Experiments in Formal Verification of Scala Code

Bachelor Thesis

Degree course:	Computer Science
Authors:	Anna Doukmak, Ramon Boss
Tutor:	Prof. Dr. Kai Brünnler
Date:	2019-06-13

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Abstract

In this thesis we try to verify some properties of Bitcoin-S.

First, we look at some theory about formal verification, Stainless – a verification tool and Bitcoin-S – an open source implementation of the Bitcoin protocol.

Then, we try to verify that a non-coinbase transaction should not generate new coins. We create a transaction and check it in Bitcoin-S. After some time we realize that it is not possible with the current tools and the given time. But because we familiarized us with the Bitcoin-S code we find a bug and send a bug fix to the Bitcoin-S project.

So we verify a less interesting property of Bitcoin-S. There is a datatype Satoshis that represents bitcoins. We verify that the addition of Satoshis with zero results in the same amount of Satoshis. Even this needs many changes in the code that we go through step-by-step.

Finally, we see that software should be written with verification tools in mind, because otherwise the code possibly must be rewritten when it comes to verification.

Contents

1 Introduction

The longer and more complex a source code is, the more difficult it is to verify its correctness. There are different approaches to show the correctness of a program. One of them is formal verification. In contrast to testing, where only predefined inputs are tested, with formal verification all possible inputs are covered. The correctness of a program is analyzed relative to its formal specification. Formal specification is a mathematical description of a software behavior that can be used by the verification tool to verify the code. We will see an example of a formal specification shortly.

In this work we use the verification framework Stainless to verify parts of the code of Bitcoin-S, a Scala implementation of the Bitcoin protocol. In the following we look at the main aspects of formal verification with Stainless and properties of Bitcoin-S we want to verify.

1.1 Formal Verification with Stainless

Stainless is developed by "Lab for Automated Reasoning and Analysis" (LARA) at EPFL's School of Computer and Communication Sciences. Using this framework we can verify the correctness of Scala programs. The following overview of the framework is taken from the section Introduction of the Stainless documentation [4].

Stainless statically verifies that a program satisfies a given specification and that a program will not crash at runtime. The framework covers all possible inputs and finds a counter examples for possible failures in a program which violate the given specification.

The main functions used to write a specification are *require* and *ensuring*.

With *require* we define a precondition of a function, which we want to verify. *require* is placed at the beginning of the function body. *require* specifies, if the parameter of the function holds against a certain condition.

With *ensuring*, we define the postcondition of a function. *ensuring* is placed at the end of the function after the body. *ensuring* specifies, if the return value or the result of the function satisfies a certain condition.

On invoking Stainless, it tries to prove that the postcondition always holds, assuming the given precondition does hold.

The following example demonstrates a simple formal specification for the function calculating a factorial. This is a modified example from the section Introduction of the Stainless documentation [4], so Stainless reports an error verifying it.

```
1 def factorial(n: Int): Int = {
2     require(n >= 0)
3     if (n == 0) {
4       1
5     } else {
6       n * factorial(n - 1)
7     }
8 } ensuring(res => res >= 0)
```

It recursively calculates the factorial of an integer number. The input of the function is constrained with *require* to a non-negative value. The result should also be non-negative. Thus, Stainless will verify whether the result of the factorial calculation is non-negative for all non-negative inputs.

Stainless can produce 3 outcomes of postcondition verification: *valid*, *invalid* and *unknown*. If the postcondition is *valid* Stainless could prove that for any inputs constrained in the precondition, the postcondition always holds. Reporting the postcondition as *invalid* the framework could find at least one counterexample which satisfies the precondition but violates the postcondition. If Stainless is unable to prove the postcondition or find a counterexample it reports the outcome *unknown*. In this case a timeout or an internal error occurred. Furthermore, Stainless checks

for calls in the code to the function with invalid arguments violating the precondition. For example if we invoke factorial with a negative number it reports *invalid*.

Let's return to our incorrect example with factorial. Stainless reports the following result for the factorial verification from our example:

```
[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

Stainless disproves the postcondition and gives the number 17 as a counterexample. Due to an overflow in the 32 bit Int the result becomes a negative value. The program would work correct by changing the type from Int to BigInt.

Stainless supports only a subset of Scala.

Anna It supports blabla, in particular it supports not blablabla. (check class with var)

why does stainless use its own BigInt etc?

They call this Pure Scala. You can find the specification of Pure Scala on the Stainless documentation website [4] in the section Pure Scala.

In addition, Stainless has its own library with annotations, reimplementation of some core data types, collections and input-output functions and more. Some of them are described on the website in the section Stainless Library. There are more details about the library in the source code of Stainless on GitHub [3] in the folder *frontends/library/stainless*.

Stainless verifies Scala programs using SMT (satisfiability modulo theories) solvers.

1.2 The Properties to Verify

Bitcoin-S is a large project found on GitHub implementing many functionalities. In this work we look at two of them, namely the No-Inflation Property and the Addition-with-Zero Property described shortly. Thus, we go through the relevant code parts to verify those properties, look at some challenges with Stainless and see why we should write software with verification in mind. In order to be able to verify Bitcoin-S, we have to rewrite parts of it in the subset of Scala supported by Stainless.

1.2.1 No-Inflation Property

The decision to verify this property was inspired by the bug found in Bitcoin Core in September 2018 [1] which allowed to spend the same unspent transaction output (UTXO) in a transaction multiple times. Hence, new coins could be created out of the air. Thus, we try to verify the property, that a non-coinbase transaction cannot generate new coins. Let's name it the No-Inflation Property.

As we can see later in chapter 2, we must rewrite a huge part of the code implementing this property. This reimplementation into Pure Scala needs a lot of time. So, we adjust the plan and verify another functionality of Bitcoin-S. Nevertheless, during the analysis of the No-Inflation Property we look at a bug in Bitcoin-S found during this work and see the code changes for the bugfix in section 2.3.

1.2.2 Addition-with-Zero Property

In Bitcoin-S there is a class `Satoshis` representing an amount of bitcoins. We look at the verification of the addition of Satoshis with zero Satoshis. This operation should result in the same amount of Satoshis. Let's call it the Addition-with-Zero Property.

Using Stainless, we see the successful verification of this property. But the process of the verification with the tool requires many changes in the code, so that Stainless can accept it. We look at all needed modifications in chapter 3.

2 Trying to Verify the No-Inflation Property

This chapter describes the part of Bitcoin-S needed to verify the No-Inflation Property described before. We are going to create a transaction and show the relevant parts of the method `checkTransaction`, where transactions are checked against some properties. Then, we will see the bug in Bitcoin-S found during this work and its fix. In the end we see why the No-Inflation Property needed to be changed to the Addition-with-Zero Property.

2.1 Creation of a Transaction

Some code in this section is copied or adapted from the Bitcoin-S-Core transaction builder example [2]. Bitcoin-S-Core has a bitcoin transaction builder class with the following constructor:

```
1 BitcoinTxBuilder(
2   destinations: Seq[TransactionOutput], // where to send the money
3   utxos: BitcoinTxBuilder.UTXOMap,      // unspent transaction outputs
4   feeRate: FeeUnit,                     // fee rate per byte
5   changeSPK: ScriptPubKey,              // where to send the change
6   network: BitcoinNetwork               // bitcoin network information
7 ): Future[BitcoinTxBuilder]
```

The return type Future does not make sense here, since the implementation calls either Future.successful or Future.fromTry which returns an already resolved Future. This might be for future purposes.

Now we create a transaction.

First, we need some money. Thus, we create a fake transaction with one single output. This transaction can be parsed from the bitcoin network, but we create one manually in order to see this process.

```
1 val privKey = ECPrivateKey.freshPrivateKey
2 val creditingSPK = P2PKHScriptPubKey(pubKey = privKey.publicKey)
3
4 val amount = Satoshis(Int64(10000))
5
6 val utxo = TransactionOutput(currencyUnit = amount, scriptPubKey = creditingSPK)
7
8 val prevTx = BaseTransaction(
9   version = Int32.one,
10  inputs = List.empty,
11  outputs = List(utxo),
12  lockTime = UInt32.zero
13 )
```

On line one and two we create a new keypair to sign the next transaction and have a scriptPubKey where the bitcoins are. This is our keypair. So the money is transferred to our public key. Line four specifies the amount of satoshis we have in the transaction. Then we create the actual transaction from line 6 to 13.

Now that we have some bitcoins, we create the new transaction where we want to spend them.

First, we need some out points. They point to outputs of previous transactions. We use the index zero, because the previous transaction has only one output that becomes the first index zero. If there were two previous outputs, the second output would become the index 1 and so on.

```
1 val outPoint = TransactionOutPoint(prevTx.txId, UInt32.zero)
2
3 val utxoSpendingInfo = BitcoinUTXOSpendingInfo(
4   outPoint = outPoint,
5   output = utxo,
6   signers = List(privKey),
7   redeemScriptOpt = None,
```

```
  scriptWitnessOpt = None,
  hashType = HashType.sigHashAll
)

val utxos = List(utxoSpendingInfo)
```

This utxos are the inputs of our transaction.

Second, we need destinations to spend the bitcoins to. For the sake of convenience we create only one.

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

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

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

We spend 5000 satoshis to the newly created random public key.

Finally, we define the fee rate in satoshis per one byte transaction size as well as some bitcoin network parameters. The bitcoin network parameters are not important, so we use some static values normally used when testing.

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

val networkParams = RegTest // some static values for testing
```

Now lets build the transaction with those data.

```
val txBuilderF: Future[BitcoinTxBuilder] = BitcoinTxBuilder(
  destinations = destinations, // where to send the money
  utxos = utxos,               // unspent transaction outputs
  feeRate = feeRate,           // fee rate per byte
  changeSPK = creditingSPK,    // where to send the change
  network = networkParams      // bitcoin network information
)

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

Line one to seven creates a transaction builder which is then signed on line ten. We can now use our transaction object on line twelve. For example, after calling *hex* on it, we can send the returned string to the bitcoin network.

2.2 Validation of a Transaction

Bitcoin-S offers a function called *checkTransaction* located in the ScriptInterpreter object. This is its type signature:

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

We can pass a transaction and it returns a Boolean indicating whether the transaction is valid or not. So for example when we pass the transaction we built before the returned value would be true, because it's a valid transaction. It might not be accepted by the bitcoin network but for a transaction on its own it's valid. We can not check context with it, because we can only pass one transaction.

There are several checks in checkTransaction. For example, it checks if there is either no input or no output. In this case we get false.

The relevant part for the bug we found:

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

It gathers all transaction ids referenced by the out points. When we call *distinct* on the returned list, we get a list with duplicate removed. If the size of the new list is the same as the size of the old, we know that there was no duplicate transaction id, because, as said, distinct removes the duplicates.

2.3 Fixing a Bug in Bitcoin-S

We can see that there is a bug in the checkTransaction function from before, recognized and fixed through this work.

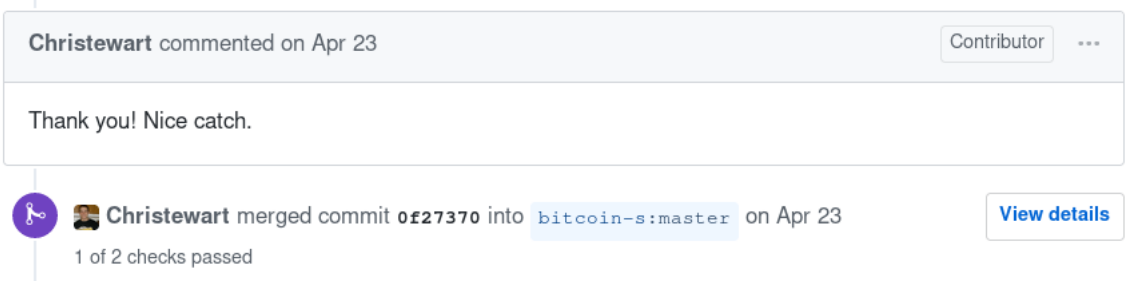


Figure 2.1: Nice catch comment on our PR #435 on GitHub

Here is the relevant code of checkTransaction again:

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

What happens if we have two TransactionOutPoints (previousOutputs) with a different index but referencing the same Transaction ID (txId)?

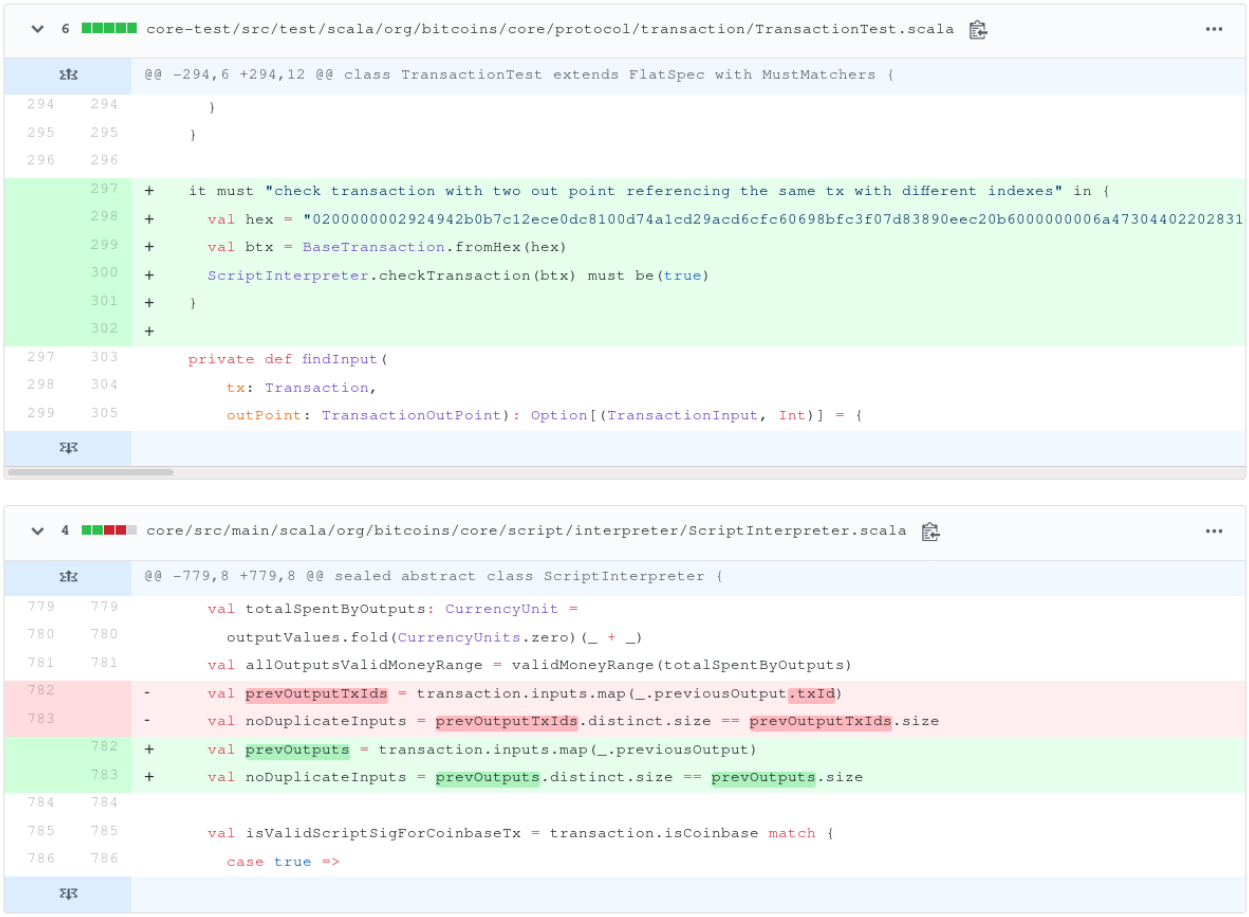
According to the Bitcoin protocol this is possible. A transaction can have multiple outputs that should be referenceable by the next transaction. So this is clearly a bug.

What should not be possible is a transaction referencing the same output twice. This bug occurred in Bitcoin Core known as CVE-2018–17144 which was patched on September 18, 2018. [1]

Here, Bitcoin-S did a bit too much and marked all transaction as invalid, if they referenced the same transaction twice. The fix is, to check on TransactionOutPoint instead of TransactionOutPoint.txId, because TransactionOutPoint contains the txId as well as the output index it references. So in pseudo code, we check on the tuple (tx, index) instead of (tx). The fixed code:

```
1 val prevOutputs = transaction.inputs.map(_.previousOutput)
2 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.==.



```
6 core-test/src/test/scala/org/bitcoins/core/protocol/transaction/TransactionTest.scala
@@ -294,6 +294,12 @@ class TransactionTest extends FlatSpec with MustMatchers {
294 294         }
295 295       }
296 296
297     +   it must "check transaction with two out point referencing the same tx with different indexes" in {
298     +     val hex = "0200000002924942b0b7c12ece0dc8100d74a1cd29acd6cfc60698bfc3f07d83890eec20b6000000006a47304402202831
299     +     val btx = BaseTransaction.fromHex(hex)
300     +     ScriptInterpreter.checkTransaction(btx) must be(true)
301     +   }
302     +
297 303     private def findInput(
298 304         tx: Transaction,
299 305         outPoint: TransactionOutPoint): Option[(TransactionInput, Int)] = {

4 core/src/main/scala/org/bitcoins/core/script/interpreter/ScriptInterpreter.scala
@@ -779,8 +779,8 @@ sealed abstract class ScriptInterpreter {
779 779       val totalSpentByOutputs: CurrencyUnit =
780 780         outputValues.fold(CurrencyUnits.zero)(_ + _)
781 781       val allOutputsValidMoneyRange = validMoneyRange(totalSpentByOutputs)
782     -     val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
783     -     val noDuplicateInputs = prevOutputTxIds.distinct.size == prevOutputTxIds.size
    782 +     val prevOutputs = transaction.inputs.map(_.previousOutput)
    783 +     val noDuplicateInputs = prevOutputs.distinct.size == prevOutputs.size
784 784
785 785       val isValidScriptSigForCoinbaseTx = transaction.isCoinbase match {
786 786         case true =>
```

Figure 2.2: Line changes for PR #435 from GitHub

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

2.4 Adjusting No-Inflation Property

Trying to integrate Stainless in Bitcoin-S caused a lot of troubles, mainly because of version conflicts. For more details see chapter 4.

After integrating Stainless in Bitcoin-S, there were many errors. It takes too much time to fix them all so it should be easier to extract the classes needed for the `checkTransaction` function.

The extracted code has more than 1500 lines. After running Stainless on it, it still throws a huge bunch of errors about what Stainless can not reason about. After fixing some of those errors, there appear new ones. So this would require changing nearly everything of the extracted code.

Let's adjust the property from the No-Inflation Property to the Addition-with-Zero Property, because there is not enough time to fix all these errors and write the verification. This is way less interesting to verify but gives still many insights in what needs to be done to verify some parts of Bitcoin-S and other source code.

So let's look at the Addition-with-Zero Property.

3 Towards Verifying the Addition-with-Zero Property

After realizing that it would consume too much time to rewrite the Bitcoin-S code and even the extracted part with checkTransaction, the smallest unit in Bitcoin-S-Core that is worthwhile to verify was extracted. This could be the addition of two CurrencyUnits. To make it even easier, the addition of CurrencyUnits with zero. CurrencyUnits is an abstract class in Bitcoin-S, representing currencies like Satoshis.

Here we can see the extracted code needed for the addition of CurrencyUnits:

```
1 package code.initial
2
3 trait BasicArithmetic[N] {
4   def +(n: N): N
5 }
6
7 sealed abstract class Number[T <: Number[T]]
8     extends BasicArithmetic[T] {
9   type A = BigInt
10
11   protected def underlying: A
12
13   def toLong: Long = toBigInt.longValue()
14   def toBigInt: BigInt = underlying
15
16   def andMask: BigInt
17
18   def apply: A => T
19
20   override def +(num: T): T = apply(checkResult(underlying + num.underlying))
21
22   private def checkResult(result: BigInt): A = {
23     require((result & andMask) == result,
24       "Result was out of bounds, got: " + result)
25     result
26   }
27 }
28
29 sealed abstract class SignedNumber[T <: Number[T]] extends Number[T]
30
31 sealed abstract class Int64 extends SignedNumber[Int64] {
32   override def apply: A => Int64 = Int64(_)
33   override def andMask = 0xffffffffffffffffL
34 }
35
36 trait BaseNumbers[T] {
37   def zero: T
38   def one: T
39 }
40
41 object Int64 extends BaseNumbers[Int64] {
42   private case class Int64Impl(underlying: BigInt) extends Int64 {
43     require(underlying >= -9223372036854775808L,
44       "Number was too small for a int64, got: " + underlying)
45     require(underlying <= 9223372036854775807L,
46       "Number was too big for a int64, got: " + underlying)
47   }
48
49   lazy val zero = Int64(0)
50   lazy val one = Int64(1)
51
52   def apply(long: Long): Int64 = Int64(BigInt(long))
```

```
53
54   def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
55 }
56
57
58 sealed abstract class CurrencyUnit
59   extends BasicArithmetic[CurrencyUnit] {
60   type A
61
62   def satoshis: Satoshis
63
64   def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis
65
66   override def +(c: CurrencyUnit): CurrencyUnit =
67     Satoshis(satoshis.underlying + c.satoshis.underlying)
68
69   protected def underlying: A
70 }
71
72 sealed abstract class Satoshis extends CurrencyUnit {
73   override type A = Int64
74
75   override def satoshis: Satoshis = this
76
77   def toBigInt: BigInt = BigInt(toLong)
78
79   def toLong: Long = underlying.toLong
80
81   def ==(satoshis: Satoshis): Boolean = underlying == satoshis.underlying
82 }
83
84 object Satoshis extends BaseNumbers[Satoshis] {
85   val zero = Satoshis(Int64.zero)
86   val one = Satoshis(Int64.one)
87
88   def apply(int64: Int64): Satoshis = SatoshisImpl(int64)
89
90   private case class SatoshisImpl(underlying: Int64) extends Satoshis
91 }
```

The additions' signature looks like this:

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

When we run Stainless on it, it throws the following errors:

```
[ Error  ] Code.scala:41:1: Objects cannot extend classes or implement traits,
           use a case object instead
         object Int64 extends BaseNumbers[Int64] {
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[ Error  ] Code.scala:60:3: Stainless doesn't support abstract type members
            type A
            ^^^^^^
[ Error  ] Code.scala:77:33: Only literal arguments are allowed for BigInt.
            def toBigInt: BigInt = BigInt(toLong)
                                          ^^^^^^
[ Error  ] Code.scala:84:1: Objects cannot extend classes or implement traits,
           use a case object instead
          object Satoshis extends BaseNumbers[Satoshis] {
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[  Info  ] Shutting down executor service.
```

So let's see how we can fix those errors.

3.1 Turning Object into Case Object

Stainless output:

```
[ Error  ] Code.scala:41:1: Objects cannot extend classes or implement traits,
           use a case object instead
         object Int64 extends BaseNumbers[Int64] {
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[ Error  ] Code.scala:84:1: Objects cannot extend classes or implement traits,
          use a case object instead
          object Satoshis extends BaseNumbers[Satoshis] {
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

Here, we can just change the objects from object to case object. Stainless recommendation is to use objects for modules and case objects as algebraic data types.

This is due to the internal design of Scala. It's possible to reason about case object but not about 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, if you want to know more.

3.2 Getting Rid of Abstract Type Member

Stainless output:

```
[ Error  ] Code.scala:60:3: 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 one of its implementing classes: Satoshis.

Simplified code.

```
1 sealed abstract class CurrencyUnit {
2   type A
3
4   def +(c: CurrencyUnit): CurrencyUnit =
5     Satoshis(satoshis.underlying + c.satoshis.underlying)
6
7   protected def underlying: A
8 }
9
10 sealed abstract class Satoshis extends CurrencyUnit {
11   override type A = Int64
12 }
```

What happens, if we typify CurrencyUnit with A, meaning to make it generic with type A?

Satoshis extends CurrencyUnit with type Int64, so it would be of type CurrencyUnit[Int64]. That's too specific, because the return type of the addition is then CurrencyUnit[Int64] not CurrencyUnit[A]. Maybe the Bitcoin-S developers should reimplement this part and not use Satoshis directly.

Since there is no easy way to fix it and the code should stay as much as possible the original, we just remove the abstract type and set it to Int64. This limits the verification a bit, but as we only want to verify the addition in satoshis, that's OK.

3.3 Replacing the BigInt Constructor Argument With A String Literal

Stainless output:

```
[ Error  ] Code.scala:77:33: Only literal arguments are allowed for BigInt.
             def toBigInt: BigInt = BigInt(toLong)
                                    ^^^^^^
```

As described before, Stainless supports only a subset of Scala. The BigInt from the Stainless library is a bit restricted. One such restriction is, that BigInt does not support dynamic BigInt construction. Thus, the constructor parameter of BitInt must be a literal argument.

Again, a simplified code version.

```
1 sealed abstract class Satoshis extends CurrencyUnit {
2   def toBigInt: BigInt = BigInt(toLong)
3   def toLong: Long = underlying.toLong
4 }
```

This would be really hard to refactor, because Bitcoin-S tries to be as dynamic as possible so it can be used with cryptocurrencies other than bitcoins. Maybe it could be impossible, because they need to parse dynamic values from the bitcoin network.

Luckily, we can use toBigInt on the field `underlying` directly instead of toLong. So, instead of converting the underlying to Long and back to BigInt we convert underlying directly to BigInt.

After fixing all Stainless errors, a new error appears.

3.4 Getting Rid of Special Generics

Stainless output:

```
[ Error  ] Code.scala:7:30: Unknown type parameter type T
           sealed abstract class Number[T <: Number[T]]
                                 ^^^^^^^^^^^^^^
```

This is a missing feature in Stainless. It does not support upper type boundaries on the class itself. To track this, issue #519 was created on GitHub during this work.

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

Despite this, in order to be able to continue, we make this a concrete type by replacing T with Int64. Int64, because Satoshis uses only Int64. There are other number types like UInt16 but for our property we don't need them.

Now, there are two new errors.

3.5 Getting Rid of Concrete Type

Stainless output:

```
[Warning ] Code.scala:9:3: Could not extract tree in class: type A =
           BigInt (class scala.reflect.internal.Trees$TypeDef)
           type A = BigInt
           ^^^^^^^^^^^^^^^
```

This is easy. We just replace all occurrence of A with BigInt, since A is not overwritten in an implementing class. This is not the exact same code, because an implementing class can not override A anymore but that's fine for our verification.

This was a missing feature in Stainless that was fixed on Mai 28, 2019 with pull request #470 on GitHub. Now it should work without this change.

3.6 Replacing the BigInt &-Function With Bound Check

Stainless output:

```
[ Error  ] Code.scala:22:14: 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, we can not use the & function either. Simplified code:

```
1 sealed abstract class Number extends BasicArithmetic[Int64] {
2   def andMask: BigInt
3
4   override def +(num: Int64): Int64 = apply(checkResult(underlying + num.underlying))
5
6   private def checkResult(result: BigInt): BigInt = {
7     require((result & andMask) == result, "Result was out of bounds, got: " + result)
8     result
9   }
10 }
```

This is a bounds check. It checks if the result of the addition is in range of the specified type, which is now the hard coded Int64.

So, we can replace the & mask with a bound check whether the result is in range of Long.MinValue and Long.MaxValue, because Int64 has the same 64-bit range as Long. Again the code gets a bit more static and it's not the exact same code anymore.

Running Stainless produces again new errors.

3.7 Extracting Private Inner Classes

Stainless output:

```
[Warning ] Code.scala:90:3: Could not extract tree in class: case private
           class SatoshisImpl extends Satoshis with Product with Serializable {
```

Stainless can not extract the private class inside the object. Bitcoin-S uses this a lot, because they separate the class from its implementation. Simplified:

```
1 object Int64 extends BaseNumbers[Int64] {
2   private case class Int64Impl(underlying: BigInt) extends Int64
3 }
```

This is easy to fix. We just extract the private class out of the object. This is not exactly the same code, because other classes in the same file could now access the private class. But for our property it does not change anything.

Now we get some weird warnings about require.

3.8 Removing Second Parameter of Require

Stainless output:

```
[Warning ] Code.scala:51:3: Could not extract tree in class: scala.this.Predef
           .require(Int64Impl.this.underlying.>=(math.this.BigInt.long2bigInt(
           -9223372036854775808L)), "Number was too small for a int64, got: "
           .+(Int64Impl.this.underlying)) (class scala.reflect.internal.Trees$Apply)
          require(underlying >= -9223372036854775808L,
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[Warning ] Code.scala:53:3: Could not extract tree in class: scala.this.Predef
           .require(Int64Impl.this.underlying.<=(math.this.BigInt.long2bigInt(
           9223372036854775807L)), "Number was too big for a int64, got: "
           .+(Int64Impl.this.underlying)) (class scala.reflect.internal.Trees$Apply)
          require(underlying <= 9223372036854775807L,
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[ Error  ] checkResult$0 depends on missing dependencies: require$1.
```

Seems like Stainless does not support the second string parameter of require or at least it throws a warning about it. We can safely remove the string parameters from the requires, since they only serve as error messages.

A new error appears.

3.9 Rewriting BigInt Comparison with Long

Stainless output:

```
[ Error  ] inv$4 depends on missing dependencies: long2bigInt$0.
```

This error is hard to understand, but we can see that there is a missing Long to BigInt conversion. So we search for all Long values in the code.

```
private case class Int64Impl(underlying: BigInt) extends Int64 {
  require(underlying >= -9223372036854775808L)
  require(underlying <= 9223372036854775807L)
}
```

Looks like it can not compare a BigInt with a Long value. We can easily convert this Long value to a BigInt with a string literal parameter by passing these numbers as strings to the BigInt constructor.

Finally, we get some output from Stainless about the verification in the code.

```
[...]
[Warning ]  => INVALID
[Warning ] Found counter-example:
[Warning ]   thiss: { x: Object | @unchecked isNumber(x) } -> Int64Impl(0)
[Warning ]   num: { x: Object | @unchecked isInt64(x) }    -> Int64Impl(9223372036854775808)
[...]
[  Info  ] +  precond. (call checkResult(thiss, underlying(thiss) + u ...)  invalid  Code.scala:16:45
[...]
[  Info  ] total: 5  valid: 4  (4 from cache) invalid: 1  unknown: 0  time:  1.317
```

This shows that there is an invalid specification in checkResult and Stainless prints a counterexample for it.

Let's ignore this for a moment and write the specification for the Addition-with-Zero Property.

3.10 Writing Specification for the Property

As specified, our verification must only support addition with zero. So we restrict the parameter to be zero in the precondition.

```
require(c.satoshis == Satoshis.zero)
```

We ensure the result is the same value as *this* in the postcondition.

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

We can use equals (==) directly on Satoshis, because it is a case class. The addition will now look like this:

```
override def +(c: CurrencyUnit): CurrencyUnit = {
  require(c.satoshis == Satoshis.zero)
  Satoshis(satoshis.underlying + c.satoshis.underlying)
} ensuring(res => res.satoshis == this.satoshis)
```

That's all we need to verify our addition.

Now we will look into the previous error.

3.11 Propagating Require

There is another problem with Bitcoin-S. Bitcoin-S-Core uses require as a fail-fast method whereas Stainless needs it to verify the code.

The Stainless output again:

```
[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 does 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 fails.

Thus, we need to add the restriction from checkResult to the addition 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, when adding one to the maximum Int64 the require does not hold anymore. Since we do only allow zero as a parameter, the easiest way is to restrict it to zero here too.

3.12 Result

Finally, everything is green and correctly verified.

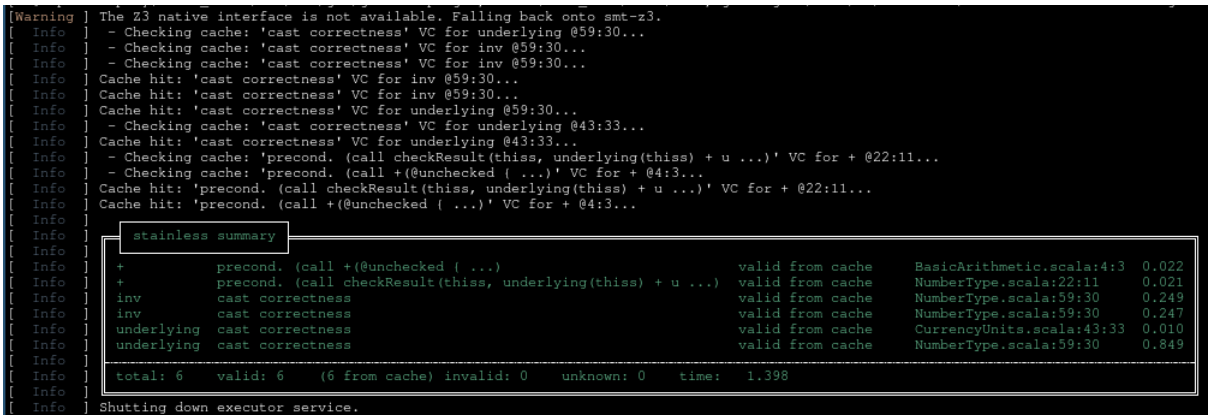


Figure 3.1: Output of Stainless verification for addition with 0 of Bitcoin-S-Cores CurrencyUnit

The verified code.

```
1 package code.end
2
3 trait BasicArithmetic[N] {
4   def +(n: N): N
5 }
6
7 sealed abstract class Number
8     extends BasicArithmetic[Int64] {
9
10  protected def underlying: BigInt
11
12  def toBigInt: BigInt = underlying
13
14  def apply: BigInt => Int64
15
16  override def +(num: Int64): Int64 = {
17    require(
18          num.underlying <= BigInt(0)
19      && this.underlying <= BigInt("9223372036854775807")
20      && this.underlying >= BigInt("-9223372036854775808")
21    )
22    apply(checkResult(underlying + num.underlying))
23  }
24
25  private def checkResult(result: BigInt): BigInt = {
26    require(
27         result <= BigInt("9223372036854775807")
28      && result >= BigInt("-9223372036854775808")
29    )
30    result
31  }
32 }
33
34 sealed abstract class SignedNumber extends Number
35
36 sealed abstract class Int64 extends SignedNumber {
37   override def apply: BigInt => Int64 = Int64(_)
38 }
39
40 trait BaseNumbers[T] {
41   def zero: T
42   def one: T
43 }
44
```

```scala
case object Int64 extends BaseNumbers[Int64] {
  lazy val zero = Int64(0)
  lazy val one = Int64(1)

  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}

private case class Int64Impl(underlying: BigInt) extends Int64 {
  require(underlying >= BigInt("-9223372036854775808"))
  require(underlying <= BigInt("9223372036854775807"))
}

sealed abstract class CurrencyUnit
    extends BasicArithmetic[CurrencyUnit] {
  def satoshis: Satoshis

  def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis

  override def +(c: CurrencyUnit): CurrencyUnit = {
    require(                                          // <--- specification
         c.satoshis == Satoshis.zero
      && this.underlying.toBigInt <= BigInt("9223372036854775807")
      && this.underlying.toBigInt >= BigInt("-9223372036854775808")
    )
    Satoshis(satoshis.underlying + c.satoshis.underlying)
  } ensuring(res => res.satoshis == this.satoshis) // <--- specification

  protected def underlying: Int64
}

sealed abstract class Satoshis extends CurrencyUnit {
  override def satoshis: Satoshis = this

  def toBigInt: BigInt = underlying.toBigInt

  def ==(satoshis: Satoshis): Boolean = underlying == satoshis.underlying
}

case object Satoshis extends BaseNumbers[Satoshis] {
  val zero = Satoshis(Int64.zero)
  val one = Satoshis(Int64.one)

  def apply(int64: Int64): Satoshis = SatoshisImpl(int64)
}

private case class SatoshisImpl(underlying: Int64) extends Satoshis
```

But is it really the original code that we verified? And why did we have to change that much? This and other questions will be answered in the next chapter.

4 Practical Challenges with Stainless

In this chapter we look at the practical challenges that could occur by using Stainless. Since Stainless is still in development and there is no 1.x release there might still be breaking changes and improvements fixing problems described now.

4.1 sbt vs JAR

We can either use the sbt plugin or a JAR file to check code with Stainless.

Invoking the JAR on our source code Stainless will verify it. If we have a bigger project, this becomes really tricky, because we must pass all files needed including the dependencies. This is in contrast to the sbt plugin, where we can integrate Stainless in our compilation process. When we call compile, Stainless verifies the code and stops the compilation if the verification fails.

Having a static version configured in the sbt build file, every developer has the same Stainless features available. This should prevent incompatibility with new or deprecated features when we use different plugins.

So the sbt plugin has clear advantages over the JAR file since its integrated directly. We do not have to download it manually and find the right version and if we bump the version we can just edit it in the build file and every developer is on the same version again.

However, currently there are some drawbacks. For example the sbt plugin does not always report errors. We will see more about that shortly.

4.2 Integration into Bitcoin-S

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

Stainless requires and Scala recommends Java SE Development Kit 8. Newer Java versions won't work.

To use the latest version of the sbt tool you have to build it locally. You can run `sbt universal:stage` in the cloned Stainless git repository. This generates *frontends/scalac/target/universal/stage/bin/stainless-scalac*.

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. Most of them can be written the sbt 0.13.17 way.

The bigger problem is, due to the different Scala and sbt versions, the following error 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 inadvertently.

The next approach is to use the stainless cli instead of sbt. Running stainless on all source files does not work, because 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' ' ')
```

.ivy2 is the dependency cache of sbt. The *tr* replaces the first char with the second so a newline with either ':' or ' '.

With this command, Stainless throws the next 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
```

So we can not know how many errors will face us. Let's go another way, because the errors may take too much time and it might lead to a next error. We extract the code needed to verify a transaction mainly the class Transaction and ScriptInterpreter with many other classes they're depending on.

After this extraction Stainless was successfully integrated with both sbt and JAR.

Running `sbt compile` in the project with Stainless ended without error. But it also ended with no output. So we are not able to change the code so Stainless would accept it since we do not know what to change.

So the sbt plugin does not always complain where the JAR file did. The open issue #484 on GitHub might describe exactly this error.

Now we can finally run Stainless on our code. But this leads us to the next findings. We must rewrite most of the code, as described in the previous chapters.

5 Conclusion

Because of the limitations of the verication tool, we could only verify a rewritten version of the original Bitcoin-S code. So we can not guarantee the correctness of the addition of Satoshis with zero in Bitcoin-S. Not all changes we made were as trivial as the replacement of objects with case objects. For these non-trivial changes, as seen for example the bound check in section 3.6, we cannot say whether they are equivalent to the original implementation or not.

So code should be written specically with formal verication in mind, in order to successfully verify it. Otherwise, it needs a lot of changes in the software because verification is mathematical and the current software is written mostly in object-oriented style. Software written in the functional paradigm would be much easier to reason about.

Thus, either Stainless must find ways to translate more of built-in object-oriented patterns of Scala to their verification tool or developers must invest more in functional programming.

Also, we found that trying to verify code reveals bugs as shown in section 2.3. Finally, our work led to some feedback to the Stainless developers to improve the tool.

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, 2019-06-13	
Last Names, First Names:	Doukmak Anna	Boss Ramon
Signatures:	.....	.....

Bibliography

[1] "Bitcoin Core Bug CVE-2018–17144: An Analysis." [Online]. Available: https://hackernoon.com/bitcoin-core-bug-cve-2018-17144-an-analysis-f80d9d373362

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

[3] Source code of Stainless. [Online]. Available: https://github.com/epfl-lara/stainless

[4] Stainless documentation. [Online]. Available: https://epfl-lara.github.io/stainless/