# Experiments in Program Verification with Stainless

Romain Ruetschi

(Draft)
January 2018

**Abstract**

(TODO: Abstract)

Master Thesis Project under the supervision of
Prof. Viktor Kuncak & Dr. Jad Hamza
Lab for Automated Reasoning and Analysis LARA - EPFL

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

# Contents

## 1 Introduction

(TODO: Introduction)

# 2 Program Verification with Stainless

(TODO: Describe Stainless/Inox pipeline)

# 3 Symbolic Evaluation of Inox Programs

## 3.1 Motivation

Consider the following program, which inserts a known list of key-value pairs in a map that is kept abstract:

```scala
def foldLeft[A, B](list: List[A], z: B)(f: (B, A) => B): B = list match {
  case Nil() => z
  case Cons(x, xs) => foldLeft(xs, f(z, x))(f)
}

def insert[A, B](kvs: List[(A, B)], map: Map[A, B]) = {
  foldLeft(kvs, map) {
    case (acc, (k, v)) => acc.updated(k, v)
  }
}

def test(map: Map[String, Int]): Boolean = {
  val xs = List("a" -> 1, "b" -> 2, "c" -> 3, "d" -> 4)
  val res = insert(xs, map)
  res("b") == 2
}.holds
```

For the sake of simplicity, let us assume that Stainless is unable to automatically prove the verification condition corresponding to `test`'s postcondition (we will look at an actual test case in Section 3.5).

So, how would one go about to help Stainless prove this program correct? We could for example break up the problem, by adding appropriate postconditions to both `foldLeft` and `insert`, and provide the necessary lemmas, assertions, or annotations in order to get these to verify.

While this is a reasonable way of going at the problem, we feel that for this specific program, there is another way of tackling the problem. Indeed, while the `map` variable is kept abstract, the list of values `xs` we want to insert is not, and it should thus be possible to simplify the initial program by unfolding some definitions, and simplifying the resulting expressions, yielding a much simpler verification condition.

If we manually unfold `insert` once, then unfold `foldLeft` "as much as possible" in the program above, we end up with the following definition for the `test` function, which is both much smaller and simpler than the original definition. We could then expect Stainless to have an easier time proving this theorem.

```scala
def test(map: Map[String, Int]): Boolean = {
  val res = map.updated("a", 1).updated("b", 2).updated("c", 3).updated("d", 4)
  res("b") == 2
}.holds
```

We believe that it is unreasonable to expect users to manually perform such a transformation, especially with large programs. We have hence implemented an automatic transformation as part of the verification pipeline, by extending the current simplification procedure found in Inox.

## 3.2 Symbolic Evaluation

The process of executing a program in the presence of abstract inputs is known as *symbolic evaluation*. The idea is to execute the program over symbolic inputs values while maintaining a *path condition*: a quantifier-free formula in conjunctive normal form that represents constraints over the symbolic variables we have encountered so far. The path condition encodes the knowledge we have gathered so far about the symbolic values we are dealing with, as well as the bindings in scope. At each evaluation step, the evaluator can query the path condition $p$ in order to hopefully determine to which of its branches a conditional expression `if (`$c$`)` $t$ `else` $e$ should be reduced. We start by symbolically evaluating $c$ under the current path condition, the result of which we will denote by $c'$. If $c'$ is a boolean literal, we then evaluate the appropriate branch, and continue from here. Otherwise, we add $c'$ as a conjunct to the path condition $p$, and check whether the resulting formula $p \wedge c'$ is satisfiable. If we are able to show that it is indeed satisfiable, we evaluate the "then" branch of the conditional expression under new path condition $p \wedge c'$. Otherwise, we instead check whether the formula $p \wedge \neg c'$ is satisfiable, and if that is the case we evaluate the "else" branch under the new path condition $p \wedge \neg c'$. If we are unable to show that either formula is satisfiable, we evaluate both branches under their respective path condition, the result of which we will denote by $t'$ and $e'$, respectively, and we return $if(c')t'elsee;$.

Listing 1 shows a simple program annotated with the current path condition are various points of execution. When evaluating such a program symbolically, the path condition enables the evaluator to pick the "else" branch of the "if-then-else" expression, because, under the path condition at step 2, the condition `x > 0` evaluates to `false`.

### 3.2.1 Example

Here is how the program we introduced in Section 3.1 is expressed in Inox's input language, formatted in PureScala syntax:

```
def foldLeft[A](list: List[A], z: B, f: (B, A) => B): B =
  if (list.isInstanceOf[Nil[A]]) {
    z
  } else {
    val x  = list.asInstanceOf[Cons[A]].head
    val xs = list.asInstanceOf[Cons[A]].tail
    foldLeft(xs, f(z, x), f)
  }
}

def insert[A, B](values: List[(A, B)], map: Map[A, B]) = {
  foldLeft(xs, map, (acc, kv) => acc.updated(kv._1, kv._2))
```

```
// Expression before evaluation:
{
  // 1. PC = true
  assume(x <= 0)
  // 2. PC = x <= 0
  if (x > 0) {
    // 3. PC = x <= 0 && x > 0
    x
  } else {
    // 4. PC = x <= 0 && !(x > 0)
    -x
  }
}

// Resulting expression after evaluation:
{
  assume(x <= 0)
  bar(-x)
}
```

Listing 1: Example program annotated with path condition

```
}

def test(map: Map[String, Int]): Boolean = {
  val xs = List("a" -> 1, "b" -> 2, "c" -> 3, "d" -> 4)
  val res = insert(xs, map)
  res("b") == 2
}.holds
```

Let's only focus on the first two lines of the `test` function, and go through the steps followed by the symbolic evaluator when ran on it.

```
val xs = List("a" -> 1, "b" -> 2, "c" -> 3, "d" -> 4)
val res = insert(xs, map)
```

1. We start by substituting the first binding into the subsequent statement:

   ```
   val res = insert(
     Cons("a" -> 1, Cons("b" -> 2, Cons("c" -> 3, Cons("d" -> 4, Nil()),
     map
   )
   ```

2. Because `insert` is non-recursive function, we can unfold it straight away:

```
val res = foldLeft(
  Cons("a" -> 1, Cons("b" -> 2, Cons("c" -> 3, Cons("d" -> 4, Nil()),
  map,
  (acc, kv) => acc.updated(kv._1, kv._2)
)
```

3. foldLeft is a recursive function, and some care is thus needed in order to not end up unfolding it indefinitely.

   The idea here is that a recursive function must contain a *guarded* recursive call to be terminating, as it would otherwise call itself recursively indefinitely and thus not terminate (we discuss termination in more detail in Section **??**). We can thus look at the function's body, and see whether it contains a conditional expression whose condition can be reduced to a boolean literal under the current path condition. If that is not the case, we cannot safely unfold the function and we stop here. Otherwise, we can replace the conditional expression by its appropriate branch, and evaluate the resulting expression under the new path condition induced by the condition. As long as the function is terminating, this process will terminate as we are either going to end up with a branch that does not contain a recursive call, or we will not be able to evaluate the condition to a boolean value and will thus not proceed further.

   In the case of `foldLeft`, there is indeed such a conditional expression, and under the current path condition, we are able to evaluate `list.isInstanceOf[Nil]` to `false` given that the path condition contains `list = Cons("a" -> 1, ...)`. We can thus unfold it, replace the "if-then-else" expression by its "else" branch, and continue:

```
val res = {
  val x  = Cons("a" -> 1, Cons("b" -> 2, ...)).asInstanceOf[Cons[A]].head
  val xs = Cons("a" -> 1, Cons("b" -> 2, ...)).asInstanceOf[Cons[A]].tail
  foldLeft(xs, ((acc, kv) => ...)(map, x), (acc, kv) => ...)
}
```

4. Here we can evaluate the bound value of `x` (note: we evaluate both bindings in one go here for brevity):

```
val res = {
  val x  = "a" -> 1
  val xs = Cons("b" -> 2, ...)
  foldLeft(xs, ((acc, kv) => ...)(map, x), (acc, kv) => ...)
}
```

5. We now inline both bindings (again, in one go for the sake of brevity):

```
val res = {
  foldLeft(
    Cons("b" -> 2, ...),
    ((acc, kv) => ...)(map, "a" -> 1),
    (acc, kv) => ...
  )
}
```

6. We can then evaluate the arguments supplied to `foldLeft`, giving us:

```
val res = {
  foldLeft(
    Cons("b" -> 2, ...),
    map.updated("a", 1),
    (acc, kv) => ...
  )
}
```

7. As we have encountered another call to a recursive function, we repeat steps 3 to 6 three times, at which point we are left with:

```
val res = {
  foldLeft(
    Nil(),
    map.updated("a", 1).updated("b", 2).updated("c", 3).updated("d", 4),
    (acc, kv) => ...
  )
}
```

8. This time, in step 3, the condition evaluates to `true`, and we thus pick the "then" branch which leaves us with the following expression:

```
val res = map.updated("a", 1).updated("b", 2).updated("c", 3).updated("d", 4)
```

9. This expression cannot be reduced further, as `map` is abstract and `updated` is a special language construct which can only be evaluated when applied to a concrete, finite map. We thus stop here.

This whole process leaves us with the very same definition we would have obtained by equational reasoning, which is admittedly much simpler, and easier to reason about for Inox.

```
def test(map: Map[String, Int]): Boolean = {
  val res = map.updated("a", 1).updated("b", 2).updated("c", 3).updated("d", 4)
  res("b") == 2
}.holds
```

## 3.3 Semantics

We subsequently denote the path condition by $\Delta$, a formula $e$ that is satisfiable under $\Delta$ by $e \in \Delta$. and a variable $v$ that is bound to an expression $e$ under $\Delta$ by $v \mapsto_{\Delta} e$. We will not delve into the details of how to decide whether a given condition is satisfiable under some path condition, but a reasonable although much too simplistic approximation would be to maintain a set of expressions and simply testing for membership. The actual implementation found in Inox (which predates our work) is much more complex and powerful implementation of this very idea. Moreover, we denote the fact that an expression $e$ evaluates to an expression $e'$ under a path condition $\Delta$ by $[\![\, e\,;\, \Delta\,]\!] \longrightarrow e'$. Given $c, v$ two expressions, and `x` a variable, we note $\Delta \cup c$ the addition of $c$ as a conjunct to the path condition $\Delta$, and $\Delta \uplus x \mapsto v$ the addition of the binding `val x =` $v$. At last, the fact that an expression $e$ can be simplified to a boolean value under a path condition $\Delta$ is captured by `e` $\Downarrow \Delta$. Figure 1 shows the operation semantics followed by our implementation, expressed in those terms.

$$\frac{v \mapsto_{\Delta} e}{\llbracket v\,;\,\Delta \rrbracket \longrightarrow e} \tag{1}$$

$$\frac{e \in \Delta}{\llbracket e\,;\,\Delta \rrbracket \longrightarrow \texttt{true}} \tag{2}$$

$$\frac{\neg\, e \in \Delta}{\llbracket e\,;\,\Delta \rrbracket \longrightarrow \texttt{false}} \tag{3}$$

$$\frac{\llbracket e\,;\,\Delta \rrbracket \longrightarrow e'}{\llbracket \lambda x_1,\ldots,x_n.\, e\,;\,\Delta \rrbracket \longrightarrow \lambda x_1,\ldots,x_n.\, e'} \tag{4}$$

$$\frac{\llbracket \neg\, l \;\texttt{||}\; r\,;\,\Delta \rrbracket \longrightarrow e'}{\llbracket l \;\texttt{==>}\; r\,;\,\Delta \rrbracket \longrightarrow e'} \tag{5}$$

$$\frac{\llbracket e_1\,;\,\Delta \rrbracket \longrightarrow \texttt{false}}{\llbracket e_1 \;\texttt{\&\&}\ldots\texttt{\&\&}\; e_n\,;\,\Delta \rrbracket \longrightarrow \texttt{false}} \tag{6}$$

$$\frac{\llbracket e_1\,;\,\Delta \rrbracket \longrightarrow e_1' \qquad \llbracket e_2 \;\texttt{\&\&}\ldots\texttt{\&\&}\; e_n\,;\,\Delta \rrbracket \longrightarrow e'}{\llbracket e_1 \;\texttt{\&\&}\ldots\texttt{\&\&}\; e_n\,;\,\Delta \rrbracket \longrightarrow e_1' \;\texttt{\&\&}\; e'} \tag{7}$$

$$\frac{\llbracket e_1\,;\,\Delta \rrbracket \longrightarrow \texttt{true}}{\llbracket e_1 \;\texttt{||}\ldots\texttt{||}\; e_n\,;\,\Delta \rrbracket \longrightarrow \texttt{true}} \tag{8}$$

$$\frac{\llbracket e_1\,;\,\Delta \rrbracket \longrightarrow e_1' \qquad \llbracket e_2 \;\texttt{||,}\ldots\texttt{||}\; e_n\,;\,\Delta \rrbracket \longrightarrow e'}{\llbracket e_1 \;\texttt{||,}\ldots\texttt{||}\; e_n\,;\,\Delta \rrbracket \longrightarrow e_1' \;\texttt{||}\; e'} \tag{9}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow \texttt{true}}{\llbracket \texttt{if}\ (c)\ t\ \texttt{else}\ e\,;\,\Delta \rrbracket \longrightarrow \llbracket t\,;\,\Delta \cup c \rrbracket} \tag{10}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow \texttt{false}}{\llbracket \texttt{if}\ (c)\ t\ \texttt{else}\ e\,;\,\Delta \rrbracket \longrightarrow \llbracket e\,;\,\Delta \cup \neg\, c \rrbracket} \tag{11}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow c' \qquad \llbracket t\,;\,\Delta \cup c' \rrbracket \longrightarrow t' \qquad \llbracket e\,;\,\Delta \cup \neg\, c' \rrbracket \longrightarrow e'}{\llbracket \texttt{if}\ (c)\ t\ \texttt{else}\ e\,;\,\Delta \rrbracket \longrightarrow \texttt{if}\ (c')\ t'\ \texttt{else}\ e'} \tag{12}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow \texttt{true} \qquad \llbracket e\,;\,\Delta \rrbracket \longrightarrow e'}{\llbracket \texttt{assume}(p)\texttt{;}\ e\,;\,\Delta \rrbracket \longrightarrow e'} \tag{13}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow \texttt{false} \qquad \llbracket e\,;\,\Delta \rrbracket \longrightarrow e'}{\llbracket \texttt{assume}(p)\texttt{;}\ e\,;\,\Delta \rrbracket \longrightarrow \texttt{assume(false);}\ e'} \tag{14}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow p' \qquad \llbracket e\,;\,\Delta \cup p' \rrbracket \longrightarrow e'}{\llbracket \texttt{assume}(p)\texttt{;}\ e\,;\,\Delta \rrbracket \longrightarrow \texttt{assume}(p')\texttt{;}\ e'} \tag{15}$$

$$\frac{\texttt{T}_2\texttt{: ADTType} \qquad \neg\texttt{isSort(T}_2\texttt{)}}{\llbracket \texttt{C(T}_1\texttt{,}\ a_1,\ldots,a_n\texttt{).isInstanceOf[T}_2\texttt{]}\,;\,\Delta \rrbracket \longrightarrow \texttt{T}_1\texttt{.id == T}_2\texttt{.id}} \tag{16}$$

$$\frac{\texttt{T: ADTType} \qquad \texttt{isSort(T)}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow \texttt{true}} \tag{17}$$

$$\frac{\texttt{T: ADTType} \qquad \llbracket e\,;\,\Delta \rrbracket \longrightarrow e' \qquad \texttt{isInstanceOf(e', T, }\Delta\texttt{) == Some(}b\texttt{)}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow b} \tag{18}$$

$$\frac{\texttt{T: ADTType} \qquad \llbracket e\,;\,\Delta \rrbracket \longrightarrow e' \qquad \texttt{isInstanceOf(}e'\texttt{, T, }\Delta\texttt{) == None}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow e'\texttt{.isInstanceOf[T]}} \tag{19}$$

Figure 1: Operational semantics of the symbolic evaluator

$$\frac{\mathtt{T:ADTType}}{\llbracket\, e\mathtt{.asInstanceOf[T]} \,;\, \Delta \,\rrbracket \longrightarrow \llbracket\, e \,;\, \Delta \,\rrbracket\mathtt{.asInstanceOf[T]}} \tag{20}$$

$$\frac{}{\llbracket\, \mathtt{let\ x:T\ =}\ v\ \mathtt{in}\ e \,;\, \Delta \,\rrbracket \longrightarrow \llbracket\, e[\mathsf{x}/v] \,;\, \Delta \,\rrbracket} \tag{21}$$

$$\frac{\llbracket\, f \,;\, \Delta \,\rrbracket \longrightarrow \lambda \mathsf{x}_1\mathtt{:T}_1\mathtt{,...,x}_n\mathtt{:T}_n\,.\,b \qquad \llbracket\, e_i \,;\, \Delta \,\rrbracket \longrightarrow e'_i,\ i \in \{1\ldots n\}}{\llbracket\, f(e_1,\ldots,e_n) \,;\, \Delta \,\rrbracket \longrightarrow \llbracket\, b[\mathsf{x}_1/e'_1,\ldots,\mathsf{x}_n/e'_n] \,;\, \Delta \,\rrbracket} \tag{22}$$

$$\frac{\llbracket\, f \,;\, \Delta \,\rrbracket \longrightarrow f' \qquad \llbracket\, e_i \,;\, \Delta \,\rrbracket \longrightarrow e'_i,\ i \in \{1\ldots n\}}{\llbracket\, f(e_1,\ldots,e_n) \,;\, \Delta \,\rrbracket \longrightarrow f'(e'_1,\ldots,e'_n)} \tag{23}$$

$$\frac{\neg\,\mathtt{isRecursive(id)} \quad \mathtt{id.params} = \langle \mathsf{x}_1,\ldots,\mathsf{x}_n \rangle \quad \llbracket\, e_i \,;\, \Delta \,\rrbracket \longrightarrow e'_i,\ i \in \{1\ldots n\}}{\llbracket\, \mathtt{id}(e_1,\ldots,e_n) \,;\, \Delta \,\rrbracket \longrightarrow \llbracket\, \mathtt{id.body}[\mathsf{x}_1/e'_1,\ldots,\mathsf{x}_n/e'_n] \,;\, \Delta \,\rrbracket} \tag{24}$$

$$\frac{\mathtt{id.body} \Downarrow \Delta \uplus \{\, \mathsf{x}_i \mapsto e'_i \,|\, 1 \le i \le n \,\} \quad \mathtt{id.params} = \langle \mathsf{x}_1,\ldots,\mathsf{x}_n \rangle \quad \llbracket\, e_i \,;\, \Delta \,\rrbracket \longrightarrow e'_i,\ i \in \{1\ldots n\}}{\llbracket\, \mathtt{id}(e_1,\ldots,e_n) \,;\, \Delta \,\rrbracket \longrightarrow \llbracket\, \mathtt{id.body}[\mathsf{x}_1/e'_1,\ldots,\mathsf{x}_n/e'_n] \,;\, \Delta \,\rrbracket} \tag{25}$$

$$\frac{\llbracket\, e_i \,;\, \Delta \,\rrbracket \longrightarrow e'_i,\ i \in \{1\ldots n\}}{\llbracket\, \mathtt{id}(e_1,\ldots,e_n) \,;\, \Delta \,\rrbracket \longrightarrow \mathtt{id}(e'_1,\ldots,e'_n)} \tag{26}$$

$$\frac{(\mathtt{cons},(e_1,\ldots,e_n)) = \mathtt{deconstruct(e)} \quad \llbracket\, e_i \,;\, \Delta \,\rrbracket \longrightarrow e'_i,\ i \in \{1\ldots n\}}{\llbracket\, e \,;\, \Delta \,\rrbracket \longrightarrow \mathtt{cons}(e'_1,\ldots,e'_n)} \tag{27}$$

$$\frac{e = \mathtt{if}\ (c)\ t_1\ \mathtt{else}\ t_2 \quad \llbracket\, c \,;\, \Delta \,\rrbracket \longrightarrow r \quad r \in \{\mathtt{true},\mathtt{false}\}}{\mathtt{e} \Downarrow \Delta} \tag{28}$$

Figure 1: Operational semantics of the symbolic evaluator

## 3.4 Implementation

We have implemented[1] the symbolic evaluator described above as a new, optional, simplification pass within Inox's existing simplification pipeline. This pass can be enabled by specifying the `--sym-eval` flag on the command-line, or by setting to corresponding option using the Inox API.

### 3.4.1 Path Condition Satisfiability

Our initial implementation

## 3.5 Case Study: Key-value Store Algebra

Listing 3 shows the implementation of a domain-specific language to manipulate a key-value store, along with a tracing interpreter over a Scala `Map`. An simple program which makes use of this DSL to insert a list of concrete values into the store is shown in Listing 2.

```scala
def insert(kvs: List[(String, String)])(after: Op): Op = kvs match {
  case Nil() => after
  case Cons((k, v), rest) => put(k, v) { () => insert(rest)(after) }
}

val xs = List("foo" -> "bar", "toto" -> "tata")
val program = insert(xs) {
  get("foo") { foo =>
    pure(foo)
  }
}

def lemma(map: Map[String, String], init: List[Label]) = {
  val (res, trace) = interpret(program)(map, init)

  res == Some("bar") &&
  trace.take(3).reverse == List(
    Label.Put("foo", "bar"),
    Label.Put("toto", "tata"),
    Label.Get("foo")
  )
} holds
```

Listing 2: Example program

While Inox is unable to prove the verification condition associated with `lemma`'s postcondition, turning on the symbolic evaluator yields the following VC, which it is then able to promptly prove is `UNSAT`, thus showing the property correct:

```scala
val x = (
```

[1]https://github.com/romac/inox/tree/sym-eval

```
  map.updated("foo", "bar").updated("toto", "tata").get("foo"),
  Get("foo") :: Put("toto", "tata") :: Put("foo", "bar") :: trace
)

x._1 != Some("bar") || {
  init.take(0).reverse :+ Put("foo", "bar") :+ Put("toto", "tata") :+ Get("foo")
  !=
  Put("foo", "bar") :: Put("toto", "tata") :: Get("foo") :: Nil
}
```

```
sealed abstract class Label
object Label {
  case class Get(key: String) extends Label
  case class Put(key: String, value: String) extends Label
}

sealed abstract class Op
case class Pure(value: Option[String]) extends Op
case class Get(key: String, next: Option[String] => Op) extends Op
case class Put(key: String, value: String, next: () => Op) extends Op

def get(key: String)(next: Option[String] => Op): Op = Get(key, next)
def put(key: String, value: String)(next: () => Op): Op = Put(key, value, next)
def pure(value: Option[String]): Op = Pure(value)

def interpret(op: Op)(kv: Map[String, String], trace: List[Label]) = op match {
  case Get(key, next) =>
        interpret(next(kv.get(key)))(kv, Label.Get(key) :: trace)
  case Put(key, value, next) =>
        interpret(next())(kv.updated(key, value), Label.Put(key, value) :: trace)
  case Pure(value) =>
        (value, trace)
}
```

Listing 3: Key-value store implementation

## 3.6 Termination

We discuss below two issues related to the termination of the symbolic evaluator. Moreover, we strongly suggest not enabling the evaluator when checking for termination in Stainless, as doing could throw prevent the termination checker to itself terminate, as well as yield wrong results.

### Evaluation of Non-Terminating Programs

While the current implementation will not attempt to unfold an "obviously" non-terminating function such as `def f(x: A): A = f(x)`, it is still fairly easy to bypass the current heuristics. For example, when evaluating `g(List(1))` the evaluator will not realise that, even though it is able to reduce the conditional expression to its "else" branch, doing so does not decrease the size of the argument of the recursive call, and will thus never terminate.

```
def bar(xs: List[BigInt]): List[BigInt] = {
  if (xs.isEmpty) Nil() else bar(Cons(xs.head, xs))
}

def test = {
  bar(List(1)) == Nil()
} holds
```

Listing 4: Non-Terminating program sending the evaluator into a loop

As Stainless already comes with a powerful termination checker, we believe that it make little sense to re-implement one within Inox. Instead, we simply suggest to only enable the symbolic evaluator on functions which have been proved terminating by Stainless, and enabling debug information with the `-debug=sym-eval` flag in order to hopefully catch any the evaluator being stuck in an infinite loop.

A possible workaround for this problem would be to add a step counter to the evaluator, and stop the evaluation after a certain number of steps have been made. Alternatively, we could stop evaluation after a given laps of time has elapsed, much like the solver underlying Inox is constrained to do. For lack of time, we have currently not implemented either solution.

### Evaluation of Terminating Programs

It unfortunately appears that, when it is currently possible for the evaluator to go into an infinite loop while evaluating a program that Stainless deems to be terminating. Listing 5 shows such a program. We suspect that functions preconditions are currently not being taken into account during the evaluation, because the latter relies on a fresh solver instance which has not been fed the corresponding assertions. It is not exactly clear to us how to solve this problem, although we suspect that either reusing the same solver instance that Inox has been initialised with or lifting the symbolic evaluator at Stainless's level rather than Inox (see Section 3.8) could fix this.

## 3.7 Conclusion

(TODO: Conclusion)

```
def foo(a: Boolean, n: BigInt): Boolean = {
  require(n != 0)

  if (n == 0)
    foo(a, 0) // this branch should never be evaluated
  else
    a
}

def test = {
  foo(true, 0) == true
} holds
```

Listing 5: Provably terminating program sending the evaluator into a loop

## 3.8 Further Work

### Symbolic Evaluation of PureScala Programs in Stainless

The technique of symbolic evaluation is not limited to the simplification of verification conditions that we covered in this chapter. It can indeed also be used on its own over regular programs. For example, one could implement an interpreter for a (domain specific) language $L$ in PureScala, and then write a program $p$ in this language. One could then symbolically execute the interpreter over this program $p$, keeping the the inputs or even the I/O operations abstract, yielding a PureScala expression. This resulting expression can be seen as a "compiled" version of the original program, represented as a PureScala program, which could then linked to a small runtime and be compiled to a lower-level language using the infrastructure provided by the Scala language itself, PureScala being a strict subset of Scala.

The technique of partially evaluating an interpreter over a known program is known as the *1st Futamura projection* [1], and can theoretically be applied to the symbolic evaluator itself, yielding a compiler (the 2nd projection) or even a converter from any interpreter to a compiler (3rd projection).

To enable such a use case, we propose adding a new annotation `@symeval` to Stainless's standard library, which when present, would instruct Stainless to symbolically evaluate the annotated function, and substitute the resulting expression for the function's original body, while also printing or dumping the full PureScala program resulting from this transformation. This would enable users to use Stainless as a symbolic evaluator for PureScala program.

Listing 6 features an example involving an interpreter for arithmetic expressions, as well as the result yielded by a preliminary experiment, which just hooks into the Inox pipeline and prints the result of the symbolic evaluation of any expression. More work is needed to properly integrate this feature within Stainless, which mainly consists of extending the symbolic evaluator with support for the full PureScala language constructs, such as pattern matches.

We also believe that lifting the evaluator at the Stainless level might potentially allow us to fix the termination issues we described in Section **??**. This because we would then be able to check whether a function's precondition holds before unfolding its definition, something that is not currently possible when working within Inox, as preconditions are encoded as assertions which are fed to the solver independently.

**Nondeterministic Symbolic Evaluator**

The evaluator described in this chapter is *deterministic*, in the sense that its evaluation function maps one expression to exactly one expression in a deterministic way. We believe that it could be worthwhile to investigate the the potential use cases of a so-called *non-deterministic*[2] symbolic evaluator. That is to say, a symbolic evaluator which, when given a single expression, would returns potentially multiple results. For example, when the evaluator would be applied to an expression $s$ `= if (`$c$`)` $t$ `else` $e$ whose condition $c$ it is not able to evaluate to a boolean literal, it would return all the possible expressions resulting from first evaluating the condition and both branches nondeterministically. We show some pseudo-code in Listing 7.

One potential use case of such an evaluator that we see, when combined with a way to trigger its application in user-land code, is for implementing a symbolic *partial order reduction* algorithm. In a bit more detail, the evaluator could be used to compute the set of (potentially symbolic) transitions that results from taking a step at a given (potentially symbolic) state.

[2]Here meant in the same sense that the lazy list monad is often use to model non-determinism.

```scala
sealed trait Expr
case class Var(name: String)     extends Expr
case class Num(value: Int)       extends Expr
case class Add(l: Expr, r: Expr) extends Expr
case class Mul(l: Expr, r: Expr) extends Expr

def interpret(expr: Expr, ctx: Map[String, Int]): Int = expr match {
  case Num(value) => value
  case Var(name)  => ctx(name)
  case Add(l, r)  => interpret(l, ctx) + interpret(r, ctx)
  case Mul(l, r)  => interpret(l, ctx) * interpret(r, ctx)
}

val program: Expr = Mul(Num(10), Add(Var("x"), Num(2)))

@symeval
def compiled(ctx: Map[String, Int]): Int = {
  interpret(program, ctx) // ==> 10 * (ctx("x") + 2)
}
```

Listing 6: Application of the 1st Futamura projection to a simple interpreter and program

```scala
def ndEval(expr: Expr, pc: PC): Stream[Expr] = {
 case IfExpr(c, t, e) =>

ndEval(e, pc) == for {
  cs <- eval(c, pc)
  ts = eval(t, pc withCond c)
  es = eval(e, pc withCond not(c))
} yield ts ++ es
```

Listing 7: Pseudo-code of a non-deterministc evaluator

# 4 Verifiying Actor Systems

## 4.1 Motivation

Over the last few decades, many different models of concurrent computation have been discovered, such as *Petri Nets*, *Communicating Sequential Processes* and the $\pi$*-calculus* [2].

Moreover, because actors do not share memory and rely on asynchronous message passing, the Actor Model is well suited to model distributed systems as well.

## 4.2 A Simple Actor Model for Verification

### 4.2.1 Introduction

### 4.2.2 Implementation

We now below the PureScala implementation of the model

#### Message

In our framework, messages are modelled as constructors of the `Msg` abstract class.

```scala
abstract class Msg
case class Hello(name: String) extends Msg
```

#### Actor Reference

Each actor is associated with a unique and persistent reference, modelled as an instance of the `ActorRef` abstract class.

```scala
abstract class ActorRef
case class Primary() extends ActorRef
```

#### In-flight Messages

In-flight messages are represented as a product of the `ActorRef` of the destination actor, and the message itself.

```scala
case class Packet(dest: ActorRef, payload: Msg)
```

#### Actor Context

When a message is delivered to an actor, the latter is provided with a context, which holds a reference to itself, and a mutable list of `Packet`s to send.

```scala
case class ActorContext(
  self: ActorRef,
  var toSend: List[Packet]
)
```

### Behavior

A behavior specifies both the current state of an actor, and how this one should process the next incoming message. In our framework, these are modelled as a subclass of the abstract class `Behavior`, which defines a single abstract method `processMsg`, to be overriden for each defined behavior.

Using the provided `ActorContext`, the implementation of the `processMsg` method can both access its own reference, and register messages to be sent after the execution of the method is complete. It is also required to return a new `Behavior`

```
abstract class Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior
}
```

### Actor System

The state of the Actor system at a given point in time is modelled as a case class, holding the behavior associated with each actor reference, and the list of in-flight messages between any two actors.

```
case class ActorSystem(
  behaviors: CMap[ActorRef, Behavior],
  inboxes: CMap[(ActorRef, ActorRef), List[Msg]]
)
```

The `ActorSystem` class is equipped with a `step` method, which takes a pair of `ActorRef` as arguments, and is in charge of delivering the oldest message found in the corresponding inbox, and which returns the new state of the system after the aforementioned message has been processed.

```
def step(from: ActorRef, to: ActorRef): ActorSystem
```

### 4.2.3 Comparison with Akka and Akka Typed

#### Akka

Akka is a powerful Scala implementation of the Actor model, suitable for both concurrent and distributed systems.

#### Akka Typed

#### Notable Differences

While we initially set to mimic the API offered by the Akka Typed library[3], we quickly ran into some limitations of Stainless's type system. Namely, the lack of an encoding for both existential types, and various issues with the encoding of co-/contravariant class hierarchies. While some of those issues got fixed over the last few months, we have been unable to provide a typed API

[3]https://doc.akka.io/docs/akka/2.5.4/scala/typed.html

similar in spirit to Akka Typed. For lack of time, we have decided to go with the untyped API described in Section 4.2.2.

Another notable difference between our model and both the one implemented in Akka[4], is that we rely on *exactly-once* delivery of messages. We experimented with a weaker model that only relied on *at-most-once* delivery but it quickly became clear that the amount of work needed to verify the very same systems we describe below would be too much in the context of this project.

## 4.3 Operational Semantics

We formulate the small-step operational semantics of our Actor model in Figure 2, where $s$ : `ActorSystem` is an Actor system, $m$ : `Msg` is a message, $n, n_{to}, n_{from}$ : `ActorRef` are references, $b, b'$ : `Behavior` are behaviors, $ps$ : `List[Packet]` a list of packets to send, $c$ : `ActorContext` is a context, and $\emptyset_n$ : `ActorContext` is the empty context for an actor whose self-reference is $n$, defined as $\emptyset_n :=$ `ActorContext`$(n,$ `Nil`$)$.

## 4.4 Proving Invariants

After having defined an Actor system with our framework, one might want to verify that this system preserves some invariant between each step of its execution. That is to say, for an `ActorSystem` $s$, any two `ActorRef` $n, m$, and an invariant `inv: ActorSystem` $\rightarrow$ **`Boolean`**, if `inv`$(s)$ holds, then `inv`$(s$`.step`$(n, m))$ should hold as well. We express this property more formally in Figure 3. This property can be easily expressed in PureScala, as shown in Listing 8.

```
def inv(s: ActorSystem): Boolean = {
  /* ... */
}

def preserveInv(s: ActorSystem, n: ActorRef, m: ActorRef): Boolean = {
  require(inv(s))
  inv(s.step(n, m))
} holds
```

Listing 8: Invariant preservation theorem in PureScala

When encoutering such a definition, Stainless will generate a verification condition equivalent to Figure 3, which will then be discharged to Inox and the underlying SMT solver.

## 4.5 Case studies

### 4.5.1 Increment-based Replicated Counter

As a first and very simple case study, we will study an Actor system which models a replicated counter, which can only be incremented by one unit. This system is composed of two actors, a

[4] https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html

$$\frac{\nexists m \in s.\texttt{inboxes}(n_{from}, n_{to})}{s.\texttt{step}(n_{from}, n_{to}) \rightsquigarrow \texttt{s}} \quad \text{(STEP-NOMSG)}$$

$$\frac{\exists m \in s.\texttt{inboxes}(n_{from}, n_{to}) \qquad s.\texttt{deliverMsg}(n_{to}, n_{from}, m) \rightsquigarrow (b, ps, t)}{s.\texttt{step}(n_{from}, n_{to}) \rightsquigarrow s \uplus (n_{to} \mapsto b, \ldots, t)} \quad \text{(STEP)}$$

$$\frac{s.\texttt{behaviors}(n_{to}).\texttt{processMsg}(m, \emptyset_{n_{to}}) \rightsquigarrow (b, c)}{s.\texttt{deliverMsg}(n_{to}, n_{from}, m) \rightsquigarrow (b, c.\texttt{toSend}, t)} \quad \text{(DELIVER-MSG)}$$

$$\frac{b.\texttt{processMsg}(m, \emptyset_{n_{to}}) = i_1 :: \ldots :: i_n :: b' :: \texttt{Nil} \qquad \langle i_1 :: \ldots :: i_n :: \texttt{Nil}, \emptyset_{n_{to}} \rangle \longrightarrow c}{b.\texttt{processMsg}(m, \emptyset_{n_{to}}) \rightsquigarrow (b', c)} \quad \text{(PROCESS-MSG)}$$

$$\frac{}{\langle \texttt{Nil}, c \rangle \longrightarrow c} \quad \text{(I-NIL)}$$

$$\frac{\langle i, c \rangle \longrightarrow c'}{\langle i :: is, c \rangle \Rightarrow \langle is, c' \rangle} \quad \text{(I-CONS)}$$

$$\frac{}{\langle n \texttt{ ! } m, c \rangle \longrightarrow (b',\ c.\texttt{copy(toSend} \mapsto \texttt{(}n\texttt{, } m\texttt{)} :: c.\texttt{toSend))}} \quad \text{(I-SEND)}$$

Figure 2: Operational semantics

$$\forall s : \texttt{ActorSystem},\ n : \texttt{ActorRef},\ m : \texttt{ActorRef}.\ \texttt{inv}(s) \implies \texttt{inv}(s\texttt{.step}(n, m))$$

Figure 3: Invariant preservation property

primary counter whose reference is `Primary()`, and a backup counter whose reference is `Backup()`. Each of these reference is associated with a behavior: the primary counter reference with an instance of `PrimaryB`, and the backup counter reference with an instance of `BackupB`, both of which hold a positive integer, representing the value of the counter. Whenever the primary actor receives a message `Inc()`, it forwards that message to the backup actor, and returns a new instance of `PrimaryB` with the counter incremented by one. When the backup actor receives an `Inc()` message, it just returns a new instance of `BackupB` with the counter incremented by one. The corresponding PureScala implementation can be found in Listing 9.

Given such a system, one might want to prove that the following invariant is preserved between each step of its execution:

This invariant specifies that the `Backup()` actor does not send itself any messages, that both actors have the proper corresponding behavior, and that, last but not least, the value of the primary counter is equal to the value of the backup counter added to the number of messages

```
case class Primary() extends ActorRef
case class Backup()  extends ActorRef

case class Inc() extends Msg

case class PrimaryB(counter: BigInt) extends Behavior {
  require(counter >= 0)

  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Inc() =>
      Backup() ! Inc()
      PrimaryB(counter + 1)
  }
}

case class BackupB(counter: BigInt) extends Behavior {
  require(counter >= 0)

  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Inc() => BackupB(counter + 1)
  }
}
```

Listing 9: Increment-based replicated counter implementation

```
def invariant(s: ActorSystem): Boolean = {
  s.inboxes((Backup(), Backup())).isEmpty && {
    (s.behaviors(Primary()), s.behaviors(Backup())) match {
      case (PrimaryB(p), BackupB(b)) =>
        p.value == b.value + s.inboxes(Primary() -> Backup()).length
      case _ => false
    }
  }
}
```

Listing 10: Increment-based replicated counter invariant

that are yet to be delivered to the backup actor.

We can now define the actual theorem we want Stainless to prove for us:

```scala
def preserveInv(s: ActorSystem, n: ActorRef, m: ActorRef): Boolean = {
  require(invariant(s))
  invariant(s.step(n, m))
} holds
```

Listing 11: Replicated counter theorem (increment)

### 4.5.2 Delivery-based Replicated Counter

Listing 12 shows a variant of the previous case study, where instead of having the primary actor forward the `Inc()` message to the backup actor, the former instead sends the latter the new value.

```scala
case object Primary extends ActorRef
case object Backup  extends ActorRef

case object Inc extends Msg
case class Deliver(c: BigInt) extends Msg

case class PrimaryB(counter: BigInt) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Inc =>
      Backup ! Deliver(counter + 1)
      PrimaryB(counter + 1)

    case _ => Behavior.same
  }
}

case class BackupB(counter: BigInt) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Deliver(c) => BackupB(c)
    case _          => Behavior.same
  }
}
```

Listing 12: Delivery-based Replicated counter implementation

The invariant now reads slightly differently, as can be seen in Listing 13.

### 4.5.3 Lock Service

Listing 14 shows the implementation of a lock service using our framework. In this case study, an actor acts as a server holding a lock on some resource, while a number of other actors (the "agents") act as clients of the lock service, each potentially trying to acquire the lock on the

```
def validBehaviors(s: ActorSystem): Boolean = {
  (s.behaviors(Primary), s.behaviors(Backup) match {
    case (p: PrimaryB, b: BackupB) => true
    case _                         => false
  }
}

def invariant(s: ActorSystem): Boolean = {
  validBehaviors(s)                     &&
  s.inboxes(Primary -> Primary).isEmpty   &&
  s.inboxes(Backup -> Backup).isEmpty     &&
  noMsgsToSelf(Backup -> Primary).isEmpty &&
  {
    val PrimaryB(p) = s.behaviors(Primary)
    val BackupB(b)  = s.behaviors(Backup)
    val bInbox      = s.inboxes(Primary -> Backup)

    p.value >= b.value && isSorted(bInbox) && bInbox.forall {
      case Deliver(Counter(i)) => p.value >= i
      case _                   => true
    }
  }
}
```

Listing 13: Delivery-based replicated counter invariant

resource. To model a variable number of actors with the same implementation, we define their reference as a case class parametrized by a (TODO: unique) identifier.

An obvious property we might want to prove is that, at any time, at most one of those agents thinks that it holds the lock. Additionally, we'd like to ensure that such an agent is actually the same one that the server granted the lock too. We express this property in Listing 15.

(TODO: Lock service invariant proof)

## 4.6 Spawning Actors

### Updating The Model

Up until now, our framework has only been able to model Actor systems with a static topology, ie. systems where no new actors besides the ones that are statically defined can be spawned. Let's now attempt to enrich our model to account for dynamic topologies.

To this end, we modify the `ActorRef` definition to include both a name and an optional field holding a reference to its parent `ActorRef` if any. We also add a new constructor of the `ActorRef` data type, which will be assigned to actors spawned from another actor.

```
abstract class ActorRef(
  name: String,
  parent: Option[ActorRef] = None()
)

case class Child(name: String, getParent: ActorRef)
  extends ActorRef(name, Some(getParent))
```

In order for actors to spawn other actors, by specifying their name and associated initial behavior, we modify the `ActorContext` class as follows:

```
case class ActorContext(
  self: ActorRef,
  var toSend: List[Packet],
  var toSpawn: List[(ActorRef, Behavior)]
) {
  def spawn(behavior: Behavior, name: String): ActorRef = {
    val id: ActorRef = Child(name, self)
    toSpawn = toSpawn :+ (id, behavior)
    id
  }
  /* ... */
}
```

As can be seen in the listing above, the context now keeps track of the names and behaviors of the actors to be spawned, and provides a `spawn` method which is in charge of constructing the `ActorRef` of the spawned actor, storing it along with the behavior within the context, and returning the newly generated reference.

### Case Study

Listing **??** defines a simple system with a dynamic topology, where one actor named `Primary` waits for a `Spawn` message to spawn a child actor, and change its behavior from `BeforeB` to `AfterB` in order to keep track of the reference to the child. The invariant we would to verify holds here, states that, if the `Primary` actor has behavior `BeforeB()`, then the behavior associated with the `ActorRef` of its child actor must be `Stopped`. On the other hand, if the `Primary` actor has behavior `AfterB(child)`, then the behavior associated with `child` must be `ChildB`. This test case verifies promptly, provided the symbolic evaluator is enabled.

## 4.7 Running an Actor System on Akka

While the verification of Actor systems is in itself an interesting endeavour, it is not of much use unless one is able to run these systems, potentially in a distributed environment. In the Scala ecosystem, the most widely used real-world Actor system is Akka (TODO: REF). Listing 16 shows a shallow shim which allows to run an Actor system developed with our framework within Akka, with only a few alterations to the original program. With this shim, the `ActorRef` type is mapped to Akka's `ActorRef` , while the `ActorContext` now only contains the actor's self-reference, as well as the underlying Akka context. The shim also defines an actual Akka actor, parametrized by an underlying `Behavior`, to which all messages of type `Msg` will be delegated. The `ActorSystem` class becomes abstract, and concrete subclasses need to provide it with an actual Akka system, as well as provide an implementation of its `run` method. Within this method, one can spawn new top-level actors, get a reference to those, and send them messages. Listing 17 shows such an implementation for the replicated counter described in Section 4.5.1.

## 4.8 Conclusion

(TODO: Conclusion)

## 4.9 Further Work

#### Real-world Case Study

#### Weaker Guarantees On Message Delivery

#### Name Uniqueness

It is important to note that, within this model, actor references are not guaranteed to be unique as a user could spawn two actors with the same name. (TODO: ++)

## 4.10 Reasoning About Traces

(TODO: Traces)

```scala
case class Server() extends ActorRef
object Server {
  case class Lock(agent: ActorRef) extends Msg
  case class Unlock(agent: ActorRef) extends Msg
}

case class Agent(id: Int) extends ActorRef
object Agent {
  case object Lock   extends Msg
  case object Unlock extends Msg
  case object Grant  extends Msg
}

// The head of ‘agents‘ holds the lock, the tail are waiting for the lock
case class ServerB(agents: List[ActorRef]) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Server.Lock(agent) if agents.isEmpty =>
      agent ! Agent.Grant
      ServerB(List(agent))

    case Server.Lock(agent) =>
      ServerB(agents :+ agent)

    case Server.Unlock(agent) if agents.nonEmpty =>
      val newAgents = agents.tail
      if (newAgents.nonEmpty) newAgents.head ! Agent.Grant
      ServerB(newAgents)

    case _ =>
      Behavior.same
  }
}

case class AgentB(holdsLock: Boolean) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Agent.Lock =>
      Server() ! Server.Lock(ctx.self)
      Behavior.same

    case Agent.Unlock if holdsLock =>
      Server() ! Server.Unlock(ctx.self)
      AgentB(false)

    case Agent.Grant =>
      AgentB(true)

    case _ =>
      Behavior.same
  }
}
```

Listing 14: Lock service implementation

```
def hasLock(s: ActorSystem, a: ActorRef): Boolean = {
  s.behaviors(a) match {
    case AgentB(hasLock) => hasLock
    case _ => false
  }
}

def mutex(s: ActorSystem): Boolean = forall { (a: ActorRef, b: ActorRef) =>
  (a != b) ==> !(hasLock(s, a) && hasLock(s, b))
}

def hasLockThenHead(s: ActorSystem): Boolean = forall { (ref: ActorRef) =>
  hasLock(s, ref) ==> {
    s.behaviors(Server()) match {
      case ServerB(Cons(head, _)) => head == ref
      case _ => false
    }
  }
}

def invariant(s: ActorSystem): Boolean = {
  mutex(s) && hasLockThenHead(s)
}
```

Listing 15: Lock service invariant

```scala
case object Primary extends ActorRef("primary")
case object Spawn extends Msg

case class BeforeB() extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case Spawn =>
      val child = ctx.spawn(ChildB(), "child")
      AfterB(child)
  }
}

case class AfterB(child: ActorRef) extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case _ => Behavior.same
  }
}

case class ChildB() extends Behavior {
  def processMsg(msg: Msg)(implicit ctx: ActorContext): Behavior = msg match {
    case _ => Behavior.same
  }
}

def invariant(s: ActorSystem): Boolean = {
  s.behaviors(Primary) match {
    case BeforeB() =>
      s.isStopped(Child("child", Primary()))
    case AfterB(child) =>
      s.behaviors(child) == ChildB()

    case _ => false
  }
}

def theorem(s: ActorSystem, from: ActorRef, to: ActorRef): Boolean = {
  require(invariant(s))
  invariant(s.step(from, to))
} holds
```

```scala
import akka.actor

type ActorRef = actor.ActorRef

case class ActorContext(self: actor.ActorRef, ctx: actor.ActorContext)

class AkkaWrapper(var behavior: Behavior)
  extends actor.Actor with actor.ActorLogging {

  implicit val ctx = ActorContext(self, context)

  def receive = {
    case msg: Msg =>
      log.info(s"${behavior}: ${msg}")
      behavior = behavior.processMsg(msg)

    case _ => ()
  }
}

abstract class ActorSystem(val system: actor.ActorSystem) {
  def spawn(behavior: Behavior, name: String): actor.ActorRef = {
    system.actorOf(actor.Props(new AkkaWrapper(behavior)), name = name)
  }

  def run(): Unit
}
```

Listing 16: Akka shim for our Actor framework

```scala
@extern
object System extends ActorSystem("rep-counter-sys") {
  def run(): Unit = {
    val backup  = spawn(BackupB(0), "backup")
    val primary = spawn(PrimaryB(0, backup), "primary")

    primary ! Inc()
  }
}

@extern
def main(args: Array[String]): Unit = {
  System.run()
}
```

Listing 17: Akka shim for our Actor system framework

# 5 Biparty Communication Protocols

## 5.1 Motivation

Most systems made of components exchanging messages between them, such as Actor systems, do so by following a *communication protocol.* A protocol is a set of rules which the components must follow when receiving, processing, and replying to a message. While it is possible to verify that a system correctly implements a given protocol, doing so is usually not trivial and often requires a sizeable amount of work, depending of course on the model one is working with. In the case of the Actor model we developed in Section 4, in which messages are *unityped*, verifying that a system followed some kind of protocol often required encoding a small type system within the invariants, for example to ensure that a message sent by an actor was of the right type. Moreover, we also had to check whether an actor would actually send a reply when it was expected of it. While being very interesting properties to verify, we would rather rely on the actual type system to catch these mistakes directly when writing the implementation.

As we have unfortunately not found a way to provide even just a typed API for our model, and do not expect to find a solution within the scope of this project, we now turn our attention to simpler, synchronous systems involving only two parties communicating over a *channel* with two endpoints, one for sending messages and one for receiving them. Such a system might be an ATM and a user willing to withdraw money communicating through a screen, or a browser and a web server attempting to initialise a secure connection through a TCP socket. The protocols governing such systems that we are going to study in this chapter are called *biparty communication protocols.* Such protocols have the interesting property that the description of the protocol from the viewpoint of one party is enough to deduce the protocol that must be followed from the viewpoint of the other party. The dialogue between two parties is commonly referred to as a *session.*

As a running example for the next sections, we will consider a very simple protocol involving two parties Alice and Bob, and four messages `Greet`, `Hello`, `Bye`, and `Quit`. From the viewpoint of Alice, the protocol is informally described as follows: Alice must send Bob either the message `Quit` or `Greet`. In the first case the sessions then ends, otherwise she can expect to receive either the message `Bye`, ending the session, or the message `Hello`, after which the session continues from the start.

It is important to note that, for the an implementation of such protocols to be correct, assuming an underlying reliable delivery channel, each party must both send the correct message when it is expected of them, and handle all possible messages that they can expect to receive at some point. A implementation of a party which fails to do so is deemed incorrect in that framework.

## 5.2 Session Types

Session types [3] provide a way to encode such protocols at the type level, thus guiding the programmer during the implementation while also guarding against mistakes such as not handling a certain message type or failing to reply to a message. Their syntax is shown in Figure 4.

$$
\begin{array}{lcll}
T & ::= & S \mid \texttt{Int} \mid \texttt{Boolean} \mid \texttt{String} \mid \ldots & \text{payload} \\
S & ::= & \&\{\, l_i : S_i \,\}_{i \in I} & \text{branching} \\
 & \mid & \oplus\{\, l_i : S_i \,\}_{i \in I} & \text{selection} \\
 & \mid & ?T.\, S & \text{receive} \\
 & \mid & !T.\, S & \text{send} \\
 & \mid & \mu_X .\, S & \text{recursion} \\
 & \mid & X & \text{variable} \\
 & \mid & \epsilon & \text{termination}
\end{array}
$$

Figure 4: Syntax of session types

$$
\begin{array}{rcl}
\overline{\overline{S}} & ::= & S \\
\overline{\&\{\, l_1 : S_1, \ldots, l_n : S_n \,\}} & ::= & \oplus\{\, l_1 : \overline{S_1}, \ldots, l_n : \overline{S_n} \,\} \\
\overline{?T.\, S} & ::= & !T.\, \overline{S} \\
\overline{\mu\alpha.\, S} & ::= & \mu\alpha.\, \overline{S[\overline{\alpha}/\alpha]} \\
\overline{X} & ::= & X \\
\overline{\epsilon} & ::= & \epsilon
\end{array}
$$

Figure 5: Duality of session types

We show below the session types $S_A$ and $S_B$ corresponding to the protocol we defined in Section 5.1, from the viewpoint of Alice and Bob, respectively.

$$
S_A = \mu\alpha.\left(!\texttt{Greet}.\left(?\texttt{Hello}.\,\alpha \;\&\; ?\texttt{Bye}.\,\epsilon\right) \oplus !\texttt{Quit}.\,\epsilon\right)
$$
$$
S_B = \mu\alpha.\left(?\texttt{Greet}.\left(!\texttt{Hello}.\,\alpha \oplus !\texttt{Bye}.\,\epsilon\right) \;\&\; ?\texttt{Quit}.\,\epsilon\right)
$$

We draw the reader's attention to the similarity between the two types, and note that the $S_B$ reads the same as $S_A$ if one substitutes ! for ?, $\oplus$ for &, and vice-versa. Each type is in fact the *dual* of the other, a property we formalise in Figure 5.

## 5.3 Session Types and Linearity

Although session types were originally meant to be implemented as a separate syntactic category of types and terms to be added to the $\pi$-calculus, it has been shown that it possible to encode them directly in a calculus or language with both *linear types* and *variants* [4, 5].

While there are many ways to perform such an encoding and provide a collection of combinators to build values of such types while enforcing the associated safety properties, the existing approaches [6, 7] either rely on more expressive type systems than the one provided by Stainless, eg. *substructural type systems* or ones which provide *path families*, *higher-kinded types* and *indexed monads*, or *path-dependent types.* In the next section we look at a solution to this problem which only require minimal and orthogonal modifications to Stainless's type system.

## 5.4 Value-Level Sessions Encoding

The *lchannels* Scala library [8] provides a lightweight, value-level encoding of sessions which does not rely on advanced type system features. This encoding essentially corresponds to the *continuation-passing-style* transformation of session types. In their library, the two endpoints (one for receiving, one for sending) of the communication channel between two parties are represented as values of the two types `In` and `Out`, respectively. These types are parametrised by the type of value that they accept or produce, and provide methods to send or receive such messages. Listing 18 shows a subset the API provided by the library.

Listing 19 shows the encoding of the protocol we have been working with so far with this framework.

## 5.5 Linear Types in Stainless

In this section, we discuss our implementation of linear types in Stainless, and note that, because the PureScala AST we are working with in Stainless is already typed, there is no need to write a full-fledged type checker. We will hence rather describe a *linearity checker* for PureScala programs.

### 5.5.1 Introducing Linear Types

First of all, we need a way to introduce to mark some types as *linear.* To this end, we define a covariant type constructor `Linear`, which simply holds a value of type `A`. This type provides a `!` method to consume the linear term and return the underlying value. This enables the user to call a method of the underlying type in a concise way. As the astute reader might have noticed, this effectively adds weakening to the linear type system, and, as we will see, some care will be needed to handle such conversions properly. For example, if one had a value `foo` of type `Linear[Option[A]]`, one could call the `isEmpty` method on the underlying value by writing `foo!.isEmpty`. While making the consumption of a linear value explicit in this way is good for reasoning about one's code, there is still a bit of clutter associated with it, we also introduce an opt-in implicit conversion `delinearize` from any `Linear[A]` to `A`. At last, because converting a non-linear value of type `A` to a linear value of type `Linear[A]` is always safe, we provide a such an implicit conversion by default, `linearize`. Listing 20 shows the full definitions. Because those will be extracted in a specific way, they are marked `@ignore`.

```
abstract class In[+A] {
  // Blocks until a message is received through the channel and returns it.
  def receive(implicit d: Duration): A

  // Map over the next message received and returns the result.
  def ?[B](f: A => B)(implicit d: Duration): B = {
    f(receive)
  }
}

abstract class Out[-A] {
  // Sends a message through the channel.
  def !(msg: A): Unit

  // Sends a message which will trigger a reply, and returns the
  // endpoint to receive the reply from.
  // Expects the provided function to embed the given
  // output endpoint into the message before returning it.
  // and returns an endpoint for receiving it.
  def !![B](h: Out[B] => A): In[B] = {
    val (in, out) = create[B]()
    this ! h(out)
    in
  }

  // Sends a message after which another message must be sent as well.
  def !![B](h: In[B] => A): Out[B]

  // Creates a new channel and return its two endpoints.
  def create[B](): (In[B], Out[B])
}
```

Listing 18: *lchannels* library interface

```
abstract class AliceBob
case class Quit()                       extends AliceBob
case class Greet(cont: Out[Response]) extends AliceBob

abstract class Response
case class Hello(cont: Out[AliceBob]) extends Response
case class Bye()                        extends Response\
```

Listing 19: *lchannels* implementation of a simple protocol

```
package stainless

import stainless.lang._
import stainless.annotation._

package object linear {

  @ignore
  class Linear[+A](_value: A) {
    def ! = _value
  }

  @ignore
  implicit def linearize[A](value: A): Linear[A] = new Linear(value)

  object implicits {
    @ignore
    implicit def delinearize[A](lin: Linear[A]): A = lin!
  }
}
```

Listing 20: Linear wrapper for Scala types and values

### 5.5.2 Preventing Weakening

We now describe what it means for a linear term to be *consumed*: a term `t` of type `Linear[A]`, for any type `A`, is deemed `consumed` in an expression $e$ when any of the following propositions is true:

- The underlying value of type `A` is extracted, via the `!` method, eg. $e =$ `t!`.
- The term is assigned to a variable, eg. `val s:Linear[A] = t`.
- The term is supplied as an argument to function, eg. given `def f(x:Linear[A]):B`, we have $e =$ `f(t)`.
- The term is supplied as an argument to a method, eg. given a class `C` with a method `def m(x:Linear[A]):B`, a value $v : C$, we have $e =$ `v.m(t)`.
- The term is supplied as an argument to a lambda, eg. given `val l:Linear[A] => B`, we have $e =$ `l(t)`.
- The term is supplied as an argument to a constructor, eg. given `case class C(x:  Linear[A])`, we have $e =$ `C(t)`.

*Note: Method calls are subsumed by the first rule, as a linear value must first be delinearized with the ! operator before one can call a method on the underlying value.*

We now must ensure that no linear term is *consumed* more than once. To this end, we must recursively walk down the AST, while keeping track of terms that have been consumed in a *usage context*, in order to disallow subsequent uses of those terms. We will denote this context by $\Delta$. (TODO: Figure **??** presents the type-checking rules used to reject invalid programs.)

The astute reader will have noticed that the presence of the `!` operator, if not handled carefully, would actually allow weakening. For example, given a value `a: Linear[A]`, one could write `val b:  A = a.!`, and thus obtain a non-linear reference to the underlying value. To counter this, the linearity checker treats any expression of the form $e$`.!`, with $e$`: Linear[A]`, as having type `Linear[A]` instead of `A`.

### 5.5.3 Preventing Contraction

Because linear logic does not allow contraction, we must also ensure that no linear term is *dropped*, that is to say, that it is *consumed* at least once. To this end, we first collect all linear variables being introduced in a function definition, for example as a parameter to the function, in a variable binding, or within a pattern in a `clause` (even as a wildcard). Then, after having ran the type-checking algorithm described in the previous section, we can make use of the resulting *usage context* $\Delta$ to check whether each and every of those variable has indeed been *consumed* at some point, and reject the program otherwise.

### 5.5.4 Linear Terms in Contracts

It is important to note that, when running the linearity checker over a function with pre- and/or post-conditions, these are ignored for the following reason: a user might want to constrain either a linear parameter of some function, or its return value. If we ran the linearity checker on such contracts, then one would not be able to re-use the linear variable that is being constrained in the precondition, or would not be able to reference any linear parameter in the postcondition. Listing 21 shows such a use-case.

```scala
def foo(x: Linear[Option[BigInt]]): BigInt = {
  require(!x.isEmpty && x.get > 0)
  x.get * 2
} ensuring { _ > 0 }
```

Listing 21: Usage of a linear variable in a function's precondition

Fortunately for us, because a function's contract will be statically verified by Stainless, there is no point to check it at runtime. Hence, in Stainless's library, both the `require` function and the `ensuring` method discard their body. For this reason, we can safely ignore linearity constraints in a function's contract.

### 5.5.5 Linear Data Types

Because data types can contain linear fields, one must be careful as to when to allow values of such types to be introduced. Indeed, if one were to define a data type `case class A(x:Linear[B])`, one should not be allowed to construct a non-linear term of type `A`. That is because doing so would permit the user to have more than one (indirect) reference to the linear `x` field, which is forbidden because of the No-Weakening rule. We must thus ensure that values such types are only introduced linearly, ie. as values of type `Linear[A]`, whether it is as a function parameter, as a variable binding, or as field of another data type. Listing 22 features a few examples of this rule in action.

```
case class A(x: Linear[BigInt])

case class B(a: A)          // error
case class C(b: Linear[A])  // ok

def f(x: A): A = { // error
  x
}

def g(x: Linear[A]): Linear[A] = { // ok
  x
}

def h(x: C): Linear[A] = c match {
  case C(b) => b // ok
}

def i(x: BigInt): A = c match { // error
  A(x)
}

def j(x: BigInt): Linear[A] = c match {
  A(x) // ok
}

def k(x: BigInt): Linear[BigInt] = c match {
  A(x) match {
    case A(y) => y // error
  }
}
```

Listing 22: Linear data types in action

### 5.5.6 Marking the Current Object as Linear

Another issue arise when dealing with data types meant to be introduced linearly if these have associated methods. To understand why, let's look at an example. Listing 23 shows a very simple API which allows to open a file, read its content line-by-line, or read all its content at once. We would like to make sure that, once a user opens a `File` and receives the associated `FileHandler`, the latter must be closed. Unfortunately, when implementing the `contents` method, nothing prevents the programmer to call `readLine` twice on the current object. That is because, Within `contents`, `this` has type `FileHandler`, and calling any method on it will thus not consume it. As it is not possible in Scala to constrain the type of the current object, even with self-annotations, we introduce a method annotation `@linear` which signals to the linearity checker that, within an annotated method of a class `C`, `this` should be considered to have type `Linear[C]`. As methods inherit the annotations of their enclosing class, it is here enough to annotate the `FileHandler` class with `@linear`.

```scala
class File {
  def open: Linear[FileHandler] = /* ... */
}

class FileHandler {
  def readLine: (Option[String], Linear[FileHandler]) = /* ... */
  def close: Unit = /* ... */

  def contents: String = {
    val (res, h) = this.readLine
    doSomething(this.readLine) // should be disallowed
    res match {
      case Some(line) =>
        line + h!.contents

      case None() =>
        h!.close
        ""
    }
  }
}
```

Listing 23: Linear File API

### 5.5.7 Higher-Order Linear Functions

(TODO: Discuss syntax for linear lambdas)

## 5.6 Sessions Library in PureScala

Listing 24 shows a PureScala implementation of the *lchannels* library. For the purpose of verification, we do not need a full-fledged implementation of the channels, but only declarations mirroring the Scala library. This way, one could run their implementation with the original library by simply linking against both it and the Stainless library, without relying on our library.

```scala
type In[A]  = Linear[InChan[A]]
type Out[A] = Linear[OutChan[A]]

@linear @library
class InChan[A] {

  @extern
  def receive(implicit d: Duration): Linear[A] = {
    ???
  }

  def ?[B](f: Linear[A] => B)(implicit d: Duration): B = {
    f(receive)
  }
}

@linear @library
class OutChan[A] {

  @extern
  def send(msg: A): Unit = {
    ???
  }

  def !(msg: A): Unit = {
    send(msg)
  }

  @extern
  def !![B](h: Out[B] => A): In[B] = {
    ???
  }

  @extern
  def create[B](): (In[B], Out[B]) = {
    ???
  }
}
```

Listing 24: Sessions library in PureScala

## 5.7 Case Studies

### 5.7.1 ATM Protocol

Let's consider a protocol involving an ATM and its user, which we informally describe below:

A The user authenticates herself by sending the ATM both her card number and PIN.

B If the authentication succeeds, the ATM displays a menu to the user, who can then choose to:

  (a) Abort the process altogether.

  (b) Ask for her account's balance, in which case the server will reply with the balance, and displays the menu again.

C If the authentication fails, the ATM notifies the user of the failure, and the process is aborted.

Listing 25 shows the encoding of such a specification using the library described in Section 5.6. Listing 26 shows the corresponding valid implementation. At last, Listing 28 shows an incorrect implementation of the protocol that would still verify without the linearity checker. We discuss the three use cases below:

1. If we provide an empty body for the `atm` function, we would then be greeted with the following error:

```
Error: linear variable ‘c‘ of type ‘Linear[In[Authenticate]]‘ is never used:
                    def atm(c: Linear[In[Authenticate]]): Unit = {
                                ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

   Re-using the same channel twice would also give rise to an error:

```
Error: linear term ‘cont‘ has already been used: doSomething(cont)
                                                            ^^^^
Info: term used here: cont !! Balance(balance(card))(_) ? menu(card)
                                                             ^^^^
```

2. In case we forget to send back a failure notification when the authentication fails. The linearity checker will realise that the linear `cont` is not consumed in every branch of the pattern match, and will pinpoint its introduction:

```
Error: linear variable ‘cont‘ of type ‘Linear[OutChan[Response]]‘ is never used:
                    case Authenticate(_, _, cont) =>
                                            ^^^^
```

3. At last, let's see what happens if we do not handle the reply to the `Success` message sent in case the authentication succeeds. Because the expression `cont !! Success(_)` has type `In[Menu]`, one could expect the Scala compiler to raise a type error, as the `atm` function

has return type `Unit`. Unfortunately, the Scala compiler will happily convert any value to `Unit` if it occurs at the end of a block. But because `In[Menu]` is a linear type, the linearity checker will notice that the corresponding value is being discarded, and will raise an error:

```
Error: linear term cannot be discarded: cont !! Success(_)
                                        ^^^^^^^^^^^^^^^^^^
```

```
// Authentication request from the user
case class Authenticate(card: String, pin: String, cont: Out[Response])

// Authentication response from the ATM
sealed abstract class Response
case class Failure()                extends Response
case class Success(cont: Out[Menu]) extends Response

// Choices available to authenticated user
sealed abstract class Menu
case class CheckBalance(cont: Out[Balance]) extends Menu
case class Quit()                           extends Menu

// User account balance
case class Balance(amount: BigInt)(cont: Out[Menu]) {
  require(amount >= 0)
}
```

Listing 25: ATM protocol description

### 5.7.2 TLS 1.2 Handshake

(TODO: TLS 1.2 Handshake)

[5]

## 5.8 Conclusion

(TODO: Conclusion)

## 5.9 Further Work

#### Multiparty Session Types for Actor Systems

[9]

[5]The Transport Layer Security (TLS) Protocol - Version 1.2: https://www.ietf.org/rfc/rfc5246.txt

```
def atm(c: In[Authenticate]): Unit = {
  c ? { auth => auth! match {
    case Authenticate(card, pin, cont) if authenticated(card, pin) =>
      cont !! Success(_) ? menu(card)

    case Authenticate(_, _, cont)  =>
      cont ! Failure()
  } }
}

def menu(card: String)(menu: Linear[Menu]) = {
  menu! match {
    case CheckBalance(cont) =>
      cont !! Balance(balance(card))(_) ? menu(card)

    case Quit() => ()
  }
}

def authenticated(card: String, pin: String): Boolean = {
  /* ... */
}

def balance(card: String): BigInt = {
  /* ... */
} ensuring { _ >= 0 }
```

Listing 26: Correct ATM protocol implementation

```
def getAmount(c: Out[Authenticate], card: String, pin: String): Option[BigInt] = {
  c !! Authenticate(card, pin, _) ? { res =>
    res! match {
      case Failure() => None()
      case Success(cont) => cont !! CheckBalance(_) ? {
        case Balance(amount)(cont) =>
          cont ! Quit()
          Some(amount)
      }
    }
  }
} ensuring { _ >= 0 }
```

Listing 27: Correct user protocol implementation

### Affine Types and Borrow Checking

After implementing a very basic borrow checker with static lifetimes within Stainless, (TODO: etc...)

```
def atm(c: In[Authenticate]): Unit = {
  c ? { auth => auth! match {
    case Authenticate(card, pin, cont) if authenticated(card, pin) =>
      // 3. do not wait for a reply to ‘Success‘ message
      cont !! Success(_)

    case Authenticate(_, _, cont) =>
      // 1. does not send back a Failure message

  } }
}

def menu(card: String)(menu: Linear[Menu]): Unit = {
  menu! match {
    case CheckBalance(cont) =>
      cont !! Balance(balance(card))(_) ? menu(card)

      // 2. ‘cont‘ has already been used
      doSomething(cont)

    case Quit() => ()
  }
}
```

Listing 28: Incorrect implementation of the ATM protocol

# 6 Conclusion

(TODO: Conclusion)

## A References

[1] Y. Futamura, “Partial evaluation of computation process&mdash;anapproach to a compiler-compiler,” *Higher Order Symbol. Comput.*, vol. 12, pp. 381–391, Dec. 1999.

[2] G. Agha and P. Thati, *An Algebraic Theory of Actors and Its Application to a Simple Object-Based Language*, pp. 26–57. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[3] K. Honda, V. T. Vasconcelos, and M. Kubo, “Language primitives and type discipline for structured communication-based programming,” in *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP ’98, (London, UK, UK), pp. 122–138, Springer-Verlag, 1998.

[4] P. Wadler, “Propositions as sessions,” in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’12, (New York, NY, USA), pp. 273–286, ACM, 2012.

[5] O. Dardha, E. Giachino, and D. Sangiorgi, “Session types revisited,” in *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP ’12, (New York, NY, USA), pp. 139–150, ACM, 2012.

[6] M. Orchard and N. Yoshida, *Session Types with Linearity in Haskell.* River publishers, 2017.

[7] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen, “Session types for rust,” in *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, (New York, NY, USA), pp. 13–22, ACM, 2015.

[8] A. Scalas and N. Yoshida, “Lightweight Session Programming in Scala,” in *30th European Conference on Object-Oriented Programming (ECOOP 2016)* (S. Krishnamurthi and B. S. Lerner, eds.), vol. 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 21:1–21:28, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016.

[9] R. Neykova and N. Yoshida, “Multiparty session actors,” *Logical Methods in Computer Science*, vol. 13, no. 1, 2017.