Introducing Reflection into a Verification System

Julie Giunta

School of Computer and Communication Sciences
Semester Project Report

Supervisor
Romain Edelmann
EPFL / LARA

Supervisor
Prof. Viktor Kunčak
EPFL / LARA

June 2019

Introducing Reflection into a Verification System

Julie Giunta
EPFL, Switzerland
julie.giunta@epfl.ch

Abstract

Stainless is a tool for verifying Scala programs. During verification, tree reflection is sometimes needed. It cans be used to send constraints to an underlying constraint solving during program execution. With tree reflection, the program can be described as an algebraic data type. You can then type check and interpret it within Stainless.

1 Introduction

The goal of this project was to implement tree reflection within Stainless. Tree reflection means to allow access to an expression as an algebraic data type. For example, if you had this expression:

```
x + y * z
```

the corresponding algebraic data type could be:

```
Plus(Var("x"), Times(Var("y"), Var("z")))
```

For this purpose, we defined which expressions we wanted to be able to reflect and how to describe them as algebraic data types. Furthermore, we defined some basic types to support type checking on our expressions. As the underlying expressions could be interpreted, we added an interpreter for our algebraic data types that returned the same result but wrapped in a data type.

2 Implementation

The system is written in Pure Scala, thus allowing Stainless to verify it. As Stainless ensures lot of properties, the code had to be as simple as it could be, otherwise Stainless was not able to terminate the verification in a sustainable time.

2.1 Expressions

Expressions are algebraic data types. They represent the abstract syntax tree (AST) of a program that can be type checked and interpreted. It was implemented as an abstract class, *Expr*, and case classes that extend *Expr*.

The expressions that will be find in the leaves of the AST are the literals. They are the "basic unit" of expressions. For example, the literal for a character is defined as:

```
case class CharLiteral(value: Char) extends Expr
```

There are also literals that represent integers, booleans, strings and fractions. They are defined in a similar manner except for fractions which take a tuple of BigInt, the first element of the tuple representing the numerator and the second one, the denominator. They all correspond to one of the basic types described in the following section.

Another type of value is lambda definition:

```
case class Lambda(params: List[(Identifier, Type)],
                  body: Expr) extends Expr
```

It represents lambda-expressions such as $\lambda\ x : BigInt,\ y : BigInt = x + y$. In the program, a list containing *(x, BigInt)* and *(y, BigInt)* represents the *params*. In fact, x and y would be Identifiers. Identifiers are defined as a case class that takes a string as parameter, the "name" of what needs to be identified. $x + y$ represents the *body*. In the body, x and y would be represented by *Variable*, which is an expression defined by:

```
case class Variable(id : Identifier) extends Expr
```

The nodes of the AST represents operations done on the leaves and on other nodes. For example, the addition operation is implemented as:

```
case class Plus(lhs: Expr, rhs: Expr) extends Expr
```

Here, *Plus* is a node in the AST and *lhs* and *rhs* are the children of this node. The other arithmetic operations (minus, unary minus, times, division, remainder and modulo) are defined in a similar way, except for the unary minus which takes only one argument.

In addition of arithmetic, there are also expressions to represent:

- String operations such as concatenation, computing the length and taking a subset of the given string expression
- Equality test on two expressions
- Comparisons on integers, fractions and characters expressions such as greater than ($>$), greater equals ($\geq$), less than ($<$) and less equals ($\leq$).
- Logical operations on booleans expressions such as logical and, logical or, logical implication and logical not
- If-then-else expression

- Let expression, an expression which gives a certain value to an identifier and then uses this definition in the subsequent expression

Another type of expression is *Application*:

```
case class Application(callee: Expr, args: List[Expr])
    extends Expr
```

It is used when we want to apply arguments to a *Lambda*. For example, if our lambda is $add = \lambda\ x : BigInt,\ y : BigInt = x + y$, then $add(1, 2)$ would correspond to an *Application* where *callee* is *add* and a list containing 1 and 2 represents *args*.

When there is a problem during the interpretation, the program outputs an expression which represents an error:

```
case class ErrorValue(error: String) extends Expr
```

2.2 Types

Types are represented as an abstract class *Type* and case classes that extend it and represent real types. There is a type for each kind of literal. For example, the type corresponding to a character is:

```
case class CharType() extends Type
```

In the same way, *IntegerType* is for integers, *BooleanType* for booleans, *StringType* for strings and *RealType* for fractions.

The type that corresponds to lambdas is:

```
case class FunctionType(from: List[Type], to: Type) extends Type
```

where *from* contains the types of the parameter *params* of *Lambda* and *to* represents the type of the body of *Lambda*.

2.3 DSL

To be able to define an expressions in a easier way, the user can call some methods of the domain specific language (DSL). For literals, the methods are named with the first letter of the literal in capital. For example, for a *CharLiteral*:

```
def C(c : Char) = CharLiteral(c)
```

For arithmetic, string concatenation, comparisons and logical operators, the methods are named *e_op* where *op* is replaced by the corresponding sign for this operation in Java. For example, for *Plus*:

```
def e_+(lhs: Expr, rhs: Expr) = Plus(lhs, rhs)
```

For expressions that needs an Identifier as argument, the user only needs to provide a string which represents the name of this Identifier. For example, to define a *Let* expression:

```
def let(name: String, tpe: Type, value: Expr)
    (body: Identifier => Expr): Expr = {
                val id = Identifier(name)
                Let(id, tpe, value, body(id))
        }
```

2.4 Type checker

The type checker was implemented in a recursive manner. It is a method that takes as arguments an expression and an environment which is represented by a Map of Identifier to Type. It does a pattern match on the given expression and applies an inference rule to obtain either *None()* if the expression does not type check or *Some(resulting type)* if the expression type checks.

The inference rules are (E representing the environment):

For literals:

$$\text{CharLiteral}\ \frac{c : Char}{E \vdash CharLiteral(c) : CharType}$$

$$\text{IntegerLiteral}\ \frac{i : BigInt}{E \vdash IntegerLiteral(i) : IntegerType}$$

$$\text{BooleanLiteral}\ \frac{b : Boolean}{E \vdash BooleanLiteral(b) : BooleanType}$$

$$\text{StringLiteral}\ \frac{s : String}{E \vdash StringLiteral(s) : StringType}$$

$$\text{FractionsLiteral}\ \frac{n : BigInt, d : BigInt}{E \vdash FractionLiteral((n, d)) : RealType}$$

$$\text{Variable}\ \frac{v : T \in E}{E \vdash v : T}$$

$$\text{Let}\ \frac{E \vdash value : T1 \quad E, id : T1 \vdash body : T2}{E \vdash Let(id, T1, value, body) : T2}$$

$$\text{If-then-else}\ \frac{E \vdash cond : BooleanType \quad E \vdash thenn : T \quad E \vdash elze : T}{E \vdash IfExpr(cond, thenn, elze) : T}$$

For lambdas:

$$\text{Lambda}\ \frac{E_{[id1:=t1,...,idN:=tN]} \vdash body : T}{E \vdash Lambda([(id_1, t_1), ..., (id_N, t_N)], body) : FunctionType([t_1, ..., t_N], T)}$$

$$\text{Application}\ \frac{E \vdash callee : FunctionType([t_1, ..., t_N], T)\ E \vdash a_1 : t_1\ ...\ E \vdash a_N : t_N}{E \vdash Application(callee, [a_1, ..., a_N]) : T}$$

For arithmetic expressions:

$$\text{General arithmetic}\ \frac{E \vdash lhs : T\ E \vdash rhs : T}{E \vdash op(lhs, rhs) : T}\ \text{Unary minus}\ \frac{E \vdash e : T}{E \vdash UnaryMinus(e) : T}$$

The general arithmetic rule is applied by *Plus, Minus, Times, Division, Modulo* and *Remainder* where *op* represents the corresponding operation. For the last two, T can only be *IntegerType*. For the others, T is either *IntegerType* or either *RealType*.
The Unary minus rule is applied by *UnaryMinus* and T can be *IntegerType* or *RealType*.

For string operations:

$$\text{Strings concatenation}\ \frac{E \vdash lhs : StringType\ E \vdash rhs : StringType}{E \vdash StringConcat(lhs, rhs) : StringType}$$

~

$$\text{Substring}\ \frac{E \vdash e : StringType\ E \vdash start : IntegerType\ E \vdash end : IntegerType}{E \vdash SubString(e, start, end) : IntegerType}$$

$$\text{String length}\ \frac{E \vdash e : StringType}{E \vdash StringLength(e) : IntegerType}$$

For comparisons expressions:

$$\text{Equals}\ \frac{E \vdash lhs : T1\ E \vdash rhs : T2}{E \vdash Equals(lhs, rhs) : BooleanType}$$

$$\text{Other comparisons}\ \frac{E \vdash lhs : T\ E \vdash rhs : T}{E \vdash op(lhs, rhs) : T}$$

For equals, *T1* and *T2* can be of any type but *lhs* and *rhs* must type check.
For other comparisons, T must be either *IntegerType, RealType* or *CharType*. This rule is applied by *LessThan, GreaterThan, LessEquals* and *GreaterEquals*.

For logical operations:

$$\text{Logical operators}\ \frac{E \vdash lhs : BooleanType\ E \vdash rhs : BooleanType}{E \vdash op(lhs, rhs) : BooleanType}$$

$$\text{Not}\ \frac{E \vdash e : BooleanType}{E \vdash Not(e) : BooleanType}$$

For logical operators, *op* can be *And, Or* or *Implies*.

2.5 Interpreter

The interpreter is a small-step interpreter. It has two main methods: interpret, which takes an expression and returns the fully interpreted expression (either a literal, a lambda or an error value) and next, which takes an expression and returns an option, either *Some(the next small-step of interpretation of the expression)* or either *None()* if it is stuck. Being stuck can happen for two reasons. First, the expression is already fully evaluated, thus we cannot progress. Second, the expression does not make sense, thus there are no inference rule to progress in the interpretation.

The interpreter substitutes the variables in the rest of the expression as soon as their value is defined. For example, if you have this pseudo-code below:

```
let x = 1; let x = x + 2; x + 4
```

substitution of the first x will return:

```
let x = 1; let x = 1 + 2; x + 4
```

because the second x masks the first one in the $x + 4$ statement. Then, after the second x has been evaluated, the substitution would return:

```
let x = 1; let x = 3; 3 + 4
```

The next method follows these inference rules, where $\rightarrow$ means a step of evaluation and v represents a fully interpreted expression (a literal, a lambda or an error):

For Let:

$$\frac{e \rightarrow e'}{Let(id, t, e, body) \rightarrow Let(id, t, e', body)}$$

$$\frac{v \ is \ a \ value}{Let(id, t, v, body) \rightarrow body_{[id:=v]}}$$

For Application:

$$\frac{callee \rightarrow callee'}{Application(callee, args) \rightarrow Application(callee', args)}$$

$$\frac{c \ is \ a \ Lambda, \ e_i \rightarrow e'_i}{Application(c, [v_1, ..., v_{i-1}, e_i, e_{i+1}, ..., e_N]) \rightarrow Application(c, [v_1, ..., v_{i-1}, e'_i, e_{i+1}, ..., e_N])}$$

$$\frac{c \ is \ a \ Lambda([(id_1, t_1), ..., (id_N, t_N)], body), \ v_1, ..., v_N \ are \ values}{Application(c, [v_1, ..., v_N]) \rightarrow body_{[id1:=v1, ..., idN:=vN]}}$$

For if-then-else expressions:

$$\frac{cond \rightarrow cond'}{IfExpr(cond, thenn, elze) \rightarrow Let(cond', thenn, elze)}$$

$$\frac{v \ is \ BooleanLiteral(true)}{IfExpr(v, thenn, elze) \rightarrow thenn} \qquad \frac{v \ is \ BooleanLiteral(false)}{IfExpr(v, thenn, elze) \rightarrow elze}$$

For arithmetic expressions:

$$\text{Arithmetic 1}\ \frac{lhs \rightarrow lhs'}{Expr(lhs, rhs) \rightarrow Expr(lhs', rhs)}$$

$$\text{Arithmetic 2}\ \frac{v\ is\ an\ IntegerLiteral,\ rhs \rightarrow rhs'}{Expr(v, rhs) \rightarrow Expr(v, rhs')}$$

$$\text{Arithmetic 3}\ \frac{v\ is\ a\ FractionLiteral,\ rhs \rightarrow rhs'}{Expr(v, rhs) \rightarrow Expr(v, rhs')}$$

$$\text{Arithmetic 4}\ \frac{v1,\ v2\ are\ IntegerLiterals}{Expr(v1, v2) \rightarrow v1\ op\ v2}$$

$$\text{Arithmetic 5}\ \frac{v1,\ v2\ are\ FractionLiterals}{Expr(v1, v2) \rightarrow v1\ op\ v2}$$

$$\frac{e \rightarrow e'}{UnaryMinus(e) \rightarrow UnaryMinus(e')}$$

$$\frac{v\ is\ an\ IntegerLiteral\ or\ a\ FractionLiteral}{UnaryMinus(v) \rightarrow -v}$$

The rules Arithmetic 1, 2, 3, 4 and 5 are used by the *Expr*: Plus, Minus, Times, Division and their corresponding operation *op*: +, -, *, / The rules Arithmetic 1, 2 and 4 are implemented by the *Expr*: Remainder, Modulo and their corresponding operation *op*: %, mod.

For operations on strings:

$$\frac{lhs \rightarrow lhs'}{StringConcat(lhs, rhs) \rightarrow StringConcat(lhs', rhs)}$$

$$\frac{v\ is\ an\ StringLiteral,\ rhs \rightarrow rhs'}{StringConcat(v, rhs) \rightarrow StringConcat(v, rhs')}$$

$$\frac{v1,\ v2\ are\ StringsLiterals}{StringConcat(v1, v2) \rightarrow v1 + +v2}$$

$$\frac{e \rightarrow e'}{SubString(e, start, end) \rightarrow SubString(e', start, end)}$$

$$\frac{v\ is\ an\ StringLiteral,\ start \rightarrow start'}{SubString(v, start, end) \rightarrow SubString(v, start', end)}$$

$$\frac{v1\ is\ an\ StringLiteral,\ v2\ is\ an\ IntegerLiteral,\ end \rightarrow end'}{SubString(v1, v2, end) \rightarrow SubString(v1, v2, end')}$$

$$\frac{v1\ is\ an\ StringLiteral,\ v2, v3\ are\ IntegerLiterals}{SubString(v1, v2, v3) \rightarrow v1.bigSubstring(v2, v3)}$$

$$\frac{e \rightarrow e'}{StringLength(e) \rightarrow StringLength(e')} \qquad \frac{v\ is\ an\ StringLiteral}{StringLength(v) \rightarrow v.bigLength}$$

For comparisons:

$$\frac{lhs \to lhs'}{Expr(lhs, rhs) \to Expr(lhs', rhs)}$$

$$\frac{v\ is\ a\ lit,\ rhs \to rhs'}{Expr(v, rhs) \to Expr(v, rhs')} \quad \frac{v1,\ v2\ are\ lits}{Expr(v1, v2) \to v1\ op\ v2}$$

For the *Expr* Equals, *v1* and *v2* can be any type of value, *v1* and *v2* can even be different type of literals and then *op* is ==. For the *Expr* LessThan, LessEquals, GreaterThan, GreaterEquals, *lit* are IntegerLiteral, FractionLiteral and CharLiteral. *v1* and *v2* must be the same kind of literals. The corresponding *op* are: $<, \leq, >, \geq$.

For logical operators:

$$\text{General logic}\ \frac{lhs \to lhs'}{Expr(lhs, rhs) \to Expr(lhs', rhs)}$$

$$\text{And and Implies 1}\ \frac{v\ is\ BooleanLiteral(true),\ rhs \to rhs'}{Expr(v, rhs) \to rhs'}$$

$$\text{And and Implies 2}\ \frac{v1\ is\ BooleanLiteral(true), v2\ is\ a\ BooleanLiteral}{Expr(v1, v2) \to v2}$$

$$\frac{v\ is\ a\ BooleanLiteral(false),\ rhs \to rhs'}{Or(v, rhs) \to rhs'}$$

$$\frac{v1\ is\ a\ BooleanLiteral(false), v2\ is\ a\ BooleanLiteral}{Or(v, v2) \to v2}$$

$$\frac{v\ is\ a\ BooleanLiteral(false)}{And(v, rhs) \to BooleanLiteral(false)}$$

$$\frac{v\ is\ a\ BooleanLiteral(true)}{Or(v, rhs) \to BooleanLiteral(true)}$$

$$\frac{v\ is\ a\ BooleanLiteral(false)}{Implies(v, rhs) \to BooleanLiteral(true)}$$

$$\frac{e \to e'}{Not(e) \to Not(e')} \quad \frac{v\ is\ a\ BooleanLiteral}{Not(v) \to !v}$$

General logic is used by And, Or and Implies.

2.6 Soundness theorem

The soundness theorem can be stated as: "If a program type checks, its evaluation does not get stuck"(1). To show that the system (or at least a subset of the Expressions) was a sound system, we proved with Stainless two lemmas, progress and preservation, on some of the Expressions.

2.6.1 Progress

Progress can be stated as: "If a program type checks, it is not stuck"(1). In the program, it is translated by a method which takes as arguments an expression *expr* and a type *t*. It has a precondition:

```
require(!isValue(expr) &&
    typecheck(expr, Map[Identifier, Type]()) == Some(t))
```

and a post condition:

```
next(expr).nonEmpty
```

where the *isValue* method returns true if *expr* is a literal, a *Lambda* or an *ErrorValue*. To ensure the post condition, we used the method *holds* of Stainless.

Stainless was not able to prove this as it was stated but fortunately, by using additional lemmas (*check* method of Stainless), Stainless did verify progress on a subset of expressions. The lemmas were added depending on the expression given. To do so, we used pattern matching. For expressions such as general arithmetic, string operations, comparisons, logical operators and if-then-else, we added an equality test on the type of the expression and checked that the fields of each expression each had a possible type. Furthermore, we pattern matched on the fields and checked the progress of the first non-evaluated field. For example, here are the added lemmas for *Plus*:

```
case Plus(lhs, rhs) => {
  check((t == IntegerType() || t == RealType()) &&
    typecheck(lhs, Map[Identifier, Type]()) == Some(t) &&
    typecheck(rhs, Map[Identifier, Type]()) == Some(t))
  (lhs, rhs) match{
        case (IntegerLiteral(_), IntegerLiteral(_)) => true
        case (FractionLiteral(_), FractionLiteral(_)) => true
        case (IntegerLiteral(_), _) => check(progress(rhs, t))
        case (FractionLiteral(_), _) => check(progress(rhs, t))
        case (_, _) => check(progress(lhs, t))
  }
}
```

For *Let*, we added:

```
case Let(id, tValue, value, body) =>
  value match{
    case _ if(isValue(value)) => true
    case _ => check(progress(value, tValue))
  }
```

For *Application*, Stainless could not verify it if it had an unbounded number of parameters, but we were able to prove progress on an *Application* which had only one or two parameters. To be able to do so, we defined:

```
case class FunctionType1(from: Type, to: Type) extends Type
case class FunctionType2(from1: Type, from2: Type , to: Type)
    extends Type

case class Lambda1(id: Identifier , t: Type, body: Expr)
    extends Expr
case class Lambda2(id1: Identifier , t1: Type,
    id2: Identifier , t2: Type, body: Expr) extends Expr

case class Application1(callee: Expr, arg: Expr) extends Expr
case class Application2(callee: Expr, arg1: Expr, arg2: Expr)
    extends Expr
```

Lambda1, Lambda2 and *Application1, Application2* follow the same rules for type checking and interpretation as *Lambda* and *Application* but with respectively one and two parameters. The lemmas that were added to prove progress on *Application1* are:

```
case Application1(callee, arg) =>
  callee match {
        case Lambda1(id, t, body) => arg match{
            case _ if(isValue(arg)) => true
            case _ => check(progress(arg, t))
        }
        case _ =>{
            val tArg = typecheck(arg,
              Map[Identifier, Type]())
            check(tArg.nonEmpty &&
              progress(callee, FunctionType1(tArg.get, t)))
        }
}
```

It is similar for *Application2*, except it checks progress on the first non-evaluated *arg* and type checks both *args*

Due to the duration of verifying progress with Stainless, we could only verify together:

- Arithmetic, string operations and if-then-else expressions (4.5 hours for the post condition of progress only and approximately 6 hours to prove all the others lemmas in progress)
- Let expressions, comparisons and logical operators (4 hours for the post condition of progress only and approximately 6 hours to prove all the other lemmas in progress)
- Application1 (3 minutes to prove progress)
- Application2 (3 minutes to prove progress)

2.6.2 Preservation

Preservation can be stated as: "If a program type checks and makes one [next] step [with the Interpreter], then the result again type checks"(1). In the program, it is translated by a method which takes as arguments an expression *e1* and a type *t*. It has a precondition:

```
require(typecheck(e1, Map[Identifier, Type]()) == Some(t) &&
    next(e1).nonEmpty)
```

and a post condition:

```
typecheck(e2, Map[Identifier, Type]()) == Some(t)
```

where *e2* represents *next(e1)*. To ensure the post condition, we used the method *holds* of Stainless.

Unforunately, as it is, Stainless is not able to prove preservation on any of the expressions (or at least not in less than ten hours). We tried to add lemmas with the *check* method of Stainless but due to the opacity of the verification system, it is too complicated to find which lemmas can improve the verification, or at least not worsen it.

2.7 Tests

We made tests to check if the type checker and the interpreter had a normal behaviour. By side effect, it also tested the DSL. Some of the tests were made using the method *holds* of Stainless. For example:

```
def testInterpretPlusInteger(): Boolean = {
                interpret(e_+(I(1), I(2))) == I(3)
        }.holds
```

Due to the duration of the verification, beside these tests, we used the ScalaTest library with the FunSuite class.

3 Conclusion

This project was challenging due to the opacity and duration of the verification in Stainless. Sometimes, I had to run Stainless for more than 8 hours to receive a result. The system did not give enough feedback to know if and where it was stuck and what could help it progress. Without the help of my supervisor, Romain Edelmann, I would not have found some of the tricks to help Stainless do the verification, like using pattern matching on list instead of if-then-else expressions to verify the ADT invariant of recursive functions.

I have also lost a lot of time creating a system too complex for Stainless to verify, groping toward the goal and following wrong paths.

Despite these complications, this project was a thrilling experience. It taught me to tackle a problem step-by-step and to use a verification system, which existence I was not aware of.

4 References

References

[1] LARA, EPFL, *Computer Language Processing, Lecture 9, CS-320, Edition 2018* in http://lara.epfl.ch/cc18:top, Date of access: 06.06.19.