

# Verifying correctness of Stainless programs using Coq

Bence Czipó, Jad Hamza

June 8, 2018

## Abstract

I am not sure if it is indeed required for a project report.

## 1 Introduction

Correctness of programs is hard to verify

## 2 Background

### 2.1 Stainless

Stainless is a verification framework for Scala programs. Among other things, it extends scala functions with the notion of pre- and postcondition. From the practical approach, a Stainless program can be divided into two parts: ADT definitions and functions. A general stainless function consists of a precondition  $pre$ , a function body  $body$  and a postcondition  $\lambda res. post(res)$  where

$$pre \implies (\lambda res. post(res))body$$

### 2.2 Coq

Coq is an proof assistant based on the calculus of constructions. It provides functionality to write definitions and theorems and an interactive environment to prove them. Even though Coq is not an automated theorem prover, it also provides some automatism through tactics.

One big advantage of Coq is that there exist several libraries that extend the core features. During our work, we used two of those.

#### 2.2.1 Program Library

Program is a Coq library that introduces the concept of dependent types in a form of  $\{e : T | P\}$  where  $T$  is a type and  $P$  is a predicate.

The program library generates obligations to plug in the "holes" in the types. The environment tries to solve the generated obligation with a user-defined obligation tactic.

### 2.2.2 Coq-Equations

”Equations provides a notation for writing programs by dependent pattern-matching and (well-founded) recursion in Coq. It compiles everything down to eliminators for inductive types, equality and accessibility, providing a definitional extension to the Coq kernel.”

### 2.2.3 Coq-std++

*Coq-std++* (*stdpp*) is a self-contained collection of data structures, lemmas and tactics that is intended to extend the functionality of the standard library of Coq. It features a set representation with all the common operations and a solver tactic to solve goals involving them.

## 3 Transforming Stainless Programs to Coq

In order to verify Stainless programs in Coq, they are needed to be transformed into a correctly typed Coq program.

**Definition 1 (Translation function)** *Let  $t$  be a function that assigns a (correctly typed) Coq program to every (correctly typed) Stainless program. Though  $t$  is not designed to be invertible, for simplicity let us introduce the  $t^{-1}$  notation where  $t^{-1}(t(p)) = p$ . Also for simplicity, let us denote  $t(p)$  by  $[p]$ .*

Many scala construct has its exact representation in Coq. For instance integers can be handled with the Z library, lambda constructs can be translated to coq lambdas, or let's can be translated directly into lets in coq. In the following we will only highlight the non-trivial translation steps. Also, the uniqueness of function and parameter names has to be ensured that we achieve through renaming. From now on, let us assume that every name is unique in its scope.

The if-then-else expressions in Scala could be translated into Coq's own if-then-else constructs, however, during verification, we would like to propagate the boolean value of the condition into the branches. For this reason, we defined our own if-then-else constructs.

```
Definition ifthenelse b A (e1: true = b → A) (e2: false = b → A): A :=
  match b as B return (B = b → A) with
  | true ⇒ fun H ⇒ e1 H
  | false ⇒ fun H ⇒ e2 H
end eq_refl.
```

We use this if-then-else to translate non-exhaustive matches and boolean and operations. Other boolean operations are translated to their Coq correspondent.

The sets and set operations are all translated to the structures of the stdpp library. Coq equality is translated to the = operator in Coq with exception of booleans, integers and sets. The equality operator has Prop result, however, the context in which we use equality usually requires booleans. To convert the two, we defined a propInBool function, however it is also requires to assume a classical type system, i.e. every Prop is either True or False.

### 3.1 Translating ADT's

The translation of whole programs consists of two steps: translating ADT's and translating functions.

In stainless, there are two kinds of ADT's, ADTSort for the root of class hierarchies and ADTConstructor for the case classes.

For example consider the following class hierarchy:

```
sealed abstract class List[T] {}

case class Nil[T]() extends List[T] {}

case class Cons[T](h: T, t: List[T]) extends List[T] {}
```

The translation starts from each ADTSort and creates an inductive definition stating that an ADTSort is one of its constructor. A constructor can be expressed as a type, that takes the types of arguments and maps it to an ADTSort. In our example, the translated version would be.

```
Inductive List (T: Type) :=
| Cons: T → ((List T) → (List T))
| Nil: List T.
```

Based on the constructor used to construct the object, we can define a recognizer, that decides for an abstract supertype if it is instance of a concrete subtype. This can be achieved through pattern matching. For example in case of Cons it would be:

```
Definition isCons (T: Type) (src: List T) : bool :=
match src with
| Cons _ _ _ => true
| _ => false
end.
```

Using this recognizers, we can also define a type for each subtype as a dependent type. For Cons from our example, it would be:

```
Definition Cons_type (T: Type) : Type :=
{src: List T | (isCons T src = true)}.
```

Now, with types corresponding to case classes, we can express the field accessors by simply pattern matching on the object. The corresponding argument of the constructor can be returned. For instance in case of the head of a list it would be:

```
Definition h (T: Type) (src: Cons_type T) : T :=
match src with
| Cons_construct _ f0 f1 => f0
| _ => let contradiction: False := _ in match contradiction with end
end.
```

Note that the second branch is only required so that the match is exhaustive. However due to obligations, that branch is impossible, and the environment is

forced to check that with deriving False.

We also generate some lemmas and tactics using them to rewrite certain expressions using ADT constructs that are not detailed here.

## 3.2 Translating Functions

Since forward declaration is difficult to implement in Coq, it is important to have a total order of functions, in which no function is referenced before it is defined. For this reason, our transforming methods does not support mutually recursive functions. It is easy to see that if there is no mutual recursivity in a set of functions, such order exists.

### 3.2.1 Translating Non-recursive Functions

A function  $f$  with body  $b$ , type parameters  $T_1, \dots, T_k$ , parameters  $p_1, \dots, p_n$  with types  $U_1, U_2, \dots, U_n$  and a return type  $U_r$  can be translated into a simple Coq definition. Preconditions can be interpreted as assertion in the beginning of the function. This can be expressed in Coq by taking an argument that states that the precondition holds, in other means, taking an argument with the type  $\text{pre} = \text{true}$ . Postconditions can be expressed by a dependent return type. In case of a postcondition  $\lambda \text{res}. \text{post}(\text{res})$  the return type changes to  $\{\text{res} : U_r \mid \text{post}(\text{res})\}$ . So a general translated function looks like this:

```
Definition f (T1: Type) ... (Tk: Type) (p1: [U1]) ... (pn: [Un]) (prec1:
  [pre] = true) :
  {res: [Ur] | [post] res} :=
    [b].
```

### 3.2.2 Translating Recursive Functions

The Program library offers `Program Fixpoint` to handle recursive functions, however, the fixpoint operator makes it difficult to rewrite recursive functions with their definitions. To overcome this, we used the Equations library.

In case of recursive functions, we decided to focus on verification, so we ignored termination. For this, we defined an axiom, that we can pass as a decreasing measure:

```
Axiom ignore_termination: nat.
```

Equations do not allow dependent types like the Program library did. In order to work around that, in case of a postcondition, we have to define the return type using a Program Definition, and after, we can use that type in the equations.

Thus in this case, a translated function would be:

```
Definition rt (T1: Type) ... (Tk: Type) (p1: [U1]) ... (pn: [Un]) (prec1:
  [pre] = true) ... : Type :=
  {res: [Ur] | [post] res}

Equations f (T1: Type) ... (Tk: Type) (p1: [U1]) ... (pn: [Un]) (prec1:
  [pre] = true) ... :
```

```

rt T1 ... Tk p1 ... pn prec1 ... :=
  f T1 ... Tk p1 ... pn prec1 ... by rec ignore_termination lt :=
  f T1 ... Tk p1 ... pn prec1 ... :=
  [b].

```

Later, if we want to rewrite with the content of the function, we can rely on the fact that it will be expressed by `f_equation_1`.

### 3.2.3 Function Invocation

The function application can be translated as function application in Coq, however, it must be taken into account, that functions with preconditions take more parameters. The program library allows us to pass unknown arguments (underscores) and requires us to construct proof for them later in form of obligation proofs. When the invoked function has a postcondition the return type is not exactly what is a refined type, that needs to be projected.

## 4 Correctness Equivalence

The goal of the previous translation was to be able to reason about the correctness of the original Stainless program, based on the provability of the Coq representation. We need that for every Stainless program  $p$  we have.

1. if a stainless program  $p$  is valid, than  $t(p)$  is also valid,
2. if  $t(p)$  is valid, than  $p$  is also valid.

contratdiction(p: False) : T := match p with end.

assume t crashes it takes a branch that leads to error than the cq representation t' take some reduction branch to a state where  $\neg p$ : False (empty context implies p of type false)

Contradiction, because every correctly typed expression should evaluate to a value of that type and there is no value of type False

We assume that if a term b evaluate to true in stainless ( $b \dashv_i \text{true}$ ) than in coq  $t(b) \dashv_{\text{coq}} \text{true}$

thus an if will chose that branch

Not so clear with params, e.g. functions

## 5 Automatically Solving the Generated Goals

## 6 Implementation/Architecture/Results

We implemented the translation methods in `stainless.verificaton.CeqEncoder.scala` and `stainless.verificaton.CeqExpression.scala`. The generated methods are written into temporary files, and `coqc` is invoked on them. For each function, a separate `.v` file is created, and each file contains all the dependencies of the function. For simplicity, we assumed a function depends on all ADT's and the functions it invokes directly or transitively. In order to speed verification and to eliminate a domino effect caused by one failing verification step, all obligations

of the dependencies are admitted. The logic handling .v files can be found in `stainless.verifi-  
cation.CoqI0.scala`

In order to integrate our solution into the Stainless framework we replaced `VerificationChecker` with `CoqVerificationChecker`. It creates a verification condition for each method (file) and puts its result to the report. Because functions are verified individually, their verification can be canceled, or timeout can be defined for them. In order to activate coq verification, a `--coq` flag has to be passed to stainless.

To benchmark the translation and verification, it has been run on the Lists library with 10 minutes timeout (per function case). The results are summarized in the following table:

## 7 Conclusions

We have implemented methods to transform Stainless programs with recursive functions and other nontrivial constructs into Coq programs. We also defined tactics to automatically generate proof for most of the cases. Our tactics are able to perform case analysis, and recursive function inlining without infinite loops. The automatic proof generation handles the majority of the functions in the Lists library. We also integrated our solution into the Stainless framework.

Future work includes extending the set of supported Stainless elements, for example with mutually recursive functions. The tactics can be also enhanced further to handle the remainder of the library. There is also room for improvement with the speed of tactics, as for example the external set-solver from `stdpp` takes a lot of time to fail, thus slows down every case where recursive inlining is needed. Checking termination of recursive functions is also a possible enhancement.

## References

- [1] K. Grove-Rasmussen og Jesper Nygård, *Kvantefænomener i Nanosystemer*. Niels Bohr Institute & Nano-Science Center, Københavns Universitet