Verifying correctness of Stainless programs using Coq

Bence Czipó

École Polytechnique Fédérale de Lausanne

bence.czipo@epfl.ch

June 14, 2018

Overview

1. Translation
   - Basic Concepts
   - Transforming Abstract Data Types
   - Transforming Methods
   - Transforming Recursive Methods
2. Relation Between Proofs and Correctness
3. Automated Verification
4. Implementation
5. Benchmark

The Translation Function

Definition (Translation function)

Let t be a function that assigns a (correctly typed) Coq program to every (correctly typed) Stainless program.

The Translation Function

Definition (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 Function

Definition (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 $[p]$.

Uniqueness of names

Translation loses some scoping rules

Uniqueness of names

Translation loses some scoping rules

Uniqueness of names is ensured by renaming.

Uniqueness of names

Translation loses some scoping rules

Uniqueness of names is ensured by renaming.

$x \to x_0$,

Uniqueness of names

Translation loses some scoping rules

Uniqueness of names is ensured by renaming.

$x \rightarrow x_0$,

$x \rightarrow x_1$,

...

Uniqueness of names

Translation loses some scoping rules

Uniqueness of names is ensured by renaming.

$x \rightarrow x_0$,
$x \rightarrow x_1$,
...

Polymorphism is not supported, methods are needed to be renamed.

Uniqueness of names

Translation loses some scoping rules

Uniqueness of names is ensured by renaming.

$x \to x_0$,
$x \to x_1$,
...

Polymorphism is not supported, methods are needed to be renamed.

```
def forall[T](l : List[T], p: T -> Boolean)-> forall0
```

Uniqueness of names

Translation loses some scoping rules

Uniqueness of names is ensured by renaming.

$x \rightarrow x_0$,
$x \rightarrow x_1$,
...

Polymorphism is not supported, methods are needed to be renamed.

```
def forall[T](l : List[T], p: T -> Boolean)-> forall0

def forall[T](o: Option[T], p: T -> Boolean)-> forall1
```

From now on, unique names are assumed.

The Program Library

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:

$$\forall\ n\ m : nat,\ n \geq m \rightarrow (\lambda\ x : nat,\ x > 2)\ (n + 1)\%nat$$

The Program Library

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
```

Translating simple types

Most constructs have exact Coq representation

- $[\text{BigInt}] = [\text{Int}] = Z$
- $[\text{Boolean}] = \mathit{bool}$ (not Prop)
- $[(a, b, \ldots)] = ([a], [b], \ldots)$
- $[f(p_1, p_2, \ldots)] = (f\ [p_1]\ [p_2] \ldots)$
- $[\lambda x.\ e] = (\text{fun } x \implies [e] \ldots)$

Translating error

Errors are translated to Coq using contradiction:

[error] = contradiction _

Contradictions can be expressed as the obligation to prove false.

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

Dependent if-then-else

In the expression `if (p) then tb else fb`, we would like to propagate the boolean value of p to the branches. There are some solutions, but for more flexibility with the expressions, we defined our own.

```
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.
```

Used for:

- if then else
- non-exhaustive matches
- boolean and (`&&`)

Translating equality

Equality is usually translated using the coq equality.

In Coq, `a = b` is of type `Prop`.

```
Definition propInBool (P: Prop): bool :=
if (classicT P)
then true
else false.
```

Decidable type system: every Prop is True or False

```
Axiom classicT: forall P: Prop, P + ¬P.
```

Translating equality

Equality is usually translated using the coq equality.

In Coq, `a = b` is of type `Prop`.

```
Definition propInBool (P: Prop): bool :=
if (classicT P)
then true
else false.
```

Decidable type system: every Prop is True or False

```
Axiom classicT: forall P: Prop, P + ¬P.
```

Exceptions: BigInt, Int, Boolean and Sets

Translating Sets

The Coq standard library comes with a limited set implementation.

Translating Sets

The Coq standard library comes with a limited set implementation.

Coq-std++ (stdpp): collection of data structures, lemmas and tactics.

Translating Sets

The Coq standard library comes with a limited set implementation.

Coq-std++ (stdpp): collection of data structures, lemmas and tactics.

Contains sets with every common operation and `set_solver` tactic to solve obligations

The Program Library

Program is a Coq library that:

- allows us to convert between types and dependent types implicitly

The Program Library

Program is a Coq library that:

- allows us to convert between types and dependent types implicitly
- generates obligations to "plug in" holes in the context

The Program Library

Program is a Coq library that:

- allows us to convert between types and dependent types implicitly
- generates obligations to "plug in" holes in the context
- also allows incomplete missing parameters, for which, it will generate obligations

The Program Library

Program is a Coq library that:

- allows us to convert between types and dependent types implicitly
- generates obligations to "plug in" holes in the context
- also allows incomplete missing parameters, for which, it will generate obligations
- defines an Obligation Tactic to solve the generated obligations

Structure

Stainless programs from our point of view are:

Structure

Stainless programs from our point of view are:

- ADT Definitions

Structure

Stainless programs from our point of view are:

- ADT Definitions
  - ADTSort

Structure

Stainless programs from our point of view are:

- ADT Definitions
  - ADTSort
  - ADTConstructor

Structure

Stainless programs from our point of view are:

- ADT Definitions
  - ADTSort
  - ADTConstructor
- Methods on them

Structure

Stainless programs from our point of view are:

- ADT Definitions
  - ADTSort
  - ADTConstructor
- Methods on them
  - Non-recursive

Structure

Stainless programs from our point of view are:

- ADT Definitions
  - ADTSort
  - ADTConstructor
- Methods on them
  - Non-recursive
  - Recursive

Structure

Stainless programs from our point of view are:

- ADT Definitions
  - ADTSort
  - ADTConstructor
- Methods on them
  - Non-recursive
  - Recursive
  - Mutually Recursive

Structure

Stainless programs from our point of view are:

- ADT Definitions
  - ADTSort
  - ADTConstructor
- Methods on them
  - Non-recursive
  - Recursive
  - ~~Mutually Recursive~~

Transforming Abstract Data Types

List Example

Best explained through an example:

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

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

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

Type definitions

Inductive type definitions for ADTSorts. The sematntics behind this is that an ADTSort is one of its constructors.

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

Recognizing Types

We can use pattern matching to check for concrete subtype

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

Using the recognizers, we can define a type for the subtypes as a refined type.

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

Accessing Fields

Now that we have the subtypes, we can have accessors to their fields

```
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.
```

Methods

A general method is built up from

- a function name f
- type parameters $T_1 \dots T_k$ and arguments p_1 of type U_1, p_2 of type U_2 ... , p_n of type U_n and a return type U_r
- a precondition pre of type `A => Boolean`, where $A \preceq \{U_1 x \dots x U_n\}$
- a body b of type $\{U_1 x \dots x U_n\} \Longrightarrow U_r$
- a postcondition *post* of type U_r`=> Boolean`

If there are more pre- and postcondition, they can be combined into one pre- and postcondition using conjunction.

Transforming Methods

Transforming non-recursive methods

If there is no recursion involved, the translation is straightforward.

Transforming non-recursive methods

If there is no recursion involved, the translation is straightforward.

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.

Transforming non-recursive methods

If there is no recursion involved, the translation is straightforward.

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 $\{\text{res} : U_r \mid \text{post(res)}\}$.

An example

```
def f[T1 ... Tk](p1: U1, ... pn: Un): Ur = {
require(pre(A))
b
} ensuring {res => post(res)}
```

Is translated to

$$
\begin{array}{l}
\texttt{Definition f } (T_1\texttt{: Type}) \texttt{ ... } (T_k\texttt{: Type})(p_1\texttt{: } [U_1]) \texttt{ ... } (p_n\texttt{: } [U_n]) \\
\quad \texttt{(prec: [pre] = true) :} \\
\texttt{\{res: } [U_r] \mid \texttt{[post] res\} :=} \\
[b]\texttt{.}
\end{array}
$$

Transforming Recursive Methods

Translating Recursive Methods

Program library has Program Fixpoint to write recursive functions.

Makes it extremely hard to rewrite.

We used *CoqEquations* instead.

```
Equations negb (b : bool) : bool :=
  negb true := false ;
  negb false := true.
```

Persevering benefits of Program

We still want to handle pre- and postconditions using Program.

We can translate preconditions into dependent types:

```
Definition prt (T_1: Type) ... (T_k: Type)(p_1: [U_1]) ... (p_n: [U_n]) :
Type := [pre] = true
```

And postconditions just the same:

```
Definition rt (T_1: Type) ... (T_k: Type)(p_1: [U_1]) ... (p_n: [U_n])
    (prec: prt T_1 ... T_k p_1 ... p_n) : Type :=
{res: [U_r] | [post] res}
```

Persevering benefits of Program

The previous example in the recursive case would be translated to:

```
Equations f (T_1: Type) ... (T_k: Type)
            (p_1: [U_1]) ... (p_n: [U_n])
            (prec: prt T_1 ... T_k p_1 ... p_n) :
            rt T_1 ... T_k p_1 ... p_n prec :=
                f T_1 ... T_k p_1 ... p_n prec by rec
                    ignore_termination lt :=
                f T_1 ... T_k p_1 ... p_n prec :=
                    [b].
```

The `ignore_termination` is the decreasing measure in the recursion. We will have an obligation proving it to be decreasing.

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`.

Function Application

How to write function application depends on whether the it has pre- or postcondition

- If the method had preconditions, we pass _, so that Program will generate obligations for us.
- If there is a postcondition, the result has to be projected

```
proj1_sig (f T1 ... Tj p1 ... pn _)
```

Relation Between Proofs and Correctness

Validity - Stainless

Definition (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

Validity - Stainless

Definition (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

Pre- and postcondition branching, failing postcondition go into an error branch. Errors are translated to Coq using contradiction:

$$[\text{error}] = \text{contradiction } \_$$

Validity - Stainless

Definition (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

Pre- and postcondition branching, failing postcondition go into an error branch. Errors are translated to Coq using contradiction:

$$[\text{error}] = \text{contradiction } \_$$

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

Validity - Coq

Definition (Coq-validity)

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

Validity - Coq

Definition (Coq-validity)

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

We can define relation between the two

Validity - Coq

Definition (Coq-validity)

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

We can define relation between the two

Theorem

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.*

Validity - Coq

Definition (Coq-validity)

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

We can define relation between the two

Theorem

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.

Proof(ish)

Assumption

For every every Scala term `b: Boolean` *we assume that if* $b \rightarrow^* c$ *where* $c \in \{true, false\}$ *then* $[b] \rightarrow^*_{coq} [c]$.

Proof(ish)

Assumption

For every every Scala term `b: Boolean` *we assume that if* $b \to^* c$ *where* $c \in \{true, false\}$ *then* $[b] \to^*_{coq} [c]$.

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 $[\Gamma]$)

Proof(ish)

Assumption

For every every Scala term `b: Boolean` *we assume that if* $b \rightarrow^* c$ *where* $c \in \{true, false\}$ *then* $[b] \rightarrow^*_{coq} [c]$.

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 $[\Gamma]$)

Assumption

Let us assume that in the program p, every decision is a branching based on a boolean condition.

Proof(ish)

Assumption

For every every Scala term `b: Boolean` *we assume that if* $b \rightarrow^* c$ *where* $c \in \{true, false\}$ *then* $[b] \rightarrow^*_{coq} [c]$*.*

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 $[\Gamma]$)

Assumption

Let us assume that in the program p, every decision is a branching based on a boolean condition.

Assumption

Let us assume that p does not contain any free variables.

Proof(ish)

By contradiction: assume $[p]$ is proven to be correct, but p crashes

Proof(ish)

By contradiction: assume $[p]$ is proven to be correct, but p crashes

there is a branch in p that contains error, and it is accessed through evaluating the boolean expressions $b_1, b_2 \ldots b_n$

Proof(ish)

By contradiction: assume $[p]$ is proven to be correct, but p crashes

there is a branch in p that contains error, and it is accessed through evaluating the boolean expressions $b_1, b_2 \ldots b_n$

$[p]$ reduces to the same branch $\rightarrow$ contradiction

Proof(ish)

By contradiction: assume $[p]$ is proven to be correct, but p crashes

there is a branch in p that contains error, and it is accessed through evaluating the boolean expressions $b_1, b_2 \ldots b_n$

$[p]$ reduces to the same branch $\rightarrow$ contradiction

Coq was able to prove it: $\{\} \vdash x : False$

Proof(ish)

By contradiction: assume $[p]$ is proven to be correct, but p crashes

there is a branch in p that contains error, and it is accessed through evaluating the boolean expressions $b_1, b_2 \dots b_n$

$[p]$ reduces to the same branch $\rightarrow$ contradiction

Coq was able to prove it: $\{\} \vdash x : False$

Fact

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

Fact

There is no value of type `False`.

Automated Verification

Automatic Verification

How to solve obligations automatically?

Automatic Verification

How to solve obligations automatically?

Define obligation tactic

Automatic Verification

How to solve obligations automatically?

Define obligation tactic

1. Fast tactics
2. Basic Tactics
3. Slow Tactics
4. Set Tactics
5. Case Analysis
6. Rewrite

Fast tactics

Some Coq tactics are fast to fail:

- cbn
- intros
- intuition
- discriminate
- ...

Fast Rewrites

Integer operations:

`forall x y : Z, (x <=? y)= false` $\leftrightarrow$ `y < x`

Boolean operations:

`forall a b : bool, eqb a b = true` $\leftrightarrow$ `a = b`

Fast Rewrites

Integer operations:

`forall x y : Z, (x <=? y)= false` $\leftrightarrow$ `y < x`

Boolean operations:

`forall a b : bool, eqb a b = true` $\leftrightarrow$ `a = b`

Own rewrite lemmas about booleans:

`forall b1 b2, negb b1 = negb b2` $\leftrightarrow$ `b1 = b2`

Fast Rewrites

Integer operations:

`forall x y : Z, (x <=? y)= false` $\leftrightarrow$ `y < x`

Boolean operations:

`forall a b : bool, eqb a b = true` $\leftrightarrow$ `a = b`

Own rewrite lemmas about booleans:

`forall b1 b2, negb b1 = negb b2` $\leftrightarrow$ `b1 = b2`

Rewrite lemmas with Props and bools:

`forall P, propInBool P = true` $\leftrightarrow$ `P`

Fast Rewrites

Integer operations:

```
forall x y : Z, (x <=? y)= false ↔ y < x
```

Boolean operations:

```
forall a b : bool, eqb a b = true ↔ a = b
```

Own rewrite lemmas about booleans:

```
forall b1 b2, negb b1 = negb b2 ↔ b1 = b2
```

Rewrite lemmas with Props and bools:

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

Own rewrite lemmas about if-then-else:

```
forall b: bool, (if b then true else false)= b.
```

Fast Rewrites

Also some rewrite lemmas with dependent if-then-else:

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

or

```
forall b (e1: true = b → bool),
   ifthenelse b bool e1 (fun _ ⇒ false) = true ↔
   exists H: true = b, e1 H = true.
```

Admitting termination obligations

Some obligations are related to termination, that we ignore currently.

We will have `(ignore_termination < ignore_termination)%nat` in the goal

Specific tactic to recognize it, and admit it.

```
match goal with
(...)
| |- (S ?T <= ?T)%nat ⇒
   unify T ignore_termination; apply False_ind; exact unsupported
(...)
end.
```

Basic Tactics

- Rewriting with boolean constants and value
- Destruction of exists, refinement, etc...
- Prop $\leftrightarrow$ bool rewrites

Slow Tactics

Coq's "not-so-fast" tactics:

- omega
- ring
- eauto (currently not included)

Set Tactics

Solve sets using `set_solver` of stdpp.

Also includes some basic set rewrites before:

- $\emptyset \cup s = s$
- $s \cup \emptyset = s$
- $s_1 = s_2 \implies s_3 \cup s_1 = s_3 \cup s_2$
- $s_1 = s_2 \leftrightarrow s_1 \subseteq s_2 \wedge s_2 \subseteq s_1$
- $x \in s_1 \vee x \in s_2 \leftrightarrow x \in (s_1 \cup s_2)$
- ...

Case Analysis

If we encounter an if-then-else (both dependent and non-dependent) or a match we can

- rewrite using some smart rule, e.g.

Case Analysis

If we encounter an if-then-else (both dependent and non-dependent) or a match we can

- rewrite using some smart rule, e.g.

```
(forall H1: true = b, e1 H1 = value)→
(forall H2: false = b, e2 H2 = value)→
ifthenelse b T e1 e2 = value.
```

Case Analysis

If we encounter an if-then-else (both dependent and non-dependent) or a match we can

- rewrite using some smart rule, e.g.

  ```
  (forall H1: true = b, e1 H1 = value)→
  (forall H2: false = b, e2 H2 = value)→
  ifthenelse b T e1 e2 = value.
  ```

- perform case-analysis

Rewrites

Rewrites can cause loops and obfuscate the goal.

The order is important:

Rewrites

Rewrites can cause loops and obfuscate the goal.

The order is important:

1. Rewrite with non-recursive types

Rewrites

Rewrites can cause loops and obfuscate the goal.

The order is important:

1. Rewrite with non-recursive types
2. Rewrite with the body of recursive functions

Rewrites

Rewrites can cause loops and obfuscate the goal.

The order is important:

1. Rewrite with non-recursive types
2. Rewrite with the body of recursive functions
3. Rewrite with recognizers (`isXY`)

Rewrites

Rewrites can cause loops and obfuscate the goal.

The order is important:

1. Rewrite with non-recursive types
2. Rewrite with the body of recursive functions
3. Rewrite with recognizers (`isXY`)

Rewriting with the body of recursive methods does not happen here, rather, they are just put into the context as equations.

Rewrites

Rewrites can cause loops and obfuscate the goal.

The order is important:

1. Rewrite with non-recursive types
2. Rewrite with the body of recursive functions
3. Rewrite with recognizers (`isXY`)

Rewriting with the body of recursive methods does not happen here, rather, they are just put into the context as equations.

Right after fast tactics, whenever we see an **appropriate** equation we rewrite with it

Appropriate?

We only rewrite with the body of recursive functions, if their body will probably not be the subject of further refinement.

```
Rw: size T l = match l with
 | Nil T ⇒ 0
 | Cons T x xs ⇒ 1 + size T xs
end.
```

This will just introduce more branches to deal with, and we would like to perform destructing before rewriting.

Appropriate?

However, if we know that `l` is cons:

```
l, ys : List T
y: T
H : l = Cons y ys
(...)
Rw : size T l = match (Cons y ys) with
| Nil T ⇒ 0
| Cons T x xs ⇒ 1 + size T xs
end.
```

We can simplify it to:

```
Rw: size T l = 1 + size T ys
```

Implementation

Implementation

Integrate into Stainless toolchain

Implementation

Integrate into Stainless toolchain

Generate separate files per function, that includes all dependencies

Implementation

Integrate into Stainless toolchain

Generate separate files per function, that includes all dependencies

- All ADT's
- (Transitively) invoked functions

Implementation

Integrate into Stainless toolchain

Generate separate files per function, that includes all dependencies

- All ADT's
- (Transitively) invoked functions

`Admit Obligations` for dependencies

- Saves time
- Eliminates domino effect

A method is valid if the verification condition generated for it is valid, and the verification conditions of all of its dependencies are correct too.

Architecture

VerificationChecker $\rightarrow$ CoqVerificationChecker

Architecture

VerificationChecker $\rightarrow$ CoqVerificationChecker

CoqIO.scala: invoke coqc and check the output

Architecture

VerificationChecker → CoqVerificationChecker

CoqIO.scala: invoke coqc and check the output

Possible output:

- Valid

Architecture

VerificationChecker $\rightarrow$ CoqVerificationChecker

CoqIO.scala: invoke coqc and check the output

Possible output:

- Valid
- Invalid: the execution terminated without solving one or more obligations

Architecture

VerificationChecker $\rightarrow$ CoqVerificationChecker

CoqIO.scala: invoke coqc and check the output

Possible output:

- Valid
- Invalid: the execution terminated without solving one or more obligations
- Timeout: the verification timed out

Architecture

VerificationChecker → CoqVerificationChecker

CoqIO.scala: invoke coqc and check the output

Possible output:

- Valid
- Invalid: the execution terminated without solving one or more obligations
- Timeout: the verification timed out
- Error: the verification failed, because of some internal error, most likely an error in the generated file

Architecture

VerificationChecker → CoqVerificationChecker

CoqIO.scala: invoke coqc and check the output

Possible output:

- Valid
- Invalid: the execution terminated without solving one or more obligations
- Timeout: the verification timed out
- Error: the verification failed, because of some internal error, most likely an error in the generated file
- Canceled: the verification was canceled

Benchmark

List library

- Recursive, but not mutually recursive methods
- Inductive ADT's
- Set operations (contains, content, intersection, ...)
- Integer operations (size, indexOf, indexWhere, ...)

List library

- Recursive, but not mutually recursive methods
- Inductive ADT's
- Set operations (contains, content, intersection, ...)
- Integer operations (size, indexOf, indexWhere, ...)
- 77 methods in total

Results

Verification run with 5 minutes timeout

- Valid: 66
- Invalid: 0
- Timeout: 11
- Error: 0

Conclusions

- Translating Stainless programs to Coq
- Tactics to solve generated goals
- Benchmark using List library (66/77 verified)

Conclusions

- Translating Stainless programs to Coq
- Tactics to solve generated goals
- Benchmark using List library (66/77 verified)

Future work:

- Extending the supported stainless expressions
- Enhance tactics so that they handle the whole library
- Speed up tactics, external `set-solver` tactic is really slow to fail, blocks every execution requiring a rewrite with definition
- Check termination

Questions?