Power and Pitfalls of Generic Smart Contracts

ANDREA BENINI, MAURO GAMBINI, SARA MIGLIORINI, FAUSTO SPOTO
UNIVERSITY OF VERONA, ITALY

Outline

- Smart contract languages
- Limits of Solidity in source code type management
- Use of high-level languages for smart contracts: Java
- Java generics in smart contract implementation
- Vulnerabilities due to the use of Java generics in smart contracts
- Prevent vulnerabilities due to generic type erasure
- Conclusion

Blockchain-based smart contracts

- A **smart contract** is a piece of code that can be **automatically enforced** when a particular event occurs, without the need for a trustworthy intermediary.
- Through smart contracts, platforms like Ethereum can build a sort of **world computer** that persists the same objects inside the memory of all computers composing the blockchain network.

Languages for Smart Contracts

DAML

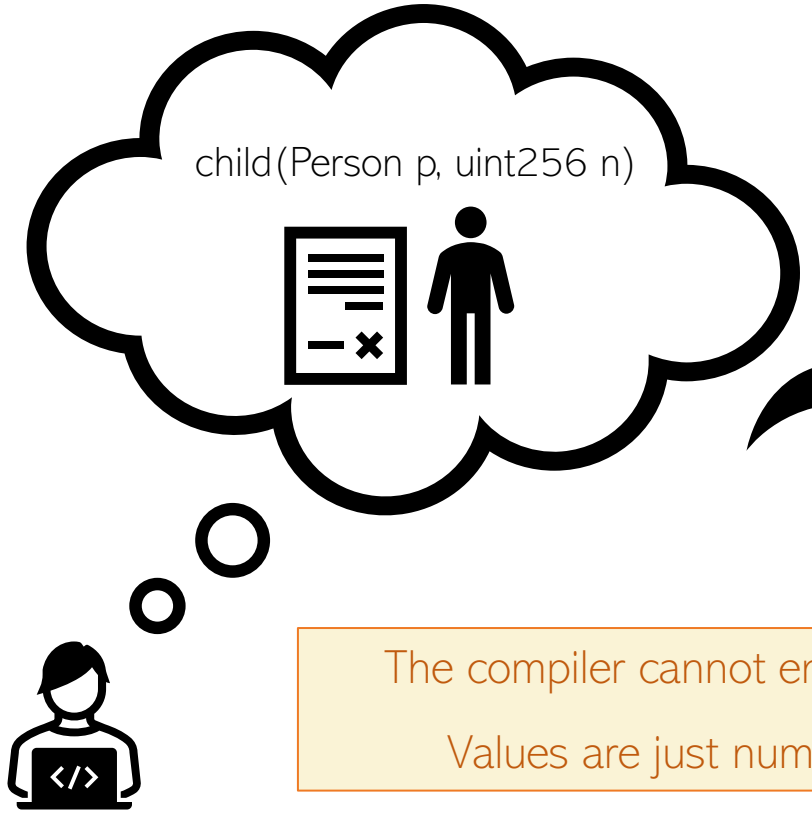
Languages for Smart Contracts

Solidity limitations

Solidity is not strongly typed.

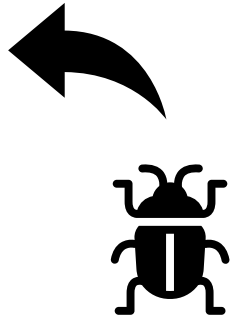
- In Solidity's bytecode, non-primitive values are referenced through a general address type.
- The compiler cannot enforce strong typing by generating defensive type instance checks and casts.
- Values are unboxed: they have no attached type information at run-time, they are just numerical addresses.

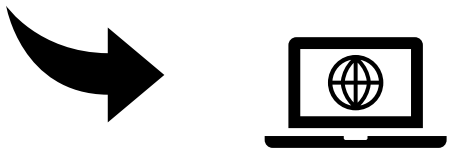
Solidity limitations



At run-time, everything can be passed to p, not just a Person instance

Bytecode publication on chain



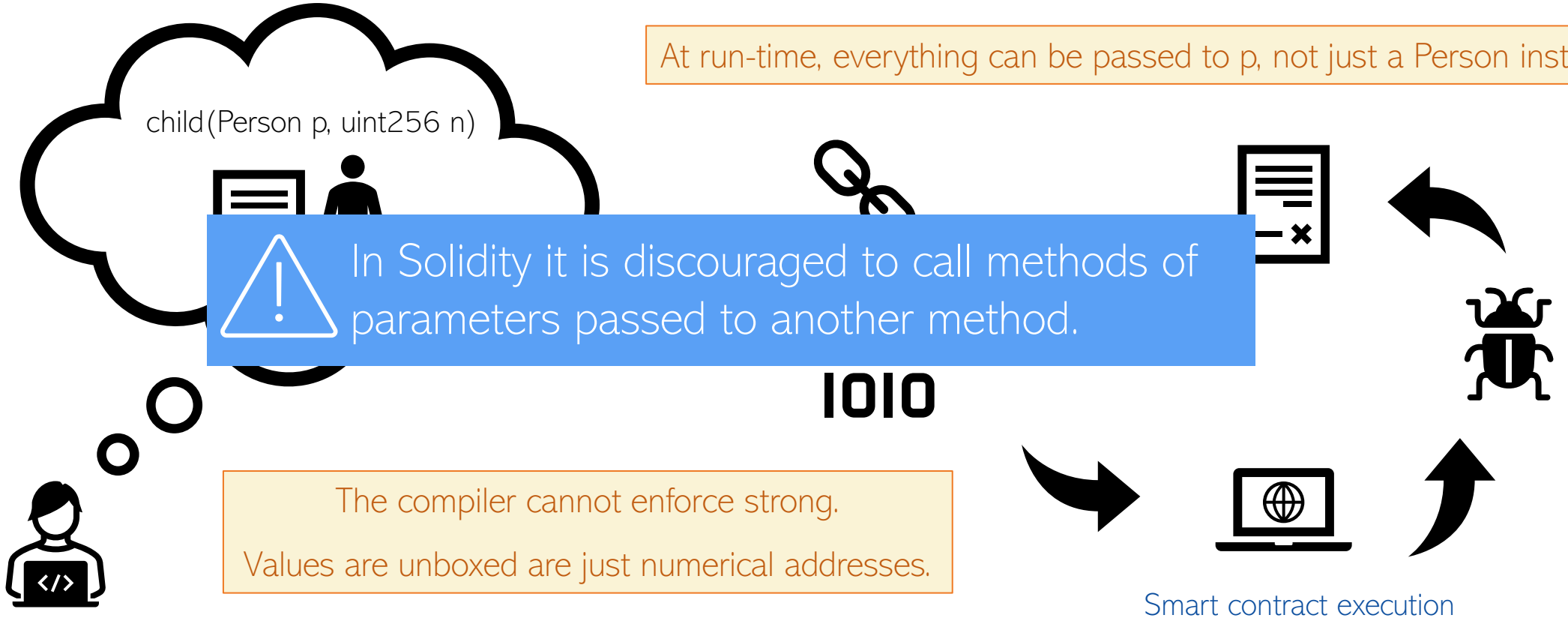


Smart contract execution

The compiler cannot enforce strong typing.

Values are just numerical addresses.

Solidity limitations



At run-time, everything can be passed to p, not just a Person instance

In Solidity it is discouraged to call methods of parameters passed to another method.

The compiler cannot enforce strong.

Values are unboxed are just numerical addresses.

Solidity limitations

Solidity misses many modern language feature, like **generics**.

- Generics allows to personalize the behavior of smart contracts and partially overcome their inherent incompleteness.

Languages for Smart Contracts

SOLIDITY

vyper

Digital Asset

DAML

GO

COSMOS
INTERNET OF BLOCKCHAINS

HYPERLEDGER

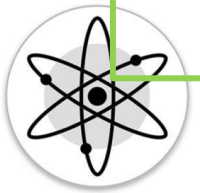
Java™

Hotmoka

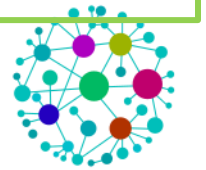
Languages for Smart Contracts

DAML

HYPERLEDGER



CØSMOS
INTERNET OF BLOCKCHAINS



Hotmoka

Languages for Smart Contracts

CØSMOS
INTERNET OF BLOCKCHAINS

HYPERLEDGER

UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Java Generics

- Java generics are strongly typed in the source code.
- Java generics are implemented through the erasure mechanism.
- The erasure mechanism weakens the type information of the compiled code.
- Java generics might have security issues at bytecode level.

The compiler guarantees type correctness of the Java code, not to the bytecode!

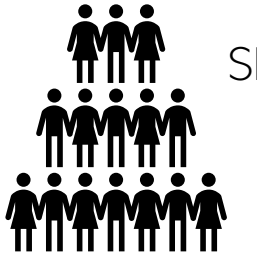
Java Generics

Heterogeneous implementation	The code is duplicated and specialized for each instance of the generic parameters. Safest approach but rarely applied since the code size can dramatically increase. In blockchain, it obliges one to reinstall all instantiations of generic code. C++ templates.
Homogeneous implementation	Only one instance of the code is maintained and shared by all generic instances. Erasure mechanism: a generic parameter is replaced by the upwards bound (e.g. Object) It is less safe, but it requires a smaller consumption of resources. Java, .Net.

Paper contribution

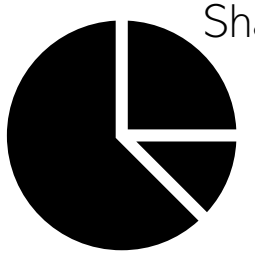
- Demonstrating that a naïve use of Java generics in smart contracts can lead to security vulnerabilities at run-time.
  - Use of a real-world example of smart contract from the Takamaka library.
- Proposing a fix to the issue through a code refactoring that forces the compiler to generate defensive checks.
- Posing the basis for the definition of new smart contract languages, by learning from the weakness of the Java bytecode.

Example: Shared Entities



Shareholders

Shares



Shared Entity

Who are the potential shareholders?

How offers for selling share work?

public interface **SharedEntity** <S extends **PayableContract**, O extends **Offer**<S>>

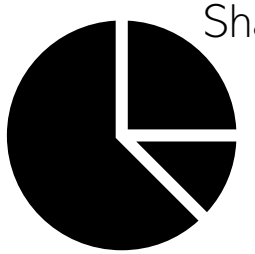
Implementations can subclass **Offer** if they need more specific offers.

- Who is the seller of the shares?
- How many shares are being sold?
- What is the requested price?
- What is the expiration of the offer?

Example: Shared Entities

Shareholders

Shares



Shared Entity

Who are the potential shareholders?

How offers for selling share work?

public interface **SharedEntity** <S extends **PayableContract**, O extends **Offer**<S>>

Consistency of Shareholders: If *se* is a **SharedEntity<S,O>** then the elements contained in the list *se*.getSharehoders() have type **S**.

Example: Shared Entities

```
public class SimpleSharedEntity <S extends PayableContract, O extends Offer<S>>
        extends Contract
        implements SharedEntity<S,O>{

        private final StorageTreeMap<S, BigInteger> shares = new StorageTreeMap<>();
        ...
        @Override @FromContract(PayableContract.class) @Payable
        public void accept( BigInteger amount, S buyer, O offer ){
                require( caller() == buyer, "only the future owner can by the shares" );
                ....
                addShares( buyer, offer.sharesOnSale );
        }
}
```

The map shares holds only values of type S as keys

The consistency property requires the dummy parameter buyer.
Alternative **addShares((S) caller(), offer.sharesOnSale);**
Unchecked cast makes the code not strongly typed!

Example: Shared Entities – Problem

- The absence of unchecked operations guarantees strong typing of Java **source** code.
- What about the bytecode?
- Malicious users might install in blockchain some manually crafted bytecode, not derived from the compilation of the provided Java source code.
- The signature of the method **accept()** declares a parameter buyer of type **S** at source code level, but during the compilation **S** is erased in favor of its superclass **PayableContract**.
- Any **PayableContract** can be passed to the method **accept()** and become shareholder!
- The Consistency of Shareholder property could be easily violated at bytecode level.

Example: Shared Entities – Problem

```
public abstract class AbstractValidators<V extends Validator>
        extends SimpleSharedEntity<V, Offer<V>> {…}

public class TendermintValidators
        extends AbstractValidators<TendermintED25519Validator>{…}
```

The shareholder are the validators of a Tendermint blockchain.

A validator can buy and sell voting power.

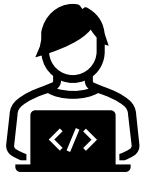
At block creation, Tendermint expects that the validators mine and vote the block validity.

The Consistency of Shareholder property holds at source code level: we need that the shareholders is of type TendermintED25519Validator.

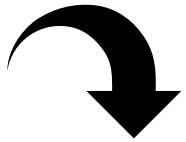
What about the bytecode?

Example: Shared Entities – Problem

public void **accept**(BigInteger amount, S buyer, O offer)



source code



bytecode

public void **accept**(BigInteger amount, PayableContract buyer, O offer)

Any instance of PayableContract can be used as parameter of accept()
Any externally owned account can become a validator of the blockchain!

Example: Shared Entities – Solution

A possible solution is to oblige the compiler to generate a more restrictive signature for the method **accept()**.

```
public class TendermintValidators
        extends AbstractValidators<TendermintED25519Validator>{
        ...
        @Override @FromContract(PayableContract.class) @Payable
        public void accept( BigInteger amount,
                            TendermintED25519Validator buyer,
                            Offer< TendermintED25519Validator> offer ){
            super.accept( amount, buyer, offer );
        }
}
```

The method **accept()** is redefined in the extended class to enforce the correct type for buyer.

Example: Shared Entities – Solution

Bytecode

```
public class TendermintValidators extends AbstractValidators{
    ...
    public void accept(BigInteger, TendermintED25519Validator, offer){
        ...
        invokespecial AbstractValidators.accept( BigInteger, PayableContract, Offer )
        ...
    }

    public void accept(BigInteger, PayableContract, offer){
        ...
        invokevirtual AbstractValidators.accept( BigInteger, TendermintED25519Validator, Offer )
        ...
    }

}
```

Method redefined in the sub-class:
it delegates to the method of the superclass.

Method of the super-class
Bridge method: all calls to the erased signatures are forwarded to the redefined accept method.

Conclusion

- Solidity provides a very limited support in terms of source code type management: all types are translated into the **address** type in the bytecode.
- Generic types can be useful in the definition of smart contracts.
- Generic types can introduce some security risks due to the erasure mechanism.
- We identify a method for preventing such security risk: redefinition of methods with specific types.
- The solution forces the compiler to generate some kinds of checks in the bytecode.
- A smarter compilers might recognize such additional code as unnecessary and remove it.
- New compilers for smart contracts could be engineered, in order to automatically produce such additional checks and guaranteeing a safe type management also in the bytecode.

Thank you for your attention.

Any question?