Towards Verifying the Bitcoin-S Library

Ramon Boss, Kai Brünnler, and Anna Doukmak

Bern University of Applied Sciences, CH-2501 Biel, Switzerland
ramon.boss@outlook.com, kai.bruennler@bfh.ch,
anna.doukmak@gmail.com

Abstract. We try to verify properties of the bitcoin-s library, a Scala implementation of parts of the Bitcoin protocol. We use the Stainless verifier which supports programs in a subset of Scala called *Pure Scala*. We first try to verify the property that regular transactions do not create new money. It turns out that there is too much code involved that lies outside of the supported fragment to make this feasible. However, in the process we find and fix a bug in bitcoin-s. We then turn to a much simpler property: that adding zero satoshis to a given amount of satoshis yields the given amount of satoshis. Here as well a significant part of the relevant code lies outside of the supported fragment. We perform a series of equivalent transformations on it until we arrive at code that we successfully verify. In that process we find and fix a second bug in bitcoin-s.

Keywords: Bitcoin · Scala · Bitcoin-S · Stainless.

1 Introduction

For software handling cryptocurrency, correctness is clearly crucial. However, even in very well-tested software such as Bitcoin Core, serious bugs occur. The most recent example is the bug found in September 2018 [6] which essentially allowed to arbitrarily create new coins. Such software is thus a worthwhile target for formal verification. In this work, we set out to verify properties of the bitcoin-s library with the Stainless verifier.

The Bitcoin-S Library. The bitcoin-s library is an implementation of parts of the Bitcoin protocol in Scala [8,9]. In particular, it allows to serialize, deserialize, sign and validate Bitcoin transactions. The library uses immutable data structures and algebraic data types but is not written with formal verification in mind. According to the website, the library is used in production, handling significant amounts of cryptocurrency each day [8].

The Stainless Verifier. Stainless is the successor of the Leon verifier [2,12,1] and is developed at EPF Lausanne [11]. It is intended to be used by programmers without training in formal verification. To facilitate that, it accepts specifications written in the programming language itself (Scala). Also, it focusses on counterexample finding in addition to proving correctness. Counterexamples are useful to programmers while correctness proofs are not – correctness is obvious or does not hold, and often both at the same time.

The example in Figure 1 adapted from the Stainless documentation [7] shows how the verifier is used. Notice how a precondition is specified using *require* and a postcondition using *ensuring*.

```
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)
```

Fig. 1. Factorial program with specification

Our function does not satisfy the specification. An overflow in the 32-bit Int leads to a negative result for the input 17, as Stainless reports in Figure 2. Changing the type from Int to BigInt will result in a successful verification.

```
[  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  ] ╚════════════════════════════════════════════════════════════════════════════════════════════════════════════╝
```

Fig. 2. Stainless output for the factorial program

The Pure Scala Fragment. The Scala fragment supported by Stainless is described in the Stainless documentation [7] in the section Pure Scala.

It comprises algebraic data types in the form of abstract classes, case classes and case objects, objects for grouping classes and functions, boolean expressions with short-circuit interpretation, generics with invariant type parameters, default values of function parameters, pattern matching, local and anonymous classes and more.

In addition to Pure Scala Stainless also supports some imperative features, such as using a (mutable) variable in a local scope of a function and while loops. They turn out not to be relevant for the current work.

What will turn out to be more relevant for us are the Scala features which Stainless does not support, such as: (concrete) class definitions, inheritance by objects, abstract type members, and inner classes in case objects.

Also, Stainless has its own library of some core data types and functions which need to be mapped correctly to functions inside of the SMT solver that Stainless ultimately relies on. Those data types in general do not have all the methods of the Scala data types. For example, the BigInt type in Scala has methods for bitwise operations while the BigInt type in Stainless does not.

Outline and Properties to Verify. In the next section we try to verify the property that a regular (non-coinbase) transaction can not generate new coins. We call it the *no-inflation property*. Trying to verify it, we uncover and fix a bug in the bitcoin-s library. We then find that there is too much code involved that lies outside of the supported fragment to currently make this verification feasible. So we turn to a simpler property to verify. The simplest possible property we can think of is the fact that adding zero satoshis to a given amount of satoshis yields the given amount of satoshis. We call it the *addition-with-zero property* and we try to verify it in Section 3. Here as well we see that a significant part of the code lies outside of the supported fragment. We perform a series of equivalent transformations on it until we arrive at code that we successfully verify. In that process we find and fix a second bug in bitcoin-s.

2 The No-Inflation Property

A crucial function for the verification of the no-inflation property is the function which validates transactions: the `checkTransaction` function in the ScriptInterpreter object, shown in Figure 3. It takes a transaction and if some basic checks succeed returns true, otherwise false. For example, one of those checks is that both the list of inputs and list of outputs need to be non-empty.

To better understand the validation of a transaction in bitcoin-s, it is useful to review how transactions are represented and created.

Creating a Transaction. The code in this subsection is adapted from the bitcoin-s documentation. To create a transaction, we first need some coins – an unspent transaction output. We could load an actual unspent transaction output from the bitcoin network, but we create one manually in order to see this process. So we first create an (invalid) transaction with one output in Figure 4.

We first create a keypair, then a lock script with its public key, then the amount of satoshis, then a transaction output (utxo) for that amount and locked with that script. Finally we create the actual transaction with that output and no inputs. Of course, that is not a valid transaction, because it creates coins out of nothing. In particular, checkTransaction(prevTx) returns false, simply because the list of inputs is empty.

Now that we have a transaction output, we create a transaction to spend it.

First, we need a so-called *out point*, which points to an output of a previous transaction. The second parameter is the index of the referenced output. Here

```
/**
  * Checks the validity of a transaction in accordance to bitcoin
      core's CheckTransaction function
  * https://github.com/bitcoin/bitcoin/blob/
      f7a21dae5dbf71d5bc00485215e84e6f2b309d0a/src/main.cpp#L939.
  */
def checkTransaction(transaction: Transaction): Boolean = {
  val inputOutputsNotZero =
    !(transaction.inputs.isEmpty || transaction.outputs.isEmpty)
  val txNotLargerThanBlock = transaction.bytes.size < Consensus.
      maxBlockSize
  val outputsSpendValidAmountsOfMoney = !transaction.outputs.exists(o
       =>
    o.value < CurrencyUnits.zero || o.value > Consensus.maxMoney)

  val outputValues = transaction.outputs.map(_.value)
  val totalSpentByOutputs: CurrencyUnit =
    outputValues.fold(CurrencyUnits.zero)(_ + _)
  val allOutputsValidMoneyRange = validMoneyRange(totalSpentByOutputs
      )
  val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
  val noDuplicateInputs = prevOutputTxIds.distinct.size ==
      prevOutputTxIds.size

  val isValidScriptSigForCoinbaseTx = transaction.isCoinbase match {
    case true =>
      transaction.inputs.head.scriptSignature.asmBytes.size >= 2 &&
        transaction.inputs.head.scriptSignature.asmBytes.size <= 100
    case false =>
      //since this is not a coinbase tx we cannot have any empty
          previous outs inside of inputs
      !transaction.inputs.exists(_.previousOutput ==
          EmptyTransactionOutPoint)
  }
  inputOutputsNotZero && txNotLargerThanBlock &&
      outputsSpendValidAmountsOfMoney && noDuplicateInputs &&
  allOutputsValidMoneyRange && noDuplicateInputs &&
      isValidScriptSigForCoinbaseTx
}
```

Fig. 3. The checkTransaction function

```
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  )
```

Fig. 4. Creating a transaction output to spend

we use zero because the previous transaction has only one output (with index zero).

Second, we need some information on how to spend the utxo from the previous transaction, in particular, how to sign the transaction to allow this. That is the information needed to create a transaction input, so essentially a transaction input.

Third, we assemble the list of transaction inputs, in our case just one.

Fourth, we set the amount of satoshis that we want to spend. The Int64 class emulates C data types in Scala, and we will look at it more in the next section.

Fifth, we create a lock script to send the coins to. To be supremely secure, we use a public key derived from a private key we immediately forget.

Sixth, we create our list of transaction outputs.

Seventh and eighth, we define the fee rate and some bitcoin network parameters.

Nineth, we create a transaction builder with those data.

Tenth, we tell the transaction builder to start signing the transaction.

Finally, we get the actual signed transaction. We could serialize it and send it to the Bitcoin network. We can also pass it to the checkTransaction function, which will return true.

A Bug in the checkTransaction Function. Note lines 16 and 17 of the checkTransaction function in Figure 3.

The value prevOutputTxIds gathers all transaction IDs referenced by the inputs of the current transaction. When we call *distinct* on the returned list, we get the list with duplicates removed. If the size of the new list is the same as the size of the old, we know that there is no duplicate transaction ID.

However, this code prevents a transaction with two inputs spending two different outputs of a previous transaction. The check is too strict: checkTransaction returns false for valid transactions.

```
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)

val destinationAmount = Satoshis(Int64(5000))

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

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

val feeRate = SatoshisPerByte(Satoshis.one)

val networkParams = RegTest // some static values for testing

val txBuilderF: Future[BitcoinTxBuilder] = BitcoinTxBuilder(
  destinations = destinations,
  utxos = utxos,
  feeRate = feeRate,
  changeSPK = creditingSPK,    // where to send the change
  network = networkParams
)

val txF: Future[Transaction] = txBuilderF.flatMap(_.sign)

val tx: Transaction = Await.result(txF, 1 second)
```

Fig. 5. Creating a transaction

The fix is simple: we perform the duplicate check on the TransactionOutPoints instead of on their transaction IDs. A TransactionOutPoint contains the txId as well as the output index it references.

Specifically, we replace lines 16-17 as follows:

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

Note that TransactionOutPoint is a case class and thus has a built in == method.

We submitted the fix together with a unit test to prevent this bug from appearing again in the future in a pull request [4], which has already been merged.

An Attempt at Verification. Naively trying Stainless on the entire bitcoin-s codebase results in many errors – as was to be expected. We tried to extract only the relevant code and to verify that. However, even the extracted code has more than 1500 lines and liberally uses Scala features outside of the supported fragment. We tried to transform the code into the supported fragment, but quickly realized that that a better approach is to first verify a simpler property with less code involved and later come back to the no-inflation property with more experience. So we now turn to the addition-with-zero property.

3 The Addition-with-Zero Property

It is of course a crucial property we are verifying here: if zero satoshis were credited to your account, you would not want your balance to change! It is also the simplest meaningful property to verify that we can think of. However, the code involved in performing an addition of two satoshi amounts in bitcoin-s is non-trivial. The reason for that is a pecularity of consensus code: agreement with the majority is more important than correctness, whatever correctness might mean. The most widely used bitcoin implementation by far is the reference implementation Bitcoin Core, written in C++. For consensus code, bitcoin-s has little choice but to be in strict agreement with the reference implementation. To achieve that, it implements C-like data types in Scala and then implements functionality using those C-like data types. For example, the Satoshis class, which is used to represent an amount of satoshis, is implemented using the class Int64 which represents the C-type `int64_t`.

Extracting the Relevant Code The relevant code for the addition of satoshis is in two files: `CurrencyUnits.scala` and `NumberType.scala`.

From those files we removed all code that is not needed for the verification of our property. For example, we removed all number types except for Int64 (so Int32, UInt64, etc.) because they are not used. We also removed the superclasses Factory and NetworkElement of CurrencyUnit and Number, respectively, because the inherited members are not used. Also, we removed all binary operations on Number that are not used, like subtraction and multiplication. The extracted code is shown in Figure 6 and Figure 7.

```scala
package reduced.number

sealed abstract class Number[T <: Number[T]] {
  type A = BigInt
  protected def underlying: A
  def toLong: Long = toBigInt.bigInteger.longValueExact()
  def toBigInt: BigInt = underlying
  def andMask: BigInt
  def apply: A => T
  def +(num: T): T = apply(checkResult(underlying + num.underlying))

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

sealed abstract class SignedNumber[T <: Number[T]] extends Number[T]

sealed abstract class Int64 extends SignedNumber[Int64] {
  override def apply: A => Int64 = Int64(_)
  override def andMask = 0xffffffffffffffffL
}

trait BaseNumbers[T] {
  def zero: T
  def one: T
  def min: T
  def max: T
}

object Int64 extends BaseNumbers[Int64] {
  private case class Int64Impl(underlying: BigInt) extends Int64 {
    require(underlying >= -9223372036854775808L,
      "Number was too small for a int64, got: " + underlying)
    require(underlying <= 9223372036854775807L,
      "Number was too big for a int64, got: " + underlying)
  }

  lazy val zero = Int64(0)
  lazy val one = Int64(1)
  lazy val min = Int64(-9223372036854775808L)
  lazy val max = Int64(9223372036854775807L)

  def apply(long: Long): Int64 = Int64(BigInt(long))
  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}
```

Fig. 6. Extracted Code from NumberType.scala

```scala
package reduced.currency

import reduced.number.{BaseNumbers, Int64}

sealed abstract class CurrencyUnit {
  type A
  def satoshis: Satoshis
  def !=(c: CurrencyUnit): Boolean = !(this == c)
  def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis
  def +(c: CurrencyUnit): CurrencyUnit = {
    Satoshis(satoshis.underlying + c.satoshis.underlying)
  }
  protected def underlying: A
}

sealed abstract class Satoshis extends CurrencyUnit {
  override type A = Int64
  override def satoshis: Satoshis = this
  def toBigInt: BigInt = BigInt(toLong)
  def toLong: Long = underlying.toLong
  def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
      underlying
}

object Satoshis extends BaseNumbers[Satoshis] {
  val min = Satoshis(Int64.min)
  val max = Satoshis(Int64.max)
  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
}
```

Fig. 7. Extracted Code from CurrencyUnits.scala

A Bug in the checkTransaction Function.
Transforming the Code.
When we run Stainless on this code (without any properties to prove), it throws the following errors: describe errors: no support for abstract types, unsupported arguments for the BigInt constructor, unsupported inheritance for objects.

...
why fix the preconditions:

- all green: new errors stand out more
- at runtime, errors are caught earlier, which is preferrable

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

```
15  def +(c: CurrencyUnit): CurrencyUnit = {
16    Satoshis(satoshis.underlying + c.satoshis.underlying)
17  } ensuring (res =>
18    (c == Satoshis.zero) ==> (res == this))
```

Result
Finally, everything is green and correctly verified.

```
[ Info ]
[ Info ] ┌─ stainless summary ─┐
[ Info ] └─────────────────────┘
[ Info ] ┌─────────┬──────────────────────────────────────────────────────┬──────────────────┬──────────┬──────────────────────────────────────────────────────────────────┬──────┐
[ Info ] │ +       postcondition                                          valid from cache           src/main/scala/addition/modified/currency/CurrencyUnits.scala:13:3   3.067 │
[ Info ] │ +       precond. (call checkResult(thiss, underlying(thiss) + u ...)  invalid           U:smt-z3  src/main/scala/addition/modified/number/NumberType.scala:17:36      0.220 │
[ Info ] │ apply   adt invariant                                          valid from cache           src/main/scala/addition/modified/currency/CurrencyUnits.scala:35:39  0.031 │
[ Info ] │ apply   adt invariant                                          invalid           U:smt-z3  src/main/scala/addition/modified/number/NumberType.scala:62:38      0.901 │
[ Info ] ├─────────┴──────────────────────────────────────────────────────┴──────────────────┴──────────┴──────────────────────────────────────────────────────────────────┴──────┤
[ Info ] │ total: 4    valid: 2    (2 from cache) invalid: 2    unknown: 0    time:   4.219                                                                                  │
[ Info ] └─────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────┘
```

Fig. 8. Output of Stainless verification for addition with 0 of Bitcoin-S-Cores CurrencyUnit

4 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 **??**, 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 **??**. Finally, our work led to some feedback to the Stainless developers to improve the tool.

conclusions: what's future work? how to change bitcoin-s? how to extend stainless?

References

1. Blanc, R., Kuncak, V.: Sound reasoning about integral data types with a reusable SMT solver interface. In: Haller and Miller [5], pp. 35–40. https://doi.org/10.1145/2774975.2774980
2. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013. pp. 1:1–1:10. ACM (2013). https://doi.org/10.1145/2489837.2489838
3. Boss, R.: Issue 519: Unknown type parameter type T in self referencing generic, https://github.com/epfl-lara/stainless/issues/519, accessed 2019-06-27
4. Boss, R.: Transaction can reference two different outputs of the same previous transaction, https://github.com/bitcoin-s/bitcoin-s/pull/435, accessed 2019-06-19
5. Haller, P., Miller, H. (eds.): Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, OR, USA, June 15-17, 2015. ACM (2015)
6. Song, J.: Bitcoin Core Bug CVE-2018–17144: An Analysis, https://hackernoon.com/bitcoin-core-bug-cve-2018-17144-an-analysis-f80d9d373362, accessed 2019-06-20
7. Stainless documentation: https://epfl-lara.github.io/stainless/, accessed 2019-06-19
8. Suredbits & the bitcoin-s developers: The bitcoin-s website, https://bitcoin-s.org, accessed 2019-06-19
9. The bitcoin-s developers: The bitcoin-s repository, https://github.com/bitcoin-s, accessed 2019-06-19
10. The Stainless developers: Pull request 470: Type aliases, type members, and dependent function types, https://github.com/epfl-lara/stainless/pull/470, accessed 2019-06-27
11. The Stainless developers: The stainless repository, https://github.com/epfl-lara/stainless, accessed 2019-06-19
12. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Haller and Miller [5], pp. 18–29. https://doi.org/10.1145/2774975.2774978

A Code Transformations

Here we see in detail how to transform the bitcoin-s code into the Scala fragment supported by Stainless. All subsections start with the Stainless error message(s) and finally a description of the changes we make to the code.

We claim that all transformations are equivalent in the sense that if the addition-with-zero property holds for the transformed code, then it also holds for the code before the transformation.

A.1 Inheriting Objects

```
[ Error  ] number/NumberType.scala:65:1: Objects cannot extend
              classes or implement traits, use a case object instead
           object Int64 extends BaseNumbers[Int64] {
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[ Error  ] currency/CurrencyUnits.scala:33:1: Objects cannot extend
              classes or implement traits, use a case object instead
           object Satoshis extends BaseNumbers[Satoshis] {
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

Here, we can just turn the objects into case objects by literally just changing the word `object` into `case object` on lines 65 and 33 in the two respective files.

That transformation is equivalent. Case objects have some additional properties (in particular, being serializable) and they inherit from `Product` instead of `AnyRef`, but none of our code depends on any of that.

A.2 Abstract Type Members

```
[ Error  ] currency/CurrencyUnits.scala:6:3: Stainless doesn't
              support abstract type members
           type A
           ^^^^^^
```

Note that we can not simply replace the unsupported abstract type member by a (supported) type parameter. The problem is that the CurrencyUnit class uses one of its implementing classes: Satoshis.

Satoshis would have to instantiate a potential type parameter with type Int64, so it would extend CurrencyUnit[Int64]. But that is too specific, because the return type of the +-method would then then be CurrencyUnit[Int64] not CurrencyUnit[A].

Since we only want to verify the addition of satoshis, and the Satoshis class overrides A with Int64 anyway, we just remove the abstract type and set it to Int64.

We remove line 6 and line 22 from CurrencyUnits.scala (to maintain line numbers we actually replace them with empty lines for now) and in line 18 we replace A by Int64.

A.3 Non-Literal BigInt Constructor Argument

```
[ Error  ] currency/CurrencyUnits.scala:26:33: Only literal arguments
             are allowed for BigInt.
           def toBigInt: BigInt = BigInt(toLong)
                                         ^^^^^^
```

As described before, the types in the Stainless library are more restricted than their Scala library counterparts. In particular, the Stainless BigInt constructor is restricted to literal arguments.

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

We replace line 26 by

```
1 def toBigInt: BigInt = underlying.toBigint
```

This is an equivalent transformation: the only thing that might go wrong in the detour via `Long` is that the BigInt underlying Int64 in turn underlying Satoshis does not fit into a Long. However, the only constructor of Int64 ensures exactly that.

A.4 Self-Reference in Type Parameter Bound

```
[ Error  ] number/NumberType.scala:8:30: Unknown type parameter type
             T
           sealed abstract class Number[T <: Number[T]] {
                                        ^^^^^^^^^^^^^^^
```

Stainless does not currently support a class with a type parameter with a type boundary that contains the type parameter itself. We opened an issue [3] on the Stainless repository and the Stainless developers have targeted Stainless version 0.4 to support such self-referential type boundaries.

For now, since our code only uses Number with type parameter T instantiated to Int64, we just remove the type parameter and replace it by Int64. We respectively replace lines 8, 44 and 49 by

```
1 sealed abstract class Number {
```

```
1 sealed abstract class SignedNumber extends Number
```

```
1 sealed abstract class Int64 extends SignedNumber {
```

and replace T by Int64 in lines 25-27.

A.5 Missing Member `bigInteger` in BigInt

```
[ Error  ] number/NumberType.scala:14:22: Unknown call to bigInteger
             on Number.this.toBigInt (BigInt) with arguments List()
             of type List()
           def toLong: Long = toBigInt.bigInteger.longValueExact()
                              ^^^^^^^^^^^^^^^^^^^
```

stainless error emssage missing

The Scala class BigInt is essentially a wrapper around java.math.BigInteger. BigInt has a member bigInteger which is the underlying instance of the Java class. The Java class has a method longValueExact which returns a long only if the BigInteger fits into a long, otherwise throws exception. Stainless does not support Java classes and in particular its BigInt has no member bigInteger.

However, our code never calls toLong anymore, so we just remove it. We replace line 14 in NumberType.scala and line 28 in CurrencyUnits.scala by an empty line.

A.6 Type Member

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

Our version of Stainless does not support type members. We just replace all occurrence of A with BigInt, since A is never overwritten in an implementing class.

We remove line 9 in NumberType.scala and replace A by BigInt in lines 12, 25, 33 and 50.

In the mean time Stainless has implemented support for type member [10]. Since version 0.2 verification should succeed without this change.

A.7 Missing Bitwise-And Method on BigInt

```
[ Error  ] number/NumberType.scala:34:14: Unknown call to & on result
             (BigInt) with arguments List(Number.this.andMask) of
             type List(BigInt)
               require((result & andMask) == result,
                       ^^^^^^^^^^^^^^^^
```

Contrary to Scala BigInt, the Stainless BigInt class does not support bitwise operations, in particular not the &-method for bitwise and.

The bitwise and with the andMask on line 34 is a bounds check. It checks if the result parameter is in range of the specified type, which in our case is the hard coded Int64.

So, we replace the & mask with a check whether the result is in range of Long.MinValue and Long.MaxValue, because Int64 has the same 64-bit range as Long.

We replace lines 34-35 by the following, squeezed into two lines to preserve line numbers. Note that the BigInt constructor requires a literal:

```
1 require(result <= BigInt("9223372036854775807")
2   && result >= BigInt("-9223372036854775808"),
3       "Result␣was␣out␣of␣bounds,␣got:␣" + result)
```

kai here we do not know if it changes semantic. ramon: oh, we don't?

A.8 Inner Class in Case Object

```
[Warning ] number/NumberType.scala:66:3: Could not extract tree in
             class: case private class Int64Impl extends Int64 with
             Product with Serializable {
...
[Warning ] } (class scala.reflect.internal.Trees$ClassDef)
              private case class Int64Impl(underlying: BigInt)
                           extends Int64 {
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...

...
[Warning ] currency/CurrencyUnits.scala:42:3: Could not extract tree
             in class: case private class SatoshisImpl extends
             Satoshis with Product with Serializable {
...
[Warning ] } (class scala.reflect.internal.Trees$ClassDef)
              private case class SatoshisImpl(underlying: Int64)
                           extends Satoshis
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Stainless does not support inner classes in a case object. Bitcoin-s uses this a lot to separate the class

ramon: i don't understand from its implementation.

This is easy to fix. We just move the inner classes out of the case objects. They do not interfere with any other code.

We remove lines 66-71 in NumberType.scala and insert them at the end of the file. We remove line 42 in CurrencyUnits.scala and insert it at the end of the file.

A.9 Message Parameter in Require

```
[Warning ] number/NumberType.scala:67:3: 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$Apply)
          require(underlying >= -9223372036854775808L,
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[Warning ] number/NumberType.scala:69:3: Could not extract tree in
             class: scala.this.Predef.require(Int64Impl.this.
             underlying.<=(math.this.BigInt.long2bigInt
             (9223372036854775807L)), "Number was too big for a int64
             , got: ".+(Int64Impl.this.underlying)) (class scala.
             reflect.internal.Trees$Apply)
          require(underlying <= 9223372036854775807L,
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[ Error  ] checkResult$0 depends on missing dependencies: require$1.
```

Here the require(condition, message) is used as an assertion: if the condition is false the fail with message. Stainless does not support the message parameter of require. For the verification we can simply remove that parameter.

A.10 Missing Implicit Long to BigInt Conversion

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

This error message does not specify a line number and it is not clear what "inv" is. However, the Scala BigInt has implict conversions from Long and they are missing in the Stainless BigInt.

We replace the two require clauses at lines 67 and following in NumberType.scala

```
  require(underlying >= -9223372036854775808L)
  require(underlying <= 9223372036854775807L)
```

by:

```
  require(underlying >= BigInt(-9223372036854775808L))
  require(underlying <= BigInt(9223372036854775807L))
```

We also replace lines 76 and 77

```
  lazy val min = Int64(-9223372036854775808L)
  lazy val max = Int64(9223372036854775807L)
```

by:

```
  lazy val min = Int64(BigInt(-9223372036854775808L))
  lazy val max = Int64(BigInt(9223372036854775807L))
```

A.11 Missing BigInt Constructor with Long Argument

```
[ Error  ] apply$14 depends on missing dependencies: BigInt$0,
             apply$15.
```

Here again, the Scala BigInt has a constructor with a Long argument which is missing in the Stainless BigInt.

We simply replace the Long values in a BigInt constructor call with a string literal.

The lines from the previous transformation respectively change into the following:

```
  require(underlying >= BigInt("-9223372036854775808"))
  require(underlying <= BigInt("9223372036854775807"))
```

and

```
  lazy val min = Int64(BigInt("-9223372036854775808"))
  lazy val max = Int64(BigInt("9223372036854775807"))
}
```