A Java Framework for Smart Contracts

Fausto Spoto

Università di Verona, Italy

WTSC 2019, Saint Kitts
February 22, 2019

Programming Smart Contracts

Transactions are atomic computation steps of a blockchain

Smart contracts are an object-oriented presentation of transactions over a shared heap

Which programming language?

- Bitcoin uses a low-level non-Turing complete bytecode for programming transactions
- Ethereum uses Turing-complete languages compiled into EVM bytecode (mainly Solidity)

Solidity

Compared to Bitcoin bytecode, Solidity was a revolution, but...

- minimal toolbelt (IDE, builders, integrators, testing, analyzers...)
- low-level semantics (memory/storage distiction is explicit)
- no exception handling
- weak typing
- no inner classes, nor anonymnous classes
- no lambda expressions, nor method references
- no generics
- very small support library
- limited production of libraries for blockchain
- another language to learn!

Solidity

Compared to Bitcoin bytecode, Solidity was a revolution, but...

- minimal toolbelt (IDE, builders, integrators, testing, analyzers...)
- low-level semantics (memory/storage distiction is explicit)
- no exception handling
- weak typing
- no inner classes, nor anonymnous classes
- no lambda expressions, nor method references
- no generics
- very small support library
- limited production of libraries for blockchain
- another language to learn!

What about using Java instead?

Not Just Us

Java for blockchain programming

- there are Java implementations of blockchain nodes
- EthereumJ allows Java clients to query Ethereum blockchains
- NEO allows one to use the Java syntax for writing smart contracts:

```java
public class ICOTemplate extends SmartContract {
  public static Object deploy() { // static Object !!!
    if (getTotalSupply().length != 0) {
      Runtime.log("Insufficient token supply-No action");
      return false;
    }
    Storage.put(Storage.currentContext(), TemplateToken.getOwner(), ...);
    Storage.put(Storage.currentContext(), NEP5Template.TOTAL_SUPPLY, ...);
    return true;
  }
}
```

- Aion seems to go in a similar direction

Takamaka: Transparent Use of Java for Smart Contracts

```java
import takamaka.lang.Contract; import takamaka.lang.Payable;
import takamaka.lang.Storage; import takamaka.util.StorageList;

public class CrowdFunding extends Contract {
  private final StorageList<Campaign> campaigns = new StorageList<>(); // generics!

  // callable from everywhere
  public int newCampaign(Contract beneficiary, int goal) {
    int campaignId = campaigns.size();
    campaigns.add(new Campaign(beneficiary, goal));
    return campaignId;
  }

  // only callable from another instance of Contract; requires payment
  public @Payable @Entry void contribute(int amount, int campaignID) {
    campaigns.elementAt(campaignID).addFunder(caller(), amount);
  }

  // callable from everywhere
  public boolean checkGoalReached(int campaignID) {
    return campaigns.elementAt(campaignID).payIfGoalReached();
  }
```

```
  private class Campaign extends Storage { // inner class
    private final Contract beneficiary;
    private final int fundingGoal;
    private final StorageList<Funder> funders = new StorageList<>();
    private int amount; // one could also use BigInteger for this field

    private Campaign(Contract beneficiary, int fundingGoal) {
      this.beneficiary = beneficiary; this.fundingGoal = fundingGoal;
    }

    private void addFunder(Contract who, int amount) {
      funders.add(new Funder(who, amount)); this.amount += amount;
    }

    private boolean payIfGoalReached() {
      if (amount >= fundingGoal) {
        pay(beneficiary, amount); amount = 0; return true;
      }
      else
        return false;
    }
  }
}
```

How Takamaka Nodes Execute Transactions

Execute `cf = new CrowdFunding()`:

```
cf = new CrowdFunding(); // Java execution in RAM only
updates = emptyset;
cf.extractUpdates(updates); // Takamaka provides this method
blockchain.store(updates); // expands the blockchain
return cfRef = cf.storageReference to the wallet
```

Execute `id = cf.newCampaign(beneficiary, 42)`:

```
cf = blockchain.deserialize(cfRef);
beneficiary = blockchain.deserialize(beneficiaryRef);
id = cf.newCampaign(beneficiary, 42); // Java execution in RAM only
updates = emptyset;
cf.extractUpdates(updates); // Takamaka provides this method
beneficiary.extractUpdates(updates);
blockchain.store(updates); // expands the blockchain
return id to the wallet
```

Key Features of Takamaka

1. constructors and methods are completely normal Java code, that operates without explicit primitives for storage manipulation
2. only white-listed methods of the standard Java library can be used: the deterministic ones
3. the wallet uses storage references to refer to objects in the blockchain, since actual memory addresses are node-dependent
4. deserialization of storage references is lazy
5. only updates are serialized at the end (this is not standard Java serialization!)

Instrumentation of Contract Classes

```
public class MyContract extends takamaka.lang.Contract {
  public @Entry T m1(args                    ) {

    body
  }

  public @Payable @Entry T m2(int amount, args                    ) {

    body
  }
}
```

Instrumentation of Contract Classes

```
public class MyContract extends takamaka.lang.Contract {
  public @Entry T m1(args, Contract caller) {
    entry(caller);
    body
  }

  public @Payable @Entry T m2(int amount, args              ) {

    body
  }
}
```

Instrumentation of Contract Classes

```
public class MyContract extends takamaka.lang.Contract {
  public @Entry T m1(args, Contract caller) {
    entry(caller);
    body
  }

  public @Payable @Entry T m2(int amount, args, Contract caller) {
    payableEntry(caller, amount);
    body
  }
}
```

Instrumentation for Gas Metering

Before each bytecode instruction, Takamaka adds a call to
`takamaka.lang.Gas.tick(int amount)`

- the `amount` depends on the bytecode instruction
- `tick` can throw an `OutOfGasException`
- that exception cannot be caught in code

White-listed Methods

They must be deterministic

- methods of `java.lang.String*`
- methods of wrapper classes in `java.lang.`
- `java.util.Arrays`
- `java.util.ArrayList`, `java.util.LinkedList`, `java.util.PriorityQueue`
- most `java.util.Date`
- ...

Black-listed

- `java.lang.System.currentTimeSystem()`
- `java.lang.Object.hashCode()`
- ...

White-listing Surprises

Should we white-list `java.util.HashSet` and `java.util.HashMap`?

1. no, since iteration on them is not deterministic! (behavioral non-determinism)
2. no, since (for instance) `set.add(element)` might call a variable number of times the `hashCode` and `equals` methods on the elements of the set (cost non-determinism)

A solution is to require that `hashCode` must be redefined on objects put inside these collections (enforced by static/dynamic verification)

JVM Code Verification

The JVM verifies that some basic Java constraints are met:

- types are strong
- visibility modifiers are honored
- `final` methods and classes are not redefined
- called methods do exist (no fall-back methods!) or an exception is thrown
- accessed fields do exist or an exception is thrown

Takamaka Further Verification

Static

- storage classes have only fields of type primitive, storage, `java.lang.String`, `java.math.BigInteger` or `java.lang.Object` (for generics)
- `@Entry` is only applied to methods of contracts
- `@Payable` is only applied to `@Entry` methods with an `int amount` first parameter
- `@Entry` methods are only called from the code of a contract
- `caller()` can only be called on `this` and from an `@Entry` method

Dynamic

- accessed storage fields of type `java.lang.Object` actually hold a storage object, a `java.lang.String` or a `java.math.BigInteger`
- `@Entry` methods are only called from a distinct contract instance
- `@Payable` methods receive a non-negative `amount` and the caller contract has enough funds

Status of the Project

Done

- September 2018: project started
- December 2018: first draft of the working principles (this paper!)
- February 2019: Java bytecode instrumenter for storage objects and contracts (the transformation described in the previous slides)

To do

- March 2019: execution of instrumented contracts on an in-memory simulation of a blockchain
- April 2019: verification of jars before instrumentation
- fall 2019: integration into a real blockchain currently being developed by an independent company

Open Questions and Future Work

- is Takamaka actually keeping track of all storage updates?
- does the updates-only approach actually support scalability?
- which security guarantees can be proved for Takamaka?
- can we add a layer of non-trivial static analysis?
  - overflow/underflow checks
  - complexity analysis
  - inference of parametric gas costs
- how much can we enlarge the set of white-listed methods?
- which other types can we allow in storage objects? (wrapper types, arrays, enums. . .)

THANKS!