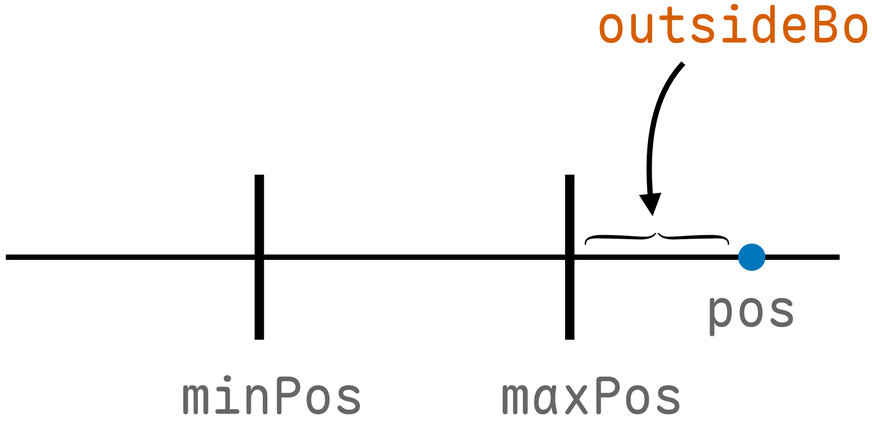



```
def outsideBorder(pos: Double, minPos: Double, maxPos: Double): Double =
    if pos < minPos then minPos - pos
    else if pos > maxPos then pos - maxPos
    else 0 // the entity is inside the map edge
```




```
def outsideBorder(pos: Double, minPos: Double, maxPos: Double): Double =
    if pos < minPos then minPos - pos
    else if pos > maxPos then pos - maxPos
    else 0 // the entity is inside the map edge
```

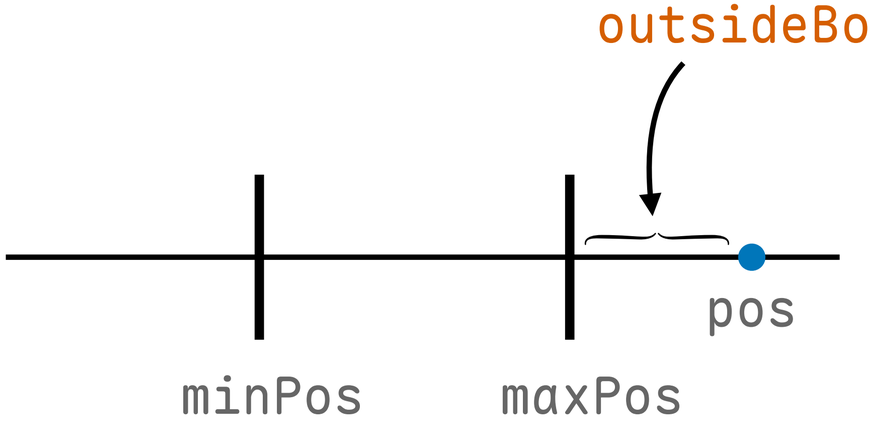
User comment




```
def outsideBorder(pos: Double, minPos: Double, maxPos: Double): Double =
    if pos < minPos then minPos - pos
    else if pos > maxPos then pos - maxPos
    else 0 // the entity is inside the map edge
```

User comment

Comparison with NaN:
position → NaN




```
def outsideBorder(pos: Double, minPos: Double, maxPos: Double): Double =
    if pos < minPos then minPos - pos
    else if pos > maxPos then pos - maxPos
    else 0 // the entity is inside the map edge
```

User comment

NaN takes this path too

Comparison with NaN:
position → NaN

```scala
case class StormDay(moves: Int, errors: Int):
```

Puzzle Storm

Move to start

```
case class StormDay(moves: Int, errors: Int):
```

Puzzle Storm

Move to start

```
case class StormDay(moves: Int, errors: Int):


    def accuracyPercent: Float = {

        100 * (moves - errors) / moves.toFloat

    }
```

Puzzle Storm

Move to start

```
case class StormDay(moves: Int, errors: Int):


    def accuracyPercent: Float = {

        100 * (moves - errors) / moves.toFloat

    } // how to check that it computes a percentage?
```

Puzzle Storm

Move to start

```
case class StormDay(moves: Int, errors: Int):

  require(moves ⩾ errors && errors ⩾ 0 && moves > 0)

  def accuracyPercent: Float = {

    100 * (moves - errors) / moves.toFloat

  }
```

Class invariant

Puzzle Storm

Move to start

```
case class StormDay(moves: Int, errors: Int):

  require(moves ⩾ errors && errors ⩾ 0 && moves > 0)

  def accuracyPercent: Float = {

    100 * (moves - errors) / moves.toFloat

  }.ensuring(result ⇒ 0 ⩽ result && result ⩽ 100)
```

Class invariant

Postcondition

Puzzle Storm

Move to start

```
case class StormDay(moves: Int, errors: Int):

  require(moves ⩾ errors && errors ⩾ 0 && moves > 0)

  def accuracyPercent: Float = {

    100 * (moves - errors) / moves.toFloat

  }.ensuring(result ⇒ 0 ⩽ result && result ⩽ 100)
```

Class invariant

Postcondition

```
Found counter-example:
  StormDay(1073741832, 730144766)
```

Puzzle Storm

Move to start

```
case class StormDay(moves: Int, errors: Int):

  require(moves ⩾ errors && errors ⩾ 0 && moves > 0)

  def accuracyPercent: Float = {

    100 * (moves - errors) / moves.toFloat

  }.ensuring(result ⇒ 0 ⩽ result && result ⩽ 100)
```

Class invariant

Postcondition

```
Found counter-example:
   StormDay(1073741832, 730144766)
    → -2.9586256E-5
```

Puzzle Storm

Move to start

```
case class StormDay(moves: Int, errors: Int):

  require(moves ⩾ errors && errors ⩾ 0 && moves > 0)

  def accuracyPercent: Float = {

    100 *((moves - errors) / moves.toFloat)

  }.ensuring(result ⇒ 0 ⩽ result && result ⩽ 100)
```

Class invariant

Postcondition

```
Found counter-example:
   StormDay(1073741832, 730144766)
    → -2.9586256E-5
```

Puzzle Storm

Move to start

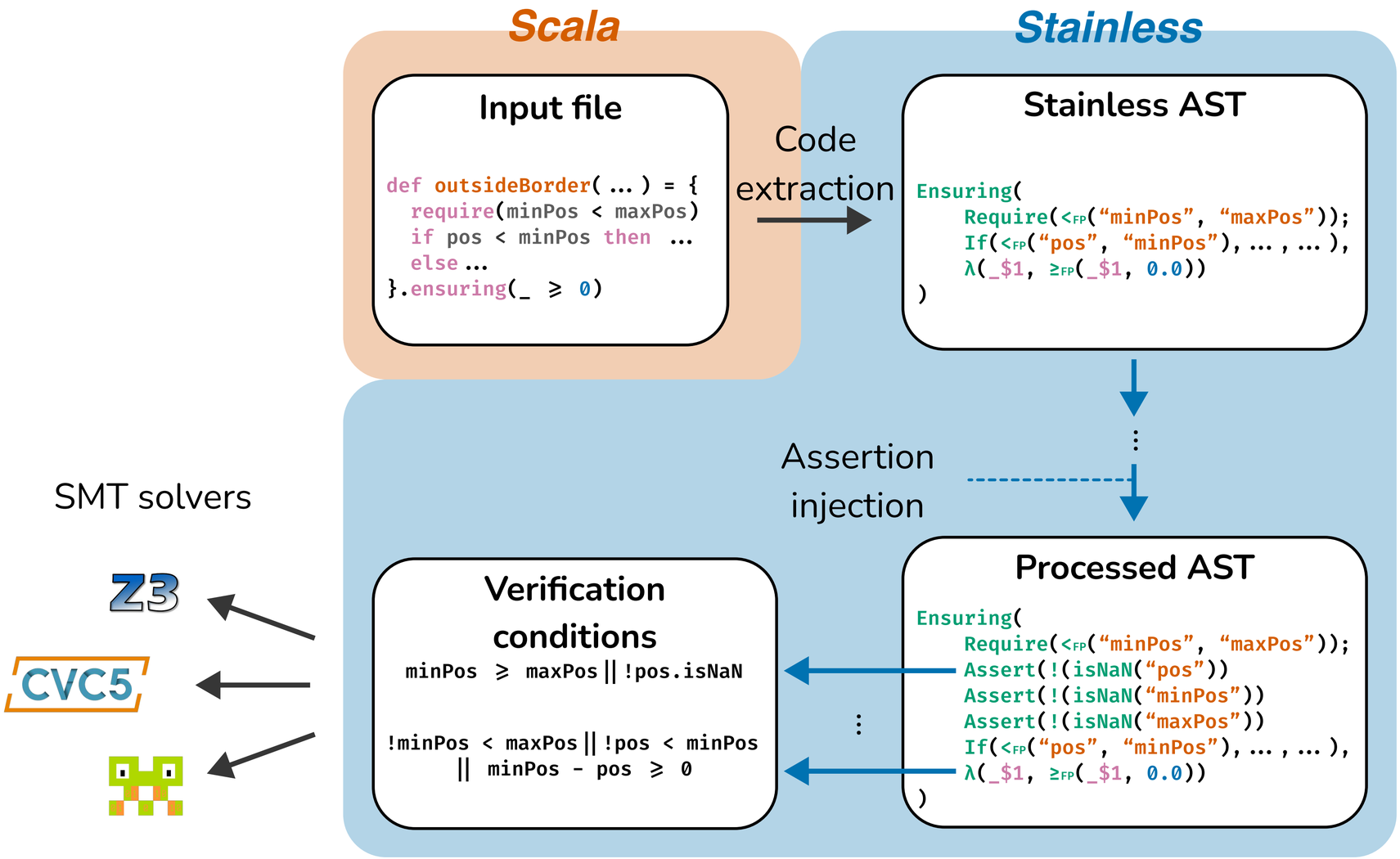
```
case class StormDay(moves: Int, errors: Int):

  require(moves ⩾ errors && errors ⩾ 0 && moves > 0)

  def accuracyPercent: Float = {

    100 *((moves - errors) / moves.toFloat)

  }.ensuring(result ⇒ 0 ⩽ result && result ⩽ 100)
```

Class invariant

Postcondition

```
accuracyPercent  postcondition  valid
```

How does it work?



Scala

Input file

```
def outsideBorder( ... ) = {
  require(minPos < maxPos)
  if pos < minPos then  ...
  else ...
}.ensuring(_ ⩾ 0)
```

Code extraction

Scala

Input file

```
def outsideBorder( ... ) = {
  require(minPos < maxPos)
  if pos < minPos then  ...
  else ...
}.ensuring(_ ⩾ 0)
```

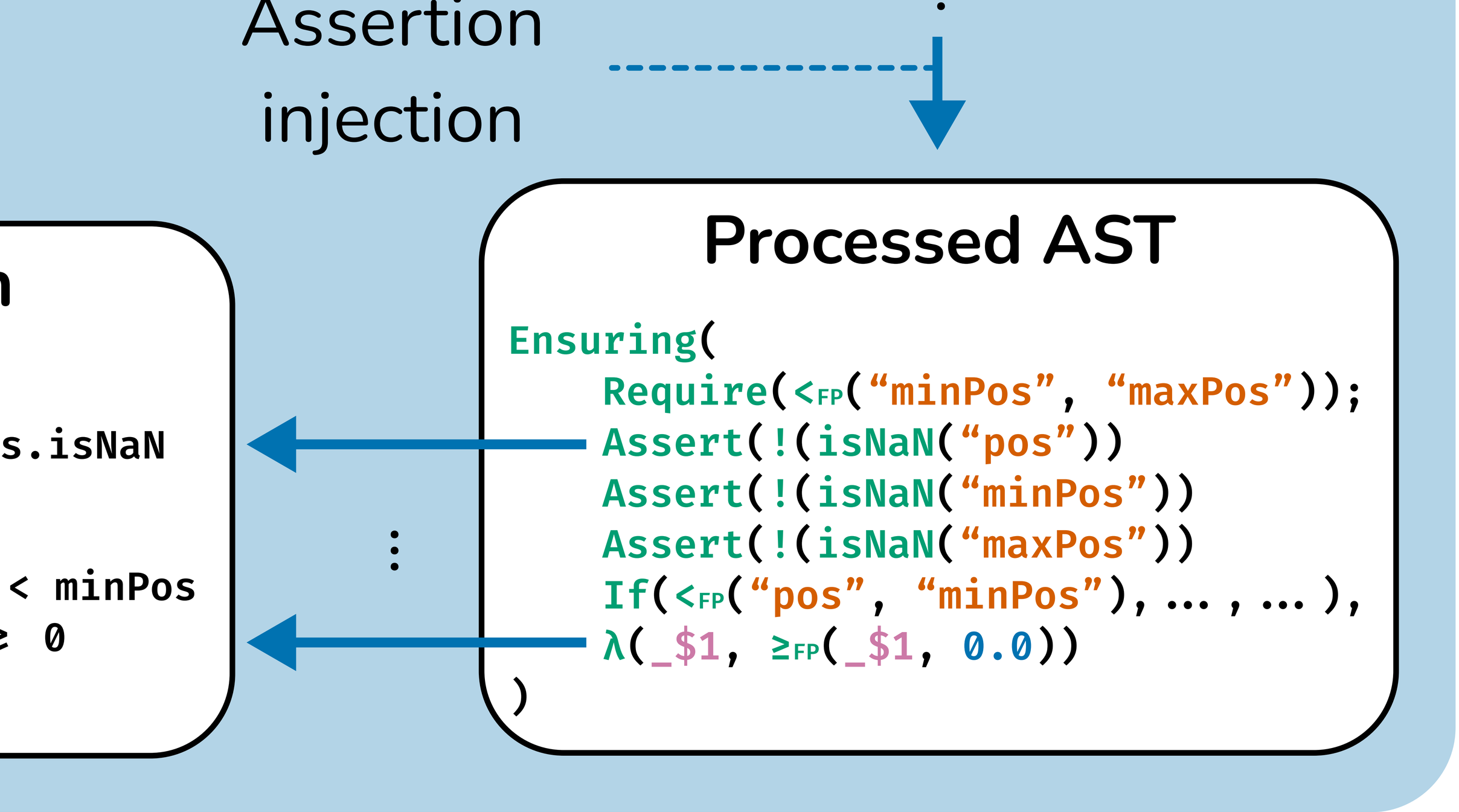
Code extraction

Stainless

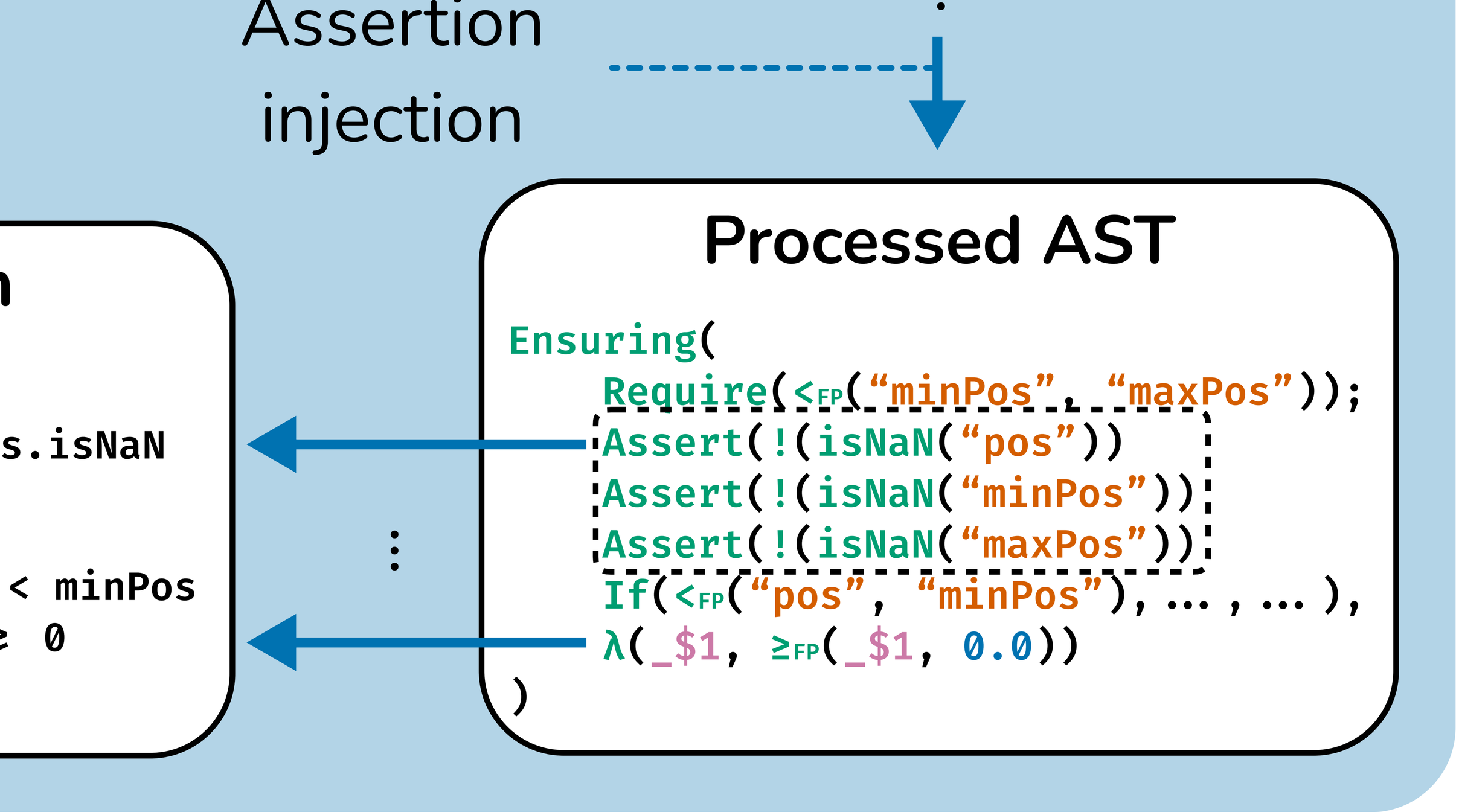
Stainless AST

```
Ensuring(
    Require(<FP(“minPos”, “maxPos”));
    If(<FP(“pos”, “minPos”), ... , ... ),
    λ(_$1, ⩾FP(_$1, 0.0))
)
```

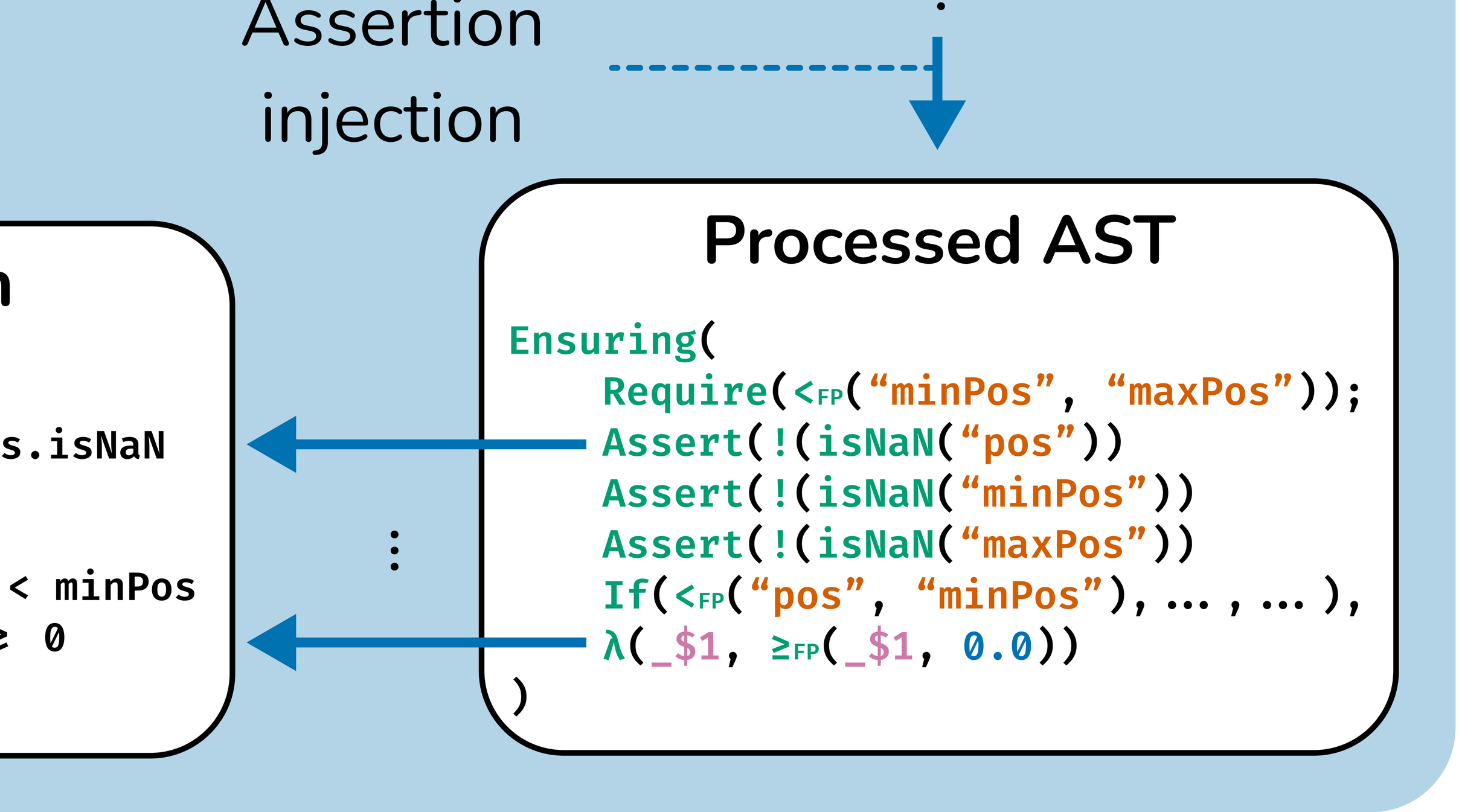
Assertion injection

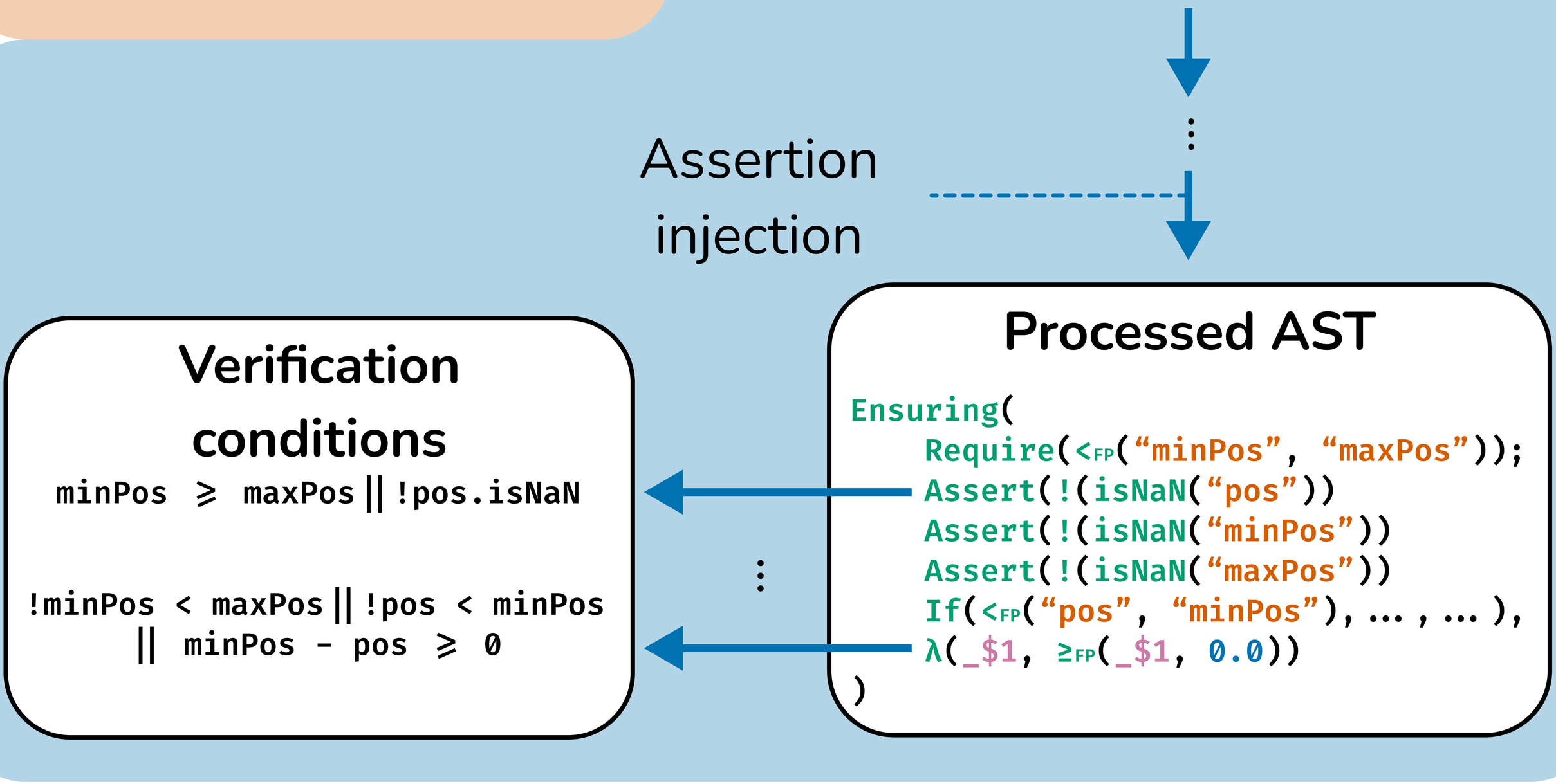


Assertion injection

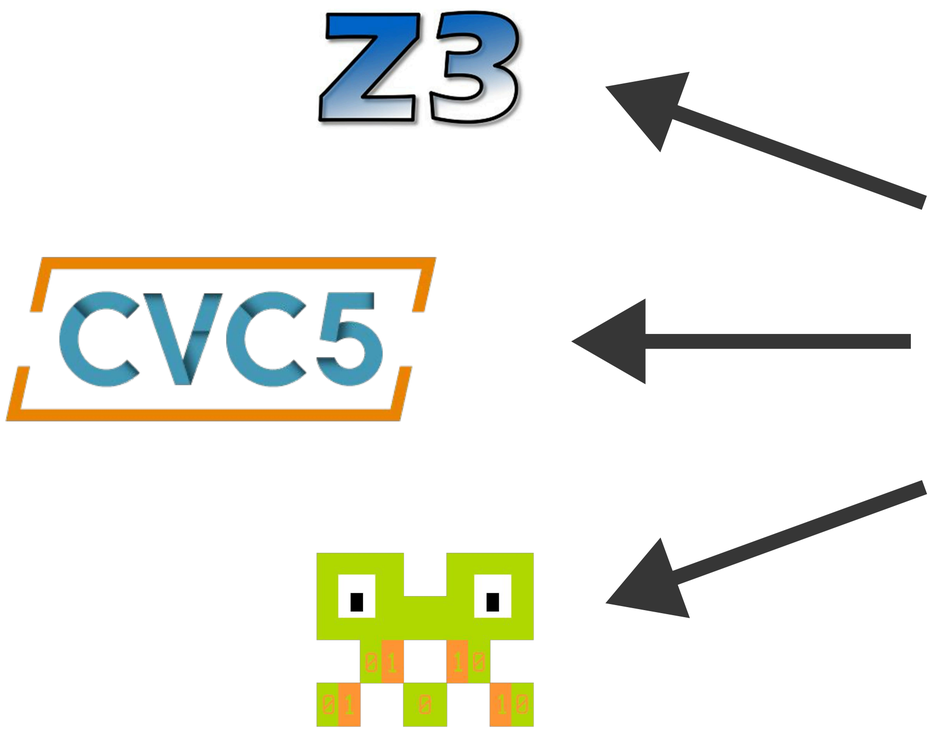


Assertion injection




Assertion injection
Processed AST
Ensuring(
Require(<FP("minPos", "maxPos"));
Assert(!(isNaN("pos"))
Assert(!(isNaN("minPos"))
Assert(!(isNaN("maxPos"))
If(<FP("pos", "minPos"), ... , ...),
λ(_$1, ≥FP(_$1, 0.0))
)
Verification conditions
minPos ⩾ maxPos || !pos.isNaN
!minPos < maxPos || !pos < minPos
|| minPos - pos ⩾ 0

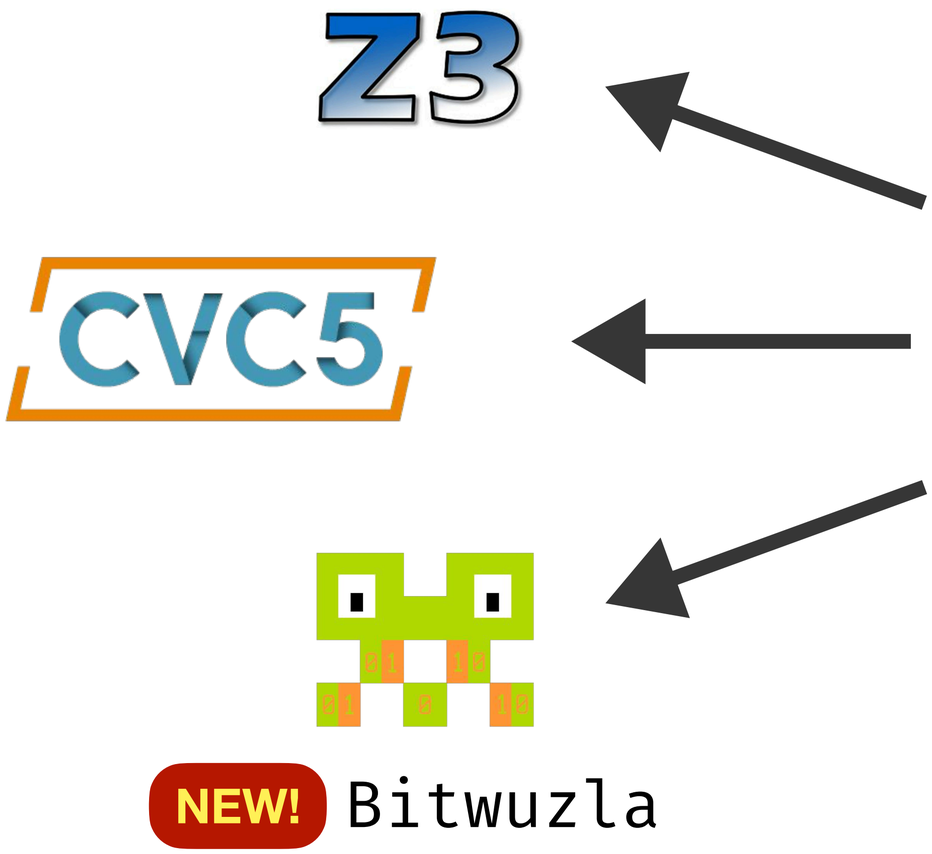
SMT solvers



Verification conditions

```
minPos ⩾ maxPos || !pos.isNaN

!minPos < maxPos || !pos < minPos
    || minPos - pos ⩾ 0
```

SMT solvers



Verification conditions

```
minPos ⩾ maxPos || !pos.isNaN

!minPos < maxPos || !pos < minPos
    || minPos - pos ⩾ 0
```

Assertion injection for floating-point programs

Assertion injection for floating-point programs

```
def outsideBorder(pos: Double, minPos: Double, maxPos: Double): Double =
   if pos < minPos then minPos - pos
   else if pos > maxPos then pos - maxPos
   else 0 // the entity is inside the map edge
```

Assertion injection for floating-point programs

```
def outsideBorder(pos: Double, minPos: Double, maxPos: Double): Double =
  if pos < minPos then minPos - pos
  else if pos > maxPos then pos - maxPos
  else 0 // the entity is inside the map edge
```

Stainless automatically detects potential comparisons with NaN

Assertion injection for floating-point programs

```
def outsideBorder(pos: Double, minPos: Double, maxPos: Double): Double =
  if pos < minPos then minPos - pos
  else if pos > maxPos then pos - maxPos
  else 0 // the entity is inside the map edge
```

Stainless automatically detects potential comparisons with NaN

Also detects unsafe typecasts to integers

Case Study: a verified math library

Case Study: a verified math library

- Problem: transcendental functions (exp, cos, pow, …) not supported by solvers

Case Study: a verified math library

- Problem: transcendental functions (exp, cos, pow, …) not supported by solvers
- KeY’s solution: function lemmas (sound, not complete)

Case Study: a verified math library

- Problem: transcendental functions (exp, cos, pow, …) not supported by solvers
- KeY's solution: function lemmas (sound, not complete)
- In Stainless:

```
object Sin {
  def compute(x: Double): Double = {
    … // ~1000 lines of code and specification
  }.ensuring(res =>
    (!x.isFinite == res.isNaN)
    && (x.equiv(+0.0d) ==> res.equiv(+0.0d))
    && (x.equiv(-0.0d) ==> res.equiv(-0.0d))
    && (res.isNaN || -1.0d <= res && res <= 1.0d)
  )
}
```

Case Study: a verified math library

- Problem: transcendental functions (exp, cos, pow, …) not supported by solvers
- KeY's solution: function lemmas (sound, not complete)
- In Stainless:

```
object Sin {
  def compute(x: Double): Double = {
    … // ~1000 lines of code and specification
  }.ensuring(res =>
    (!x.isFinite == res.isNaN)
    && (x.equiv(+0.0d) ==> res.equiv(+0.0d))
    && (x.equiv(-0.0d) ==> res.equiv(-0.0d))
    && (res.isNaN || -1.0d <= res && res <= 1.0d)
  )
}
```

- KeY *axiomatizes* the lemmas; we *prove* them in Stainless

Case Study: a verified math library

Case Study: a verified math library

- We add a math library based OpenJDK/FdLibm

Case Study: a verified math library

- We add a math library based OpenJDK/FdLibm
- In total: 84 lemmas for 18 functions

Case Study: a verified math library

- We add a math library based OpenJDK/FdLibm
- In total: 84 lemmas for 18 functions
- Straightforward for 15/18 functions
  - by re-writing in a functional style + using Bitwuzla

Case Study: a verified math library

- We add a math library based OpenJDK/FdLibm
- In total: 84 lemmas for 18 functions
- Straightforward for 15/18 functions
  - by re-writing in a functional style + using Bitwuzla
- sin(), cos(), and tan() are more complicated
  - multiple modifications required

Evaluation: benchmark sets

Evaluation: benchmark sets

`cos(x)`

`exp(x)` `pow(x, y)`

FdLibm
implementations

18 benchmarks / 256 VCs

Evaluation: benchmark sets

```
cos(x)
exp(x) pow(x, y)
```

FdLibm
implementations

18 benchmarks / 256 VCs

KeY benchmarks
(adapted from Java)

8 benchmarks / 176 VCs

Evaluation: benchmark sets

```
cos(x)
exp(x) pow(x, y)
```

FdLibm
implementations

18 benchmarks / 256 VCs

KeY benchmarks
(adapted from Java)

8 benchmarks / 176 VCs

GitHub mined
user functions

79 benchmarks / 1600 VCs

Evaluation: benchmark sets

```
cos(x)
exp(x) pow(x, y)
```

FdLibm
implementations

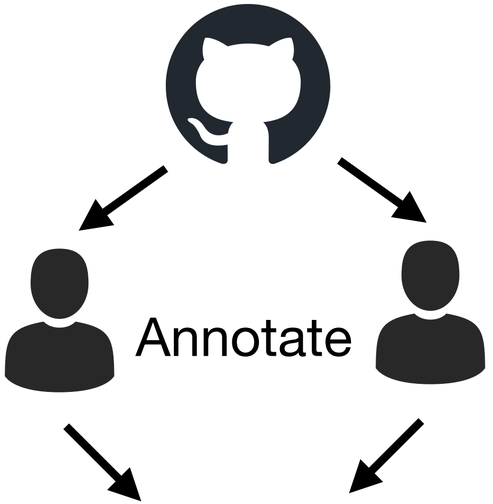
↓

18 benchmarks / 256 VCs

KeY benchmarks
(adapted from Java)

↓

8 benchmarks / 176 VCs



Common
specification

↓

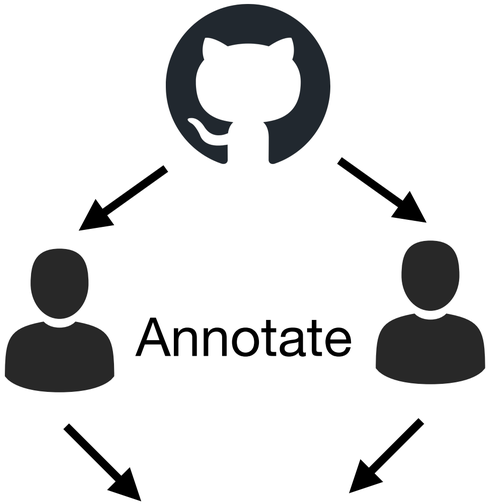
79 benchmarks / 1600 VCs

Evaluation: benchmark sets

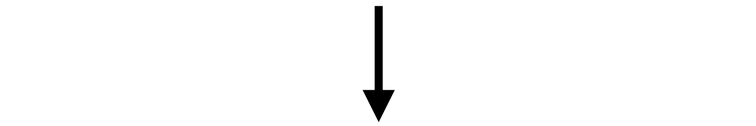
```
cos(x)
exp(x) pow(x, y)
```

FdLibm implementations

KeY benchmarks (adapted from Java)



Common specification



18 benchmarks / 256 VCs

8 benchmarks / 176 VCs

79 benchmarks / 1600 VCs

Using a solver portfolio, Stainless solves:

18/18 benchmarks	8/8 benchmarks	73/79 benchmarks

Evaluation: SMT-solver comparison

- Using one solver at a time:

benchmark set	#VCs	cvc5	Z3	Bitwuzla	Any solver
KeY	176	100%	85.2%	67.0%	100%
FdLibm	256	88.7%	78.1%	84.8%	99.2%
User Functions	1600	99.7%	95.5%	30.6%	99.8%

Evaluation: SMT-solver comparison

- Using one solver at a time:

benchmark set	#VCs	cvc5	Z3	Bitwuzla	Any solver
KeY	176	100%	85.2%	67.0%	100%
FdLibm	256	88.7%	78.1%	84.8%	99.2%
User Functions	1600	99.7%	95.5%	30.6%	99.8%

Bitwuzla limited by theory support, not performance

Evaluation: SMT-solver comparison

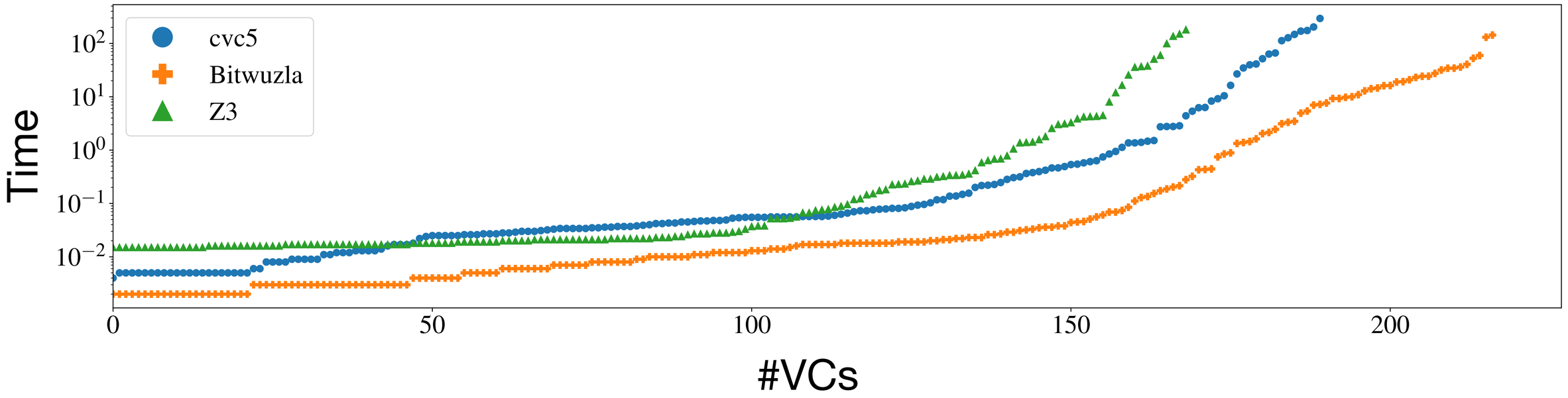
- Using one solver at a time:

benchmark set	#VCs	cvc5	Z3	Bitwuzla	Any solver
KeY	176	100%	85.2%	67.0%	100%
FdLibm	256	88.7%	78.1%	84.8%	99.2%
User Functions	1600	99.7%	95.5%	30.6%	99.8%

Bitwuzla limited by theory support, not performance

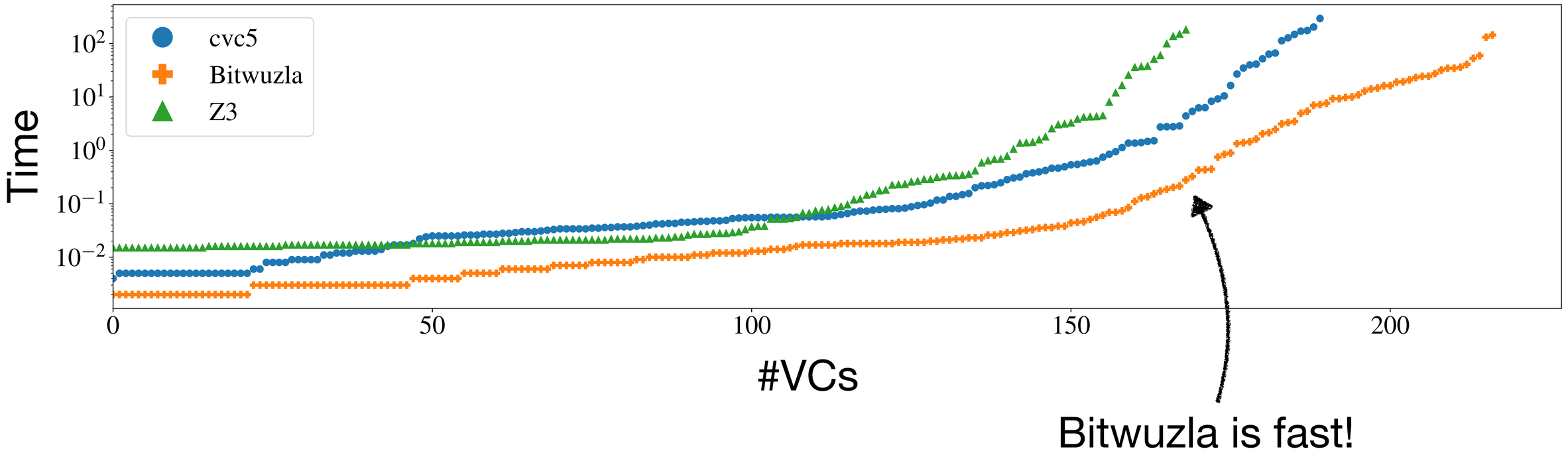
Evaluation: SMT-solver comparison

- FdLibm solving time, VCs supported by all solvers:



Evaluation: SMT-solver comparison

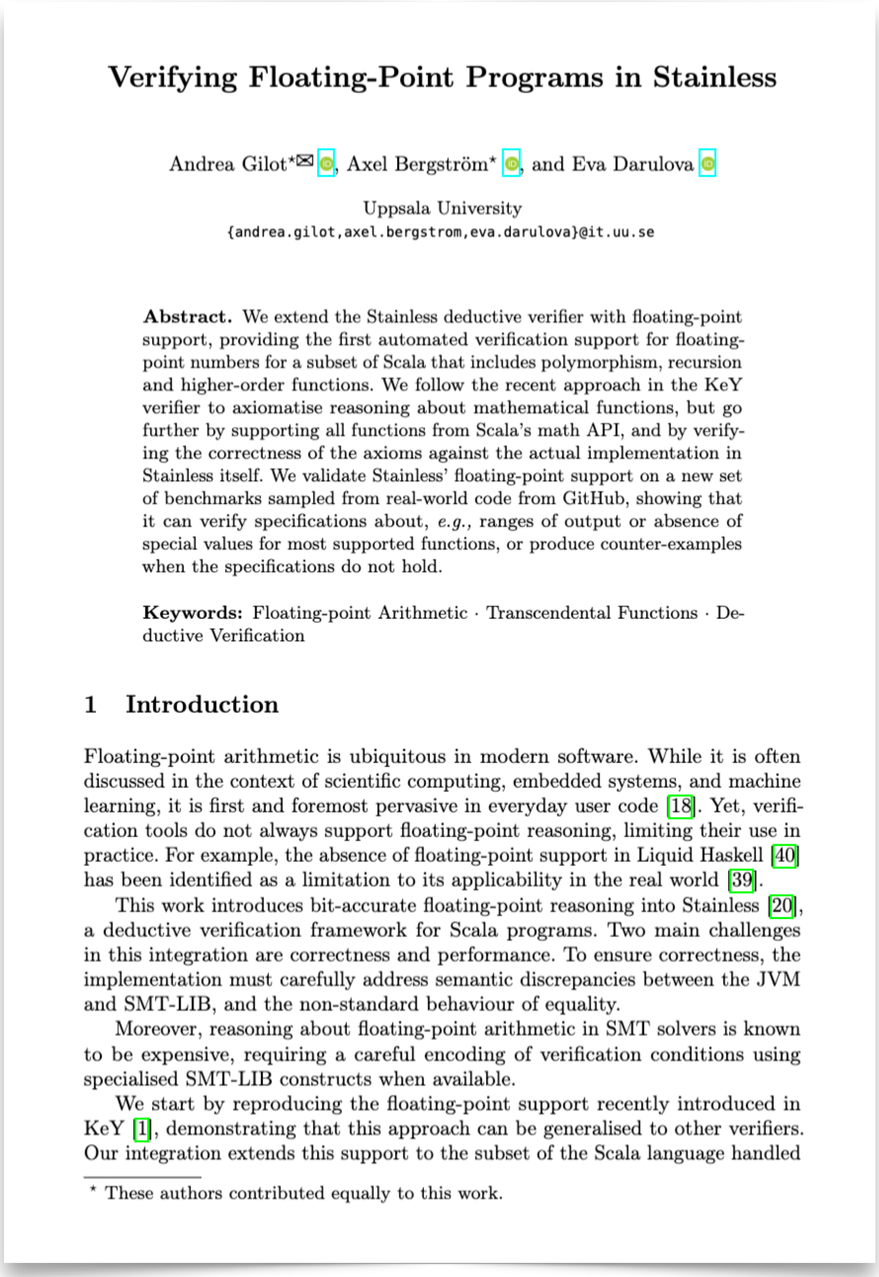
- FdLibm solving time, VCs supported by all solvers:



Also in the paper…

More details about:

- the case study
- reconciling JVM vs SMT-LIB semantics
- keeping polymorphism sound
- mining and annotating GitHub functions
- the evaluation results



Verifying Floating-Point Programs in Stainless

Andrea Gilot*✉, Axel Bergström*, and Eva Darulova

Uppsala University
{andrea.gilot,axel.bergstrom,eva.darulova}@it.uu.se

Abstract. We extend the Stainless deductive verifier with floating-point support, providing the first automated verification support for floating-point numbers for a subset of Scala that includes polymorphism, recursion and higher-order functions. We follow the recent approach in the KeY verifier to axiomatise reasoning about mathematical functions, but go further by supporting all functions from Scala's math API, and by verifying the correctness of the axioms against the actual implementation in Stainless itself. We validate Stainless' floating-point support on a new set of benchmarks sampled from real-world code from GitHub, showing that it can verify specifications about, *e.g.,* ranges of output or absence of special values for most supported functions, or produce counter-examples when the specifications do not hold.

Keywords: Floating-point Arithmetic · Transcendental Functions · Deductive Verification

1 Introduction

Floating-point arithmetic is ubiquitous in modern software. While it is often discussed in the context of scientific computing, embedded systems, and machine learning, it is first and foremost pervasive in everyday user code [18]. Yet, verification tools do not always support floating-point reasoning, limiting their use in practice. For example, the absence of floating-point support in Liquid Haskell [40] has been identified as a limitation to its applicability in the real world [39].

This work introduces bit-accurate floating-point reasoning into Stainless [20], a deductive verification framework for Scala programs. Two main challenges in this integration are correctness and performance. To ensure correctness, the implementation must carefully address semantic discrepancies between the JVM and SMT-LIB, and the non-standard behaviour of equality.

Moreover, reasoning about floating-point arithmetic in SMT solvers is known to be expensive, requiring a careful encoding of verification conditions using specialised SMT-LIB constructs when available.

We start by reproducing the floating-point support recently introduced in KeY [1], demonstrating that this approach can be generalised to other verifiers. Our integration extends this support to the subset of the Scala language handled

* These authors contributed equally to this work.