Fungible and Non-Fungible Tokens with Snapshots in Java

Marco Crosara[1†], Luca Olivieri[1,2†], Fausto Spoto[1*†] and Fabio Tagliaferro[1,3†]

[1*]Università degli Studi di Verona, Verona, Italy.
[2] Corvallis S.r.l., Padova, Italy.
[3] Commercio.network S.p.A., Schio, Italy.

*Corresponding author(s). E-mail(s): fausto.spoto@univr.it;
Contributing authors: marco.crosara@studenti.univr.it; luca.olivieri@univr.it;
fabio.tagliaferro@univr.it;
†These authors contributed equally to this work.

Abstract

Many blockchain applications exchange tokens, such as bitcoin and ether, or implement them through smart contracts. A trend in blockchain is to apply standards for token interoperability, unchanged, from platform to platform, easing the design challenges with trusted and widely-used specifications. However, the exploitation of the target language semantics can result in technological advantages and more efficient contracts. This paper presents a re-engineering of OpenZeppelin's implementation of the ERC-20 and ERC-721 standards in Takamaka, a Java framework for programming smart contracts. It describes a sound solution to the issue about the types allowed for the token holders and a novel implementation for making snapshots of tokens, based on tree maps, that is possible in Java, but not in Solidity, more efficient than the literal translation in Java from Solidity, within the Java Virtual Machine. Moreover, it applies to ERC-721 as well, where a snapshot mechanism was previously missing.

Keywords: Smart contract, software reengineering, blockchain, token, ERC-20, ERC-721

1 Introduction

Blockchains exploit the redundant, concurrent execution of the same transactions on a decentralized network of machines to enforce their execution in accordance with a set of predefined rules. Namely, blockchains make it hard, for a single machine, to disrupt the semantics of the transactions or their ordering: a misbehaving single machine gets immediately put out of consensus and isolated. Bitcoin [3, 19] has been the first blockchain's success story. Bitcoin transactions are transfers of cryptocurrency between accounts, with the specific rule that the same inputs cannot be spent twice. Bitcoin's cryptocurrency is called bitcoin itself and has been the first example of a blockchain *token*.

A few years after Bitcoin, another blockchain, called Ethereum [4, 6], introduced the possibility of programming transactions in an actual, imperative programming language, called Solidity, whose code is compiled for the Ethereum Virtual Machine (EVM). Ethereum's transactions are still paid in terms of its native *ether* token, but they execute much more than native token transfers.

Namely, transactions can also run object constructors and methods of code units called *smart contracts*, so that the Ethereum blockchain becomes a sort of *world computer* that persists the same objects in the memory of all the machines in the blockchain's network. The transactions included in the blocks of the blockchain must be the same in every machine of the network and must lead to the same outcome. Machines that do not abide to this rule will be put out of consensus and their future transactions will be rejected by the other machines.

Typically, smart contracts are written using domain-specific languages (DSLs) such as Solidity, that have specific features and restrictions for blockchain. More recently, the trend in smart contract development shifted to the usage of well-known general-purpose programming languages providing useful syntactical features missing in DSLs, along with ready-to-use available developer toolbelts. This has opened the opportunity to re-engineer and optimize the implementation of several existing standards for different blockchains. Java is among these general-purpose languages, since it enjoys large popularity [1,2] and a modern set of programming tools.

A popular class of Solidity smart contracts implements a dynamic ledger of coin transfers between accounts. These coins are not native tokens, but rather new, derived tokens, implemented in software through a smart contract[1]. Native and derived tokens can be categorized in many ways [13,20,30]. The most popular classification is between *fungible* and *non-fungible* tokens. Fungible tokens are interchangeable with each other since they have an identical nominal value, that is not tied to each specific token instance. Both native tokens and traditional (*fiat*) currencies are fungible tokens. Their main application is in the area of crowdfunding and in initial coin offers to support startups. On the contrary, non-fungible tokens have a value that depends on their specific instance. Hence, in general, they are not interchangeable. Their main application is currently in the art market, where they represent a written declaration of the author's rights concession to the holder, in gaming and, in general, in notarization.

A few standards have emerged for fungible and non-fungible tokens, that should guarantee correctness [23], accessibility, interoperability, management and security of the smart contracts that run the tokens. Among them, the Ethereum Request for Comment #20 (ERC-20 [12]) and #721 (ERC-721 [11]) are the most popular for fungible and non-fungible tokens, respectively, also outside Ethereum [15,16,18]. They provide developers with a list of rules required for the correct integration of tokens with other smart contracts and with applications external to the blockchain, such as wallets, block explorers, decentralized finance protocols and games.

The most popular implementations of the ERC-20 standard are in Solidity, by OpenZeppelin [21], a team of programmers in the Ethereum community who deliver useful and secure smart contracts and libraries, and by ConsenSys [7], later deprecated in favor of OpenZeppelin's. OpenZeppelin extends ERC-20 with snapshots, *ie.* immutable views of the state of a token contract, that show its ledger at a specific instant of time. They are useful for investigating the consequences of an attack, for creating forks of the token and for implementing mechanisms based on token balances such as weighted voting. Snapshots are essential also to provide an immutable view of the ledger that can be queried by a client without the risk that it changes during the query, which would result in a race condition.

In the case of ERC-721, the standard implementation is in Solidity, again by OpenZeppelin [22]. That implementation does not provide a snapshot mechanism, despite the usefulness of such feature. The reason is that the already very tricky implementation in Solidity of snapshots for ERC-20 becomes intractable for the more complicated ERC-721 standard.

A controversial issue about ERC-20 and ERC-721 tokens is about who can hold tokens. It is universally accepted that externally owned accounts can hold tokens: they are accounts controlled by humans or external applications, hence their behavior is not fixed. Once they receive a token, the human or the application can decide to keep it or sell it forward. In this case, there is no risk that the token remains stuck. However, the

[1]Perhaps confusingly, the term *token* is used here for the smart contract that tracks coin transfers, for the single coin units and for the category of similar coins.

ERC-20 and ERC-721 standards also allow contracts to hold tokens. This is problematic, since contracts are controlled by their code, which is immutable. If a contract receives a token and its code is not prepared to deal with it, the result is that the token gets stuck forever: the contract will never use it and it will never sell it forward either. There is no solution to this problem, since both externally owned accounts and contracts are represented by the same `address` type in Solidity and are consequently indistinguishable. ERC-721 implementations have tried to limit this problem in a buggy and fragile way, by requiring that contracts receiving ERC-721 tokens must implement an interface `IERC721Receiver`. At least, this acknowledges that the programmer of the contract was aware that the latter could receive tokens. Unfortunately, Solidity has no `instanceof` operator to check for implementation of an interface, because `address` values are unboxed in Ethereum and they carry no dynamic type information [8]. As a consequence, the solution, based on the ERC-165 standard [24], is tricky and fragile and can be circumvented very easily if contracts cheat about the interfaces they implement.

The contributions of this paper are the following:

- a detailed analysis of OpenZeppelin's Solidity implementation of ERC-20 and ERC-721;
- a re-engineered solution that exploits Java features, not applicable in Solidity, to create cleaner implementations of both standards;
- a sound check that contracts holding ERC-721 tokens actually implement `IERC721Receiver`;
- an implementation of a mechanism for efficient snapshots within the Java Virtual Machine (JVM), for both ERC-20 and ERC-721.

The smart contracts that we developed are available in the support library of Takamaka, as non-proprietary, open-source code [27]. They run in the Hotmoka blockchain [14] hence *not* in Ethereum-like blockchains. In particular, the smart contracts of Hotmoka are written in a subset of Java called Takamaka [28,29], hence not in Solidity. Takamaka uses bytecode instrumentation and code annotations (marks starting with `@`, such as `@View` or `@FromContract`) to implement concepts specific to smart contracts. ERC-20 and ERC-721 tokens were not ported previously to Takamaka, hence this is the first version of ERC-20 and ERC-721 for that platform. Moreover, this is the first implementation of ERC-721 tokens that correctly check holders to implement `IERC721Receiver` and that provide snapshots, as far as we know.

Paper structure

Sec. 2 presents the ERC-20 standard and its OpenZeppelin implementation. Sec. 3 presents the ERC-721 standard and its OpenZeppelin implementation. Sec. 4 shows how to perform a code language migration from Solidity to Takamaka and which heuristics have helped in the translation of OpenZeppelin's implementations. Sec. 5 shows an implementation of ERC-20 contracts in Takamaka, with snapshots, that mimics as much as possible the Solidity code structure of OpenZeppelin's implementation, discussing its drawbacks. Sec. 6 shows a more efficient implementation of snapshots, possible in Takamaka but not in Solidity, and that works for both ERC-20 and ERC-721 tokens. Sec. 7 shows, experimentally, that this new implementation is more efficient than what discussed in Sec. 5, inside the JVM. Sec. 8 concludes.

This paper is an extended version of [9]. Compared to that previous version, which was limited to ERC-20 tokens only, the current one adds the ERC-721 tokens as well and the solution to the issue related to the kind of holders allowed to hold tokens, that Solidity tries to solve by using the partial and fragile approach of the ERC-165 standard. Moreover, all sections have been expanded and clarified.

2 ERC-20 and its OpenZeppelin Implementation

The ERC-20 standard [12] defines an interface with nine functions and two *events*, *ie.* immutable marks saved in blockchain to attest some logical turning points. Owners of tokens are *addresses*. In Solidity, these are untyped pointers to *externally owned accounts* (sort of bank accounts controlled by an external app or human) or to *contracts* (objects geared by their code). Although in principle contracts can hold tokens, this could be problematic if their code is not programmed to deal with such tokens. In such a case, the tokens could remain stuck forever, since only the contract can transfer them but the code of the contract

```
contract ERC20 is IERC20 {
  mapping (address => uint) private _balances;
  mapping (address => mapping (address => uint)) private _allowances;
  uint private _totalSupply;
  string private _name;
  string private _symbol;

  constructor(string name_, string symbol_) { _name = name_;  _symbol = symbol_; }

  function totalSupply() public view virtual override returns (uint) {
    return _totalSupply;
  }

  function balanceOf(address owner) public view virtual override returns (uint) {
    return _balances[owner];
  }

  function transfer(address to, uint value) public virtual override {
    _transfer(msg.sender, to, value);
  }

  function allowance(address owner, address delegate) public view virtual override returns (uint) {
    return _allowances[owner][delegate];
  }

  function transferFrom(address owner, address to, uint value) public virtual override {
    _transfer(owner, to, value);
    uint currentAllowance = _allowances[owner][msg.sender];
    require(currentAllowance >= value, "transfer excess");
    _approve(owner, msg.sender, currentAllowance - value);
  }

  function _transfer(address owner, address to, uint value) internal virtual {
    require(owner != address(0), "transfer zero address");
    require(to != address(0), "transfer to zero address");
    _beforeTokenTransfer(owner, to, value);
    uint senderBalance = _balances[owner];
    require(senderBalance >= value, "transfer excess");
    _balances[owner] = senderBalance - value;
    _balances[to] += value;
    emit Transfer(owner, to, value);
  }

  function _mint(address account, uint amount) internal virtual {
    require(account != address(0), "mint to zero address");
    _beforeTokenTransfer(address(0), account, amount);
    _totalSupply += amount;
    _balances[account] += amount;
    emit Transfer(address(0), account, amount);
  }

  function _beforeTokenTransfer(address from, address to, uint amount) internal virtual {
  }
}
```

Fig. 1: A portion of OpenZeppelin's ERC-20 implementation in Solidity. Its full code is available at https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol.

does not deal with token transfers. Therefore, it is normally assumed that only externally owned accounts own tokens, but the implementations of ERC-20 do not check this constraint and do not forbid to transfer tokens to contracts, even inadvertently. Sec. 3 will show that the same problem occurs for ERC-721 tokens, whose implementations have tried to solve the issue in a cumbersome and finally ineffective way.

The functions of the ERC-20 standard are for:

1. Direct transfers: `totalSupply()` yields the integer total amount of tokens in circulation. `balanceOf(address owner)` yields the amount of tokens that `owner` owns. `transfer(address to, uint value)` transfers `value` tokens from the balance of the caller to the balance of `to` (`uint` is an unsigned integer of 256 bits). This function must emit a `Transfer` event.
2. Delegated transfers: `approve(address delegate, uint cap)` allows `delegate` to transfer up to `cap` tokens on behalf of the caller. It must emit an `Approval` event. `transferFrom(address owner, address to, uint value)` transfers `value` tokens from `owner` to `to`, but only if `owner` has `approved` the caller to do so. This function must emit a `Transfer` event. `allowance(address owner, address delegate)` yields the amount of tokens that `delegate` has been `approved` to transfer on behalf of `owner`.
3. Optional info: `name()` yields the name of the tokens. `symbol()` yields the symbol of the tokens. `decimals()` yields the number of decimal digits of the tokens.

The first part of this interface is just the API of a dynamic ledger of token balances. Not surprisingly, OpenZeppelin's code, shown in Fig. 1, stores the user's balance in a field `_balances`[2] of type `mapping (address => uint)`, that binds each address to the amount of tokens it holds, and with an integer field `_totalSupply`, assigned at contract creation time. The second part of the interface allows token owners to delegate, to other participants, the transfer of a capped amount of tokens. OpenZeppelin implements this through a field `_allowances` of type `mapping (address => mapping (address => uint))`: a map from each token owner to another map from each delegate to its allowed cap. The third, optional part is just manifest information about the tokens.

Both `transfer` and `transferFrom` use an internal function `_transfer`, that shifts the tokens from the `owner` to the destination `to`, calling the handler `_beforeTokenTransfer`. This does not do anything by default, but subclasses can redefine it to add extra functionalities to the contract. Function `_transfer` checks, defensively, for missing values (`address(0)`) that might arise from incorrect use of the contract. Function `transferFrom` additionally checks if the `owner` of the tokens has actually delegated `msg.sender` (the caller of the function) to transfer at least `value` tokens on its behalf. This check occurs after the call to `_transfer`, which is fine since Solidity's functions do not commit their side-effects if they fail. The code of `transferFrom` ends with a call to `_approve` (not shown), which reduces the allowance. OpenZeppelin adds a `_mint` function that initializes the total supply of the token: it is internal since it is meant to be called from the constructors of subclasses that deploy actual instances of the contract. This function uses `address(0)` to represent the fact that minted tokens come *from nowhere*.

[2]It is customary in Solidity to start non-`public` properties with underscore.

3 ERC-721 and its OpenZeppelin Implementation

The ERC-721 standard [11] defines an interface with ten functions and three events. As for the ERC-20 standard, token owners can be both externally owned accounts and contracts, but contracts should be avoided, unless they have been explicitly programmed to deal with ERC-721 tokens. We will be back on this issue in a moment.

The functions of the ERC-721 standard are for:

1. Direct transfers: `balanceOf(address owner)` yields the amount of tokens that `owner` owns. `ownerOf(uint tokenId)` yields the owner of the given token, if any. `transferFrom(address from, address to, uint tokenId)` transfers the given token from `from` to `to`. In general, the caller of this function must coincide with `from`, or at least be authorized to transfer the given token on behalf of `from` (see later). This function does not even try to check that `to` is an externally owned account or a contract that will be able to deal with the token. If that is not the case, the token will be transferred to `to` and stuck forever. Because of that, this function is considered to be *unsafe*. This function must emit a `Transfer` event. `safeTransferFrom(address from, address to, uint tokenId)` behaves like `transferFrom`, but

```
contract ERC721 is IERC721 {
    string private _name, _symbol;
    mapping(uint => address) private _owners; // Mapping from token ID to owner address
    mapping(address => uint) private _balances; // Mapping owner address to token count
    mapping(uint => address) private _tokenApprovals; // Mapping from token ID to approved address
    mapping(address => mapping(address => bool)) private _operatorApprovals; // Mapping from owner to
         approved operators

    constructor(string name_, string symbol_) { _name = name_; _symbol = symbol_; }

    function balanceOf(address owner) public view virtual override returns (uint) { return _balances[
         owner]; }
    function ownerOf(uint tokenId) public view virtual override returns (address) { return _owners[
         tokenId]; }
    function name() public view virtual override returns (string) { return _name; }
    function symbol() public view virtual override returns (string) { return _symbol; }

    function approve(address to, uint tokenId) public virtual override {
        address owner = ownerOf(tokenId); require(to != owner, "approval to current owner");
        require(_msgSender() == owner or isApprovedForAll(owner, _msgSender()), "caller is not owner
             nor approved");
        _approve(to, tokenId);
    }

    function getApproved(uint tokenId) public view virtual override returns (address) { return
         _tokenApprovals[tokenId]; }

    function isApprovedForAll(address owner, address operator) public view virtual override returns (
         bool) { return _operatorApprovals[owner][operator]; }

    function transferFrom(address from, address to, uint tokenId) public virtual override {
        require(_isApprovedOrOwner(_msgSender(), tokenId), "caller is not owner nor approved");
        _transfer(from, to, tokenId);
    }

    function _isApprovedOrOwner(address spender, uint tokenId) internal view virtual returns (bool) {
        address owner = ownerOf(tokenId);
        return spender==owner or isApprovedForAll(owner, spender) or getApproved(tokenId) == spender;
    }

    function _transfer(address from, address to, uint tokenId) internal virtual {
        require(ownerOf(tokenId) == from, "transfer from incorrect owner");
        require(to != address(0), "transfer to the zero address");
        _beforeTokenTransfer(from, to, tokenId);
        _approve(address(0), tokenId); // Clear approvals from the previous owner
        _balances[from] -= 1; _balances[to] += 1; _owners[tokenId] = to;
        emit Transfer(from, to, tokenId);
    }

    function _approve(address to, uint tokenId) internal virtual {
        _tokenApprovals[tokenId] = to; emit Approval(ownerOf(tokenId), to, tokenId);
    }
}
```

Fig. 2: A simplified portion of OpenZeppelin's ERC-721 implementation in Solidity. Its full code is available at https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol.

additionally tries to ensure that `to` is an externally owned account or a contract able to deal with the token. In this sense, it is considered to be *safe*.

2. Delegation: function `approve(address delegate, uint tokenId)` allows `delegate` to transfer the given token on behalf of the caller of the function, that must be the owner of the token or itself an authorized operator for the

token. The previous delegate (if any) loses its delegation after this function has been called. This function emits an `Approval` event. The function `setApprovalForAll(address operator, bool approved)` allows `operator` to transfer all tokens owned by the caller of the function (if `approved` is true) or removes that right (if `approved` is false). It is possible to allow more operators per token owner. This function emits an `ApprovalForAll` event. `getApproved(uint tokenId)` yields the delegate for the given token, if any. `isApprovedForAll(address owner, address operator)` determines if `operator` has been authorized to transfer all tokens owned by `owner`.

3. Optional info: `name()` yields the name of the tokens. `symbol()` yields the symbol of the tokens.

OpenZeppelin's implementation of the ERC-721 standard is relatively long, so we only report a portion of the code in Fig. 2. Most information is kept in four maps: `_owners` specifies who is the owner of each given token; `_balances` tells how many tokens each given owner owns; `_tokenApprovals` specifies which delegate has been authorized for each given token (if any); and `_operatorApprovals` yields the set of approved operators for each token owner. Note that `mapping (address => bool)` is actually a set of approved operators: Solidity has no set type, hence sets are encoded as their characteristic map.

Fig. 2 shows that `transferFrom` calls an auxiliary function `_transfer` that decreases the balance of the sender, increases the balance of the receiver and assigns the token to the receiver (`to`). There is no check on the fact that `to` is actually an externally owned account, or a contract, able to deal with the token it receives. This check exists for function `safeTransferFrom` (not shown in Fig. 2). The idea is that contracts ready to receive ERC-721 tokens must be explicitly labeled by their programmer as implementing an `IERC721Receiver` interface, whose only method `onReceive` is called when the contracts receive an ERC-721 token. In general, it would be enough to check that `to instanceof IERC721Receiver` in order to be sure that the programmer was actually expecting the contract to receive ERC-721 tokens and to call `onReceive` in that case. But this is not possible in Solidity, since that language lacks the `instanceof` operator and, in general, it misses any way to check the dynamic type of values. This is not just a missed feature: it is actually impossible to implement such a check, since Ethereum implements data as unboxed values, so that their dynamic type is not available and no `instanceof` operator can ever be implemented. Because of this limitation, Solidity programmers use a very cumbersome technique, based on the ERC-165 standard [24], consisting in adding a function that yields a hash of the signatures of the methods implemented by a contract. By calling that function, it is possible, at run time, to guess the interfaces implemented by a contract. This technique (that we have highly simplified but is much more complicated than what we could express here) is very weak, since contracts are free to cheat and pretend to implement an interface that they actually do not implement. However, it is the best that a programmer can do in Solidity. There is an even weaker approach to cope with this problem. Namely, the ERC-223 token standard [10] requires to cast the token receiver to an interface `IERC223Recipient` and then call its `tokenReceived` method. If the receiver does not implement such method, the transaction fails. This is even weaker than ERC-165 since it makes no attempt to guarantee that the receiver was actually declared to implement `IERC223Recipient`: casts are unchecked in Solidity, they are pure decorations to make the compiler accept the code, but they are not verified at run time.

4 From Solidity to Takamaka

OpenZeppelin's implementations of ERC-20 (Fig. 1) and of ERC-721 (Fig. 2) are only around a few hundred non-comment lines of Solidity. Their code is not particularly complex, although their correctness has never been proved formally, in particular against overflows and underflows, by using formal techniques such as abstract interpretation, as already possible in Java [26]. Bugs are not a theoretical possibility, as the *iToken* incident shows [17] (that, however, did not affect OpenZeppelin's, but another ERC-20 implementation). Bugged contracts cannot be patched and replaced in blockchain, but only redeployed at another address. Their correctness is hence of major importance. Years of exposure to the open-source community and 35 Github contributors give some confidence in OpenZeppelin's code.

Hence, if an ERC-20 or ERC-721 implementation must be provided in another programming language, a literal translation of OpenZeppelin's code is a more reliable starting point than a complete rewriting from scratch.

However, code migration between different programming languages can be tricky, also for relatively simple code. There is no formal way that one can follow to perform such a translation. Therefore, we are not going here to provide any formal proof of equivalence between the original Solidity code and its translation into Takamaka, but only a re-engineering approach and some translation patterns.

Languages might have different semantics for apparently similar constructs or might require different coding styles, for efficiency, which is more often the case if they compile towards different virtual machines. For instance, Vyper [31] and Solidity compile for the same EVM and the translation from Solidity to Vyper [32] is almost immediate. Takamaka compiles for the JVM and the translation from Solidity to Takamaka is more difficult. In many cases, different programming languages have specific solutions that cannot be translated literally: for instance, Java has an `instanceof` operator, hence it is pointless to translate the ERC-165-based technique used in Solidity to allow contracts to hold ERC-721 tokens only if they explicitly declare to implement a specific interface. Just use `instanceof` in Java instead. Nevertheless, our analysis of both languages highlights some translation patterns from Solidity to Takamaka, as shown below.

Visibility modifiers. Solidity's `public` and `private` have direct Java equivalents. Solidity's `internal` corresponds to Java's `protected`, but the latter grants access also to code in the same package of the class `C` where `protected` is used, which is not the case for `internal` (Solidity has no packages). This might be dangerous since an attacker might place a new class in `C`'s package and get access to `C`'s methods that were meant to be `C`'s implementation details. To avoid this scenario, the verifier of Takamaka code, that Hotmoka runs before installing code in blockchain, rejects split packages, *ie.* does not allow two classes in the same package to occur in different jars (Java archives) in the classpath (Java enforces the same constraint only from Java 9). Thanks to this constraint, `internal` can be safely translated into Java's `protected`. Solidity's `external` grants access to a function only to other contracts and, in this sense, it is used to specify the public API of a contract. There is no such visibility notion in Java. However, Takamaka introduces the `@FromContract` annotation, which restricts the callers of a method or constructor to be contracts. Hence `external` can be translated into `public @FromContract`.

The following table summarizes the translation:

Solidity	Takamaka (Java)
`public`	`public`
`private`	`private`
`internal`	`protected`
`external`	`public @FromContract`

`view` modifier. In Solidity, this states that a function (such as `balanceOf` in Fig. 1) has no side-effects and can consequently be executed outside of transactions, in every single node of the blockchain. This translates into Takamaka's `@View` annotation, with the same semantics.

`override` and `virtual` modifiers. Solidity and Java take opposite approaches to non-`private` methods redefinition. Namely, methods can be redefined in Solidity only if they are marked with `virtual` and redefinitions must be marked with `override`. In Java, methods can always be redefined unless they are marked with `final` and redefinitions do not need any special syntactical mark, although the `@Override` annotation has become customary. Consequently, the translation of these modifiers from Solidity to Takamaka is the following:

Solidity	Takamaka (Java)
`virtual f(args) returns T`	`T f(args)`
`override f(args) returns T`	`@Override T f(args)`
`f(args) returns T`	`final T f(args)`

`uint` type. Solidity uses `uint` (short form of `uint256`) to represent unsigned, potentially very large integers (up to $2^{256} - 1$). For instance, ERC-20 implementations use `uint` to represent token balances (Fig. 1). This type suffers from (silent) underflows and overflows. To cope with this problem, Solidity code can use the SafeMath library that provides arithmetic functions with defensive checks against underflows, overflows and divisions by zero. The latest versions of

```java
public class ERC20 extends Contract implements IERC20 {
  private final UnsignedBigInteger ZERO = new UnsignedBigInteger("0");
  private final StorageMap<Contract,UnsignedBigInteger> _balances = new StorageTreeMap<>();
  private final StorageMap<Contract,StorageMap<Contract,UnsignedBigInteger>> _allowances = new
      StorageTreeMap<>();
  private UnsignedBigInteger _totalSupply = ZERO;
  private final String _name, _symbol;

  public ERC20(String name, String symbol) { _name = name; _symbol = symbol; }

  public final @Override @View UnsignedBigInteger totalSupply() {
    return _totalSupply;
  }

  public final @Override @View UnsignedBigInteger balanceOf(Contract owner) {
    return _balances.getOrDefault(owner, ZERO);
  }

  public final @Override @FromContract void transfer(Contract to, UnsignedBigInteger value) {
    _transfer(caller(), to, value);
  }

  public final @Override @View UnsignedBigInteger allowance(Contract owner, Contract delegate) {
    return _allowances.getOrDefault(owner, StorageTreeMap::new).getOrDefault(delegate, ZERO);
  }

  protected final void transferFrom(Contract owner, Contract to, UnsignedBigInteger value) {
    _transfer(caller(), to, value);
    _approve(caller(), owner, allowance(owner, caller()).subtract(value, "transfer excess"));
  }

  protected void _transfer(Contract owner, Contract to, UnsignedBigInteger value) {
    require(owner != null, "transfer from null account");
    require(to != null, "transfer to the null account");
    require(value != null, "value cannot be null");
    _beforeTokenTransfer(owner, to, value);
    _balances.put(owner, balanceOf(owner).subtract(value, "transfer excess"));
    _balances.put(to, balanceOf(to).add(value));
    event(new Transfer(owner, to, value));
  }

  protected void _mint(Contract account, UnsignedBigInteger amount) {
    require(account != null, "mint to the null account");
    require(amount != null, "amount cannot be null");
    _beforeTokenTransfer(null, account, amount);
    _totalSupply = _totalSupply.add(amount);
    _balances.put(account, balanceOf(account).add(amount));
    event(new Transfer(null, account, amount));
  }

  protected void _beforeTokenTransfer(Contract from, Contract to, UnsignedBigInteger amount) {
  }
}
```

Fig. 3: A portion of our ERC-20 implementation in Takamaka. Its full code is available at https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/tokens/ERC20.java.

Solidity implement such checks in the language, natively, at an increased gas cost. Takamaka code can use `UnsignedBigInteger` for that, a wrapper of Java's `BigInteger` class, from Takamaka's support library, whose operations include defensive checks, with the extra advantage that they are unbounded unsigned integers, hence do not suffer from overflows.

`mapping` type. Solidity uses the `mapping` type for maps between values, as for field `_balances` in Fig. 1. These are not data structures, but rather an algorithm that spreads the bindings of the mapping in the key/value store of Ethereum (with an *unlikely* risk of hash collision). Takamaka can use an actual, generic data structure `StorageTreeMap<Key,Value>` instead, an implementation of the interface `StorageMap<Key,Value>`, from Takamaka's support library. Solidity's maps default to 0, hence one must use `getOrDefault(index, 0)` calls on `StorageTreeMap` in Takamaka. If `mapping` is used in Solidity as a trick to implement a set (as in the codomain of `_operatorApproval` in Fig. 2), then in Takamaka it is simpler and more efficient to use a `StorageTreeSet<Value>` instead, an implementation of the interface `StorageSet<Value>`, from Takamaka's support library.

`msg.sender`. This Solidity expression refers to the contract that calls a function. In Takamaka, this corresponds to `caller()` inside a `@FromContract` method.

`address(0)`. This Solidity expression refers to a contract or account at address 0. It is assumed that nobody controls that contract or account. Hence, traditionally, it stands for a missing value or for the sign of missing information in a transaction request. In Takamaka, the same can be achieved with `null`.

Fig. 3 shows our manual translation in Takamaka of the Solidity code for ERC-20 in Fig. 1, by following the heuristics above. The translation is almost literal, with a few exceptions. For instance, function `transferFrom` in Fig. 1 enforces a non-negative allowance through a `require` assertion. In Fig. 3, that same check is moved inside the `subtract` method of the `UnsignedBigInteger` class.

Fig. 4 shows our manual translation in Takamaka of the Solidity code for ERC-721 in Fig. 2. Also, this translation is almost literal. We observe that the `_operatorApprovals` field uses a `StorageSet` in Takamaka, instead of the Solidity trick of using a map to represent a set. Token instances are represented as `BigInteger` in Takamaka, hence they are more general than in Solidity, where they are limited to be `uint`, hence 256 bits only. The `_balances` field uses `BigInteger` to represent the balance of each token holder. This is cheaper than `UnsignedBigInteger` and has been preferred in this case since the code of the contract guarantees such values to be non-negative, hence the run-time checks of `UnsignedBigInteger` are not useful here. Maps in Takamaka cannot use the handy indexing notation of Solidity and do not use `null` to represent a missing binding. This explains why the Takamaka code is sometimes a bit more verbose (see for instance the methods `isApprovedForAll` and `_approve`). In Takamaka, both methods `transferFrom` and `safeTransferFrom` have been collapsed into a single method `transferFrom` that safely checks if the receiver of the token is an externally owned account or a contract that implements `IERC721Receiver`. In this latter case, its `onReceive` method is called. The check on the type of the receiver is sound in Takamaka and doesn't need the tricky and fragile ERC-165 machinery, since Java has an `instanceof` operator that fails if the test is false.

5 Snapshots of ERC-20 Ledgers

OpenZeppelin has subclassed its `ERC20` implementation (Sec. 2) to provide extra functionalities, for instance for tokens that can be (further) minted, burned, capped or paused. Among them, this paper focuses on the `ERC20Snapshot` subclass only, that supports *snapshots*, shown in Fig. 5. Namely, it adds a `_snapshot` function that performs a snapshot of the ledger and yields its progressive identifier (starting at 1). Then it overloads methods `balanceOf` and `totalSupply` from Fig. 1 with variants that receive a snapshot identifier and yield the balance and the total supply *at the time of that snapshot* (Fig. 5). For that, it stores the modification history of an integer variable by using the following data structure:

```
struct Snapshots {
  uint[] ids;
  uint[] values;
}
```

For instance, if a variable v is associated with a `Snapshots` structure with fields `ids={5,8,15}` and `values={6,7,20}`, then the value of v was 20 for snapshot identifiers from 9 to 15; it was 7 for snapshot identifiers from 6 to 8; it was 6 for snapshot identifiers from 1 to 5; for snapshot identifiers after 20, the value of v is v's current value in the ledger. A function `_valueAt` (not shown in Fig. 5) reconstructs the value of a variable at a snapshot. There is one `Snapshots`

```java
public class ERC721 extends Contract implements IERC721 {
  private final StorageMap<BigInteger,Contract> _owners = new StorageTreeMap<>();
  private final StorageMap<Contract,BigInteger> _balances = new StorageTreeMap<>();
  private final StorageMap<BigInteger,Contract> _tokenApprovals = new StorageTreeMap<>();
  private final StorageMap<Contract,StorageSet<Contract>> _operatorApprovals = new StorageTreeMap<>();
  private final String _name, _symbol;

  public ERC721(String name, String symbol) { _name = name; _symbol = symbol; }
  public final @Override @View BigInteger balanceOf(Contract owner) { return _balances.getOrDefault(
      owner, ZERO); }
  public final @Override @View Contract ownerOf(BigInteger tokenId) { return _owners.get(tokenId); }
  public final @View String name() { return _name; }
  public final @View String symbol() { return _symbol; }

  public @Override @FromContract void approve(Contract to, BigInteger tokenId) {
    Contract owner = ownerOf(tokenId); require(owner != to, "approval to current owner");
    Contract caller = caller();
    require(caller == owner or isApprovedForAll(owner, caller), "caller is not owner nor approved");
    _approve(to, tokenId); }

  public @Override @View Contract getApproved(BigInteger tokenId) { return _tokenApprovals.get(tokenId
      ); }

  public @Override @View boolean isApprovedForAll(Contract owner, Contract operator) {
    StorageSet<Contract> approvedForAll = _operatorApprovals.get(owner);
    return approvedForAll != null && approvedForAll.contains(operator); }

  public @Override @FromContract void transferFrom(Contract from, Contract to, BigInteger tokenId) {
    require(_isApprovedOrOwner(caller(), tokenId), "caller is not owner nor approved");
    require(to instanceof ExternallyOwnedAccount or to instanceof IERC721Receiver,
      "transfer destination must be an externally owned account or implement IERC721Receiver");
    _transfer(from, to, tokenId); }

  protected boolean _isApprovedOrOwner(Contract spender, BigInteger tokenId) {
    Contract owner = ownerOf(tokenId);
    return spender == owner or isApprovedForAll(owner, spender) or getApproved(tokenId) == spender;
  }

  protected void _transfer(Contract from, Contract to, BigInteger tokenId) {
    require(ownerOf(tokenId) == from, "transfer from incorrect owner");
    require(to != null, "transfer to null");
    _beforeTokenTransfer(from, to, tokenId); _approve(null, tokenId);
    _balances.put(from, balanceOf(from).subtract(BigInteger.ONE));
    _balances.put(to, balanceOf(to).add(BigInteger.ONE)); _owners.put(tokenId, to);
    if (to instanceof IERC721Receiver) ((IERC721Receiver) to).onReceive(this, from, to, tokenId);
    event(new Transfer(from, to, tokenId)); }

  protected void _approve(Contract to, BigInteger tokenId) {
    if (to == null) _tokenApprovals.remove(to); else _tokenApprovals.put(tokenId, to);
    event(new Approval(owner, to, tokenId)); }
}
```

Fig. 4: A portion of our ERC-721 implementation in Takamaka. Its full code is available at https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/tokens/ERC721.java.

instance for each address that takes part in the token, inside a new field `mapping (address => Snapshots) private _balancesSnapshots`, and for `_totalSupply`, with a new field `Snapshots private _totalSupplySnapshots`. Such structures are allocated and populated whenever a balance gets updated or the total supply changes (the latter situation occurs if mints or burns are allowed). This is achieved by overriding the internal function `_beforeTokenTransfer` (see Fig. 5).

The code of `ERC20Snapshot` (that is very technical and consequently we do not show) has good

```
abstract contract ERC20Snapshot is ERC20 {
  Counters.Counter private _currentSnapshotId;
  struct Snapshots { uint[] ids; uint[] values; }
  mapping (address => Snapshots) private _balancesSnapshots;
  Snapshots private _totalSupplySnapshots;

  function _snapshot() internal virtual returns (uint) {
    _currentSnapshotId.increment();
    uint currentId = _getCurrentSnapshotId();
    // ...emit Snapshot event...
    return currentId;
  }

  function balanceOfAt(address account, uint snapshotId) public view virtual returns (uint) {
    (bool snapshotted, uint value) = _valueAt(snapshotId, _balancesSnapshots[account]);
    return snapshotted ? value : balanceOf(account);
  }

  function totalSupplyAt(uint snapshotId) public view virtual returns (uint) {
    (bool snapshotted, uint value) = _valueAt(snapshotId, _totalSupplySnapshots);
    return snapshotted ? value : totalSupply();
  }

  function _beforeTokenTransfer(address from, address to, uint amount) internal virtual override {
    super._beforeTokenTransfer(from, to, amount);

    if (from == address(0)) {          // mint
      _updateAccountSnapshot(to);
      _updateTotalSupplySnapshot();
    } else if (to == address(0)) {     // burn
      _updateAccountSnapshot(from);
      _updateTotalSupplySnapshot();
    } else {                           // transfer
      _updateAccountSnapshot(from);
      _updateAccountSnapshot(to);
    }
  }

  // _valueAt, _updateAccountSnapshot, _updateTotalSupplySnapshot not shown
}
```

Fig. 5: A portion of OpenZeppelin's Solidity ERC-20 contract with snapshots. Its full code is available at https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC20Snapshot.sol.

computational complexity: it creates snapshots in $O(1)$; since the `ids` fields are sorted, it retrieves balances and total supply at each given snapshot in $O(\log n)$, by binary search, where n is the number of snapshots already performed. Nevertheless, it has some drawbacks:

1. It is complex and tricky. We found it very hard to reach a sufficient trust in its correctness. It is so complicated and specific to ERC-20 that its extension from ERC-20 to ERC-721 tokens has never been done.
2. It induces a significant overhead for the manipulation of the `Snapshots`, also because it needs the extra `_balancesSnapshots` map.
3. All participants pay the overhead of the previous point when they transfer tokens, not just those who create snapshots. That is, if a participant creates a snapshot, then the other participants will later pay the overhead during transfers, even though they were not interested in the snapshot.
4. If a large number of snapshots is generated, arrays `ids` and `values` might become so long that their manipulation exceeds the maximal gas (metering of code execution) allowed for Ethereum transactions, which is the perfect surface for a denial of service attack. That is why function `_snapshot` is internal: subclasses

must implement some security policy to control its access.

We have translated in Takamaka the Solidity code from Fig. 5. The result of this translation is at https://github.com/Hotmoka/hotmoka/blob/master/io-hotmoka-examples/src/main/java/io/hotmoka/examples/tokens/ERC20OZSnapshot.java. It works perfectly but suffers from the same issues highlighted above for its Solidity counterpart. Hence, it is interesting to investigate whether a better implementation of ERC-20 contracts with snapshots exists, at least in Takamaka, which is our target language. Moreover, it is interesting to see if that implementation can also work for ERC-721 tokens, currently missing the snapshot feature in Solidity.

6 An Efficient Algorithm for Snapshots

By looking at OpenZeppelin's code in Fig. 1, it would be convenient to implement the `_snapshot` function in a way completely different from that described in Sec. 5: it should return an actual snapshot (not its identifier), *ie.* a data structure containing an immutable view of the ledger. This new implementation does not increase the length of any array and can be safely `public`. In Solidity-like pseudocode, this would look like in Fig. 6.
However, this code cannot be written in Solidity. The main reason is that Solidity maps cannot be cloned, since they are not data structures, but just an algorithm for distributing key/value pairs in the storage of Ethereum. Solidity maps do not even know their set of keys, whose iteration would at least allow a (very expensive) clone of the map. Moreover, at the time we conducted the analysis and experiments, Solidity functions could not return a `struct` (from Solidity v0.8, ABIEncoderV2 implements that feature).

Fig. 7 shows that the corresponding code can well be written in Takamaka instead. The local inner class `SnapshotImpl` plays the role of the `struct` in Solidity. At creation time, it clones fields `_totalSupply` and `_balances` from the outer `ERC20` object. Class `SnapshotImpl` actually implements a new superinterface `IERC20View` of `IERC20`, that has only the read-only methods of ERC-20, *ie.* `totalSupply` and `balanceOf`. Fig. 8 shows the UML diagram of these interfaces and classes. It shows that there is no special class for ERC-20 contracts with snapshots anymore: all ERC-20 contracts can be snapshotted.

The *magic* of this Java code is that, in Takamaka, an immutable clone of `_balances` is simply `_balances.snapshot()` (the `snapshot` method of `StorageMap`), that runs in $O(1)$. Therefore, the problem is now to understand how the class `StorageTreeMap` and its `snapshot` method work. They exploit the same idea used, for instance, in the Git version control system and in the storage of Ethereum, allowing one to check out their full history of states, by simply swapping a root pointer. They favor the re-creation of immutable data structures instead of updates to mutable data structures. More in detail, in our case class `StorageTreeMap<K,V>` implements red/black trees [25], a special kind of balanced binary search trees that orders keys of type `K` by their *storage reference*, *ie.* a machine-independent pointer to the keys in the memory of the blockchain [28]. Such references are 32 bytes long, *ie.* 256 bits. Since a red/black tree is balanced, the length of a path from root to leaf is 256 at most and get and put operations run in $O(256)$, *ie.* in $O(1)$. Fig. 9(a) shows a `StorageTreeMap<Contract,UnsignedBigInteger> _balances` that implements the mapping with the following insertion order: 81af $\mapsto$ 14, 77b1 $\mapsto$ 18, da89 $\mapsto$ 14, 71a0 $\mapsto$ 19, fa31 $\mapsto$ 35 and 9100 $\mapsto$ 5 (for simplicity, this example assumes that storage references are only two bytes long, *ie.* four hexadecimal digits or 16 bits). We remember that the O notation states a *worst-case* scenario. Namely, the cost for get and put is often smaller than 256 operations, being in general dependent on the number of elements in the tree. We are not stating that get and put cost always exactly 256 operations, which would need the Θ notation instead. What we are stating is that it is never higher than 256, which is the meaning of the O notation. The fact that get and put run in constant worst-case time is made possible by the choice of a particular kind of keys, whose size is fixed a priori. The situation here is similar to the use of Merkle-Patricia tries for implementing the storage of Ethereum, whose get and put operations are considered to run in constant time as well, since their cost increases with the size of the trie but

```
contract ERC20 is IERC20 {
  mapping (address = > uint) private _balances;
  uint private _totalSupply;

  struct SnapshotImpl {
    immutable mapping (address => unit) public balances;
    immutable uint public totalSupply;
  }

  function snapshot() public returns (SnapshotImpl) {
    return (immutable clone of _balances, copy of _totalSupply) }
}
```

Fig. 6: The pseudocode of an alternative implementation of snapshots in Solidity, that its compiler does not accept.

```
public class ERC20 extends Contract implements IERC20 {
  private UnsignedBigInteger _totalSupply = ZERO;
  private final StorageMap<Contract, UnsignedBigInteger> _balances = new StorageTreeMap<>();

  public final IERC20View snapshot() {

    class SnapshotImpl extends Storage implements IERC20View {
      private final UnsignedBigInteger totalSupply = _totalSupply;
      private final StorageMapView<Contract,UnsignedBigInteger> balance = _balances.snapshot();

      public @Override @View UnsignedBigInteger totalSupply() {
        return totalSupply;
      }

      public @Override @View UnsignedBigInteger balanceOf(Contract account) {
        return balances.getOrDefault(account, ZERO);
      }

      // the snapshot of a snapshot is itself
      public @Override @View IERC20View snapshot() {
        return this;
      }
    }

    return new SnapshotImpl();
  }
}
```

Fig. 7: The `snapshot` method added to the code in Fig. 3.

is bounded from above by a constant [4, 6]. Also, in that case, constant worst-case time is possible since keys are Ethereum addresses of fixed size.

Fig. 9(a) shows also the computation of a clone of `_balances`: it is another `StorageTreeMap` whose root is the same root of `_balances`. The independence between `_balances` and its clones is obtained by making the nodes of the trees immutable data structures: destructive updates of the tree actually create new nodes instead of modifying old nodes. For instance, Fig. 9(b) shows an update to `_balances`, that changes the value bound to da89, from 14 to 30. It shows that both nodes for 81af and da89 are recreated (darkened in the figure), and the root of `_balances` is updated. The clone's root remains unchanged instead and points to the old tree. Note that computing a clone means just creating a new root cell that points to the current root of the tree. Hence, a clone is computed in $O(1)$. The idea of creating independent clones of a tree by using immutable nodes and a new root pointer is not new. We have borrowed this idea from the way the Git version control system works internally. Git allows very inexpensive creation of

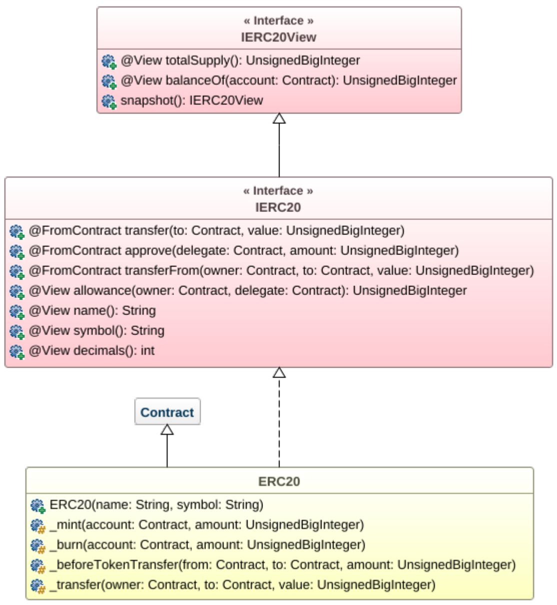
« Interface »
IERC20View
@View totalSupply(): UnsignedBigInteger
@View balanceOf(account: Contract): UnsignedBigInteger
snapshot(): IERC20View
« Interface »
IERC20
@FromContract transfer(to: Contract, value: UnsignedBigInteger)
@FromContract approve(delegate: Contract, amount: UnsignedBigInteger)
@FromContract transferFrom(owner: Contract, to: Contract, value: UnsignedBigInteger)
@View allowance(owner: Contract, delegate: Contract): UnsignedBigInteger
@View name(): String
@View symbol(): String
@View decimals(): int
Contract
ERC20
ERC20(name: String, symbol: String)
_mint(account: Contract, amount: UnsignedBigInteger)
_burn(account: Contract, amount: UnsignedBigInteger)
_beforeTokenTransfer(from: Contract, to: Contract, amount: UnsignedBigInteger)
_transfer(owner: Contract, to: Contract, value: UnsignedBigInteger)

Fig. 8: The UML class diagram of the `IERC20View` and `IERC20` interfaces, implemented by the `ERC20` class. The `IERC20View` interface is a very abstract view of a ledger: it has methods for read-only access and for creating snapshots.

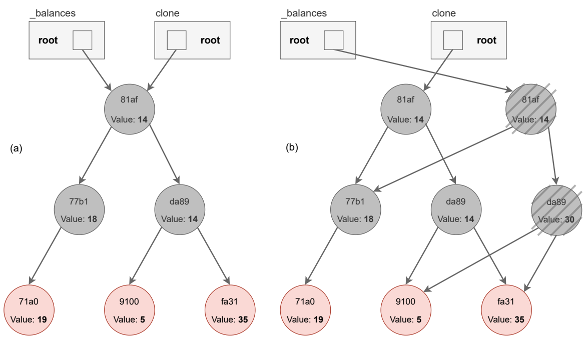

Fig. 9: A red/black tree with immutable nodes, with snapshots in $O(1)$.

branches of a repository in $O(1)$, since a branch is just a reference to the root of the repository at the time of the branch creation.

The code in Fig. 7 has the same asymptotical complexity as OpenZeppelin's ERC-20 contracts with snapshots, but overcomes all its drawbacks reported at the end of Sec. 5:

1. It is simple and intuitive. Class `StorageTreeMap` might look complex but comes with the support library of Takamaka and needn't be reimplemented.
2. It has no overhead because of snapshots and no `_balancesSnapshots` map exists anymore.
3. Who creates a snapshot pays gas. The other participants can transfer coins without paying any overhead because of that snapshot.
4. There are no arrays that grow in size when snapshots are created, hence a denial of service attack is not possible.

Moreover, the same technique can be used to implement a snapshot of an ERC-721 token ledger as well. There is no extra difficulty in comparison with ERC-20 ledgers. The only difference is that the snapshot must be performed for *two* maps this time: for the `_balances` and for the `_owners` maps of the implementation in Fig. 4. The `snapshot` method added to the code in Fig. 4 is shown in Fig. 10. Also in this case, there is an `IERC721View` interface that collects the read-only methods of `IERC721`.

7 Performance Evaluation

This section compares the performance of the literal translation into Takamaka of OpenZeppelin's ERC-20 contracts with snapshots (Sec. 5) against that of our implementation in Takamaka that uses a more efficient snapshot algorithm (Figs. 3 and 7), which we call **Native**, in terms of gas consumed for code execution. Gas is the standard cost measure for smart contracts, since it reflects the actual number of resources (CPU cycles, RAM allocations, storage slots) that each node of a blockchain must consume. However, gas is a low-level, bytecode-specific measure and Solidity and Takamaka use two completely different bytecode languages. Because of that, what we are actually going to compare is OpenZeppelin's ERC-20 contract with snapshots translated in Takamaka (end of Sec. 5), that we call **OpenZeppelin**, against our **Native**. Both are written in Takamaka and both are compiled into Java bytecode. Hence, the comparison gives a measure of the relative efficiency of the two algorithmic solutions, which is what we are looking for. Instead, this is *not* a comparison between OpenZeppelin's Solidity code and our Takamaka code, or more generally between

```
public class ERC721 extends Contract implements IERC721 {

  public final IERC721View snapshot() {

    class SnapshotImpl extends Storage implements IERC721View {
      private final StorageMapView<BigInteger,Contract> owners = _owners.snapshot();
      private final StorageMapView<Contract,BigInteger> balances = _balances.snapshot();

      public @Override @View BigInteger balanceOf(Contract owner) {
        return balances.getOrDefault(owner, ZERO);
      }

      public @Override @View Contract ownerOf(BigInteger tokenId) {
        return owners.get(tokenId);
      }

      // the snapshot of a snapshot is itself
      public @Override @View IERC721View snapshot() {
        return this;
      }
    }

    return new SnapshotImpl();
  }
}
```

Fig. 10: The `snapshot` method added to the code in Fig. 4.

Solidity and Takamaka, that would be meaningless and that we cannot provide, since they compile into distinct bytecode languages, have different gas models and do not allow the same algorithmic solutions: maps can be cloned in Takamaka but not in Solidity.

We have written a JUnit test case that simulates a typical usage scenario for an ERC-20 contract: it creates the contract in blockchain, spreads its tokens among a set of investors (other contracts), play for some time with the ERC-20 contract (we assumed for ten days), performing random token transfers between them, burning some random tokens or minting new random tokens. At the end of each day, it takes a snapshot. The test case is implementation-agnostic: given an implementation of ERC-20 with snapshots (such as **OpenZeppelin** or **Native**), the test case will reproduce the scenario and report the gas consumption. Moreover, in order to be deterministic and fair, the test case uses a fixed seed for random choices. Hence its execution is exactly the same at each run, with both **OpenZeppelin** and **Native**. Similarly, the number and kind of transactions executed by the test case do not change. The interested reader can inspect and run the test case by cloning the repository of Hotmoka (`git clone https://github.com/Hotmoka/hotmoka.git -b erc20-comparison`) and following the instructions in the file `README.txt`. TABLE 11 shows the results. The experiment has been done on an Intel Core i5-8259U machine with 16GB of RAM running Ubuntu Linux 20.04.2. For instance, the test with 1000 investors generates 22372 transactions. With our **Native** contract, it consumes a total of 9718604702 units of gas (CPU+RAM+Storage) and takes 223 seconds. With the **OpenZeppelin** contract, it consumes 17726118750 units of gas (CPU+RAM+Storage, almost twice as **Native**) and takes 344 seconds.

This experiment shows that our **Native** solution with efficient snapshots (Fig. 7) saves gas units (hence money) and reduces the overall time for the execution of the test case. This time reduction is more apparent when there are many investors, as the overhead of **OpenZeppelin**'s solution consequently grows.

8 Conclusion

This paper has shown some patterns for the code migration from Solidity to Takamaka, applied to the specific examples of ERC-20 and ERC-721 contracts. It has shown an improvement of the literal code translation of ERC-20 contracts with snapshots, by using maps with immutable clones, not available in Solidity but implemented in other programming languages. It has shown that the same technique applies to ERC-721 contracts, where snapshots were previously missing. The result has been validated with a test case that shows the reduced gas and time costs of our implementation *wrt.* OpenZeppelin's within the JVM.

The possibility of creating immutable clones of maps in $O(1)$ is useful to simplify the code of other contracts as well, where Solidity must use tricky code instead (as that described in Sec. 5) or recur to events, to mark the historical evolution of data and allow its recovery (with extra gas costs). For instance, we have added snapshots also to *shared entities*, that are used to implement DAOs and the set of validators of a proof-of-stake blockchain [5].

Implementation	Investors	Transfers	Mints	Burns	Txs	CPU	RAM	Storage	Time
Native	100	219	103	99	433	**2326293**	**3589999**	**66336137**	**1.98**
OpenZeppelin	100	219	103	99	433	**4020320**	**5808375**	**130334125**	**2.02**
Native	200	832	205	194	1243	**7636110**	**11627281**	**285906113**	**4.61**
OpenZeppelin	200	832	205	194	1243	**13766079**	**19617649**	**529992972**	**5.86**
Native	300	1776	302	316	2406	**15645862**	**23705622**	**655534336**	**9.82**
OpenZeppelin	300	1776	302	316	2406	**28372984**	**40299922**	**1216043831**	**12.43**
Native	400	3260	383	411	4066	**27995748**	**42184186**	**1272430439**	**16.22**
OpenZeppelin	400	3260	383	411	4066	**51587088**	**72881258**	**2332030574**	**24.02**
Native	500	5170	512	506	6200	**43846203**	**65859592**	**2086138985**	**27.68**
OpenZeppelin	500	5170	512	506	6200	**81836726**	**115260504**	**3801993384**	**43.42**
Native	600	7326	590	599	8527	**61657573**	**92597805**	**3064332633**	**44.20**
OpenZeppelin	600	7326	590	599	8527	**115871428**	**163144629**	**5600364411**	**68.47**
Native	700	10038	738	710	11498	**85337821**	**127833698**	**4327160882**	**68.61**
OpenZeppelin	700	10038	738	710	11498	**160102275**	**225029886**	**7815254989**	**107.20**
Native	800	12896	759	871	14538	**110260986**	**164858626**	**5673424050**	**98.56**
OpenZeppelin	800	12896	759	871	14538	**208781103**	**293035390**	**10340769047**	**160.49**
Native	900	15939	884	901	17736	**137069383**	**204568208**	**7154706461**	**144.09**
OpenZeppelin	900	15939	884	901	17736	**261476515**	**366375520**	**13058660548**	**231.03**
Native	1000	20390	939	1031	22372	**175274120**	**261148704**	**9282181878**	**223.00**
OpenZeppelin	1000	20390	939	1031	22372	**333622702**	**467160332**	**16925335716**	**344.23**

Fig. 11: The result of running our test that simulates ten days of interaction with an ERC-20 contract, performing a snapshot at the end of each day. *Implementation* is the implementation under test: native Takamaka with efficient snapshots or translated from OpenZeppelin into Takamaka. *Investors* is the number of accounts that invest in the ERC-20 contract. *Transfers*, *Mints* and *Burns* are the number of transfer, mint and burn transactions performed during the test, respectively. *Txs* is the total number of transactions performed by the test, including those for the creation and initialization of the ERC-20 contract and for the computation of its snapshots. *CPU*, *RAM* and *Storage* are the gas units consumed for CPU execution, RAM allocation and persistent storage in blockchain, respectively. *Time* is the time for the execution of the test, in seconds.

References

[1] URL: https://www.tiobe.com/tiobe-index/ [accessed: 2021-07-23].
[2] URL: https://pypl.github.io/PYPL.html [accessed: 2021-07-23].
[3] A. M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain.* O'Reilly, 2nd edition, 2017.
[4] A. M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and Dapps.* O'Reilly, 2018.
[5] A. Benini, M. Gambini, S. Migliorini, and F. Spoto. Power and Pitfalls of Generic Smart Contracts. In *Third International Conference on Blockchain Computing and Applications (BCCA'21)*, pages 179–186, Tartu, Estonia, November 2021. IEEE.
[6] V. Buterin. Ethereum Whitepaper, 2013. https://ethereum.org/en/whitepaper.
[7] ConsenSys. Consensys Tokens. https://github.com/ConsenSys/Tokens.
[8] S. Crafa, M. Di Pirro, and E. Zucca. Is Solidity Solid Enough? In A. Bracciali, J. Clark, F. Pintore, P. B. Rønne, and M. Sala, editors, *3rd Wokshop on Trusted Smart Contracts (WTSC'19)*, volume 11599 of *Lecture Notes in Computer Science*, pages 138–153, St. Kitts and Nevis, 2019. Springer.
[9] M. Crosara, L. Olivieri, F. Spoto, and F. Tagliaferro. Re-engineering ERC-20 Smart Contracts with Efficient Snapshots for the Java Virtual Machine. In *Third International Conference on Blockchain Computing and Applications (BCCA'21)*, pages 187–194, Tartu, Estonia, November 2021. IEEE.
[10] Dexaran. ERC223 Token Standard. https://github.com/Dexaran/ERC223-token-standard, 2021.
[11] W. Entrinken, D. Shirley, J. Evans, and N. Sachs. EIP-721: ERC-721 Token Standard, Ethereum Improvement Proposals, no. 721. https://eips.ethereum.org/EIPS/eip-721, 2018.
[12] V. F. and V. Buterin. EIP-20: ERC-20 Token Standard, Ethereum Improvement Proposals, no. 20. https://eips.ethereum.org/EIPS/eip-20, 2017.
[13] P. Freni, E. Ferro, and R. Moncada. Tokenization and Blockchain Tokens Classification: A Morphological Framework. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 1–6, Rennes, France, July 2020. IEEE.
[14] Hotmoka – Blockchain and IoT with Smart Contracts in Java. https://www.hotmoka.io, 2021.
[15] Hyperledger. ERC-20 Token Scenario, 2021. https://github.com/hyperledger/fabric-samples/tree/main/token-erc-20#erc-20-token-scenario.
[16] Hyperledger. ERC-721 Token Scenario, 2021. https://github.com/hyperledger/fabric-samples/tree/main/token-erc-721#erc-721-token-scenario.
[17] K. J. Kistner. iToken Duplication Incident Report. https://bzx.network/blog/incident, September 2020.
[18] M. Koscina, M. Lombard-Platet, and P. Cluchet. PlasticCoin: An ERC20 Implementation on Hyperledger Fabric for Circular Economy and Plastic Reuse. In *IEEE/WIC/ACM International Conference on Web Intelligence - Companion Volume*, pages 223–230. ACM, 2019.
[19] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Available at https://bitcoin.org/bitcoin.pdf, 2008.
[20] L. Oliveira, L. Zavolokina, I. Bauer, and G. Schwabe. To Token or not to Token: Tools for Understanding Blockchain Tokens. In *Proc. of the International Conference on Information Systems - Bridging the Internet of People, Data, and Things, ICIS 2018*, San Francisco, CA, USA, December 2018. Association for Information Systems.
[21] OpenZeppelin. ERC-20 Docs. https://docs.openzeppelin.com/contracts/4.x/api/token/erc20.
[22] OpenZeppelin. ERC-721 Docs. https://docs.openzeppelin.com/contracts/4.x/api/token/erc721.
[23] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. A Formal Verification Tool for Ethereum VM Bytecode. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 912–915, New York, NY, USA, 2018.
[24] C. Reitwießner, N. Johnson, F. Vogelsteller, J. Baylina, K. Feldmeier, and W. Entriken.

EIP-165: ERC-165 Standard Interface Detection: Ethereum Improvement Proposals, no. 165. https://eips.ethereum.org/EIPS/eip-165, 2018.

[25] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, fourth edition, 2014.

[26] F. Spoto. The Julia Static Analyzer for Java. In *Proc. of the 23rd International Symposium on Static Analysis (SAS 2016)*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57, Edinburgh, UK, September 2016. Springer.

[27] F. Spoto. *Hotmoka Github Repository*. GitHub Inc., 2018–2022. Available at https://github.com/Hotmoka/hotmoka.

[28] F. Spoto. A Java Framework for Smart Contracts. In *3rd Wokshop on Trusted Smart Contracts (WTSC'19)*, volume 11599 of *Lecture Notes in Computer Science*, pages 122–137, St. Kitts and Nevis, February 2019. Springer.

[29] F. Spoto. Enforcing Determinism of Java Smart Contracts. In *4th Wokshop on Trusted Smart Contracts (WTSC'20)*, volume 12063 of *Lecture Notes in Computer Science*, pages 568–583, Kota Kinabalu, Malaysia, February 2020. Springer.

[30] D. Tapscott. Token Taxonomy: The Need for Open-Source Standards around Digital Assets, 2020. https://www.blockchainresearchinstitute.org/project/token-taxonomy-the-need-for-open-source-standards-around-digital-assets.

[31] Vyper Documentation. https://vyper.readthedocs.io.

[32] Vyper ERC20 Implementation. https://github.com/vyperlang/vyper/blob/master/examples/tokens/ERC20.vy.