Systems Modeling 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

Contents

1 Introduction

(TODO: Introduction)

2 Program Verification with Stainless

(TODO: Describe Stainless/Inox pipeline)

3 Symbolic Partial Evaluation of PureScala Programs

3.1 Motivation

Let's consider the following program:

```
def foldLeft[A, B](list: List[A], z: B)(f: (B, A) => B): B = list match {
  case Nil     => z
  case 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)) if !acc.contains(k) => acc.updated(k, v)
    case (acc, _)                          => acc
  }
}

def test(map: Map[String, Int], x: Int, y: Int): (Int, Map[String, Int]) = {
  require(!map.contains("a") && map.contains("b"))

  val xs  = ("a", x) :: ("b", y) :: Nil
  val res = insert(xs, map)

  res("a") == x && res("b") == map("b")
} holds
```

Listing 1: Example program

Although this program is correct, Stainless is unfortunately unable to prove the verification condition corresponding to `test`'s postcondition on its own. To remedy to this problem, one could refine the program by adding contracts to the `foldLeft` and `insert` functions, or introduce additional lemmas. While this is a reasonable way of going at the problem, we will discuss in this chapter an alternative way of proving such programs correct. The main insight we will make use of is that, although none of the arguments to `insert` are fully known within the `test` function, the knowledge we have of `map` from `test`'s precondition combined with the known structure of `xs` is enough to *partially evaluate* in a *symbolic* way the body of the function, yielding a much simpler expression which in turn makes the postcondition much easier to verify. Before we discuss this idea in greater details, let's look at how one would "manually" prove correct the above program:

1. Let's consider the call to `insert` in the `test` function:

   ```
   insert(xs, map)
   ```

2. We start by inlining the definition of `xs`:

```
insert(("a", x) :: ("b", y) :: Nil, map)
```

3. We then unfold the definition of `insert`:

```
foldLeft(("a", x) :: ("b", y) :: Nil, map) {
  case (acc, (k, v)) if !acc.contains(k) => acc.updated(k, v)
  case (acc, _)                          => acc
}
```

4. We now unfold the definition of `foldLeft`:

```
val f = {
  case (acc, (k, v)) if !acc.contains(k) => acc.updated(k, v)
  case (acc, _)                          => acc
}

(("a", x) :: ("b", y) :: Nil) match {
  case Nil     => map
  case x :: xs => foldLeft(xs, f(z, x))(f)
}
```

   For legibility, we have assigned the supplied match expression to a variable instead of inlining it straight away.

5. We select the second case of the pattern match:

```
val f = {
  case (acc, (k, v)) if !acc.contains(k) => acc.updated(k, v)
  case (acc, _)                          => acc
}

foldLeft("b" -> y :: Nil, f(map, ("a", x)))(f)
```

6. We now evaluate the first two arguments of the call to `foldLeft`. The first one stays as is as it is already in normal form. The second argument `f(map, ("a", x))` becomes:

```
(map, ("a", x)) match {
  case (acc, (k, v)) if !acc.contains(k) => acc.updated(k, v)
  case (acc, _)                          => acc
}
```

7. Recall the precondition of the `test` function: `!map.contains("a") && map.contains("b")`. When we inline the scrutinee into the guard of the first case, we obtain `!map.contains("a")`, which by the aforementioned precondition we can deduce is **`true`**. We thus select the first case, and inline the scrutinee within it, yielding:

```
map.updated("a", x)
```

8. We are now left with this expression:

```
foldLeft("b" -> y :: Nil, map.updated("a", x))(f)
```

9. We can now unfold `foldLeft` one more time:

```
(("b", y) :: Nil) match {
  case Nil     => map.updated("a", x)
  case x :: xs => foldLeft(xs, f(z, x))(f)
}
```

10. Once again, we recurse in the second case of the match expression:

```
foldLeft(Nil, f(map.updated("a", x), ("b", y)))(f)
```

11. We now unfold the definition of `f` in `f(map.updated("a", x), ("b", y))`, yielding:

```
(map.updated("a", x), ("b", y)) match {
  case (acc, (k, v)) if !acc.contains(k) => acc.updated(k, v)
  case (acc, _)                          => acc
}
```

12. Recall once again the precondition of the `test` function: `!map.contains("a") && map.contains("b")`. This time, the guard reads as `!map.contains("b")`, which by precondition we can deduce is **`false`**. We thus select the second case, and inline the scrutinee within it, yielding:

```
map.updated("a", x)
```

13. We are now left with this expression:

```
foldLeft(Nil, map.updated("a", x))(f)
```

14. We can now unfold `foldLeft` one last time:

```
Nil match {
  case Nil     => map.updated("a", x)
  case x :: xs => foldLeft(xs, f(z, x))(f)
}
```

15. This time, we recurse in the first case, which leaves us with:

```
map.updated("a", x)
```

16. At last, we substitute this expression for the call to `insert` within `test`'s body, resulting in:

```
def test(map: Map[String, Int], x: Int, y: Int): (Int, Map[String, Int]) = {
  require(!map.contains("a") && map.contains("b"))

  val xs  = ("a", x) :: ("b", y) :: Nil
  val res = map.updated("a", x)

  res("a") == x && res("b") == map("b")
} holds
```

17. 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.

18. If one knows the semantics of `Map`, it becomes fairly obvious that the postcondition is valid, for any initial value of `map`, `x` and `y`.

We want to draw the reader's attention to the fact that the actual values `x` and `y` did not play any role in this process. Moreover, while the exact content of the `Map` underlying the `map` variable was unknown, we knew enough about it from the precondition of `test` to successfully reduce the program to a simple expression. We formalize this whole process in the next section.

3.2 Symbolic Partial Evaluation

Let's first get a couple of definitions out of the way:

- *Partial evaluation* [1] is the process of executing a program in the presence of partial inputs, effectively specializing it program that is more performant or easier to reason about than the initial one.

- *Symbolic evaluation* [2] is the process of gathering knowledge about symbolic values in a program by executing it, in order to verify whether or not a given program satisfies some properties, such as the absence of divisions by zero, or the preservation of some invariant during execution [3].

- A *path condition* is a quantifier-free formula in conjunctive normal form which encodes the knowledge about the symbolic values encountered so far that the evaluator has gathered along the way.

Combining both techniques into a single evaluation procedure, ie. a procedure which partially evaluates a program with symbolic inputs while maintaining a *path condition*, has been shown to yield a much more powerful evaluator than each procedure on its own, and to increase both performance and reasoning abilities of verifications systems [4]. We call such a procedure a

symbolic partial evaluator. The key benefit of this technique is that, whenever the evaluator encounters a boolean expression while evaluating a program, it can make use of the path condition to hopefully determine whether the expression is **`true`** or **`false`**.

Listing 2 shows a simple program annotated with the current path condition at various points of execution. This path condition enables the partial evaluator to reduce the whole expression to the **`else`** branch of the conditional expression because, under the path condition at step 2, the condition `x > 0` is provably **`false`**.

```
// 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 2: Program annotated with path conditions and result of its symbolic partial evaluation

3.3 Semantics

We subsequently denote the path condition by Δ, an boolean expression e that evaluates to **`true`** under Δ by $e \in \Delta$. and a variable v that is bound to an expression e under Δ 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 Δ 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 Δ, and $\Delta \uplus x \mapsto v$ the addition of the binding `val x =` v. At last, the fact that a function invocation f can be safely unfolded under path condition Δ is captured by `f` $\Downarrow \Delta$. Figure 1 shows the operational semantics followed

by our implementation, expressed in those terms. We informally discuss the most interesting of these rules below.

Evaluation of Boolean expressions

To determine the value of an expression c of type `Boolean`, the evaluator proceeds as follows:

1. If c is a boolean literal, the evaluator returns c.
2. Otherwise, it checks whether the formula $\Delta_{CNF} \implies c$ is valid, where Δ_{CNF} is the CNF formula corresponding to the path condition Δ:
   (a) If it is valid, the evaluator returns **`true`**.
   (b) Otherwise, it checks whether the formula $\Delta_{CNF} \implies \neg c'$ is valid, in which case it returns **`false`**.
   (c) If none are valid, the expression goes through the other evaluation rules based on its structure.

Evaluation of conditional expressions

When encountering a conditional expression $E =$ `if (`c`)` t `else` e, under a path condition Δ, the evaluator proceeds as follows:

1. It partially evaluates c, resulting in an expression c'.
2. If c' is the boolean literal **`true`**, it recurses into the `then` branch of the conditional expression under the path condition $\Delta \cup c'$.
3. If c' is the boolean literal **`false`**, it recurses into the `else` branch under the path condition $\Delta \cup \neg c'$.
4. If c' is not a literal, it evaluates both branches under their respective path conditions. We denote the result of their evaluation respectively by t' and e'. It then returns the expression `if (`c'`)` t' `else` e'.

This procedure is formalized in rules 10, 11, and 12 of Figure **??**.

Evaluation of pattern matches

Pattern matches are converted to an equivalent conditional expression with explicit type checks, casts, and field accesses, before being fed back to the evaluator. The corresponding rule (21) can be found in Figure **??**.

Evaluation of function invocations

Finding good rules for the evaluation of function invocations, and more specifically recursive one, was by far the most challenging and subtle aspect of this work. Indeed, the naive approach of unfolding every function call that is encountered during evaluation send the evaluator into an infinite loop. Instead, the evaluation of a function call proceeds as follows:

1. The arguments are recursively evaluated.
2. The precondition of the function is then checked against the evaluated arguments. If is is invalid, the evaluator stops there and replace the original arguments to the invocation by their counterpart computed in the step 1.
3. If the function that is invoked is not recursive, the invocation is unfolded, and the evaluator recurses over the resulting expression.
4. If the function is recursive, some more care is needed. Indeed, if one were to unfold every calls to a recursive function, this would often send the evaluator into an infinite loop, even when the function has been deemed terminating. To understand why, let's consider the following example:

```
def fact(n: Int): Int = {
  require(n >= 1)
  if (n == 1) 1 else n * fact(n - 1)
}

def test(n: Int): Int = fact(n)
```

   If we were to unfold `fact` in function `test` and continue evaluation from there, because we do not know anything about the value of `n`, the evaluator would recursive into both branches of the conditional expression. In the **`else`** branch, the recursive call to `fact` would itself be unfolded. Because we still do not know anything about the value of `n - 1`, both branches of the expression resulting from unfolding `fact` once more would also be explored, and so on. This obviously send the evaluator into an infinite loop, and prevent termination of the evaluation. On the other hand, if we knew more about `n`, eg. that `n >= 3`, the evaluator would be able to unfold `fact` a number of times already, eg. twice resulting in `n * n * fact(n - 2)`.

   In other words, we only want to unfold an invocation to a recursive function if we can determine, under the current path condition, whether or not a subsequent, valid recursive call will definitely be reached or not. We denote such a function invocation by $f(e_1, \ldots, e_2) \Downarrow \Delta$. We discuss the impact of this heuristic on termination in more details in Section 3.7.
5. If the function invocation cannot be productively unfolded, the evaluator stops there and replace the original arguments to the invocation by their counterpart computed in step 1.

The corresponding rules (25, 26, and 27) can be found in Figure **??**.

$$\frac{v \mapsto_{\Delta} e}{[\![v \,;\, \Delta]\!] \longrightarrow e} \tag{1}$$

$$\frac{e \in \Delta}{[\![e \,;\, \Delta]\!] \longrightarrow \texttt{true}} \tag{2}$$

$$\frac{\neg e \in \Delta}{[\![e \,;\, \Delta]\!] \longrightarrow \texttt{false}} \tag{3}$$

$$\frac{[\![e \,;\, \Delta]\!] \longrightarrow e'}{[\![\lambda x_1, \ldots, x_n . e \,;\, \Delta]\!] \longrightarrow \lambda x_1, \ldots, x_n . e'} \tag{4}$$

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

$$\frac{[\![e_1 \,;\, \Delta]\!] \longrightarrow \texttt{false}}{[\![e_1 \ \texttt{\&\&} \ldots \texttt{\&\&} \ e_n \,;\, \Delta]\!] \longrightarrow \texttt{false}} \tag{6}$$

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

$$\frac{[\![e_1 \,;\, \Delta]\!] \longrightarrow \texttt{true}}{[\![e_1 \ \texttt{||} \ldots \texttt{||} \ e_n \,;\, \Delta]\!] \longrightarrow \texttt{true}} \tag{8}$$

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

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

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

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

Figure 1: Operational semantics of the symbolic partial evaluator

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

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

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

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

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

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

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

$$\frac{\texttt{T: ADTType}}{[\![e\texttt{.asInstanceOf[T]}\,;\,\Delta]\!] \longrightarrow [\![e\,;\,\Delta]\!]\texttt{.asInstanceOf[T]}} \tag{20}$$

$$\frac{e\texttt{.getType} = \texttt{MatchExpr} \quad [\![\texttt{matchToIfThenElse}(e)\,;\,\Delta]\!] \longrightarrow e'}{[\![e\,;\,\Delta]\!] \longrightarrow e'} \tag{21}$$

Figure 1: Operational semantics of the symbolic partial evaluator

$$\frac{}{[\![\, \texttt{let x:T = } v \texttt{ in } e \,;\, \Delta \,]\!] \longrightarrow [\![\, e[\texttt{x}/v] \,;\, \Delta \,]\!]} \tag{22}$$

$$\frac{[\![\, f \,;\, \Delta \,]\!] \longrightarrow \lambda \texttt{x}_1\texttt{:T}_1\texttt{,...,x}_n\texttt{:T}_n\texttt{.}\, b \qquad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow e'_i,\ i \in \{1 \dots n\}}{[\![\, f(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow [\![\, b[\texttt{x}_1/e'_1, \dots, \texttt{x}_n/e'_n] \,;\, \Delta \,]\!]} \tag{23}$$

$$\frac{[\![\, f \,;\, \Delta \,]\!] \longrightarrow f' \qquad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow e'_i,\ i \in \{1 \dots n\}}{[\![\, f(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow f'(e'_1, \dots, e'_n)} \tag{24}$$

$$\frac{\neg\, \texttt{f.isRecursive} \quad \texttt{f.params} = \langle \texttt{x}_1, \dots, \texttt{x}_n \rangle \quad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow e'_i,\ i \in \{1 \dots n\} \quad [\![\, \texttt{f.pre}(e'_1, \dots, e'_n) \,;\, \Delta \,]\!] \longrightarrow \texttt{true}}{[\![\, \texttt{f}(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow [\![\, \texttt{f.body}[\texttt{x}_1/e'_1, \dots, \texttt{x}_n/e'_n] \,;\, \Delta \,]\!]} \tag{25}$$

$$\frac{\texttt{f.params} = \langle \texttt{x}_1, \dots, \texttt{x}_n \rangle \quad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow e'_i,\ i \in \{1 \dots n\} \quad [\![\, \texttt{f.pre}(e'_1, \dots, e'_n) \,;\, \Delta \,]\!] \longrightarrow \texttt{true} \quad \texttt{f}(e'_1, \dots, e'_n) \Downarrow \Delta}{[\![\, \texttt{f}(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow [\![\, \texttt{f.body}[\texttt{x}_1/e'_1, \dots, \texttt{x}_n/e'_n] \,;\, \Delta \,]\!]} \tag{26}$$

$$\frac{[\![\, e_i \,;\, \Delta \,]\!] \longrightarrow e'_i,\ i \in \{1 \dots n\}}{[\![\, \texttt{f}(e_1, \dots, e_n) \,;\, \Delta \,]\!] \longrightarrow \texttt{f}(e'_1, \dots, e'_n)} \tag{27}$$

$$\frac{(\texttt{cons}, (e_1, \dots, e_n)) = \texttt{deconstruct(e)} \quad [\![\, e_i \,;\, \Delta \,]\!] \longrightarrow e'_i,\ i \in \{1 \dots n\}}{[\![\, e \,;\, \Delta \,]\!] \longrightarrow \texttt{cons}(e'_1, \dots, e'_n)} \tag{28}$$

Figure 1: Operational semantics of the symbolic partial evaluator

3.4 Implementation

We have implemented the symbolic partial evaluator described above within Stainless[1] in about 900 lines of code, leveraging Inox to check for the validity of the formulas underlying the path condition.

In order to invoke the partial evaluator on a function definition, a user must supply the new `-partial-eval` flag and annotate the function with the `@partialEval` annotation. Any such function will then be partially evaluated, and its body will be substituted by the resulting expression. Enabling the `partial-eval` debug section instructs Stainless to print the result of the evaluation. This effectively enable users to use Stainless as a symbolic partial evaluator for PureScala program.

3.5 Case Study: Key-value Store Algebra

In Listing 5, we define a monadic domain-specific language to manipulate a key-value store, along with a tracing interpreter which maps the operations over a PureScala `Map`. A simple program which makes use of this DSL to insert a list of concrete values into the store is shown in Listing 4. Without additional lemmas, Stainless is unfortunately unable to verify the postcondition of the `result` function. On the other hand, turning on the partial evaluator turns the associated verification condition into a much more concrete form, which in turns allow the verification to promptly go through. We will note that, although the exact amount of *fuel* we give to the interpreter is unknown, the precondition of the `result` function constrains it to be greater than 10. Because the program expressed in that DSL requires less than 10 steps to be fully interpreted, the evaluator is able to reduce it to normal form.

```
val (res, trace) = (
  map.updated("foo", "bar").updated("toto", "tata")("foo"),
  Get("foo") :: Put("toto", "tata") :: Put("foo", "bar") :: init
)

res == Some("bar") &&
trace.take(3) == Get("foo") :: Put("toto", "tata") :: Put("foo", "bar") :: Nil()
```

Listing 3: Partially evaluated verification condition

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

```
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)
  }
}

@partialEval
def result(map: Map[String, String], init: List[Label], fuel: BigInt) = {
  require(fuel > 10)
  interpret(program)(map, init, fuel)
} ensuring { case (res, trace) => prop(res, trace) }

@inline
def prop(res: Option[String], trace: List[Label]) = {
  res == Some("bar") &&
  trace.take(3) == List(
    Label.Get("foo"),
    Label.Put("toto", "tata"),
    Label.Put("foo", "bar")
  )
}
```

Listing 4: Example program

```scala
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], fuel: BigInt): (Option[String], List[Label]) = {
  require(fuel >= 0)
  decreases(fuel)

  op match {
    case Pure(value) =>
      (value, trace)

    case Get(key, next) if fuel > 0 =>
      interpret(next(kv.get(key)))(kv, Label.Get(key) :: trace, fuel - 1)

    case Put(key, value, next) if fuel > 0 =>
      interpret(next())(kv.updated(key, value), Label.Put(key, value) :: trace, fuel - 1)

    case _ =>
      (None(), trace)
  }
}
```

Listing 5: Key-value store implementation

3.6 1st Futamura Projection

Let us consider the case where one were to define a language, implement an interpreter for it in PureScala, and then write a program in this language. One could then partially evaluate the interpreter applied to the program, while keeping the inputs and I/O operations abstract. The resulting expression could then be seen as a "compiled" version of the original program, represented as a PureScala program, which can be linked to a small runtime and 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* [5], and can theoretically be applied to the partial evaluator itself, yielding a compiler (the 2nd projection) or even a converter from any interpreter to a compiler (3rd projection).Listing 7 features an example involving an interpreter for a simple language with variables, arithmetic expressions and an I/O operation.

3.7 Termination

When the partial evaluator is invoked over a program that is not provably terminating, there are no guarantees that the evaluation procedure will terminate as well. If the program is provably non-terminating, such as the one in Figure 8, then the evaluator will not terminate either. Indeed, in the current state, checking for program termination using Stainless's powerful termination checker during evaluation is impractical for performance reasons. While it would certainly be possible to perform termination checking before the partial evaluator kicks in, and then only attempt to evaluate functions which have been proved terminating, we for now simply suggest to only enable partial evaluation on functions which have been proved terminating by Stainless, by running the termination checker manually, as is the recommended practice when doing verification as well. As an additional measure, we have bounded the maximum number of evaluation steps that can be performed by the evaluator. This ensure that, should one still run the evaluator on a non-termination function, Stainless will not hang or crash.

Moreover, we argue that, when ran over a program that Stainless deems terminating, the evaluator terminates as well. We note that, at point in the pipeline, the only source of non-termination are calls to recursive functions. As such, because the evaluator will only unfold a function invocation when the precondition is satisfied, and when any of the following two statement is true:

(a) The unfolded expression does not contain a recursive call

(b) Otherwise, it is possible to determine under the current path condition whether a recursive call will be reached or not

The understand why the second property is important, consider the program in Listing 9. If the evaluator only looked at the outermost conditional statement to determine whether the call to `tricky` in `test` could be unfolded in productive way, it would end up in an infinite loop. Indeed, because `n` is greater than 0 under the path condition, the `then` branch of the outermost `if` will be selected, and evaluation will continue recursively from there. Because we don't know

anything about `xs`, the evaluator will then speculatively evaluate both branches of the match expression, and will thus encounter another recursive call, which it will deem worthy of unfolding because, by the same reasoning, the outermost `if` can be reduced to its `then` branch. On the other hand, property (b) will forbid the evaluator to unfold `tricky` because it is not possible to deduce, under the current path condition, whether or not the recursive call will be reached.

On the other hand, if we supply `tricky` with a partially known list, such as in the function `test2`, `tricky` will still be unfolded until the function is eventually applied to the abstract list `xs`. While this no formal argument, we believe that these conditions are sufficient to ensure termination of evaluation of provably terminating programs.

Regarding termination checking itself, we have ensured that the partial evaluator cannot be enabled when checking termination in Stainless, as doing so could prevent the termination checker to itself terminate, or even yield incorrect results.

3.8 Conclusion

We have presented, formalized and implemented a symbolic partial evaluator for PureScala programs which:

1. enhances the verification capability of Stainless by allowing the system to reason about programs at a semantic level closer to those of PureScala, effectively simplifying the verification conditions then generated by the system. This in turn helped us reason about a tracing interpreter for a simple algebra over a key-value store.
2. effectively turns Stainless into a symbolic partial evaluator for PureScala programs, thanks to user-land annotations which allow users to selectively simplify parts of their programs. This allowed us to turn an interpreter for a simple programming language into a compiler from that language to PureScala.

3.9 Further Work

Nondeterministic 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 potential use cases of a so-called *non-deterministic* evaluator. That is to say, an evaluator which, when given a single expression, would return 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 10.

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.

```
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
case class Rand(max: Expr)       extends Expr

case class Context(bindings: List[(String, Expr)]) {
  def contains(name: String): Boolean =
        apply(name).isDefined

  def apply(name: String): Option[Expr] =
    bindings.find(_._1 == name).map(_._2)
}

implicit val state = Random.newState

@extern
def random(max: Int): Int = {
  Random.nextInt(max)
}

case class Error(msg: String)
def interpret(expr: Expr, ctx: Context): Either[Error, Int] = {
  expr match {
    case Num(value) => Right(value)

    case Var(name)  => ctx(name) match {
      case None()      => Left(Error("Unbound variable: " + name))
      case Some(value) => interpret(value, ctx)
    }

    case Add(l, r)  => for {
      le <- interpret(l, ctx)
      re <- interpret(r, ctx)
    } yield le + re

    case Mul(l, r)  => for {
      le <- interpret(l, ctx)
      re <- interpret(r, ctx)
    } yield le * re

    case Rand(max) =>
      interpret(max, ctx).map(random(_))
  }
}
```

Listing 6: Implementation an interpreter for a simple language

```
val program: Expr = Mul(Num(10), Add(Var("x"), Rand(Num(42))))

@partialEval
def compiled(ctx: Context): Int = {
  require(ctx contains "x")
  interpret(program, ctx).get.            // 10 * (ctx("x") + random(42))
}

@partialEval
def test(y: Int) = {
  val ctx = Context(Map("y" -> Num(y)))
  interpret(program, ctx)                 // Left(Error("Unbound variable: x"))
} ensuring { _.isLeft }
```

Listing 7: "Compilation" of a simple program

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

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

Listing 8: Example of non-terminating program

```
def tricky(n: Int, xs: List[Int]): Int = {
  if (n > 0) {
    xs match {
      case Nil()       => n
      case Cons(x, xs) => x + tricky(n, xs)
    }
  } else {
    0
  }
}

@partialEval
def test1(n: Int, xs: List[Int]) = {
  require(n > 0)
  tricky(n, xs) // tricky(n, xs)
}

@partialEval
def test2(n: Int, xs: List[Int]) = {
  require(n > 0)
  tricky(n, Cons(1, Cons(2, Cons(3, xs)))) // 1 + (2 + (3 + trick(n, xs)))
}
```

Listing 9: Program requiring some care during evaluation

```
def ndEval(expr: Expr, pc: PC): Stream[Expr] = {
 case IfExpr(c, t, e) =>
  for {
    c  <- ndEval(c, pc)
    ts  = ndEval(t, pc withCond c)
    es  = ndEval(e, pc withCond not(c))
  } yield if (c == true) ts else if(c == false) es else (ts ++ es)

  /* ... */
}
```

Listing 10: Pseudo-code of a non-determinstc 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 π*-calculus* [6].

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.

```
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 `Packets` 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
```

4.2.3 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[2], 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

[2] 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[3], 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. Because we are essentially doing a proof by induction over execution steps here, one needs also to ensure the invariant holds for some initial system. These two properties can be easily expressed in PureScala, as shown in Listing 11.

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

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

def initial: ActorSystem = /* ... */

def initialInv: Boolean = {
  invariant(initial)
} holds
```

Listing 11: Invariant preservation theorem in PureScala

[3] 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

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 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 the message `Inc()`, it forwards that message to the backup actor, and returns a new instance of `PrimaryB` with the counter value incremented by one. When the backup actor receives the `Inc()` message, it too returns a new instance of `BackupB` with the counter value incremented

by one. The corresponding PureScala implementation can be found in Listing 12.

We now want to verify that the following properties stay valid between at execution step: the `Backup()` actor does not send itself any messages, both actors have the proper corresponding behavior, the value of the primary counter is equal to the value of the backup counter added to the number of `Inc()` messages that are yet to be delivered to the backup actor. Listing 13 presents the corresponding invariant, while Listing 14 shows the invariant preservation theorem that we want to prove.

```
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 12: Increment-based replicated counter implementation

4.5.2 Delivery-based Replicated Counter

Listing 15 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.

The invariant now reads slightly differently, as can be seen in Listing 16.

4.5.3 Lock Service

Listing 17 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 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 13: Increment-based replicated counter invariant

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

Listing 14: Replicated counter theorem (increment)

```
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 15: Delivery-based Replicated counter implementation

```
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 16: 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 18.

(TODO: Lock service invariant proof)

4.6 Spawning Actors

4.6.1 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.

4.6.2 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 that the partial 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. Thanks to the shim presented in Listing 19, it is effectively possible run an Actor system developed with our framework on top of 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 20 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)

```
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 17: 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 18: Lock service invariant

```
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
```

```
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 19: Akka shim for our Actor framework

```
@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 20: 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 any mistakes directly when writing the implementation.

As we have unfortunately not found a way to provide 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.

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 [7] 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.

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

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.

$$
\begin{aligned}
S_A &= \mu\alpha.\, \Big(!\texttt{Greet}.\, \big(?\texttt{Hello}.\, \alpha \;\&\; ?\texttt{Bye}.\, \epsilon \big) \oplus !\texttt{Quit}.\, \epsilon \Big) \\
S_B &= \mu\alpha.\, \Big(?\texttt{Greet}.\, \big(!\texttt{Hello}.\, \alpha \oplus !\texttt{Bye}.\, \epsilon \big) \;\&\; ?\texttt{Quit}.\, \epsilon \Big)
\end{aligned}
$$

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 formalize 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 π-calculus, it has been shown that it possible to encode them directly in a calculus or language with both *linear types* and *variants* [8, 9].

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 [10, 11] 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 [12] provides a lightweight, term-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 parametrized by the type of value that they accept or produce, and provide methods to send or receive such messages. Listing 21 shows a subset the API provided by the library, and Listing 22 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[4] of linear types in Stainless, which consists of about 750 lines of code. 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 23 shows the full definitions. Because those will be extracted in a specific way, they are marked `@ignore`.

[4] https://github.com/romac/stainless/tree/linear

```
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 21: *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 22: *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 23: 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 linear terms must first be delinearized with the ! operator before one can call methods on the underlying values.

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 Δ. (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* Δ 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 24 shows such a use-case.

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

Listing 24: 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 25 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 25: 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 26 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`.

```
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 26: Linear File API

5.5.7 Higher-Order Linear Functions

Listing 28 shows the syntactic sugar provided by our implementation, which allow to refer to the type a linear lambda, ie. a lambda which consumes its argument, by `A -*> B` instead `Linear[A] => B`. The library also provides an implicit conversion from function of `Linear[A] => B` to a `A -*> B`. In Listing **??**, we show the definition of a linear list, along with its `map` method, which accepts a linear lambda.

```
case class -*>[A, B](f: Linear[A] => B) {
  def apply(x: Linear[A]): B = f(x)
}

implicit def toLin[A, B](f: Linear[A] => B): A -*> B = -*>(f)
```

Listing 27: Syntactic sugar for linear lambdas

```
@linear
sealed abstract class LList[A] {
  def map[B](f: A -*> B): Linear[LList[B]] = this match {
    case LNil()      => LNil[B]()
    case LCons(h, t) => LCons(f(h), t.map(f))
  }
}

case class LNil[A]() extends LList[A]
case class LCons[A](head: Linear[A], tail: Linear[LList[A]]) extends LList[A]

def ok(xs: Linear[LList[Int]]): Linear[LList[Int]] = {
  xs.map((x: Linear[Int]) => x + 1)
}

def bad(xs: Linear[LList[Int]]): Linear[LList[Int]] = {
  xs.map((x: Linear[Int]) => x + x + 1) // err: linear term ‘x‘ has already been used
}
```

Listing 28: Linear list

5.6 Sessions Library in PureScala

Listing 29 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.

5.7 Case Study: ATM

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.

```
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 29: Sessions library in PureScala

(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 30 shows the encoding of such a specification using the library described in Section 5.6. Listing 31 shows the corresponding valid implementation. At last, Listing 33 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(_)
                                        ^^^^^^^^^^^^^^^^^
```

5.8 Conclusion

(TODO: Conclusion)

```
// 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 30: ATM protocol description

5.9 Further Work

Term-level Multiparty Session Types for Actor Systems

We believe that it would be worth investigating the interplay between linear types, sessions types and the actor model developed in Section 4, provided that we manage to enrich the model with a typed API. It is of course already possible to mark actor references as linear within our model, effectively requiring the holder of such a reference to send it a message. But without a typed API, it is current not possible to model a session within this framework. On top of that, more work is probably needed to properly encode the resulting so-called *multiparty session types* [13] at the term level.

Affine Types and Borrow Checker

In the early days of this project, we implemented a very basic borrow checker within Stainless[5]. While the results were encouraging, such a mechanism is not enough to implement linear channels, as it corresponds to an enriched affine type system. We nonetheless believe that this could also be an interesting addition to Stainless. This because one could implement a pass within Stainless which would insert the appropriate invocations for allocating and freeing the memory associated with an affine term which could then be linked and compiled with the Scala Native project[6], thus effectively allowing one to opt out of garbage collection for the parts of a program requiring a high memory throughput. It is worth noting that this technique could also apply to Leon's C code generator [14, 15] but as a way to safely introduce dynamic memory allocations in the generator, which currently only support statically allocated memory.

[5] https://github.com/romac/stainless/tree/borrow-checker
[6] https://github.com/scala-native/scala-native

```
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 31: 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 32: Correct user protocol implementation

```
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 33: Incorrect implementation of the ATM protocol

6 Conclusion

(TODO: Conclusion)

A References

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

[2] J. C. King, *A Program Verifier*. PhD thesis, Pittsburgh, PA, USA, 1970. AAI7018026.

[3] 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.

[4] R. Bubel, R. Hähnle, and R. Ji, *Interleaving Symbolic Execution and Partial Evaluation*, pp. 125–146. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] M. Antognini, "From Verified Functions to Safe C Code." https://github.com/mantognini/GenC, 2016.

[15] M. Antognini, "Extending Safe C Support In Leon." https://github.com/mantognini/GenC, 2017.