Experiments with Formal Verification of Scala Code @TODO ok?

Place your subheading here

Bachelor thesis

[Insert short text (abstract) if desired]
This document serves as a template for the compilation of reports according to the guidelines of the BFH. The template is written in LATEX and supports the automatic writing of various directories, references, indexing and glossaries. This small text is a summary of this document with a length of 4 to max. 8 lines.
The cover picture may be turned on or off in the lines 157/158 of the file template.tex.

Degree course:	Computer Science
Authors:	Anna Doukmak, Ramon Boss
Tutor:	Kai Brünnler // TODO Dr.?
Experts:	Urs Keller // TODO remove?
Date:	TODO fix this

Versions

Version	Date	Status	Remarks
1.0	TODO fix this	Final	Final Version

Abstract

In this thesis some experiments with formal verification are performed and examined. To this end, parts of the implementation of the Bitcoin protocol in Bitcoin-S are verified for correctness using the verification framework Stainless.

In the first part the goal of the work is stated. A brief overview of the main blocks of the study is given: concepts of formal verification, the verification framework Stainless and the Scala implementation of the Bitcoin protocol.

Chapter 2 outlines the first practical steps of using Stainless. The integration process of the Stainless framework into a Scala project is described, and some aspects of language compatibility are visited in detail.

In chapter 3 the principle and process of transaction creation are demonstrated with Bitcoin-S.

In chapter 4 the correctness of transaction verification in Bitcoin-S is examined. The difficulties emerged during the integration of Bitcoin-S and Stainless are covered.

Chapter 5 highlights the verification process of coin addition in Bitcoin-S using the Stainless framework. The challenges encountered and solutions found are described.

Contents

1. Introduction (Aufgabestellung)

In this work some experiments in formal verification of Scala code will be accomplished. The framework Stainless is used as a verification tool. The code of Bitcoin-S-Core, Scala implementation of Bitcoin protocol, is taken as an input for Stainless to be verified. In following the main aspects of formal verification, Stainless and Bitcoin-S are described.

1.1. Formal verification (formal verification vs. testing)

The longer and complexer a source code is, the more difficult it is to verify its correctness. There are different approaches for verification of program correctness.

The commonly used one is testing. A set of some practical scenarios and assertions is created by a developer to test a software design and implementation. While testing, it is checked whether the actual results match the expected results. Tests help to identify bugs and missing requirements. But it is not achievable to test all possible inputs. Thus, a long-term test coverage model should be created to test a software design enough. Furthermore, it is challenging to test a software keeping the ability to observe the side effects of a specific part of code. Practically, a developer runs tests, debugs appeared failures, adjusts a code of a software and extends a test bench verifying previously uncovered aspects. [7]

Another powerful method to check whether a program works correct is formal verification. This is a systematic process based on mathematical modeling. It is unnecessary to create a simulation tests with some possible inputs. With formal verification all possible input values are explored algorithmically and exhaustively. Correctness of a program is analyzed relative to its formal specification. Formal specification is a mathematical description of a software behavior that can be provided to formal verification tools for proving it. [7]

One of such tools is Stainless. This framework is used in the work to verify a Scala code.

1.2. Stainless (allgemeine Beschreibung: Struktur with pre- and postconditions, Solvers, Terminierung und mögliche Outputs)

Stainless is a framework developed by "Lab for Automated Reasoning and Analysis" (LARA) at EPFL's School of Computer and Communication Sciences. The framework is used to verify Scala programs. It verifies statically that a program satisfies a specification given by a developer and that a program will not crash at runtime. Stainless explores all possible input values, reports inputs for which a program fails and demonstrates counterexamples which violate a given specification.[3]

The main functions used to write a specification are *require* and *ensuring*. A precondition should be written at the beginning of a function body with *require*. Its argument is a Boolean expression which corresponds to constraint for inputs of a function being verified. A postcondition should be written after a function body with *ensuring*. It is a verification condition on an output of a function. While compiling Stainless tries to prove that the postcondition always holds, assuming a given precondition does hold.[3]

3 outcomes of verification with Stainless are possible: valid, invalid and unknown. If the postcondition is valid, Stainless could prove that for any inputs constrained in a precondition, the postcondition always holds. With invalid postcondition the framework could find at least one counterexample which satisfies a precondition but violates a postcondition. Also, an output unknown is possible when Stainless is unable to prove a postcondition or find a counterexample. In this case a timeout or an internal error occurred. Furthermore, it will be verified by Stainless that a precondition cannot be violated.[3]

The following example demonstrates a simple structure of formal specification for calculating factorial.

```
def factorial(n: Int): Int = {
    require(n >= 0)
    if (n == 0) {
      1
    } else {
      n * factorial(n - 1)
    }
} ensuring(res => res >= 0)
```

Listing 1.1: Example of verifying the function calculating factorial of an integer number

The function recursively calculates factorial of an integer number. An input to the function is constrained in *require* with non-negative value. A result of calculation should also be non-negative, what will be verified by Stainless. While compiling Stainless disproves the postcondition and gives the number 17 as a counterexample. A number of type Int is a 32 bit value. That is why calculating factorial of 17 causes an overflow and results to a negative value. The program would work correct by changing a type of number to BigInt. The outcome of Stainless verification of the function factorial from Listing 1.1 is shown bellow.

```
[Warning ] The Z3 native interface is not available. Falling back onto smt-z3.
[  Info  ]  - Checking cache: 'postcondition' VC for factorial @10:3...
[  Info  ]  - Checking cache: 'precond. (call factorial(n - 1))' VC for factorial @15:11...
[  Info  ]  - Checking cache: 'postcondition' VC for factorial @10:3...
[  Info  ] Cache miss: 'postcondition' VC for factorial @10:3...
[  Info  ] Cache hit: 'precond. (call factorial(n - 1))' VC for factorial @15:11...
[  Info  ] Cache hit: 'postcondition' VC for factorial @10:3...
[  Info  ]  - Now solving 'postcondition' VC for factorial @10:3...
[  Info  ]  - Result for 'postcondition' VC for factorial @10:3:
[Warning ]  => INVALID
[Warning ] Found counter-example:
[Warning ]   n: Int -> 17
[  Info  ]
[  Info  ] ╔═╡ stainless summary ╞═══════════════════════════════════════════════════════════════════════════════════════════════╗
[  Info  ] ╟─────────────────────┘                                                                                                 ╢
[  Info  ] ╟ factorial  postcondition                   valid from cache           src/TestFactorial.scala:10:3   1.055 ╢
[  Info  ] ╟ factorial  postcondition                   invalid          U:smt-z3  src/TestFactorial.scala:10:3   7.861 ╢
[  Info  ] ╟ factorial  precond. (call factorial(n - 1)) valid from cache           src/TestFactorial.scala:15:11  1.054 ╢
[  Info  ] ╠═══════════════════════════════════════════════════════════════════════════════════════════════════════════════════╣
[  Info  ] ╟ total: 3    valid: 2    (2 from cache) invalid: 1    unknown: 0    time:   9.970                                 ╢
[  Info  ] ╚═══════════════════════════════════════════════════════════════════════════════════════════════════════════════════╝
[  Info  ] Shutting down executor service.
```

Figure 1.1.: Output of Stainless verification for calculating factorial of Int number

The code being verified with Stainless should be written in Pure Scala, that corresponds to the functional programming. But also some imperative features are supported by Stainless.

Stainless verifies Scala programs using SMT (satisfiability modulo theories) solvers. A program code will be transformed by Stainless to the Pure Scala fragment supported by Inox solver. This solver was also developed by LARA and allows usage of SMT solvers such as Z3, CVC4 and Princess, adding support of other higher-level features. Preconditions, postconditions and assertions will be translated into SMT formulas, so an SMT solver can determine if all properties hold.

1.3. Bitcoin-S (Projekten und Packages, bitcoin-s-core, Eigenschaften zu prüfen)

In the work some fragments of the code of Bitcoin-S has to be verified. Bitcoin-S is an open source Scala implementation of the Bitcoin protocol. There are different projects belonging to Bitcoin-S. Protocol data structures are defined in the package **bitcoin-s-core**. This package is the main target to be verified because it specifies the

base of Bitcoin protocol, and it is crucial for correct protocol functionality. There is also the package **bitcoind-rpc** which is an RPC client implementation to communicate with *bitcoind*, daemon of Bitcoin Core running on a machine. Developers of Bitcoin-S are working on the implementation of a lightweight bitcoin client in the package **bitcoin-s-spv-node**. The whole list of packages can be found on GitHub [1].

Bitcoin-S is a large project implementing many functionalities. In this work one of them is going to be examined and worked on, namely the property of the Bitcoin that a non-coinbase transaction can not generate new coins. Thus, the code of Bitcoin-S should be analyzed, and fragments implementing this feature should be identified. Afterwards, the fragments should be rewritten, so that Stainless can accept it. After that a formal specification for functions should be defined and annotated according Stainless libraries, so that Stainless can verify correctness of the code.

2. Using Stainless

This chapter describes the setup and integration process of Stainless in a new or already existing project. It also shows the compatibility of Stainless with Scala as Stainless supports only a purely functional subset of Scala which they call *Pure Scala*.

2.1. Configuration

There are two ways to integrate Stainless in a Scala project, the Scala build tool (sbt) plugin or a command line tool. Both, when run, analyses the code passed and report warnings to the console about the given code. Stainless requires and Scala recommends Java SE Development Kit 8. Newer Java versions won't compile.

2.1.1. sbt

sbt for Scala is like gradle or maven for Java. It can compile Scala code continuously or manual, manage dependencies with support for Maven-formatted repositories, mixing Scala and Java projects and much more.

A simple sbt project has the following structure:

```
.
├── project
│   ├── build.properties
│   └── plugins.sbt
├── src
│   └── main
│       └── scala
│           └── Main.scala
└── build.sbt
```

build.properties specifies the sbt version used for this project. If the version is not available locally, sbt will download it.

In *plugins.sbt* new sbt plugins can be added. A plugin extends the build definition. Mostly this means adding and overriding settings.

build.sbt defines the build definition. There can be several projects or subprojects as sbt doc calls it.

Here an example for a single project in *build.sbt*:

```
scalaVersion := "2.12.8"

lazy val root = (project in file("."))
```

The project is called root and its source files are located in the files' directory. Executing `sbt compile` should now compile the code.

The Stainless web page has a guide on how to integrate Stainless in an existing project. The simplifies steps are:

- Install an external solver.
- Add Stainless sbt plugin to *plugins.sbt*
- Enable the plugin in *build.sbt* for the project.

After this setup, Stainless will report errors to the console, when running `sbt compile`.

2.1.2. Command Line Tool

There are two ways to use the command line tool.

Either download a prebuilt JAR file from efpl-lara/stainless GitHub repository or built a binary from source. Prebuild versions are released by Stainless. The latest was released on January 14, when there was no support for Scala 2.12. The 'Bump Scala to 2.12.8' branch was merged on March 4.

If latest features are needed, like support for Scala 2.12, the build from source is required. Here a short installation description. Full description can be found on the Stainless documentation pages.

- Install sbt.
- Check out GitHub repository.
- Run `sbt universal:stage` inside the project.

This generates *frontends/scalac/target/universal/stage/bin/stainless-scalac*.

To check the source code with one of those either `java -jar downloaded.jar source.scala` or `stainless-scalac source.scala` must be invoked. The file *source.scala* is the file to be checked.

As compiling without a build tool, this command will become really complex for bigger projects. All dependencies must be on the classpath and all source files appended. Those are added with `-classpath Dep1.jar:Dep2.jar:...:DepN.jar src1.scala src2.scala ... srcM.scala`.

2.2. Scala compatibility (Pure Scala, imperative features, dedicated BigInt, Generics...)

To be verified the Scala code extended with Stainless annotations is passed to the framework as input. The source code should be written in so called Pure Scala, a functional subset of Scala, so that Stainless can perform the verification. Additionally, the framework supports some imperative features. We will see some of these restrictions we have detected during this work in the next chapter.

2.2.1. Functional Pure Scala

The following functional elements are a part of Pure Scala:

- **Algebraic Data Types:** abstract class, case class, case object. The root of ADT is an abstract class, unless only one case class or case object is defined. An abstract class cannot define fields and constructor arguments. Case class and case object can extend an abstract class. An example of the ADT definition:

```
abstract class MyList
case object MyEmpty extends MyList
case class MyCons(elem: BigInt, rest: MyList) extends MyList
```

- **Objects** for grouping classes and functions. An example of object definition:

```
object Specs {
  def increment(a: BigInt): BigInt = {
    a + 1
  }
case class Identifier(id: BigInt)
}
```

- **Booleans:** logical operators *&&*, *||*, *!*, relational operator *==*, Stainless syntax for boolean implication *==>*. The short-circuit interpretation is used, when the second argument of a Boolean expression is evaluated only when needed.
- **Generics.** Only invariant type parameters are supported, what means if S is subtype of T then List[S] and List[T] do not have sub-typing relationship. Covariant and contravariant type parameters, such as List[+T] and List[-T], are not supported. Furthermore, only simple hierarchies are allowed, when subclass must define the same type parameters in the same order as a parent class. An example using Generics:

```
object Test {
  abstract class List[T]
  case class Cons[T](hd: T, tl: List[T]) extends List[T]
  case class Nil[T]() extends List[T]
  def contains[T](l: List[T], el: T) = { ... }
}
```

- **Methods** can be defined in an abstract class and implemented or overridden in case classes. The call of superclass methods with the keyword *super* is allowed. The method parameters can have default values. An example of method definition in case of inheritance:

```
object Test {
  sealed abstract class Base {
    def double(x: BigInt): BigInt = x * 2
  }

  case class Override() extends Base {
    override def double(x: BigInt): BigInt = {
      super.double(x + 1) + 42
    }
  }
}
```

- **Pattern matching.** Simple nested pattern matching, matching on type only, pattern guards and custom pattern matching with *unapply* method are supported.
- **Values**. Only immutable variables are allowed. Thus, they must be defined with the keyword *val* (and not with *var*).
- **Inner functions.**
- **Anonymous functions.**
- **Anonymous classes** with an explicit parent. An anonymous class should have the same number of public members as its parent. An example of anonymous class implementation:

```
abstract class Foo {
  def bar: Int
}

def makeFoo(x: Int): Foo = new Foo {
  def bar: Int = x
}
```

- **Local classes**. The creation of local classes within a function are allowed. [4]
- **Functional arrays** which use the operation that do not modify an array.

2.2.2. Imperative features

Stainless supports extensions to Pure Scala implemeting some imperative features. They will be translated by Stainless into Pure Scala during a preprocessing phase.

The following imperative components are supported:

- **Mutable variables** which can be declared with the keyword *var*. Declaration of mutable variables is allowed only as a local variable within a function, or in a case classe constuctor, or in trait if a default value is not assigned.
- **While loops**. Because besides main calculations in a loop the state of a loop counter changes at every iteration as a side effect, this kind of programming component does not belong to the functional paradigm and cannot be handled by solvers. Thus, Stainless generates automatically a postcondition with the negation of the loop condition as an argument. Moreover, to make it stronger, the automatically generated postcondition can be customised with the keyword *invariant*. An example of while loop implementation:

```
var res = 0
var i = 0
```

```
(while(i < 10) {
  res = res + i
  i = i + 1
}) invariant(i >= 0 && res >= i)
```

- **Function old** can be used in postconditions. It allows to get value of a variable before the execution of the block. This feature is useful dealing with mutable objects. An example of using the function *old*:

```
case class A(var x: Int)
def inc(a: A): Unit = {
  a.x = a.x + 1
} ensuring(_ => a.x == old(a).x + 1)
```

- **Arrays.** Stainless extends functional arrays allowing the reassignment of arrays elements. [2]

2.2.3. Stainless library

Besides annotations and functions helping for verification some core data types and operation on them is defined in the Stainless library. The own implementation and dedicated mapping of data types allows Stainless to transform it correctly into mathematical expressions of SMT solvers. Thus, the functionality of some data types can be restricted.

For example, BigInt implemented in Stainless has constructors only with a parameter of the type Int or String, while other types of a parameter is allowed in Scala. Furthermore, Stainless defines its own collection List[T] with the standard and some additional functions because SMT solvers do not support it. Moreover, the own Sets and Maps are used in Stainless. [5]

3. Using Bitcoin-S-Core

This chapter describes the relevant parts needed to verify that a non-coinbase transaction cannot generate new coins in Bitcoin-S-Core. We will see, how to create valid and invalid transactions with Bitcoin-S-Core and how these transactions are validated against the described property.

3.1. Creation of a Transaction

Some parts of the code in this section are from Bitcoin-S-Core transaction builder example.[6] Bitcoin-S-Core has a bitcoin transaction builder class with the following signature:

```
BitcoinTxBuilder(
  destinations: Seq[TransactionOutput], // where we send money
  utxos: BitcoinTxBuilder.UTXOMap,      // unspent transaction outputs
  feeRate: FeeUnit,                     // fee rate per byte
  changeSPK: ScriptPubKey,              // public key
  network: BitcoinNetwork               // bitcoin network information
): Future[BitcoinTxBuilder]             // sign TxBuilder to get tx
```

Here is how those parameters are generated.

First, a previous transaction with outputs is needed to spend some money from. This is created here, to show the process. It could also be parsed from a transaction in the bitcoin network. A single output is sufficient for this example. So, first create a new key pair to sign the next transaction and have a scriptPubKey where the money is. Then define the amount to spend (here 10000 Satoshis) collect this information in a transaction output and add it to the previous transaction.

```
val privKey = ECPrivateKey.freshPrivateKey
val creditingSPK = P2PKHScriptPubKey(pubKey = privKey.publicKey)

val amount = Satoshis(Int64(10000))

val utxo = TransactionOutput(currencyUnit = amount, scriptPubKey = creditingSPK)

val prevTx = BaseTransaction(
  version = Int32.one,
  inputs = List.empty,
  outputs = List(utxo),
  lockTime = UInt32.zero
)
```

Next, the new transaction should point to an output of the previous transaction. Thus, an outpoint is created with the id of the previous transaction and the index pointing to the specific output of it. This is collected in an utxo spending info which is then put in the list of all utxos (only one here).

```
val outPoint = TransactionOutPoint(prevTx.txId, UInt32.zero)

val utxoSpendingInfo = BitcoinUTXOSpendingInfo(
  outPoint = outPoint,
  output = utxo,
  signers = List(privKey),
  redeemScriptOpt = None,
  scriptWitnessOpt = None,
  hashType = HashType.sigHashAll
)

val utxos = List(utxoSpendingInfo)
```

Then, the destination, where the money goes, is defined. This includes a destination script pub key, as well as the amount to spend to it.

```
val destinationAmount = Satoshis(Int64(5000))

val destinationSPK = P2PKHScriptPubKey(pubKey = ECPrivateKey.freshPrivateKey.publicKey)

val destinations = List(
  TransactionOutput(currencyUnit = destinationAmount, scriptPubKey = destinationSPK)
)
```

Finally, define a fee rate, the network params and create a transaction builder.

```
val feeRate = SatoshisPerByte(Satoshis.one)

val networkParams = RegTest // soem static values for testing

val txBuilder: Future[BitcoinTxBuilder] = BitcoinTxBuilder(
  destinations = destinations,
  utxos = utxos,
  feeRate = feeRate,
  changeSPK = creditingSPK,
  network = networkParams
)
```

After calling sign on the transaction builder, a valid transaction is returned.

```
val signedTxF: Future[Transaction] = txBuilder
  .flatMap(_.sign)
  .map {
    (tx: Transaction) => println(tx.hex) // transaction in hex for the bitcoin network
  }
```

There is no need for a transaction input, since a transaction out point is kind of the pointer to a transaction input.

3.2. Validation of a Transaction

Bitcoin-S-Core offers a function called *checkTransaction*. This is its type signature.

```
checkTransaction(transaction: Transaction): Boolean
```

It takes a transaction as input and returns a Boolean whether the transaction is valid or not. So for example when passing the built tx from above to this function the returned value would be true. There are several checks in checkTransaction. For example, it checks if there is either no input or no output. In this case it returns false.

The relevant parts for the property to be verified are the following two lines.

```
val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
val noDuplicateInputs = prevOutputTxIds.distinct.size == prevOutputTxIds.size
```

It gathers all inputs output transaction ids. On this previous outputs ids it calls distinct and checks if the size stays the same. Distinct removes duplicates, so if there was two times the same input transaction id the size of those ids would be greater before calling distinct.

There is a bug in those two lines, as described in the next chapter.

4. Trying to Verify checkTransaction

This chapter describes the process we've made on trying to integrate Stainless in Bitcoin-S. It shows the differences between the error reporting by the sbt plugin and the JAR. Finally, during our work on Bitcoin-S we found a bug that we explain along its corresponding bug fix.

4.1. Integration

During this work, Stainless bumped the sbt plugin from version 0.13.17 to 1.2.8 and Scala from 2.11.12 to 2.12.8. So this section might be out of date now.

Bitcoin-S-Core uses sbt 1.2.8 and Scala 2.12.8, while Stainless sbt plugin is on sbt 0.13.17 and Scala 2.11.12.

Sbt introduced new features in the 1.x release used by Bitcoin-S.

For example, they introduced the new slash syntax.

```
lazy val root = (project in file("."))
  .settings(
    name := "hello",
    version in ThisBuild := "1.0.0"
  )
```

The version line written without slash becomes now one with slash. The right and left parameter of in-operator reverse and instead of the in-operator is now a slash.

```
ThisBuild / version := "1.0.0",
```

This is just one example of many others.

The bigger problem is, due to the different Scala and sbt versions, the error message, after trying to go in a sbt shell.

```
[warn] There may be incompatibilities among your library dependencies; run 'evicted'
       to see detailed eviction warnings.
[error] java.lang.NoClassDefFoundError: sbt/SourcePosition
...
Project loading failed: (r)etry, (q)uit, (l)ast, or (i)gnore?
```

Downgrading Bitcoin-S sbt version to 0.13.17 fixes the error but then it can not load some libraries only compiled for newer versions. So this would take too much time to fix and changes the Bitcoin-S code unwantingly.

The next approach is to use stainless cli instead of sbt. Running stainless on all source files does not work, because the dependencies are missing. The parameter *-classpath* can resolve it but the value of this parameter must be the paths to all the dependencies separated by a ':'. Finally, *core* depends on *secp256k1jni*, another package of Bitcoin-S written in Java. So this needs to be in the source files to.

The final command looks like this in *core* folder of Bitcoin-S:

```
$ stainless
  -classpath ".:$(find ~/.ivy2/ -type f -name *.jar | tr '\n' ':')"
  $(find . -type f -name *.scala | tr '\n' ' ')
  $(find ../secp256k1jni -type f -name *.java | tr '\n' ' ')
```

The output was some Stainless error.

```
[Internal] Error: object scala.reflect.macros.internal.macroImpl in compiler mirror
           not found.. Trace:
[Internal] - scala.reflect.internal.MissingRequirementError$.signal
```

```
            (MissingRequirementError.scala:17)
...
[Internal] object scala.reflect.macros.internal.macroImpl in compiler mirror not found.
[Internal] Please inform the authors of Inox about this message
```

After some discussion, we decided to go another way, because the errors may take too much time and it might lead to a next error. Let's extract the code needed to verify a transaction mainly the class Transaction and ScriptInterpreter with many other classes their depending on.

After this extraction Stainless was successfully integrated with both, sbt and JAR. This leads us to another finding as described in the next section.

4.2. Error Reporting with sbt and JAR

// TODO https://github.com/epfl-lara/stainless/issues/484?

4.3. Bugfix

After so much digging and trying to understand the checkTransaction code, we found a bug even without Stainless.

Here the relevant code of checkTransaction again:

```
val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
val noDuplicateInputs = prevOutputTxIds.distinct.size == prevOutputTxIds.size
```

What happens, if a TransactionOutPoint (previousOutput) references the same Transaction ID (txId) another does?

According to the Bitcoin protocol this is possible. A transaction can have multiple outputs that should be referenceable by the next transaction. What should not be possible is a transaction referencing the same output twice. This was a bug in Bitcoin Core known as CVE-201817144 which was patched on September 18, 2018.

Here, Bitcoin-S did a bit too much and marked all transaction as invalid, if they referenced a transaction twice in the next transaction. The fix is, to check on TransactionOutPoint instead of TransactionOutPoint.txId, because TransactionOutPoint contains the txId as well as the output index it references.

```
val prevOutputs = transaction.inputs.map(_.previousOutput)
val noDuplicateInputs = prevOutputs.distinct.size == prevOutputs.size
```

Since TransactionOutPoint is a case class and Scala has a built in == for case classes there is no need to implement TransactionOutPoint.==.

This was fixed in pull request #435 on GitHub at April 23, 2019 by us along a unit test to prevent this bug from appearing again in the future.

5. Towards Verifying Addition with Zero

After so many failures, we have decided to search for the smallest unit in Bitcoin-S-Core that is worthwhile to verify. We found the addition of two Satoshis would be a good candidate. To make it even easier, we decided to verify only the addition, where we add zero to an amount of Satoshis. The signature looks like the following.

```
+(c: CurrencyUnit): CurrencyUnit
```

Where the class Satoshis extends CurrencyUnit. The post condition would be, that the parameter c must be zero.

```
require(c.satoshis.underlying == Int64.zero)
```

c.satoshis is an abstract method of CurrencyUnit that must return an instance of the class Satoshis.

c.satoshis.underlying is also an abstract method of CurrencyUnit that must return an instance of the abstract type A.

Both are implemented in Satoshis where A is set to Int64. So the underlying number of the parameter must be zero.

To ensure that the result is the same value as *this*:

```
ensuring(res => res.satoshis == this.satoshis)
```

Here, equals (==) can be used directly on Satoshis, because it is a case class and Int64, the only parameter of Satoshis is a case class too.

After writing the verification, running Stainless on the source files was the next step. It prints a lot of errors because of code incompatibility. So let's describe the code rewriting in the following sections.

5.1. Rewriting Abstract Type Member

Stainless output:

```
[ Error  ] CurrencyUnits.scala:5:3: Stainless doesn't support abstract type members
           type A
```

This should be easy to rewrite by using generics instead of an abstract type, right? Unfortunately not. The problem is, CurrencyUnit uses its implementing class Satoshis.

Simplified code.

```
sealed abstract class CurrencyUnit {
  def +(c: CurrencyUnit): CurrencyUnit =
    Satoshis(satoshis.underlying + c.satoshis.underlying)
}

sealed abstract class Satoshis extends CurrencyUnit
```

Thus, changing it to generics, Satoshis is not of type CurrencyUnit anymore and neither of type CurrencyUnit[A], where A is the new generic type. Satoshis extends CurrencyUnit with type Int64, so it is of type CurrencyUnit[Int64]. That's too specific.

Since there is no easy way to fix it and the code should stay as much as possible the original, removing the abstract type might be the only choice. The verification is a bit more limited, to only Satoshis, but that's no problem, because the goal is to verify the addition of Satoshis.

So by removing the abstract type and fix it to Int64 as defined by Satoshis, there was one error less in Stainless.

5.2. Rewriting Generics

Stainless output:

```
[ Error  ] NumberType.scala:3:30: Unknown type parameter type T
           sealed abstract class Number[T <: Number[T]]
```

This is a missing feature in Stainless. To track this, there is the issue #519 on GitHub.

```
sealed abstract class Number[T <: Number[T]] extends BasicArithmetic[T]
```

For the sake of convenience this is also made static by parameterize T with Int64.

5.3. Rewriting Objects

Stainless output:

```
[ Error  ] CurrencyUnits.scala:54:1: Objects cannot extend classes or implement
           traits, use a case object instead
           object Satoshis extends BaseNumbers[Satoshis] {
```

Changing the objects that extends classes to case objects solves this problem. This is, due to the internal design of Scala and Java. It's possible to reason about a case object but not about an object. This needs a fundamental knowledge of Scala and some functional paradigms that should not be part of this thesis. The issue #520 on Stainless GitHub gives some thoughts about this.

5.4. Rewriting BigInt Constructor (only literal argument, no long argument, etc.)

Stainless output:

```
[ Error  ] CurrencyUnits.scala:47:33: Only literal arguments are allowed for BigInt.
           def toBigInt: BigInt = BigInt(toLong)
```

As described before, Stainless supports only a subset of Scala. They provide their own dedicated BigInt library. This library does not have support for dynamic BigInt construction only for string literals.

Again, a simplified version.

```
sealed abstract class Satoshis extends CurrencyUnit {
  def toBigInt: BigInt = BigInt(toLong)
  def toLong: Long = underlying.toLong
}

object Int64 extends BaseNumbers[Int64] {
  private case class Int64Impl(underlying: BigInt) extends Int64
}
```

This would be really hard to refactor, because Bitcoin-S-Core tries to be as much dynamic as possible so it can be used with other cryptocurrencies too. Maybe it would even be impossible, because they need to parse a lot from the bitcoin network.

Restricting the code one more time helps. *toBigInt* can directly return underlying, because in case of Int64, which is what Satoshis use, it is a BigInt. So it does not be converted from BigInt to Long and back to BigInt.

5.5. Rewriting Private Inner Classes

Stainless output (simplified):

```
[Warning ] CurrencyUnits.scala:36:3: Could not extract tree in class:
           case private class SatoshisImpl extends Satoshis
```

Stainless is not able to extract private classes inside other objects. Bitcoin-S-core uses it a lot, because they separate the class from its implementation.

```
object Int64 extends BaseNumbers[Int64] {
  private case class Int64Impl(underlying: BigInt) extends Int64
}
```

This is an easy one. Just extract the inner class out of the object. This is not exactly the same code but since the class is still private it can not be extended outside of the file.

5.6. Rewriting Type Member

Stainless output:

```
[Warning ] NumberType.scala:5:3: Could not extract tree in class: type A
           = BigInt (class scala.reflect.internal.Trees$TypeDef)
           type A = BigInt
```

Since the type A is not overridden in any of its subclasses, this can simply be hard coded.

```
sealed abstract class Number {
  type A = BigInt
  protected def underlying: A
  def apply: A => T
}
```

Replace every occurrence of A with BigInt. This might be a missing feature in Stainless that should not be too hard to fix. There is an open pull request #470 on GitHub for this issue.

5.7. Rewriting Usage of BigInt &-Function

Stainless output:

```
[ Error  ] NumberType.scala:19:14: Unknown call to & on result (BigInt) with
           arguments List(Number.this.andMask) of type List(BigInt)
           require((result & andMask) == result,
```

Due to the restrictions on BigInt, the & function on it is not supported too.

```
sealed abstract class Number extends BasicArithmetic[Int64] {
  def andMask: BigInt

  override def +(num: Int64): Int64 = apply(checkResult(underlying + num.underlying))

  private def checkResult(result: BigInt): BigInt = {
    require((result & andMask) == result, "Result was out of bounds, got: " + result)
    result
  }
}
```

This is a bound check. It checks, if the result of the addition is in range of the specified type (Int64 here). So here the & mask can be replaced with a bound check whether the result is in range of Long.MinValue and Long.MaxValue. Again the code gets a bit more static but for this use case it is OK.

5.8. Rewriting require

Stainless output:

```
[Warning ] NumberType.scala:54:3: Could not extract tree in class:
           scala.this.Predef.require(Int64Impl.this.underlying.<=(
           math.this.BigInt.long2bigInt(9223372036854775807L)),
           "Number was too big for a int64, got: ".+(Int64Impl.this.underlying))
           (class scala.reflect.internal.Trees$Apply)
           require(underlying <= 9223372036854775807L,
```

This is, because Stainless does not support require with a second String parameter. Removing this String fixes the error.

5.9. Propagate require

Finally, the code works with Stainless. But there is another problem. Bitcoin-S-Core uses require like a fail-fast method whereas Stainless needs it to verify the code.

Stainless output:

```
[Warning ] Found counter-example:
[Warning ]   thiss: { x: Object | @unchecked isNumber(x) } -> Int64Impl(0)
[Warning ]   num: { x: Object | @unchecked isInt64(x) }    ->
               Int64Impl(9223372036854775808)
```

Corresponding code:

```
sealed abstract class Number extends BasicArithmetic[Int64] {
  override def +(num: Int64): Int64 = apply(checkResult(underlying + num.underlying))

  private def checkResult(result: BigInt): BigInt = {
    require(
      result <= BigInt("9223372036854775807")
     && result >= BigInt("-9223372036854775808")
    )
    result
  }
}
```

But how can Stainless find a counter example ignoring the require in checkResult? Since Stainless is a static verification tool, it tests every possibility. So it can use a number bigger than the maximum Int64 and pass it to the addition. The require in checkResult will then fail. Thus, the addition need to have the restrictions of checkResult too.

```
override def +(num: Int64): Int64 = {
  require(
    num.underlying <= BigInt("9223372036854775807")
    && num.underlying >= BigInt("-9223372036854775808")
    && this.underlying <= BigInt("9223372036854775807")
    && this.underlying >= BigInt("-9223372036854775808")
  )
  apply(checkResult(underlying + num.underlying))
}
```

Stainless finds another counter example:

```
[Warning ] Found counter-example:
[Warning ]   num: { x: Object | @unchecked isInt64(x) }    -> Int64Impl(1)
[Warning ]   thiss: { x: Object | @unchecked isNumber(x) } ->
               Int64Impl(9223372036854775807)
```

Sure, it can just add one to the maximum Int64 and the require does not hold anymore. The goal is to check the addition of zero so let's add a restriction for num to be zero.

Finally, it works and verifies correctly.

5.10. Result

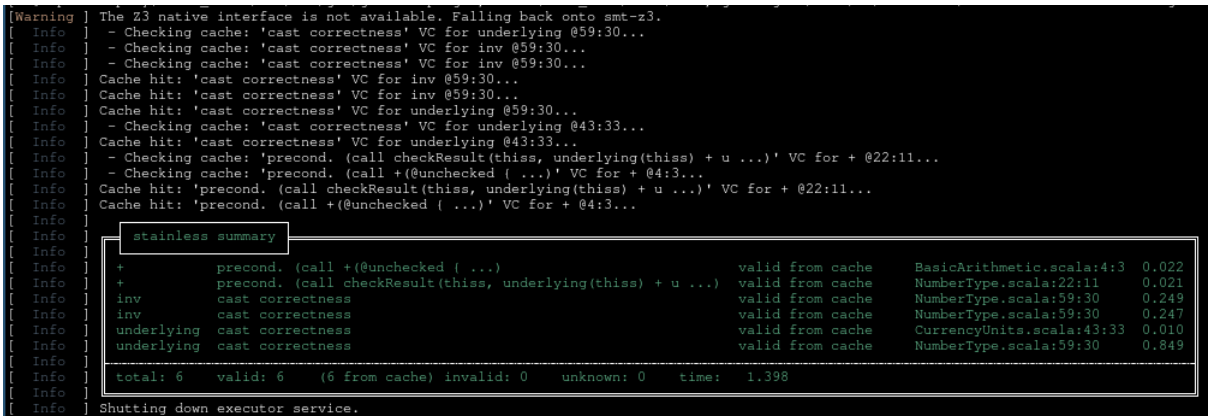

Figure 5.1.: Output of Stainless verification for addition with 0 of Bitcoin-S-Cores CurrencyUnit

6. Conclusion

6.1. Future Work

Declaration of primary authorship

We hereby confirm that we have written this thesis independently and without using other sources and resources than those specified in the bibliography. All text passages which were not written by me are marked as quotations and provided with the exact indication of its origin.

Place, Date:	Biel, TODO fix this	
Last Names, First Names:	Doukmak Anna	Boss Ramon
Signatures:	.....	.....

Bibliography

[1] "Bitcoin-S Project." [Online]. Available: https://github.com/bitcoin-s

[2] "Stainless Documentation - Imperative." [Online]. Available: https://epfl-lara.github.io/stainless/imperative.html

[3] "Stainless Documentation - Introduction." [Online]. Available: https://epfl-lara.github.io/stainless/intro.html

[4] "Stainless Documentation - Pure Scala." [Online]. Available: https://epfl-lara.github.io/stainless/purescala.html

[5] "Stainless library." [Online]. Available: https://epfl-lara.github.io/stainless/library.html

[6] "Transaction Builder Example." [Online]. Available: https://github.com/bitcoin-s/bitcoin-s/blob/master/docs/core/txbuilder.md

[7] A. Sanghave, "What is formal verification?"

APPENDICES

A. Arbitrary Appendix

The European languages are members of the same family. Their separate existence is a myth. For science, music, sport, etc, Europe uses the same vocabulary. The languages only differ in their grammar, their pronunciation and their most common words. Everyone realizes why a new common language would be desirable: one could refuse to pay expensive translators. To achieve this, it would be necessary to have uniform grammar, pronunciation and more common words. If several languages coalesce, the grammar of the resulting language is more simple and regular than that of the individual languages. The new common language will be more simple and regular than the existing European languages. It will be as simple as Occidental; in fact, it will be Occidental.