

# Symbolic Partial Evaluation of PureScala Programs

And Other Things

Romain Ruetschi

Laboratory for Automated Reasoning and Analysis, EPFL

February 2018

# Outline

- Symbolic Partial Evaluation of PureScala Programs
- Verifying Invariants of Actor Systems
- Bi-party Communication Systems with Linear Types

# Symbolic Partial Evaluation of PureScala Programs

# Motivation

```
def insert[A, B](kvs: List[(A, B)], map: Map[A, B]) = {
  kvs.foldLeft(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) = {
  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
```

```
$ stainless --timeout=7200 test.scala
```

**TIMEOUT**<sup>1</sup>

---

<sup>1</sup>This example actually verifies as-is with the latest Stainless, but we are using it for the sake of the argument

# Idea

While some input to test are abstract, we know enough about the whole program to reduce it to a simpler form, which has higher chances of verifying, or will verify quicker.

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

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

We start by inlining the definition of `xs`:

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



We then unfold the definition of `insert`:

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

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 => xs.foldLeft(f(z, x))(f)
}
```

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

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  
}
```

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

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
}
```

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)
```

We are now left with this expression:

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

We can now unfold foldLeft one more time:

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

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

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



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
}
```

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)
```

We are now left with the following expression:

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

We can now unfold foldLeft one last time:

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

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

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

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) = {  
  require(!map.contains("a") && map.contains("b"))  
  
  val res = map.updated("a", x)  
  
  res("a") == x && res("b") == map("b")  
} holds
```

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.

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`.

# Definitions

- *Partial evaluation* [5] 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.



# Definitions

- *Partial evaluation* [5] 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* [7, 6] 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 [1].

# Definitions

- *Partial evaluation* [5] 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* [7, 6] 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 [1].
- 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 the two techniques

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 partial evaluator, and to increase both performance and reasoning abilities of verifications systems [2].

We call the result a *symbolic partial evaluation procedure*.

# Implementation

- Implemented within Stainless as a new type of evaluator.
- Leverages Inox to check for feasibility of path under current path condition.
- The user must supply the new `-partial-eval` flag, and annotate the function with the `@partialEval` annotation.

# Termination

# Non-terminating programs

- 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, then the evaluator will not terminate either. We alleviate this problem by bounding the maximum number of steps the evaluator can take.

# Terminating programs

We argue that, when ran over a program that Stainless deems terminating, the evaluator terminates as well.

**Note:** The only source of non-termination are calls to recursive functions.

# Recursive function unfolding

The evaluator only unfolds a function invocation when

- The precondition is satisfied, and:
- When any of the following two statement is true:
  - The unfolded expression does not contain a recursive call, or:
  - It is possible to determine under the current path condition whether the recursive call will be (transitively) reached or not.



# Example

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

```
@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, xs))) // 1 + (2 + tricky(n, xs))
}
```

# Results

- Reasoning about the result of a tracing interpreter for a simple algebra over a key-value store
- Specializing an interpreter for a simple programming language into a compiler from that language to PureScala

# Conclusion

The symbolic partial evaluator we have implemented within Stainless:

- enhances its verification capabilities by allowing the system to reason about programs at a semantic level closer to those of PureScala.
- allows users to selectively specialize parts of their programs.

# Further work

## Nondeterministic evaluator

- Implement an evaluator which, when faced with a conditional expression whose condition it is not able to evaluate to a boolean literal, would return a list of expressions resulting from (recursively) evaluating both branches.
- Let users splice back the list of results in their programs.

# Verifying Invariants of Actor Systems

# Motivation

- The actor model is a popular model for building concurrent and distributed programs. Such programs often have invariants that need to be respected at any point in time for the program to be deemed correct.
- Akka is the de-facto standard for building actor-based applications in Scala.
- We want to describe actor systems in PureScala, then prove that their invariants hold, before running them on top of Akka.

# Our model

# Implementation

```

abstract class Msg
abstract class ActorRef {
  def !(msg: Msg)(implicit ctx: ActorContext): Unit = /* ... */
}

case class Packet(dest: ActorRef, payload: Msg)
case class ActorContext(
  self: ActorRef,
  var toSend: List[Packet]
)

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

```



# Implementation

```
case class ActorSystem(  
  behaviors: Map[ActorRef, Behavior],  
  inboxes: Map[(ActorRef, ActorRef), List[Msg]]  
) {  
  def step(from: ActorRef, to: ActorRef): ActorSystem = /* ...  
}
```

# Specifying an invariant

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

# Proving correctness

## 1. Invariant initially holds

$$\exists i : \text{ActorSystem}. \text{inv}(i)$$

## 2. Invariant is preserved between each execution step

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

# Proving correctness (PureScala)

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

// 1. Invariant initially holds
def initialInv = invariant(initial).holds

// 2. Invariant is preserved between each execution step
def preserveInv(s: ActorSystem, n: ActorRef, m: ActorRef) = {
  require(inv(s))
  inv(s.step(n, m))
} holds
```

# Example

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

case class Inc() extends Msg
```

# Example

```

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

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

```

## Example

```
case class BackupB(counter: BigInt) extends Behavior {  
  require(counter >= 0)  
  
  def processMsg(msg: Msg)(implicit ctx: ActorContext) =  
    msg match {  
      case Inc() => BackupB(counter + 1)  
    }  
}
```

# Example

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

    case _ => false
  }
}
```



# Dynamic topologies

We extended the model to handle dynamic topologies as well, where actors can spawn other actors.

# Running actor systems on Akka

We provide a shallow shim which allows a user to run an actor system built with our framework on top of Akka.

```
import stainless.actors.akkashim._

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()
  }
}

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

# Conclusion

- Developed a simple model for actor systems implemented as a PureScala library.
- Modeled a few simple systems with it.
- Specified and verified global invariants
- Shown how to run these systems on top of a Akka

## Further work

- Model more complex systems
- Weaken guarantees on message delivery
- Ensure uniqueness of actor names
- Take a closer look at the possible interplay with the partial evaluator described in the first part

# Modelling Bi-party Communication Systems with Linear Types

# Motivation

We want to:

- Model bi-party communication systems which follow a given protocol
- Ensure by construction that the implementation properly follows the protocol

# Definitions

- **Bi-party communication system:** System involving two parties communicating synchronously over a channel with two endpoints, one for sending messages and one for receiving them.
- **Bi-party communication protocol:** Protocol governing such systems.

## Session types

Session types [4, 10] provide a way to encode a protocol in the type system. The type checker can then both guide and ensure that the implementation follows the protocol. Mistakes such as not handling a certain message or failing to reply to a message are caught statically.



# Session types

$T$	$::=$	$S$	$ $	$\text{Int}$	$ $	$\text{Boolean}$	$ $	$\text{String}$	$ $	$\dots$	payload
$S$	$::=$	$\&\{l_i : S_i\}_{i \in I}$									branching
		$ $	$\oplus\{l_i : S_i\}_{i \in I}$								selection
		$ $	$?T.S$								receive
		$ $	$!T.S$								send
		$ $	$\mu X.S$								recursion
		$ $	$X$								variable
		$ $	$\epsilon$								termination

Figure 1: Syntax of session types

# Protocol example

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

$$S_B = \overline{S_A}^1 = \mu\alpha. \left( ?\text{Greet}. \left( !\text{Hello}. \alpha \oplus !\text{Bye}. \epsilon \right) \ \& \ ?\text{Quit}. \epsilon \right)$$

---

<sup>1</sup> $\overline{S}$  denotes the *dual* of session type  $S$ .

# Type-level encoding of session types

Although session types were originally meant to be implemented as a separate syntactic category of types and terms to be added to the  $\pi$ -calculus, in mainstream languages with sufficiently powerful type systems (eg. Haskell, Scala), they are usually encoded using graded/indexed monads [8].

Unfortunately, Stainless's type system is currently not powerful enough to model such constructs.

# Term-level encoding of session types

It has been shown that it possible to encode session types directly in a calculus featuring both linear types and variants (sum types) [10, 3].

Despite the lack of linear types in Scala, there already exists such an encoding in the form of the *lchannels* library [9].

# Term-level encoding of session types with *lchannels*

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

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

```
def alice(c: Out[AliceBob]) = {  
  c !! Greet("Eve") ? {  
    case Hello(name, cont) =>  
      println(s"Hello, $name")  
      cont ! Quit() // or restart with `alice(cont)`  
    case Bye() =>  
      println("Bye")  
  }  
}
```

```
def bob(c: In[AliceBob]) = {  
  c ? {  
    case Greet(name)(cont) if name == "Eve" =>  
      cont !! Hello(name)  
    case Quit() =>  
      c ! Bye()  
  }  
}
```

# The need for linearity

As mentioned in the paper, this encoding is sufficient to model any session type but, in the absence of linearity, cannot prevent all programmer mistakes, such as a using a channel more than once, or forgetting the listen for/reply to a message.



# Linear types in PureScala

```
class Linear[+A](_value: A) {  
  def ! = _value  
}  
  
implicit def linearize[A](value: A): Linear[A] = new Linear(value)  
  
object implicits {  
  implicit def delinearize[A](lin: Linear[A]): A = lin!  
}
```

# Definition

A term  $t$  of type  $\text{Linear}[A]$ , for any type  $A$ , is deemed consumed in an expression  $e$  when any of the following propositions is true:

- The underlying value is extracted, via the `!` method.
- The term is assigned to a variable.
- The term is supplied as an argument to function.
- The term is supplied as an argument to a method.
- The term is supplied as an argument to a lambda.
- The term is supplied as an argument to a constructor.

# Preventing weakening

To ensure that no linear term is *consumed* more than once, we 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.

# The ! operator

If it were not handled carefully, the ! operator would actually allow weakening:

```
val a: Linear[A] = ...  
val b: A = a.!
```

To prevent this, the linearity checker treats any expression of the form  $e.!$ , with  $e: \text{Linear}[A]$ , as having type  $\text{Linear}[A]$  instead of  $A$ .

# Preventing contraction

Because linear logic does not allow contraction, we must also ensure that no linear term is *dropped*, that is to say, ensure that every linear terms are consumed at least once.

To this end, we check every linear variables being introduced within a function against the *usage context* resulting from the linear type checking algorithm to determine whether each and every of those variable has indeed been *consumed* at some point, and reject the program otherwise.

## Linear terms in contracts

Because in PureScala semantics, contracts are not executed at runtime, we can safely disable the linearity checker within contracts. This allows us to refer to linear variable in a pre- or post-condition without effectively consuming it.

# Linear datatypes

The linearity checker also ensures that any value of type  $A$ , where  $A$  is type with at least one constructor (transitively) containing linear fields, is introduced linearly. Failing to do so would allow having multiple references to a linear value.

# Marking the current object as linear

Within a class method, one can specify that the current object `this` must be treated as a linear variable (both in the caller and the callee) by marking the method with an `@linear` annotation.



# Higher-order linear functions

For convenience, we define a type alias<sup>2</sup> for higher-order linear functions:

```
type -*>[A, B] = Linear[A] => B  
def map[A, B](l: LinList[A], f: A -*> B): LinList[B] = ...
```

---

<sup>2</sup>only very recently added to Stainless

# Linear channels in PureScala

```

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

@linear class InChan[A] {
  def ?[B](f: A -> B)(implicit d: Duration): B = ???
}

@linear class OutChan[A] {
  def !(msg: A): Unit = ???
  def !![B](h: Out[B] => A): In[B] = ???
  def create[B]():(In[B], Out[B]) = ???
}

```

## Catching common errors

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

Linear variable ``c`` of **type** ``Out[AliceBob]`` is never used:

```
def alice(c: Out[AliceBob]): Unit = {
  ~~~~~
```

## Catching common errors

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

Linear term ``cont`` has already been used:

```
doSomething(cont)
      ^^^^
```

Term used here:

```
cont !! Greet(name)(_) ? alice(_)
      ^^^^
```

In case we forget to reply to the Greet message within the bob function, we get:

Linear variable ``cont`` of type ``Out[Response]`` is never used:

```
case Greet(name)(cont) =>
      ^^^^
```

## Catching common errors

At last, let's see what happens if we do not handle the reply to the Greet message sent in case the authentication succeeds.

Linear term cannot be discarded:

```
cont !! Greet(name)
~~~~~
```

In this case, the linearity checker is able to determine that the linear value return by the `!!` is discarded because the Scala compiler implicitly converts it to `Unit`.

# Results

- Implementation of a simple protocol involving an ATM and its user.
- Partial implementation of the TLS 1.2 handshake protocol (*work in progress*).

# Conclusion

- We provide an implementation of linear types for PureScala, an object-oriented language.
- We implemented linear channels in PureScala, fully compatible with the original Scala version.
- We have shown that linearity helps catching some potential errors that could arise while writing the program.
- Can also be put to use for designing safe APIs.

## Furter work

- Assuming soundness of PureScala's type system, prove soundness of the linear extension.
- Implement affine types and investigate further how to implement a borrow checker on top.
- Investigate how to leverage linearity to model multiparty session types for actor systems.









Thank you

# Thank you

Special thanks to Prof. Viktor Kuncak and Dr. Jad Hamza for their mentorship all along the project, as well as to Dr. Stephan Merz for his very constructive remarks. Many thanks as well to Nicolas Voirol, Dr. Andreas Pavlogiannis, Manos Koukoutos, Marco Antogini, Dr. Ravi Kandhadai, Georg Schmid, Romain Edelmann and Mikael Mayer for their support and insights.

# References I

-  Roberto Baldoni et al. “A Survey of Symbolic Execution Techniques”. In: *CoRR* abs/1610.00502 (2016).
-  Richard Bubel, Reiner Hähnle, and Ran Ji. “Interleaving Symbolic Execution and Partial Evaluation”. In: *FMCO 2009*. Ed. by Frank S. de Boer et al. 2010, pp. 125–146.
-  Ornela Dardha, Elena Giachino, and Davide Sangiorgi. “Session Types Revisited”. In: *PPDP '12*. ACM, 2012.
-  Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. “Language Primitives and Type Discipline for Structured Communication-Based Programming”. In: *ESOP '98*. 1998.
-  Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., 1993.
-  James C. King. “Symbolic Execution and Program Testing”. In: *Commun. ACM* (1976), pp. 385–394.

# References II



James Cornelius King. “A Program Verifier”. PhD thesis. 1970.



Marc Orchard and Nobuko Yoshida. “Session Types with Linearity in Haskell”. In: *Behavioural Types: from Theory to Tools*. River publishers, 2017.



Alceste Scalas and Nobuko Yoshida. “Lightweight Session Programming in Scala”. In: *ECOOP 2016*. 2016.



Philip Wadler. “Propositions As Sessions”. In: *ICFP '12*. 2012, pp. 273–286.