Bachelor Project
Smart Contracts In Stainless
Verification & Compilation

Romain Jufer

Spring 2018

Contents

Introduction **4**

1 Smart Contracts In Stainless **5**
1.1 Solidity Library . . . 5
1.1.1 Address . . . 5
1.1.2 Environment . . . 5
1.1.3 Message . . . 5
1.1.4 Contract Interface . . . 6
1.1.5 Contract . . . 6
1.1.6 Mapping . . . 6
1.1.7 Utilities . . . 6
1.2 Declaring a Contract . . . 6
1.3 Interface . . . 7
1.4 Types . . . 7
1.5 Functions . . . 7
1.5.1 Fallback Function . . . 7
1.5.2 Calling Methods With Ether . . . 8
1.6 Annotations . . . 8
1.7 Example - Loan . . . 9
1.7.1 Concept . . . 9
1.7.2 Code . . . 9

2 Verification **11**
2.1 Method Call & Implicit Values . . . 11
2.2 Pay Function . . . 12
2.3 Payable Modifier . . . 12
2.4 Gas And Transaction Price . . . 13
2.5 Solidity Library . . . 13
2.6 Example 1 - Contract Interaction . . . 13
2.7 Example 2 - Loan . . . 13
2.7.1 Interface ERC20 . . . 14
2.7.2 Loan Contract . . . 15

3 Compilation **17**
3.1 Compiler Architecture . . . 17
3.2 Solidity Code . . . 17
3.2.1 Contract . . . 18
3.2.2 Constructor . . . 18
3.2.3 Functions & Methods . . . 19
3.2.4 Contract Interface . . . 20
3.2.5 Enumeration . . . 20
3.2.6 Types equivalences . . . 21
3.2.7 Globally Available Variables . . . 21
3.2.8 Loops and Switch . . . 21

3.2.9 Assert & Require 21

Conclusion **22**

Listings

1.1 Declaring a contract . . . 6
1.2 Declaring an interface . . . 7
1.3 Declaring Fallback Function . . . 7
1.4 Call with ether . . . 8
1.5 Declaring a payable function . . . 8
1.6 Loan Contract . . . 9
2.1 Solidity globally available variables . . . 11
2.2 Transformation of the message variable . . . 11
2.3 Transformation of the environment variable . . . 12
2.4 Transformation of the function 'pay' of the library . . . 12
2.5 Contract interaction . . . 13
2.6 Interface ERC20 . . . 14
2.7 Loan contract invariant . . . 15
2.8 Loan contract ADT invariant . . . 15
2.9 Loan contract payback method . . . 16
3.1 Contract compilation . . . 18
3.2 Address cast compilation . . . 18
3.3 Constructor compilation . . . 18
3.4 Function return type compilation . . . 19
3.5 Pay Function compilation . . . 19
3.6 Interface compilation . . . 20
3.7 Enumeration declaration compilation . . . 20
3.8 Enumeration expression compilation . . . 20

Introduction

This project was conducted during my last semester of bachelor at EPFL under the supervision of Professor Viktor Kunčak and his assistants Jad Hamza and Romain Ruetschi. The focus of this work is to create a language that can be formally verified and that allows us to write smart contracts which can be deployed on the Ethereum blockchain. The Ethereum blockchain was chosen as it is the most widely used infrastructure for smart contracts.

Our language uses Stainless, which is a tool developed by the laboratory for automated reasoning at EPFL. It is used to verify mostly functional programs but also supports imperative features.

The first chapter presents our language and its features. It covers the particularities of Solidity and how we emulate them in our language. The next chapter presents the verification phase that we have integrated in Stainless. A detailed example is presented to illustrate the subtleties of our languge. Finally, the last chapter covers the compilation phase and how we translate our language to Solidity.

Chapter 1

Smart Contracts In Stainless

In this section we discuss the design choices we made for our language. One of the key consideration was to make it look like Solidity code. The reason is that Solidity is the most widely used language to write smart contracts and we wanted to facilitate the adoption of our language.

Our language is based on a subset of Stainless and thus can be verified. It can also be compiled to solidity code using our compiler. In the next paragraphs we first present how we implemented the particular features of the solidity language then we go through a complete example of a loan contract written in our language.

1.1 Solidity Library

In order to represent some of the features of the Solidity language, we provide the Solidity library. To write a contract that can be verified and compiled to Solidity, it is mendatory to import it. The library provides the following utilities.

1.1.1 Address

The address class is used to modelize the type *address* in Solidity. The value of an address[1] in Solidity can be specified by an hexadecimal literals of 39 to 41 digits long. In our language we choose to represent that value as a BigInt. The class also provides two methods, *balance* and *send*. They mimic the behavior of the corresponding methods defined on the address type in Solidity.

1.1.2 Environment

The environment is defined both by an object and a case class. The user must not use the environment directly but should use the methods *send* and *transfer* defined on *Address* and *Contract*. However, if the user needs to use the environment, it must do so by using the object and not the case class. The justification is given in section 2.1.

Both the object and the class environment represent the balance, in ether, of the participants. In the current implementation, the amount of ether is typed as BigInt.

1.1.3 Message

We have defined in the library both an object and a case class *Msg*. The same rule than for the environment applies, which means that the user must not use the case class *Msg*. *Msg* mimic the actual variable *msg*[2] that exists in Solidity and represents the information on the caller of a contract's method. There are two useful methods defined on *Msg*, *sender* and *value*. The first one returns the address of the caller. The second returns the amount of ether sent during the call.

[1] http://Solidity.readthedocs.io/en/v0.4.21/types.html#address
[2] http://solidity.readthedocs.io/en/v0.4.21/units-and-global-variables.html#block-and-transaction-properties

1.1.4 Contract Interface

Equivalent of *interface*[3] in Solidity. An example is given in the next section.

1.1.5 Contract

Base class to define a contract. More detail is given in the section 1.2. Note however that *Contract* inherits from *ContractInterface*.

1.1.6 Mapping

The mapping class represents the corresponding type in Solidity[4]. A companion object provides a function *constantMapping[A,B](b: B)* to create a constant mapping for all values. This is typically used by the environment since every address must have a positive balance.

1.1.7 Utilities

Along the mentionned classes, the library also provides the following functions

- def now() : return the current block timestamp
- def pay[A](f: A, amount: BigInt) : call function f and forwards it an *amount* of ether. Modelize call with ether in Solidity.
- def address(contract: ContractInterface) : return the address of a certain contract. Used to represent the cast address(contract) allowed in Solidity.

1.2 Declaring a Contract

All the code written in our language must be contained in an object named *SmartContract*. This is due to the way we do the verification and the compilation. A contract is declared as a case class that must extend the abstract class *Contract* from the library.

```
1 object SmartContract {
2   case class MyContract(
3     // Contract Fields
4   ) extends Contract {
5     // Contract Code
6   }
7 }
```

Listing 1.1: Declaring a contract

The user can specify several fields for its contract, both *val* and *var* are allowed. A contract inherits an address field from the super class. In the next examples the object surrounding our code is implicit but not explicitly written.

[3] http://Solidity.readthedocs.io/en/v0.4.21/contracts.html#interfaces
[4] http://Solidity.readthedocs.io/en/v0.4.21/types.html#mappings

1.3 Interface

In Solidity, it is possible to define interfaces. This is useful to create a contract that can interact with another contract that is already deployed. Typically, the ERC20 interface is widely used to ensure compatibility between several contracts. This is illustrated by the example given in section 1.7. In our language, we specify an interface by inheriting the class *ContractInterface*. While it is possible to define parameters and implemented methods in an interface, it is important to note that they are removed during compilation. This is discussed in the chapter on compilation and it is due to a restriction in the Solidity language.

```
1 case class MyInterface() extends ContractInterface {
2   def balance(addr: Address) = {
3     // Code
4   }
5 }
```

Listing 1.2: Declaring an interface

1.4 Types

In the current version, the user can use the following types :

- Boolean
- Int
- BigInt
- String
- Unit
- Address
- ContractInterface
- Contract

The user can declare new types by defining other contracts or case classes. However, only contracts can have parameters. The reason is that we choose to compile every other case classes as enumerations. More details is given in section 3.2.5

1.5 Functions

Fuctions are defined as usual. Note that we do not support higher-order functions nor lambda. Futhermore, functions that are outside a contract are not compiled to Solidity. Hence the user must write every functions related to the proof of the contract outside the contract.

1.5.1 Fallback Function

In Solidity, a contract can have a particular function named *fallback* which is called when a contract receives ether directly. If the function does not exist, the transfer of ether fails. Therefore, every contract who wants to receive money should declare this function. In our language, the function is defined as follows :

```
1 case class MyContract(
2   // Contract Fields
3 ) extends Contract {
4   def fallback() = { ... }
5 }
```

Listing 1.3: Declaring Fallback Function

Note that the function must be called fallback and takes no parameters.

1.5.2 Calling Methods With Ether

It is possible, when calling a method of a contract in Solidity, to also forward an amount of ether. We provide this feature through the use of the function *pay* defined in the library. The function must take a call to the method of a contract that is annotated as *payable* as well as a positive amount of ether. More detail on the implementation of the function is given in section **??**.

```
1  case class ERC20() extends ContractInterface {
2    @smart contractsPayable
3    def payable() = Msg.value
4  }
5
6  case class MyContract(otherContract: ERC20) extends Contract {
7    def pay() = pay(otherContract.payable, 20) // transfer 20 ether when calling the
         method
8  }
```

Listing 1.4: Call with ether

1.6 Annotations

Solidity provides several keywords to specify the definition of a function[5]. However, most of them are not implemented in the current version of the language.

Modifier	Status
payable	implemented
view	not implemented
pure	not implemented

Since both *pure* and *view* are not implemented in Solidity, we have chosen to only provide *payable*. Payable is used to specify that a function can be called with an amount of ether. Furthermore, a call with an amount of ether to a function that is not explicitly defined as payable will fail. In our language, we denote a function as payable using the annotation *@smart contractsPayable*. The user must import the package *Stainless.annotation*.

```
1  case class MyContract(
2    // Contract Fields
3  ) extends Contract {
4    @smartcontractsPayable
5    def payableFun() = { ... }
6  }
```

Listing 1.5: Declaring a payable function

[5] http://Solidity.readthedocs.io/en/v0.4.21/contracts.html#functions

1.7 Example - Loan

1.7.1 Concept

We now present an example of a contract that manages a loan of ether between two parties. Conceptually the contract works as follow :

1. Someone, call it the borrower, wants to borrow an amount of ether. He setups the contract by specifying the amount he wants, the duration of the loan and the amount he will refund at the end. In addition to that, we require that the borrower also provides a form of guarantee to the person who is lending the money. Otherwise one could just borrow the money and never returns it back. We have chosen as a guarantee an amount of digital tokens managed by another contract that lives on the Ethereum blockchains. The only requirement is that the contract must implement the ERC20 interface so that we know how to interact with it. ERC20 is the standard interface used by all the ICOs. In order to provide the guarantee, the borrower specifies the contract that holds the tokens and transfer the guarantee in favor of the loan contract.

2. Once the contract is setup and deployed on the blockchain, the borrower must call the function *checkTokens* from the loan contract. This function verifies that the loan contract owns the guarantee by querying its balance on the contract that holds the digital tokens. If it is the case, it will now transition in a state where it will wait for someone to lend the money.

3. To lend the money, the lender has to call the function *lend* with an amount of ether. If the amount is correct, the loan contract forwards the money to the borrower and then enters a new state to wait for either the refund of the loan or the end of the duration of the loan.

4. Suppose that the borrower refunds the money. It calls the function *payback* with an amount of ether. If the amount is right then the loan contract forwards it to the lender and transfers the guarantee back to the borrower. The loan contract then ends.

5. Suppose that the duration of the contract is reached. The lender can call the function *requestDefault* which will forward the guarantee that the loan contract holds to the lender. The contract then ends.

Here is a state diagram that resume the life of the contract

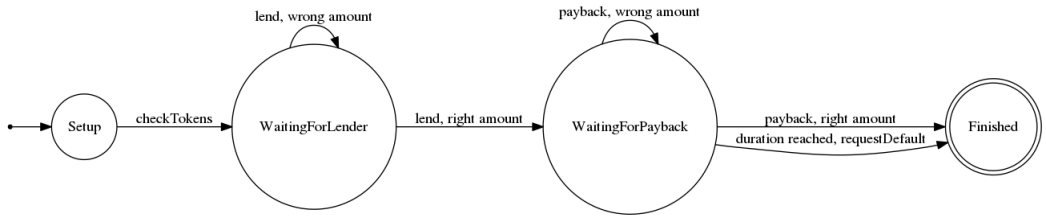

Figure 1.1: State Machine Loan Contract

1.7.2 Code

```
1  object SmartContract {
2      case class ERC20Token(var s: BigInt) extends ContractInterface {
3          def transfer(to: Address, amount: BigInt): Boolean
4          def balanceOf(from: Address): BigInt
5      }
6
7      sealed trait State
8      case object Setup extends State
9      case object WaitingForLender extends State
10     case object WaitingForPayback extends State
11     case object Finished extends State
12
```

```
13      sealed case class Loan (
14          val borrower: Address,        // Amount of ether to borrow
15          val wantedAmount: BigInt,     // Interest in ether
16          val premiumAmount: BigInt,    // The amount of digital token guaranteed
17          val tokenAmount: BigInt,      // Name of the digital token, e.g GNT
18          val tokenName: String,        // Reference to the contract that holds the tokens
19          val tokenContractAddress: ERC20Token,
20          val daysToLend: BigInt,
21          var currentState: State = WaitingForData,
22          var start: BigInt,
23          var lender: Address )  extends Contract {
24          def checkTokens(): Unit = {
25              if(currentState == WaitingForData) {
26                  val balance = tokenContractAddress.balanceOf(addr)
27                  if(balance >= tokenAmount) {
28                      currentState = WaitingForLender
29                  }
30              }
31          }
32
33          @smartcontractPayable
34          def lend(): Unit = {
35              // Condition to prevent self funding.
36              if(Msg.sender != borrower) {
37                  if(currentState == WaitingForLender && Msg.value >= wantedAmount) {
38                      lender = Msg.sender
39                      // Forward the money to the borrower
40                      if(borrower.send(wantedAmount)) {
41                          currentState = WaitingForPayback
42                          start = now()
43                      }
44                  }
45              }
46          }
47
48          @smartcontractPayable
49          def payback(): Unit = {
50              if(currentState == WaitingForPayback &&
51                  Msg.value >= premiumAmount + wantedAmount  &&
52              Msg.sender == lender) {
53                  // Forward the money to the lender
54                  if(lender.send(Msg.value)) {
55                      // Transfer all the guarantee back to the borrower
56                      val balance = tokenContractAddress.balanceOf(addr)
57                      if(tokenContractAddress.transfer(borrower, balance)) {
58                          currentState = Finished
59                      }
60                  }
61              }
62          }
63
64          def requestDefault(): Unit = {
65              if(currentState == WaitingForPayback &&
66                  now() > (start + daysToLend) &&
67              Msg.sender == lender) {
68                  // Transfer all the guarantee to the lender
69                  var balance = tokenContractAddress.balanceOf(addr)
70                  if(tokenContractAddress.transfer(lender, balance)) {
71                      currentState = Finished
72                  }
73              }
74          }
75      }
76  }
```

Listing 1.6: Loan Contract

Chapter 2

Verification

In this section we discuss how the verification of smart contracts written in Stainless is implemented. We first present the particularities of the smart contracts and how we modelize them in our framework. Then we present two examples that illustrate our implementation.

2.1 Method Call & Implicit Values

In Solidity, inside the body of a contract's method, we have access to information about the caller. More precisely, we have access to the address of the contract who called the method as well as the amount of ether sent to the method. This is done through a globally available variable named *msg*. Here is an example of Solidity code that illustrates this feature.

```
1 contract Example {
2   function foo() payable public {
3     if(msg.value > 0)
4       msg.sender.send(msg.value)
5   }
6 }
```

Listing 2.1: Solidity globally available variables

The function *foo* sends the amount of ether just received back to the caller. In our framework, we use the object *Msg* defined in the library to represent this feature. It is important to note that the value represented by this object changes at each call made by a contract. To modelize this aspect we add, during the verification, a new parameter to the signature of every method that uses the object *Msg*. The parameter is of type *Msg*, which explains why we need both an object and a class for *Msg*. Furthermore, in the body of every method, we replace the uses of the object *Msg* by the newly added parameter.

In addition to that, when a method of a contract calls another method, we have to pass a new message that specifies the contract as the caller. If the call to the method is done using the function *pay* defined in the library, we also pass the amount paid in the message. Here is an example of the transformation performed.

```
1 case class MyContract() extends Contract {
2   def foo() = {
3     Msg.value > 0
4   }
5 }
```

becomes

```
1 case class MyContract() extends Contract {
2   def foo(msg: Msg) = {
3     msg.value > 0
4   }
5 }
```

Listing 2.2: Transformation of the message variable

Note a few things here. We said that we add the parameter if the corresponding object is used in the code. It includes the case where, even though there is no explicit usage of *Msg*, we call another method that does require this extra parameter. Furthermore we chose not to add the parameter if it is not needed in order to avoid extra complexity in the proofs.

The same idea is also applied to the environment. This justifies the existence of both an object and a class named *Environment*. The environment is implicitly used by the methods *send* and *balance* available in the class *Address*. Therefore, the extra parameter is added if an explicit or implicit call is made to these functions.

```
1 case class MyContract() extends Contract {
2   def foo() = {
3     Msg.sender.send(20)
4   }
5 }
```

becomes

```
1 case class MyContract() extends Contract {
2   def foo(msg: Msg, env: Environment) = {
3     msg.sender.send(20, env)
4   }
5 }
```

Listing 2.3: Transformation of the environment variable

2.2 Pay Function

In the Solidity library we provide a function *pay* that takes a call to the method of a contract and an amount of ether. As presented in the first chapter, this is used to modelize a call with ether in Solidity. During the verification we replace the call to the function *pay* by a call to the method *send* on the address of the target contract to send the specified amount of ether and then we call the actual method. The amount of ether is also forwarded to the method that is called by the mean of the *msg* parameter. Here is an illustration of the transformation. Suppose that we also have defined another contract named *MyOtherContract* which has a payable function named *payable* that takes no argument.

```
1 case class MyContract(otherContract: MyOtherContract) extends Contract {
2   def foo() = pay(otherContract.payable, 20)
3 }
```

becomes after desugaring

```
1 case class MyContract(otherContract: MyOtherContract) extends Contract {
2   def foo(env: Environment) = {
3     otherContract.addr.send(20, env)
4     otherContract.payable(...)
5   }
6 }
```

Listing 2.4: Transformation of the function 'pay' of the library

2.3 Payable Modifier

As presented in the first chapter, a function can be annotated as payable. In Solidity, a call with ether to a function succeeds only if the function is specified as *payable*. In our current implementation we decided to fail the verification if this property is not satisfied.

2.4 Gas And Transaction Price

In our current implementation we do not have any notion of gas. It could be interesting to specify properties on gas to guarantee that a function cannot cost an infinit amount of ether. It could also be interesting to verify that a call to the method of another contract does not forward too much gas to avoid reentrancy or the execution of malicious code. However it is difficult to get a correct equivalence between the amount of gas used by the real Solidity code and ours.

2.5 Solidity Library

While inspecting the Solidity library, one can remark that certain classes possess what seems to be unused method. For example the method *self* in the object *Environment* that returns the class *Environment* and has an unspecified body. This is due to the fact that unused code is optimized away by the Scala compiler. Hence since the user must not use the class *Msg* for exemple, then it is removed from the verified code and we cannot do the substitution we presented above. Therefore, until we find a better fix, we have added these methods that reference the unused code to guarantee that it is not optimized away.

2.6 Example 1 - Contract Interaction

We first consider a very basic example that illustrate the transformation made during the verification process. Consider the following two contracts

```
1  case class Target() extends Contract {
2    @smartcontractPayable
3    def receiveMoney() = { }
4  }
5
6  case class Source(
7    val targetContract:Target
8  ) extends Contract {
9    def send() = {
10     require (
11       address(this).balance >= 50 &&
12       address(targetContract).balance == 0
13     )
14
15     pay(targetContract.receiveMoney, 20)
16     assert(address(targetContract).balance >= 20)
17   }
18 }
```

Listing 2.5: Contract interaction

The idea is that we have a function *send* that will call the method of another contract, forwarding some ether along the way. We then assert that after the call, since the method of the target contract does nothing, the balance of the target must be greater than 20. In the original Stainless this assertion should not be true since initially there is no notion of environment. However in our framework this assertion is indeed true. This is because the *pay* call is desugared as a transfer of ether done on the environment and then a call to the method of the contract.

2.7 Example 2 - Loan

We return to our loan contract and explain how we verify it in Stainless. The major property we want to prove for this contract is the following :

$$state(loanContract) \in \{WaitingForLender, WaitingForPayback\}$$
$$\implies ERC20Token.balance(loanContract) >= guarantee$$

Which means that the guarantee must be held by the contract during the loan. This will be our invariant for our proof. However, this property uses the interface ERC20 for which we don't have an implementation. Therefore we have to specify the behavior of the methods of the interface through pre- and post-conditions if we want to be able to prove our invariant. We we first consider the code used for the interface

2.7.1 Interface ERC20

```
1  def transferUpdate(a: Address, to: Address, sender: Address, amount: BigInt, thiss:
       ERC20Token, oldThiss: ERC20Token) = {
2      ((a == to) ==> (thiss.balanceOf(a) == oldThiss.balanceOf(a) + amount)) &&
3      ((a == sender) ==> (thiss.balanceOf(a) == oldThiss.balanceOf(a) - amount)) &&
4      (a != to && a != sender) ==> (thiss.balanceOf(a) == oldThiss.balanceOf(a))
5  }
6
7  def transferSpec(b: Boolean, to: Address, sender: Address, amount: BigInt, thiss:
       ERC20Token, oldThiss: ERC20Token) = {
8      (!b ==> (thiss == oldThiss)) &&
9      (b ==> forall((a: Address) => transferUpdate(a, to, sender,amount,thiss,oldThiss)))
           &&
10   (thiss.addr == oldThiss.addr)
11 }
12
13 def snapshot(token: ERC20Token): ERC20Token = {
14     val ERC20Token(s) = token
15     ERC20Token(s)
16 }
17
18 case class ERC20Token(var s: BigInt) extends ContractInterface {
19     @library
20     def transfer(to: Address, amount: BigInt): Boolean = {
21         require(amount >= 0)
22         val oldd = snapshot(this)
23         s = s + 1
24
25         val b = choose((b: Boolean) => transferSpec(b, to, Msg.sender, amount, this,
               oldd))
26         b
27     } ensuring(res => transferSpec(res, to, Msg.sender, amount, this, old(this)))
28
29     @library
30     def balanceOf(from: Address): BigInt = {
31         choose((b: BigInt) => b >= 0)
32     } ensuring {
33         _ => old(this).addr == this.addr
34     }
35 }
```

Listing 2.6: Interface ERC20

The functions *transferUpdate* and *transferSpec* are used to specify what happens on a call to the methods *transfer*. We assume that transfer works as follows :

If transfer returns *true*, we can assert the following properties :

1. The balance of the address who receives the money is equals to its previous balance plus the amount transfered.

2. The balance of the address who is transfering the money is equals to his previous balance minus the amount transfered.

3. In the case where both the sender and the receiver are the same then the balances stay the same.

In the case where transfer returns false the we assert that the balances are not changed. We also assert that a call to *transfer* cannot modify the address of the contract. Even though we don't know the real implementation of the function, those assumptions are fair since it is the expected behavior of the interface ERC20.

The actual body of the function *transfer* is a bit complicated and not really interesting. It uses a trick to simulate the mutation of the contract after the call to transfer and then assert the specifications presented above.

For the function *balance* the only property comes from the fact that the balance of an address cannot be negative. Therefore we always returns a value that is positive. We also ensure that the function does not modify the address of the contract.

2.7.2 Loan Contract

We want to prove that the following invariant holds for all our functions. The invariant specifies exactly the property about the guarantee. The reason for not having a strict equality is that any user can transfer an amount of token to this contract at any time. We do not have control over that since it happens on the ERC20 contract.

```
1 def tokenInvariant(
2     loanContractAddress: Address,
3     contractState: State,
4     tokenAmount: BigInt,
5     tokenContractAddress: ERC20Token
6 ): Boolean = {
7     (contractState == WaitingForLender ==> (tokenContractAddress.balanceOf(
          loanContractAddress) >= tokenAmount)) &&
8     (contractState == WaitingForPayback ==> (tokenContractAddress.balanceOf(
          loanContractAddress) >= tokenAmount))
9 }
```

Listing 2.7: Loan contract invariant

We also have some properties specified as an ADT invariant.

```
1 require (
2 addr != borrower &&
3 wantedAmount > 0 &&
4 premiumAmount > 0 &&
5 tokenAmount > 0 &&
6 daysToLend > 0 &&
7 start >= 0 &&
8 addr != tokenContractAddress.addr)
```

Listing 2.8: Loan contract ADT invariant

In Solidity, ether amount should be specified as uint256. In our framework, we do not have such type at the moment hence we use BigInt as an equivalent. This is why we assert that some quantities must be positive. However it is important to remark that a bounded type is subject to overflow which is not the case of BigInt. We did not have the time to implement the correspondance between bounded types in Stainless/Inox and the ones in Solidity, therefore we use BigInt for the moment.

We also have to assert properties on the address of both contract. While those assertions are trivial on the Ethereum blockchain, since two contracts cannot be deployed at the same address, it is much more difficult to define in our framework.

We now turn to the actual verification of the body of our contract. We want to ensure that calls to the function of the contract do not break the invariant. The proof is trivial once we have setup all the requirements that are due to the EVM. For example the fact that no two contract can have the same address and that the address, once set, cannot change.

The function *checkTokens* and *lend* are trivial and therefore we do not present them here. The code can be found in the repository. Consider now the function *payback*.

```
@smartcontractPayable
def payback(): Unit = {
  require (
    tokenInvariant(addr, currentState, tokenAmount, tokenContractAddress) &&
    addr.balance >= Msg.value
  )
  if(currentState == WaitingForPayback &&
    Msg.value >= premiumAmount + wantedAmount) {

    if(lender.send(Msg.value)) {
      val balance = tokenContractAddress.balanceOf(addr)

      if(tokenContractAddress.transfer(borrower, balance)) {
        currentState = Finished
      }
    }
  }
} ensuring {
  tokenInvariant(addr, currentState, tokenAmount, tokenContractAddress)
}
```

Listing 2.9: Loan contract payback method

The proof is simple, however we see that we have to require that the contract possess at least the amount of ether specified in the field value of the message. This is trivially true, however since we do not have a complet axiomatisation of the EVM we have to require this property explicitly.

Finally the function *requestDefault* is also very simple hence not presented here. Initially we also wanted to prove that any other transaction on the blockchain that does not involve this contract directly could not break the invariant. We simulated this idea through a method in the contract that only forwarded the calls and we showed that the invariant could not be broken. However when doing the verification, we change the signature of every methods in a contract as presented above. Therefore we would have changed this method to replace the message passed with a new one specifying our contract as the caller, which is not what we want. To resolve this problem we thought about providing an annotation to allow the user a finer grain control over which message is passed when calling a function. However this lead to an added complexity and it was not clear how to correctly achieve that, hence we decided to not implement it for the moment.

We also had another invariant that assert a property on the chronological order of the states of the contract. We wanted to ensure that we can not go from *WaitingForPayback* back to the start of the contract for example. However in order to implement this invariant, we need a new variable in the contract that keep the history of the visited state. But we do not want to compile this variable, this is a ghost variable used only in the proof. We also thought about providing an annotation to specify that such variable are only used in the proof. The problem is that we have to ensure that the user does not use it in statement where "*real*" variable are used. The reason is that if we do not compile the variable we also have to remove every statement that references the variable and it is not clear for the user what we are going to remove and how. This could lead to uncorrect compiled code and other problems therefore we decided to not implement this feature in this version.

Chapter 3

Compilation

In this section we present the compilation from Stainless to Solidity. We discuss what we currently support with our compiler and what can be added in a next version.

3.1 Compiler Architecture

Our compiler is directly integrated to the Stainless code. One should use the argument *–solidity* to call the compiler. It is implemented as a new callback to a new component. Our compiler is composed of the following classes

SmartContractCallBack This is the entry point to the compiler. We do not support parallel compilation at the moment hence the compilation phase starts at the end of the extraction to ensure that we have gathered all the dependancies.

SmartContractComponent The component takes a Stainless program and gather the contracts, the interfaces and the enumerations found in the input files. It then calls the Solidity transformer to create what we call a SolidityModule. A SolidityModule is a collection of transformed contracts, interfaces and enumerations found in the same input file. When every module has been generated, it forwards them to the printer to create the Solidity files.

SolidityCode This class transforms the mentionned contracts, interfaces and enumerations into a Solidity module.

SolidityPrinter As the name suggests, this class output a file for a given Solidity module.

3.2 Solidity Code

As mentioned in the first chapter, the code must be declared inside an object name *SmartContract*. This restriction is due to the way we compile the code, we have to reference certain function by their lookup path and we need to know the name of the package in which they are defined. This is a temporary constraint until we find a better approach.

3.2.1 Contract

Every class that inherits from Contract is compiled to a new contract in Solidity. The parameters of the class are compiled as class fields. Here is an basic example to illustrate the process :

```
case class MyContract(owner: Address) extends Contract {
  // More Code
}
```

becomes

```
contract MyContract {
  address owner;

  // More code
}
```

Listing 3.1: Contract compilation

In Solidity, the keyword *this* returns the address of the contract it references. Therefore the compiler replaces every call to *addr* in a contract by *this*. In the library we also provide a function *address* that can be applied to either a contract or a contract interface. At compile time, a call to this function is replaced by a cast to address in Solidity. It is due to the fact that in Solidity, we can explicitly cast a contract to an address by using *address(c)* where *c* is a contract. Here is an example that illustrates both features.

```
case class MyContract(otherContract: MyContract) extends Contract {
  def getBalance = addr.balance
  def getOtherBalance = Address(otherContract).balance
}
```

becomes

```
contract MyContract {
  MyContract otherContract;

  function getBalance() public returns(uint256) { this.balance; }
  function getOtherBalance() public returns(uint256) { address(otherContract).balance; }
}
```

Listing 3.2: Address cast compilation

Note that even though it is not recommanded, the user can replace *Address(otherContract)* by *otherContract.addr*. We compile both alternative the same way as presented above.

3.2.2 Constructor

Every contract has a default constructor that is automatically added if none is specified by the user. In Solidity a contract can have at most one constructor. Solidity does not support constructor overloading, neither do we.

To define a new constructor the user has to create a companion object to its contract. Inside the object, a constructor is a function named *constructor* that should return an instance of the contract. Here is an example to illustrate this feature.

```
case class MyContract(a: BigInt) extends Contract { // Code }
object MyContract {
  def constructor(_a: BigInt) = MyContract(a)
}
```

This will be compiled to

```
contract MyContract {
  uint256 a;

  constructor(uint 256 _a) public { a = _a }
  // Code
}
```

Listing 3.3: Constructor compilation

The name of the arguments taken by the constructor the user must use different names than the names of the corresponding fields. The reason is that *this* references the address of the contract, hence we cannot write *this.a = a* but we have to write *address(this).a = a*. Since every line of code costs an amount of gas, it is better to avoid this type of syntax. Therefore we chose to throw an error if an argument of the constructor has the same name as one of the field of the class.

3.2.3 Functions & Methods

In Solidity, every piece of code must be in a contract or an interface. We do not compile code that is not in one of them.

Functions

We support the following definition of a function in Solidity :

function _name_ (_params_) [external|public] [payable] [returns (_type_)

The access modifier external is used for the methods in a contract interface as recommanded by the Solidity compiler. Every methods defined in a contract is specified as *public* for the moment. The modifier *payable* is specified through the annotation *@smartcontractPayable*. In a next version of the compiler other modifiers will be added.

In Solidity, a function can return multiple values however we do not support that for the moment. In our compiler a function can return at most one value. Here is an example

```
case class MyContract() extends Contract {
  def add(a: BigInt) = a + 5
}
```

becomes

```
contract MyContract {
  function add(uint256 a) public returns (uint256) {
    return a + 5;
  }
}
```

Listing 3.4: Function return type compilation

Pay function

Suppose that we have defined another contract named *MyOtherContract* that has a payable function *receiveEther*. The pay function is transformed as follows

```
case class MyContract(otherContract: MyOtherContract) extends Contract {
  def pay = pay(otherContract.receiveEther, 20)
}
```

```
contract MyContract {
  MyOtherContract otherContract;

  function pay() public { otherContract.receiveEther.value(20)(); }
}
```

Listing 3.5: Pay Function compilation

In the case where the pay function does not take a call to the method of a contract in argument, the compiler will throw an error.

3.2.4 Contract Interface

Every class that inherits directly *ContractInterface* is compiled to an interface. In the current version of the compiler, every methods in an interface class is compiled as an abstract method, even if it posses a body in the source code. This is due to the fact that an interface in Solidity cannot have implemented function. The signature of the function is compiled appropriately. Note however that the access modifier is *external*. This is to avoid the warnings emitted by the official Solidity compiler if we specify something else. The following example illustrates the translation.

```
case class ERC20() extends ContractInterface {
  def balance(addr: Address) = {
    // Can have a body in Stainless
  }

  def transfer(to: Address, amount: BigInt)
}
```

```
interface ERC20 {
  function balance(addr: address) external;
  function transfer(to: address, amount: uint256) external;
}
```

Listing 3.6: Interface compilation

3.2.5 Enumeration

Solidity allows the user to define enumerations. In Stainless we provide this construction through case object that inherits a same trait. Every case object defined in the source code is considered as being part of an enumeration. Furthermore, such a case object cannot have parameters.

```
sealed trait State
case object Start() extends State
case object Running() extends State
case object Finish() extends State
```

Is compiled to

```
enum State { Start, Running, Finish}
```

Listing 3.7: Enumeration declaration compilation

During the compilation we gather every case object and we group them by their parent trait. Then we compile each group into a proper enumeration. Every use of the case object in the body of a contract is replaced by a reference to the corresponding value in the enumeration.

```
case class MyContract(currState: State) extends Contract {
  def done = {
    currState = Finish()
  }
}
```

becomes

```
contract MyContract {
  State currState;

  function done() public {
    currState = State.Finish;
  }
}
```

Listing 3.8: Enumeration expression compilation

In Solidity, enumeration must be defined in the contract's body. Currently, to simplify the compiler, we decided to add the enumerations built during compilation to every contracts that is being compiled. As a next step, we would optimize the compiler so that we only add the enumerations that are actually used.

3.2.6 Types equivalences

The following table present how the types in Stainless are translated to existing types in Solidity. We do not support all the types that exist in Solidity. Furthermore, some of the mappings are temporary. For exemple BigInt should not be transformed to *uint256* since it is both unbounded and can hold positive and negative values which is not the case of *uint256*. However we chose to temporarily do so because we did not have the time to implement the right type in Stainless/Inox.

Stainless	Solidity
Int	int32
BigInt	uint256
Boolean	bool
String	string
Address	address
Mapping[A,B]	mapping(a => b)

3.2.7 Globally Available Variables

Solidity offers several variables that can be access from everywhere in the body of a contract. For the time being we only support the following

Stainless Msg (object)	Solidity msg
sender	sender
value	value
unsupported	gas
unsupported	sig
unsupported	data

Stainless <Address>	Solidity <address>
balance	balance
send	send
unsupported	transfer
unsupported	call
unsupported	callcode
unsupported	delegatecall

3.2.8 Loops and Switch

Switch are not supported by Solidity hence we do not implement them. Loops exist however we do not support them in the current version of our compiler. Moreover, since every transaction in Solidity has a cost, it is not advised to use loop.

3.2.9 Assert & Require

Assert and Require exist both in Stainless and Solidity. In order to avoid conflicts we do not support those keywords in Solidity for the moment. All the expressions that are used for the verification in Stainless are not compiled to Solidity. Hence the user should not use require and assert to ensure that some properties are respected at runtime.

Conclusion

The main objective of this project is to provide a bridge between a language that we can formally verify and Solidity, the language of smart contracts. There is still much to do if we want to have a solid and reliable implementation. However, the work done in this project provides a foundation on which it is possible to build more advanced features.

The next steps will be to develop our language to support all the functionalities of Solidity. We have seen, for example, that we do not use the notion of gas, which is inherent to the blockchain. The difficulty here resides in finding a good correspondance between the cost of instructions in Solidity and our code. We could, for example, compile our code to Solidity and evaluate the cost of our program using the compiled code.

Another step is to find a way to define formally the EVM, the rules and constraints by which it obeys, in our language. We have seen that some requirements could be automatically added to each functions. However, by going down this path, we hide information from the user. This could lead to a more difficult usage of the language. Nonetheless, this should allow us to express more complex and interesting properties on the way smart contracts interact with their environment.

Finally, the compiler has to be extended to get a complete correspondance between our language and Solidity. For example, we have to decide how to deal with case classes and pattern matching. For the case classes, we can try to compile them to *struct* in Solidity. The problem resides mostly in the fact that case classes are often used with pattern matching. It is not natural to use them the same way we use a *struct* in an imperative language. Therefore, we have to find a way to compile pattern matching to a Solidity expression while minimizing the gas cost of such translation. We have also noted some minor problems due to the optimization made by the Scala compiler.

In conclusion, this work is a first step towards a language that is formally verifiable and in which it is possible to write secure and reliable Ethereum smart contracts.