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
Date:	TODO fix this

Versions

Version	Date	Status	Remarks
1.0	TODO fix this	Final	Final Version

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus scelerisque, leo sed iaculis ornare, mi leo semper urna, ac elementum libero est at risus. Donec eget aliquam urna. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc fermentum nunc sollicitudin leo porttitor volutpat. Duis ac enim lectus, quis malesuada lectus. Aenean vestibulum suscipit justo, in suscipit augue venenatis a. Donec interdum nibh ligula. Aliquam vitae dui a odio cursus interdum quis vitae mi. Phasellus ornare tortor fringilla velit accumsan quis tincidunt magna eleifend. Praesent nisl nibh, cursus in mattis ac, ultrices ac nulla. Nulla ante urna, aliquet eu tempus ut, feugiat id nisl. Nunc sit amet mauris vitae turpis scelerisque mattis et sed metus. Aliquam interdum congue odio, sed semper elit ullamcorper vitae. Morbi orci elit, feugiat vel hendrerit nec, sollicitudin non massa. Quisque lacus metus, vulputate id ullamcorper id, consequat eget orci .

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, adjustes a code of a software and extands a testbench verifying previously uncovered aspects. [**?**]

Another powerfull 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. [**?**]

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.

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 outcomes of verification with Stainless are possible: valid, unvalid 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 occured. Furthermore, it will be verified by Stainless that a precondition cannot be violated.

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 a 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 to be verified should be written in Pure Scala belonging to functional programming paradigm. To extend this subset of the Scala language Stainless supports also some imperative features translating them into Pure Scala concepts. For more details about Stainless its official website can be explored. [**?**]

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

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 (abt) plugin or a command line tool. Both, when run, analyses the passed code 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 continously 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 abt 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 webpage 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. The 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...)

Stainless supports only a subset of Scala. They call it Pure Scala. We will see some of this restrictions we have detected during this work in the next chapter. There is a list of what's supported in the Stainless documentation.

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 this 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.[**?**] Bitcoin-S-Core has a bitcoin transcation 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 keypair to sign the next transaction and have a scriptPubKey where the money is. Then define the amount to spend (here 10000 Satishis) 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, a 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 a 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 prevOutputs = transaction.inputs.map(_.previousOutput)
val noDuplicateInputs = prevOutputs.distinct.size == prevOutputs.size
```

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

4. Trying to Verify checkTransaction

4.1. Integration (Versionskonflikte, neues Plugin)

4.2. Error Reporting with sbt and JAR

4.3. Findings

4.4. Bugfix

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 more easy, 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, one can directly use equals (==) on Satoshis, because its 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 lets 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, changeing 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 to 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

// I dont understand this.. maybe just a missing feature in Stainless? Stainless output:

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

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

5.3. Rewriting Objects

// TODO reason?

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.

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 occurence of A with BigInt. This might be a missing feature in Stainless that should not be to 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 usecase its 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 where as 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 lets add a restriction for num to be zero.

Finally, it works and verifies correctly.

5.10. Result

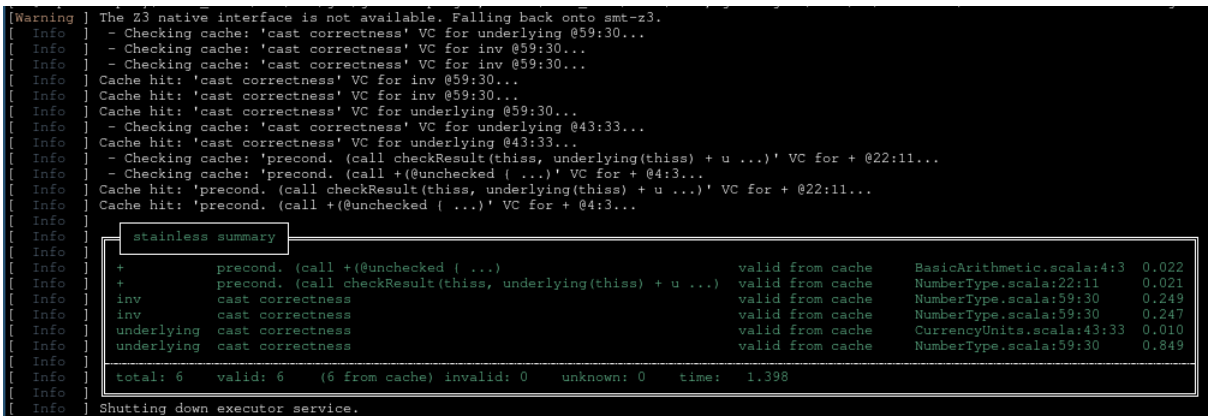

Figure 5.1.: Output of Stainless verification for addition with 0 of Bitcoin-S-Cores CurrencyUnit

6. Conclusion

But I must explain to you how all this mistaken idea of denouncing pleasure and praising pain was born and I will give you a complete account of the system, and expound the actual teachings of the great explorer of the truth, the master-builder of human happiness. No one rejects, dislikes, or avoids pleasure itself, because it is pleasure, but because those who do not know how to pursue pleasure rationally encounter consequences that are extremely painful. Nor again is there anyone who loves or pursues or desires to obtain pain of itself, because it is pain, but because occasionally circumstances occur in which toil and pain can procure him some great pleasure. To take a trivial example, which of us ever undertakes laborious physical exercise, except to obtain some advantage from it? But who has any right to find fault with a man who chooses to enjoy a pleasure that has no annoying consequences, or one who avoids a pain that produces no resultant pleasure? On the other hand, we denounce with righteous indignation and dislike men who are so beguiled and demoralized by the charms of pleasure of the moment, so blinded by desire, that they cannot foresee the pain and trouble that are bound to ensue; and equal blame belongs to those who fail in their duty through weakness of will, which is the same as saying through shrinking from toil and pain. These cases are perfectly simple and easy to distinguish. In a free hour, when our power of choice is untrammelled and when nothing prevents our being able to do what we like best, every pleasure is to be welcomed and every pain avoided. But in certain circumstances and owing to the claims of duty or the obligations of business it will frequently occur that pleasures have to be repudiated and annoyances accepted. The wise man therefore always holds in these matters to this principle of selection: he rejects pleasures to secure other greater pleasures, or else he endures pains to avoid worse pains.

But I must explain to you how all this mistaken idea of denouncing pleasure and praising pain was born and I will give you a complete account of the system, and expound the actual teachings of the great explorer of the truth, the master-builder of human happiness. No one rejects, dislikes, or avoids pleasure itself, because it is pleasure, but because those who do not know how to pursue pleasure rationally encounter consequences that are extremely painful. Nor again is there anyone who loves or pursues or desires to obtain pain of itself, because it is pain, but because occasionally circumstances occur in which toil and pain can procure him some great pleasure. To take a trivial example, which of us ever undertakes laborious physical exercise, except to obtain some advantage from it? But who has any right to find fault with a man who chooses to enjoy a pleasure that has no annoying consequences, or one who avoids a pain that produces no resultant pleasure? On the other hand, we denounce with righteous indignation and dislike men who are so beguiled and demoralized by the charms of pleasure of the moment, so blinded by desire, that they cannot foresee the pain and trouble that are bound to ensue; and equal blame belongs to those who fail in their duty through weakness of will, which is the same as saying through shrinking from toil and pain. These cases are perfectly simple and easy to distinguish. In a free hour, when our power of choice is untrammelled and when nothing prevents our being able to do what we like best, every pleasure is to be welcomed and every pain avoided. But in certain circumstances and owing to the claims of duty or the obligations of business it will frequently occur that pleasures have to be repudiated and annoyances accepted. The wise man therefore always holds in these matters to this principle of selection: he rejects pleasures to secure other greater pleasures, or else he endures pains to avoid worse pains.

6.1. Future Work

Declaration of primary authorship

I / We hereby confirm that I / 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/Burgdorf], TODO fix this	
Last Name/s, First Name/s:	[Test Peter]	[Müster Rösä]
Signature/s:	.....	.....

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.

B. Additional Appendix

B.1. Test 1

To an English person, it will seem like simplified English, as a skeptical Cambridge friend of mine told me what Occidental is. 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.

B.1.1. Environment

It will be as simple as Occidental; in fact, it will be Occidental. To an English person, it will seem like simplified English, as a skeptical Cambridge friend of mine told me what Occidental is. 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.

C. Content of CD-ROM

Content of the enclosed CD-ROM, directory tree, etc.