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 simplifying a program by evaluating it as much as possible, even in the presence of unknown inputs is often referred to as *symbolic evaluation.* The idea is to execute the program over symbolic inputs values while maintaining a *path condition*, ie. a quantifier-free formula in conjunctive normal form that represents the constraints over the inputs we have discovered so far, as well as the branching decisions made. The actual implementation also keeps track of bound variables in a specific way for performance.

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 current path condition, the formula `x <= 0`, `x > 0` is unsatisfiable. As a result of the symbolic evaluation, the resulting program is now equivalent to the code featured in Listing 2.

```
// 1. PC = true
assume(x <= 0)
// 2. PC = x <= 0
val y = if (x > 0) {
  // 2. PC = x <= 0 && x > 0
  x
} else {
  // 3. PC = x <= 0
  -x
}
// 4. PC = y == -x && x <= 0
```

Listing 1: Initial program with path condition

```
assume(x <= 0)
val y = -x
// PC = x <= 0 && y == -x
```

Listing 2: Simplified program

Here is how the *PureScala* program above 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))
}

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 here. We look at its body and find a toplevel "if-else" statement. We now ask ourselves whether or not its condition can be reduced to a boolean literal under the current path condition, ie. whether `list.isInstanceOf[Nil]` evaluates to `true` or `false` given that `list = Cons("a" -> 1, ...)`. It is clear here that this expression reduces to `false`, and we can thus unfold `foldLeft`'s definition and continue:

   ```
   val res = if (Cons("a" -> 1, Cons("b" -> 2, ...)).isInstanceOf[Nil[A]]) {
     map
   } else {
     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) => ...)(z, x), (acc, kv) => ...)
   }
   ```

4. Once again, we see that the condition evaluates to `false`, and we thus replace the "if-else" statement by its "else" branch (note: our implementation actually performs this step and the previous one in go for performance):

```
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) => ...)
}
```

5. 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) => ...)
}
```

6. We now inline both bindings:

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

7. And we now evaluate the arguments supplied to `foldLeft`, giving us:

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

8. We now repeat steps 3 to 7 until we have traversed the entire list, leaving us with:

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

9. We once again look ahead and see that, this time, the condition will evaluate to `true`, and can thus unfold `foldLeft` once again, getting us to:

```
val res = if (Nil().isInstanceOf[Nil[A]]) {
  map.updated("a", 1).updated("b", 2).updated("c", 3).updated("d", 4)
} else {
  val x  = Nil().asInstanceOf[Cons[A]].head
  val xs = Nil().asInstanceOf[Cons[A]].tail
  foldLeft(xs, ((acc, kv) => ...)(z, x), (acc, kv) => ...)
}
```

10. This time we pick the "then" branch, and end up the following expression, which we cannot reduce further, as `map` is abstract and `updated` is a special language construct which can only be evaluated when applied to a concrete, finite map:

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

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 Δ, and a formula e that is satisfiable under Δ by $e \in \Delta$. 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 Δ by $[\![\, e \,;\, \Delta \,]\!] \longrightarrow e'$. Given c, v two expressions, and `x` a variable, we note $\Delta \cup c$ the addition of c to the path condition Δ, 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 Δ is captured by `e` $\Downarrow \Delta$. Figure 1 shows the operation semantics followed by our implementation, expressed in those terms.

$$\frac{e \in \Delta}{\llbracket e\,;\,\Delta \rrbracket \longrightarrow \texttt{true}} \tag{1}$$

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

$$\frac{}{\llbracket \lambda x_1, \ldots, x_n.\, e\,;\,\Delta \rrbracket \longrightarrow \lambda \llbracket x_1\,;\,\Delta \rrbracket, \ldots, \llbracket x_n\,;\,\Delta \rrbracket.\, \llbracket e\,;\,\Delta \rrbracket} \tag{3}$$

$$\frac{}{\llbracket l\ \texttt{==>}\ r\,;\,\Delta \rrbracket \longrightarrow \llbracket \neg l\ \texttt{||}\ r\,;\,\Delta \rrbracket} \tag{4}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow \texttt{true}}{\llbracket \texttt{if (}c\texttt{)}\ t\ \texttt{else}\ e\,;\,\Delta \rrbracket \longrightarrow \llbracket t\,;\,\Delta \cup c \rrbracket} \tag{5}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow \texttt{false}}{\llbracket \texttt{if (}c\texttt{)}\ t\ \texttt{else}\ e\,;\,\Delta \rrbracket \longrightarrow \llbracket e\,;\,\Delta \cup c \rrbracket} \tag{6}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow c' \qquad \llbracket t\,;\,\Delta \cup c' \rrbracket \longrightarrow t' \qquad \llbracket e\,;\,\Delta \cup \neg c' \rrbracket \longrightarrow e' \qquad t' = e'}{\llbracket \texttt{if (}c\texttt{)}\ t\ \texttt{else}\ e\,;\,\Delta \rrbracket \longrightarrow t'} \tag{7}$$

$$\frac{\llbracket c\,;\,\Delta \rrbracket \longrightarrow c' \qquad \llbracket t\,;\,\Delta \cup c' \rrbracket \longrightarrow t' \qquad \llbracket e\,;\,\Delta \cup \neg c' \rrbracket \longrightarrow e' \qquad t' \neg = e'}{\llbracket \texttt{if (}c\texttt{)}\ t\ \texttt{else}\ e\,;\,\Delta \rrbracket \longrightarrow \texttt{if (}c'\texttt{)}\ t'\ \texttt{else}\ e'} \tag{8}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow \texttt{true}}{\llbracket \texttt{assume(}p\texttt{, }e\texttt{)}\,;\,\Delta \rrbracket \longrightarrow \llbracket e\,;\,\Delta \rrbracket} \tag{9}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow \texttt{false}}{\llbracket \texttt{assume(}p\texttt{, }e\texttt{)}\,;\,\Delta \rrbracket \longrightarrow \texttt{assume(false, }\llbracket e\,;\,\Delta \rrbracket} \tag{10}$$

$$\frac{\llbracket p\,;\,\Delta \rrbracket \longrightarrow p'}{\llbracket \texttt{assume(}p\texttt{, }e\texttt{)}\,;\,\Delta \rrbracket \longrightarrow \texttt{assume(}p'\texttt{, }\llbracket e\,;\,\Delta \cup p' \rrbracket\texttt{)}} \tag{11}$$

$$\frac{\texttt{T}_2\texttt{: ADTType} \qquad \neg\texttt{isSort(T}_2\texttt{)}}{\llbracket \texttt{C(T}_1\texttt{, }a_1, \ldots, a_n\texttt{).isInstanceOf[T}_2\texttt{]}\,;\,\Delta \rrbracket \longrightarrow \texttt{T}_1\texttt{.id == T}_2\texttt{.id}} \tag{12}$$

$$\frac{\texttt{T: ADTType} \qquad \texttt{isSort(T)}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow \texttt{true}} \tag{13}$$

$$\frac{\texttt{T: ADTType} \qquad \llbracket e\,;\,\Delta \rrbracket \longrightarrow e' \qquad \texttt{isInstanceOf(e', T, }\Delta\texttt{) == Some(}b\texttt{)}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow b} \tag{14}$$

$$\frac{\texttt{T: ADTType} \qquad \llbracket e\,;\,\Delta \rrbracket \longrightarrow e' \qquad \texttt{isInstanceOf(}e'\texttt{, T, }\Delta\texttt{) == None}}{\llbracket e\texttt{.isInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow e'\texttt{.isInstanceOf[T]}} \tag{15}$$

$$\frac{\texttt{T: ADTType}}{\llbracket e\texttt{.asInstanceOf[T]}\,;\,\Delta \rrbracket \longrightarrow \llbracket e\,;\,\Delta \rrbracket\texttt{.asInstanceOf[T]}} \tag{16}$$

Figure 1: Operational semantics of the symbolic evaluator

$$\frac{}{\llbracket\, \texttt{let x:T =}\ v\ \texttt{in}\ e\,;\,\Delta\,\rrbracket \longrightarrow \llbracket\, e[\mathsf{x}/v]\,;\,\Delta\,\rrbracket} \tag{17}$$

$$\frac{\llbracket\, f\,;\,\Delta\,\rrbracket \longrightarrow \lambda\mathsf{x}_1\texttt{:}\mathsf{T}_1\texttt{,...,}\mathsf{x}_n\texttt{:}\mathsf{T}_n\,.\,b \qquad \llbracket\, e_i\,;\,\Delta\,\rrbracket \longrightarrow e_i',\ i \in \{1\ldots n\}}{\llbracket\, f(e_1,\ldots,e_n)\,;\,\Delta\,\rrbracket \longrightarrow \llbracket\, b[\mathsf{x}_1/e_1',\ldots,\mathsf{x}_n/e_n']\,;\,\Delta\,\rrbracket} \tag{18}$$

$$\frac{\llbracket\, f\,;\,\Delta\,\rrbracket \longrightarrow f' \qquad \llbracket\, e_i\,;\,\Delta\,\rrbracket \longrightarrow e_i',\ i \in \{1\ldots n\}}{\llbracket\, f(e_1,\ldots,e_n)\,;\,\Delta\,\rrbracket \longrightarrow f'(e_1',\ldots,e_n')} \tag{19}$$

$$\frac{\neg\,\texttt{isRecursive(id)}\quad \texttt{id.params} = \langle \mathsf{x}_1,\ldots,\mathsf{x}_n\rangle \quad \llbracket\, e_i\,;\,\Delta\,\rrbracket \longrightarrow e_i',\ i \in \{1\ldots n\}}{\llbracket\, \texttt{id}(e_1,\ldots,e_n)\,;\,\Delta\,\rrbracket \longrightarrow \llbracket\, \texttt{id.body}[\mathsf{x}_1/e_1',\ldots,\mathsf{x}_n/e_n']\,;\,\Delta\,\rrbracket} \tag{20}$$

$$\frac{\texttt{id.body} \Downarrow \Delta \uplus \{\, \mathsf{x}_i \mapsto e_i' \mid 1 \le i \le n \,\} \quad \texttt{id.params} = \langle \mathsf{x}_1,\ldots,\mathsf{x}_n\rangle \quad \llbracket\, e_i\,;\,\Delta\,\rrbracket \longrightarrow e_i',\ \ i \in \{1\ldots n\}}{\llbracket\, \texttt{id}(e_1,\ldots,e_n)\,;\,\Delta\,\rrbracket \longrightarrow \llbracket\, \texttt{id.body}[\mathsf{x}_1/e_1',\ldots,\mathsf{x}_n/e_n']\,;\,\Delta\,\rrbracket} \tag{21}$$

$$\frac{\llbracket\, e_i\,;\,\Delta\,\rrbracket \longrightarrow e_i',\ i \in \{1\ldots n\}}{\llbracket\, \texttt{id}(e_1,\ldots,e_n)\,;\,\Delta\,\rrbracket \longrightarrow \texttt{id}(e_1',\ldots,e_n')} \tag{22}$$

$$\frac{e = \texttt{if}\ \texttt{(}c\texttt{)}\ t_1\ \texttt{else}\ t_2 \quad \llbracket\, c\,;\,\Delta\,\rrbracket \longrightarrow r \quad r \in \{\texttt{true},\texttt{false}\}}{\texttt{e} \Downarrow \Delta} \tag{23}$$

$$\frac{(\texttt{cons},(e_1,\ldots,e_n)) = \texttt{deconstruct(e)} \quad \llbracket\, e_i\,;\,\Delta\,\rrbracket \longrightarrow e_i',\ i \in \{1\ldots n\}}{\llbracket\, e\,;\,\Delta\,\rrbracket \longrightarrow \texttt{cons}(e_1',\ldots,e_2')} \tag{24}$$

Figure 1: Operational semantics of the symbolic evaluator

3.4 Implementation

We have implemented the symbolic evaluator described above as a new, optional, simplification pass within Inox' 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.

3.5 Case Study: Key-value store

Listing 4 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 3.

```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 3: Example program

While Stainless is unable to prove the verification condition associated with `lemma`'s post-condition, 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 = (
  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 4: Key-value store implementation

3.6 Conclusion

(TODO: Conclusion)

3.7 Further Work

3.7.1 Function Annotations

(TODO: Further work)

4 Verifiying Actor Systems

4.1 Motivation

(TODO: Motivation)

4.2 The Actor Model

(TODO: Actor Model)

4.3 Our Framework

Message

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

```
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.

```
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.

```
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.

```
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
```

4.4 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 $\emptyset_n := \texttt{ActorContext}(n, \texttt{Nil})$.

4.5 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 $\texttt{inv}(s)$ holds, then $\texttt{inv}(s\texttt{.step}(n, m))$ 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 5.

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.

4.6 Case studies

4.6.1 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 primary counter whose reference is `Primary()`, and a backup counter whose reference is `Backup()`.

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

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

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

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

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

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

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

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

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

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

Listing 5: Invariant preservation theorem in PureScala

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 6.

```scala
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 6: Increment-based replicated counter implementation

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

```scala
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 7: Increment-based replicated counter invariant

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 that are yet to be delivered to the backup actor.

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

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

Listing 8: Replicated counter theorem (increment)

4.6.2 Delivery-based Replicated Counter

Listing 9 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.

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

case class 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 9: Delivery-based Replicated counter implementation

The invariant now reads slightly differently, as can be seen in Listing 10.

4.6.3 Lock Service

Listing 11 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 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.

```
def invariant(s: ActorSystem): Boolean = {
  validBehaviors(s)                             &&
  noMsgsToSelf(Primary()).isEmpty               &&
  noMsgsToSelf(Backup()).isEmpty                &&
  noMsgsToSelf(Backup() -> Primary()).isEmpty && {
    val PrimBehav(p) = s.behaviors(Primary())
    val BackBehav(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 10: Delivery-based replicated counter invariant

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 12.

(TODO: Lock service invariant proof)

4.7 Spawning Actors

(TODO: Name Uniqueness)

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]
)

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.

Let's now update the case study in Listing 6 to accommodate these changes, while noting that we are not making use of this new feature yet. Only the two `ActorRef` definitions need to be touched, becoming:

```
case class Primary() extends ActorRef("primary", None())
case class Backup()  extends ActorRef("backup", None())
```

Unfortunately, when we now feed the updated benchmark to Stainless, the latter is be unable to prove the very same theorem it previously had no issue whatsoever with. All is not lost though, as turning on the symbolic evaluator described in Section 3 enables Stainless to verify the program in less than 10 seconds.

Listing **??** defines a simple system with a dynamic topology, where one actor, deemed `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.

4.8 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 13 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.

(TODO: Explain shim)

To this end, one must define a subclass of `ActorSystem`, and 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 14 shows such an implementation for the replicated counter described in Section 4.6.1.

4.9 Conclusion

(TODO: Conclusion)

4.10 Further Work

(TODO: Further work)

```
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 11: Lock service implementation

```scala
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 12: 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 Wrapper(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 name: String) {
  lazy val system = actor.ActorSystem(name)

  def spawn(behavior: Behavior, name: String): actor.ActorRef = {
    system.actorOf(actor.Props(new Wrapper(behavior)), name = name)
  }

  def run(): Unit
}
```

Listing 13: Akka shim for our Actor system 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 14: Akka shim for our Actor system framework

5 Biparty Communication Protocols

5.1 Motivation

5.1.1 Session Types

(TODO: Session Types)

5.1.2 Value-level Encoding of Sessions

(TODO: Value-level Encoding of Sessions)

5.2 Linear Types in Stainless

We now discuss our implementation of linear types in Stainless. One thing to note is that because the AST we are working with within 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.

Introducing Linear Types

We introduce a way 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 15 shows the full definitions. Because those will be extracted in a specific way, they are marked `@ignore`.

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)`.

```
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 15: Linear wrapper for Scala types and values

- The term is supplied as an argument to a constructor, eg. given `case class C(x: Linear[A])`, we have $e =$ `C(t)`.

We now must ensure that no linear term can be *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 context so that we can disallow subsequent uses of those terms. We will denote this context by Δ. (TODO: Figure **??** presents the type-checking rules.)

Preventing Contraction

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 **??** shows such a use-case.

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' 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.

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

Listing 16: Usage of a linear variable in a function's precondition

Linear Data Types

(TODO: Discuss which types can contain linear fields)

Marking the Current Object as Linear

(TODO: Discuss @linear annotation)

Higher-Order Linear Functions

(TODO: Discuss syntax for linear lambdas)

5.3 Sessions Library in PureScala

Listing 17 shows the PureScala implementation of the *lchannels* library. For the purpose of verification, we do not need a full-fledged implementation, 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 our implementation.

5.4 Case Studies

5.4.1 ATM Protocol

Let's consider a protocol involving an ATM and its user.

1. The user authenticates herself by sending the ATM both her card number and PIN.
2. 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 18 shows the encoding of such a specification using the library described in Section 5.3. Listing 19 shows the corresponding valid implementation. At last, Listing 20 shows an invalid implementation of the protocol that would still verify without the linearity checker. The three mistakes are discussed below.

1. If we make a terrible mistake while implemented the `atm` function by not doing anything with its linear parameter `c`, eg. just left its body empty, we would be greeted with the following error:

```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 17: Sessions library in PureScala

```
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)
```

```
//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])
```

Listing 18: ATM protocol description

```
def atm(c: In[Authenticate]) = {
  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() => ()
  }
}

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

@extern
def balance(card: String): BigInt = {
  /* ... */
}
```

Listing 19: ATM protocol implemenation

```
                                                          ^^^^
```

2. In case we forget to send back a failure notification when the authentication fails. The linearity checker will realize 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(_)
                                        ^^^^^^^^^^^^^^^^^^
```

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

      // 3. do not wait for reply to ‘Success‘ message
      cont !! Success(_)

    case Authenticate(_, _, cont) =>

      // 1. forgot the 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 20: Wrong implementation of the ATM protocol

5.4.2 TLS 1.2 Handshake

(TODO: TLS 1.2 Handshake)

5.5 Conclusion

(TODO: Conclusion)

5.6 Further Work

(TODO: Further Work)

6 Conclusion

(TODO: Conclusion)

A References

[1] S. Yasutake and T. Watanabe, “Actario: A framework for reasoning about actor systems,” in *Workshop on Programming based on Actors, Agents, and Decentralized Control*, AGERE 2015, 2015.

[2] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, “Verdi: A framework for implementing and formally verifying distributed systems,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’15, (New York, NY, USA), pp. 357–368, ACM, 2015.

[3] 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.

[4] C. Hewitt and H. Baker, “Applications of the laws for communicating parallel processes,” in *IFIP Working Conf. on the Formal Desc. of Prog. Concepts*, 1977.

[5] J. He, P. Wadler, and P. Trinder, “Typecasting actors: From akka to takka,” in *Proceedings of the Fifth Annual Scala Workshop*, SCALA ’14, (New York, NY, USA), pp. 23–33, ACM, 2014.

[6] P. Haller and A. Loiko, “Lacasa: Lightweight affinity and object capabilities in scala,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), pp. 272–291, ACM, 2016.

[7] P. Haller and F. Sommar, “Towards an Empirical Study of Affine Types for Isolated Actors in Scala,” *ArXiv e-prints*, Apr. 2017.

[8] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986.

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

[10] 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.

[11] 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.

[12] P. Wadler, “Linear types can change the world!,” in *PROGRAMMING CONCEPTS AND METHODS*, North, 1990.

[13] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[14] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[15] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *CoRR*, vol. abs/1610.00502, 2016.

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, *Conflict-Free Replicated Data Types*, pp. 386–400. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011.

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

[19] P. Wadler, "Propositions as sessions," *ACM SIGPLAN Notices*, vol. 47, no. 9, pp. 273–286, 2012.

[20] P. Wadler, "Linear types can change the world,"

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

[22] P. Suter, A. S. Köksal, and V. Kuncak, "Satisfiability modulo recursive programs," in *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, (Berlin, Heidelberg), pp. 298–315, Springer-Verlag, 2011.

[23] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter, "An overview of the leon verification system: Verification by translation to recursive functions," in *Proceedings of the 4th Workshop on Scala*, SCALA '13, (New York, NY, USA), pp. 1:1–1:10, ACM, 2013.

[24] N. Voirol and V. Kuncak, "Automating verification of functional programs with quantified invariants," tech. rep., 2016.