Verifying correctness of Stainless programs using Coq

Bence Czipó

Advisors: Jad Hamza, Viktor Kunčak

June 9, 2018

1 Introduction

Since the appearance of the first software, formally proving their correctness has been a hot topic. There are several approaches to perform formal verification using different foundations such as satisfiability modulo theories (SMT) or the calculus of inductive constructions.
Stainless, a software verification framework developed at École Polytechnique Fédérale de Lausanne, uses SMT solvers, whereas Coq is the most known representer of proof assistant using the calculus of inductive constructions.
In case of formal verification tools, the question always remains, whether the tool is correct. In other terms, who is going to verify the verification checker? One approach to do so can be to transform the input of one to an other, broadly accepted tool, and compare the verification results.
In this report, I going to propose a verification method for Stainless programs using the Coq proof assistant. The translation and verification is done automatically, and integrates to the Stainless toolchain. During this report, I only focus on correctness, and ignore termination of programs, moreover I limit the allowed Stainless expressions. To benchmark my work, at the end of the report, I include the verification result of the method run in the Lists library of Stainless.

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 (for simplicity, let us assume that it takes only one parameter of type A) consists of a precondition pre : $A- > Bool$, a function body body and a postcondition λres. post(res) where

$$\forall x : A,\ \text{pre}(x) \implies (\lambda\text{res. post(res)})\text{body}$$

2.2 Coq

Coq is an proof assistant based on the calculus of inductive 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 allows the users to convert between types and dependent types implicitly while generating obligations to plug in holes in the context. The library also allows us to write incomplete expressions using the underscore character, and will generate further obligations to fill in the holes. The environment tries to solve the generated obligations with a user-defined obligation tactic, which gives the possibility to automate the proofs.
For example, we can have the following definition:

```
Definition add (n : nat)(m: nat) (p: n >= m) : { x : nat | x > 2 } :=
Nat.add n 1.
```

The refined type will generate an obligation we have to prove:

```
∀ n m : nat, n ≥ m → (λ x : nat, x > 2) (n + 1)%nat
```

Using add, we can give an example for obligations generated to plug in holes in types:

```
Definition mul2(n: nat): nat := add n n _.
```

It will generate require us to give an expression of the type of the missing part, specifically in this case

```
∀ n: nat, n ≥ n
```

2.2.2 Coq-Equations

The Equations library provides a notation to write recursive programs in Coq. It defines equations between function definition and its body which will come in hand when rewriting recursive functions.

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 fetaures 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. During verification, we would like to propagate the boolean value of the condition into the branches. Even though if-then-else of Program and Equations does so, in order to have more flexibility with the expressions, 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 reutrned. For instnace in case of the ead 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

The functions are ordered into a total order, where a function is preceded by each of its dependencies. The translation currently does not support mutually recursive functions, so it can be pointed out that this way 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 expressed in Coq by taking an argument that states that the precondition holds, in oder means, taking an argument with the type pre = true. Postconditions can be expressed by a dependent return type. In case of a postcondition λres. post(res) the return type changes to $\{\text{res} : U_r \mid \text{post(res)}\}$. 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.
Equations do not allow dependent types like the Program library did. In order to work around that, in case of pre- and postconditions, we have to define the 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 prt (T1: Type) ... (Tk: Type)(p1: [U1]) ... (pn: [Un]) :
 Type := [pre] = true
...
Definition rt (T1: Type) ... (Tk: Type)(p1: [U1]) ... (pn: [Un]) (prec1:
    prt T1 ... Tk p1 ... pn) ... : Type :=
     {res: [Ur] | [post] res}

Equations f (T1: Type) ... (Tk: Type)(p1: [U1]) ... (pn: [Un]) (prec1:
    prt T1 ... Tk p1 ... pn) ... :
   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

We only checked two aspects of validity of stainless programs, so let us define validity accordingly

Definition 2 (Stainless-validity) *A Stainless program p is valid if*

- *the preconditions hold for every call of f*
- *for any input satisfying the preconditions, the postcondition holds*

In stainless, pre- and postconditions can be expressed as evaluating a boolean condition, and if the condition does not hold, go into an error branch. So in other terms, p is correct, if these error branches are never active.

Errors are translated to Coq using contradiction: $[\text{error}] = \text{contradiction}$ _ Contradictions can be expressed as the obligation to generate false.

```
Definition contradiction (T: Type)(p: False) : T := match p with end.
```

Applying `contradiction` on an unknown variable will result in an obligation to derive `False` from the context.

Definition 3 (Coq-validity) *A Coq program p is valid if there is a proof (an expression of that type) for every obligation generated by it.*

The goal of the translation presented in Section 3 was to be able to reason about the validity (or correctness) of the original Stainless program, based on the validity of the translated Coq representation.

Theorem 1 *For every (correctly typed) Stainless program p and its translation $[p]$, if $[p]$ is proved to be valid by Coq, than p is also valid.*

Instead of proving this we will just sketch some notions why is it true.

Assumption 1 *For every every Scala term `b: Boolean` we assume that if $b \to^* c$ where $c \in \{true, false\}$ then $[b] \to^*_{coq} [c]$, so if b evaluates to a boolean constant in Stainless (under the context Γ), its translated representation evaluate to the representation of that boolean constant in Coq (under the context $[\Gamma]$).*

Let us examine only the cases where every decision in the program p is a branching based on a boolean variable. We will also assume that p does not have any free variables. Our statement can be proved by contradiction, assuming that $[p]$ is proven to be correct, but p crashes. In this case, there is a branch in p that contains error, and it is accessed through evaluating the boolean expressions $b_1, b_2 \dots b_n$. We can apply Assumption 1, add conclude that $[p]$ reduces to the same branch. We know that Coq was able to verify $[p]$, which means it was able to prove the obligation that was generated by contradiction. This means that we had $\{\} \vdash x : False$ as hypothesis.
Here, we can use two facts:

Fact 1 *In Coq, if $\{\} \vdash t : T$, then $t \to^*_{coq} v$, where v is a value of T*

Fact 2 *There is no value of type `False`.*

We have reached contradiction, so we have "proved" our statement.

5 Automatically Solving the Generated Goals

In order to automate the generation of proofs, we defined several tactics, that are invoked in a strict order.
The tactics are carried out in the following order:

1. First of all, we try to solve the goal with some fast tactics.
2. Some basic rewrites are performed, along with some fast tactics.
3. More time-demanding tactics are performed, concretely `omega`, `ring`, `eauto` and the `set-solver` tactic of the stdpp library.
4. Case analysis of if-then-else branches.
5. Rewrites with function bodies and recognizers.

5.1 Fast tactics

Fast tactics involve tactics that fail fast, so that they do not block execution. They include basic tactics such as `cbn`, `intros`, `intuition`, ... We also rewrite with a basic set of rewrite rules concerning integer and boolean operations. We also define some lemmas to rewrite with, for example ones handling boolean and Prop relations:

```
   forall P, propInBool P = true ↔ P
```

or ones handling the dependent of-then-else:

```
ifthenelse b T e1 e2 = value ↔ (
   (exists H1: true = b, e1 H1 = value) ∨
   (exists H2: false = b, e2 H2 = value)
).
```

In case of recursive functions, we decided to focus on verification, so we ignored termination. For this, we defined a tactic to admit all obligations related to termination, which is also included among the fast tactics, so that Coq does not waste much time solving it.

5.2 Basic tactics

Basic tactics involve

- rewriting with boolean constants or values
- destruction of simple constructs such as exists expressions or refinement
- performing property and boolean rewrites (rewrites of `propInBool`)

5.3 Case analysis

Case analysis involves branching on dependent and non-dependent (Coq's built in) if-then-else expressions with simplifications whenever possible.

5.4 Rewrites

The rewrites performed at the very end, mostly because they usually overcomplicate simple problems. First, we rewrite with non-recursive types. This step is guaranteed to terminate, as mutual recursion is not allowed.
After, we rewrite with the equations defined for recursive types. Note that in some cases, this process might loop. We only rewrite with the body of recursive functions, if their body will probably not be the subject of further refinement. For example, for a list, if we rewrite size, we have to perform case analysis later, to check if the argument was Nil or Cons. This just introduce more branches to deal with. However, if we know that the list is Nil or Cons, we can rewrite with `0` or `1 + size tail`.
Eventually, if the tactics reach it, we rewrite with the definition of recognizers.

6 Implementation/Architecture/Results

We implemented the translation methods in `stainless.verification.CoqEncoder.scala` and `stainless.verification.CoqExpression.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.verification.CoqIO.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.
The tactics can be found in a separate slc-lib directory, where they have been separated based on their functionality. The tactics are included automatically into every generated file, however, before running the verification, the SLC module has to be generated by running make in the slc-lib directory.
To benchmark the translation and verification, it has been run on the Lists library with 5 minutes timeout (per function case). The results are summarized in Table 1. Out of 77 methods, 48 is valid, 9 is invalid (there are obligations remaining that the tactics can not solve) and 20 times out (most likely because looping rewrite of recursive functions).

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 analyis, 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.

Table 1: Verification results for the list library

Function name	**Status**	**Time**	**Function name**	**Status**	**Time**
&	timeout	300.002	insertAtImpl	invalid	35.279
++	timeout	300.003	isDefined	valid	2.005
-	timeout	300.001	isEmpty	valid	2.004
–	timeout	300.002	isEmpty	valid	1.972
:+	timeout	300.017	last	valid	58.399
::	valid	2.04	lastOption	valid	4.262
apply	invalid	7.736	length	valid	2.138
chunk0	invalid	7.569	map	valid	5.812
chunks	valid	31.083	map	valid	2.342
contains	valid	7.43	nonEmpty	valid	1.903
content	valid	2.093	nonEmpty	valid	1.945
count	timeout	300.012	orElse	valid	2.017
drop	timeout	300.006	padTo	timeout	300.005
dropWhile	timeout	300.02	partition	timeout	300.011
empty	valid	1.992	replace	timeout	300.014
evenSplit	valid	28.316	replaceAt	invalid	91.058
exists	valid	2.055	replaceAtImpl	invalid	49.727
exists	valid	1.99	reverse	timeout	300.002
filter	timeout	300.005	rotate	timeout	300.001
filter	valid	2.069	scanLeft	valid	2.256
filterNot	timeout	300.006	scanRight	valid	24.657
find	valid	6.975	size	valid	2.198
flatMap	valid	5.927	slice	valid	30.364
flatMap	valid	1.997	split	invalid	3.786
flatten	valid	5.268	splitAt	valid	2.944
foldLeft	valid	2.206	splitAtIndex	timeout	300.432
foldRight	valid	2.181	tail	valid	2.288
forall	valid	2.104	tailOption	valid	2.703
forall	valid	1.983	tails	valid	2.532
get	valid	2.167	take	timeout	300.001
getOrElse	valid	2.01	takeWhile	timeout	300.336
groupBy	valid	3.65	toSet	valid	2.282
head	valid	2.142	toSet	valid	2.063
headOption	valid	2.126	unique	valid	3.191
indexOf	valid	178.239	updated	invalid	33.731
indexWhere	valid	185.303	withFilter	valid	278.839
init	timeout	300.001	withFilter	valid	2.156
insertAt	invalid	33.998	zip	timeout	300.003
insertAt	invalid	29.234			

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.