Verifying correctness of Stainless programs using Coq

Bence Czipó, Jad Hamza

June 5, 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 λres. post(res) where

$$\text{pre} \implies (\lambda\text{res. post(res)})\text{body}$$

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 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 $t^{-1}(t(p)) = p$.*

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

3.1 Translating ADT's

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

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.

5 Automatically Solving the Generated Goals

6 Implementation/Architecture/Results

Replaced verification checker with own verification checker.

Generate a separate coq file for every function. Admit obligation for all the dependencies, and only verify that one.

Include results in a table form

References

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