Vyper

Portland State
Computer Science

Blockchain Programming Languages

Turing complete vs. non-Turing complete

- Not to be confused with the "Turing" test for whether you are human!

- Article on whether "Turing-completeness" is necessary for smart contracts

But first… a Turing machine

- Machine with an infinite amount of RAM that can run a finite program that controls the reading and writing of that RAM
- Program also dictates when to terminate itself

Turing completeness

- Computability on a Turing machine
  - Has the ability to implement any computable function
  - Has the ability to have a function that won't terminate by itself (e.g. infinite loop)
  - Has the ability to use an infinite amount of memory
- Q: Sound like something a smart contract needs?
- Q: Then, why do we have Solidity?

Non-Turing completeness

- Does not support
  - Loops
  - Recursion
  - Goto constructs which are not guaranteed to terminate
  - Constructs that prevent analysis (for security issues)
- Has finite computational and memory resources

Analysis of Ethereum contracts

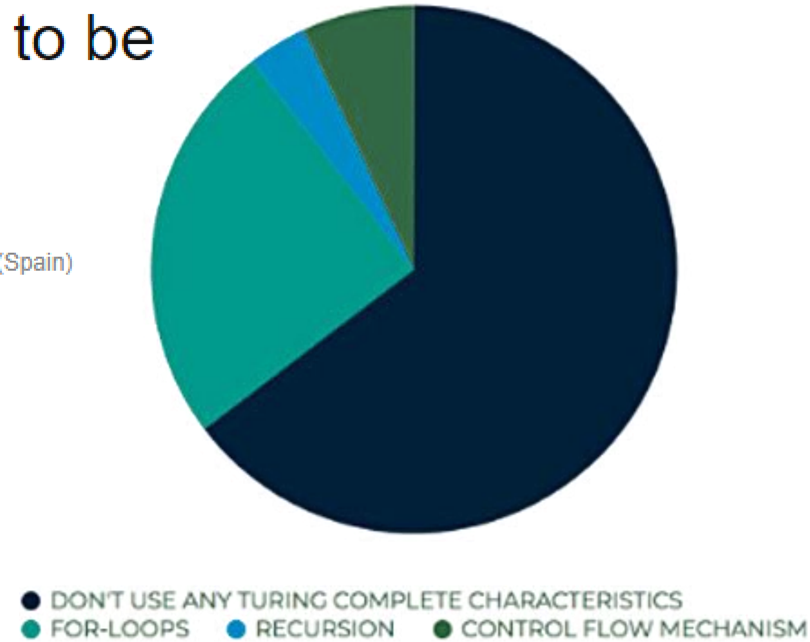
- Study in 3/2019

Do Smart Contract Languages Need to be Turing Complete?

Conference Paper (PDF Available) · March 2019 *with* 250 Reads

Conference: International Congress on Blockchain and Applications. Publisher: Springer, At Ávila (Spain)

- 6.9% use while loops
- 3.6% use recursion
- 24.8% use for loops
  - But not all are unbounded



"Turing-incompleteness is not even that big a limitation; out of all the contract examples we have conceived internally, so far only one required a loop"

Vitalik Buterin

Vyper

Overview

- Non-turing complete Pythonic programming language
  - Language and compiler much simpler
  - Limits functionality to remove common avenues for vulnerabilities
    - Allows one to build secure contracts more easily
  - Simplified programming model to make programs
    - Maximally human-readable
    - Maximally difficult to have misleading code
    - Easy to analyze and audit
  - Compiles to EVM bytecode
- Links
  - On-line interpreter
  - Project page
  - Example contracts

Enforcing simplicity

- Removes modifiers

  `function withdraw() ctf { … }`
  - SI `ctf` modifier defined in a separate file
  - Typically, modifiers are single-condition checks
  - Vyper encourages these to be done as in-line asserts for readability
- Removes class inheritance
  - Similar issue of code across multiple files
  - Inheritance requires knowledge of precedence rules in case of conflicts
    - Inheriting from 2 classes that both implement a particular function call
- Removes in-line assembly
  - Removes the potential for having assembly-level aliases to variables to improve code auditability
- Removes function overloading
  - SI CTF: `withdraw(uint8 amount)` vs `withdraw(uint amount)`
  - Confusion over which version is being executed
- Removes operator overloading
  - Similar issues to above

Avoiding vulnerable patterns

- Removes infinite or arbitrary-length loops
  - Hard to analyze run-time execution for (e.g. gas)
  - Recall DoS contract bricking attacks on while loops in contracts
- Removes recursive calling (e.g. re-entrancy)
  - Prevents one from estimating upper bound on gas consumption for a call
- All integers 256-bit
- Other details
  - `address(this)` in Solidity replaced by `self` in Vyper
  - `address(0)` in Solidity replaced by `ZERO_ADDRESS` in Vyper
  - `require` in Solidity is `assert` in Vyper

Other features

- Strongly and statically typed
- Bounds and overflow checking on array accesses
- Overflow and underflow checks for arithmetic operations
- Decimal fixed point numbers
- Precise upper bounds on gas consumption (execution deterministic)

Language syntax

https://vyper.readthedocs.io

Variables and types

- State variables
  - Stored in contract storage
  - Must have type specified
  - Declare `myStateVariable` as a signed, 128-bit integer

```
myStateVariable: int128
```

- Boolean type
  - Can be either `True` or `False`

```
myBooleanFlag: bool
```

- Integer types
  - Only 256-bit unsigned and 128-bit signed integers

```
myUnsignedInteger: uint256
mySignedInteger: int128
```

- Decimal fixed-point type
  - Values from -2^{127} to $(2^{127}-1)$ at a precision of 10 decimal places

```
myDecimal: decimal
```

- Address type
  - 20-byte Ethereum address

```
myWalletAddress: address
```

  - Contains built-in members (e.g. `myWalletAddress.<member>`)
    - `balance` (returns `wei_value` for address)
    - `codesize` (returns amount of bytes in bytecode for address)
    - `is_contract` (returns whether address is a contract versus a wallet)

- Strings (as in Python)
  - Stored strings with maximum length specified so it can be allocated

```
exampleString: String[100] = "Test String"
```

- Byte Arrays
  - Fixed to 32 bytes (e.g. the size of a `uint256`)

```
codehash: bytes32
```

- Lists
  - Fixed-size array of elements of a specified type
  - Example
    - Declare a list of 3 signed integers, initialize it, then set an element of it

```
myIntegerList: int128[3]
myIntegerList = [10, 11, 12]
myIntegerList[2] = 42
```

- Mappings (hash tables)
  - Example
    - Declare a mapping called `myBalances` that stores values of unit type `decimal` and is keyed by an `address`

```
myBalances: HashMap(address, decimal)
```

    - Set the sender's balance to 10.5

```
myBalances[msg.sender] = 10.5
```

- Structs
  - Declare custom struct data type with attributes and their types
  - Cannot contain mappings

```
struct Bid:
    id: uint256
    deposit: decimal
```

  - Instantiate an instance, initialize it, then change one of its attributes

```
myBid: Bid

myBid = Bid({id: 10, deposit: 10.5})

myBid.deposit = 11.5
```

- Operators
  - All similar to Python and Solidity
  - `true` and `false` booleans
  - `not, and, or, ==, !=` logical operators
  - `<, <=, ==, !=, >=, >` arithmetic comparisons
  - `+, -, *, /, **, %` arithmetic operators
  - Bitwise operators
    - Done as function calls
    - `bitwise_and(), bitwise_not(), bitwise_or(), bitwise_xor(), shift()`
- Built-in functions (selected)
  - `send()` to send a recipient a specified amount of Wei
  - `clear()` to reset datatype to default value
  - `len()` to return the length of a variable
  - `min(), max()` to return smaller or larger of two values
  - `floor(), ceil()` to round a decimal down or up to nearest int

- Defining your own functions
  - Done via Pythonic method via `def` keyword

```
def bid():
  # Check if bidding period is over.
  assert block.timestamp < self.auctionEnd
```

  - Return types specified via `->` operator

```
def returnBalance() -> wei_value:
    return self.balance
```

- Visibility declarations
  - Default setting on everything is private
  - Explicitly denote public variables (via wrapping with `public()`)
  - Explicitly denote public functions (via `@external` decorator)

```
# Keep track of refunded bids so we can follow the withdraw pattern
pendingReturns: public(HashMap(address, uint256))

@external
def withdraw():
    pending_amount: wei_value = self.pendingReturns[msg.sender]
    self.pendingReturns[msg.sender] = 0
    send(msg.sender, pending_amount)
```

- Other function decorators
  - `@internal` (Can only be called within current contract)
  - `@payable` (Can receive Ether)
  - `@nonreentrant` (Cannot be called back into during an external call to stop re-entrancy attacks)
  - `@view` (Does not alter contract state)
- Default function (a.k.a. Fallback function)
  - Function that is executed when receiving a payment only
  - Function that is executed when no function matches
  - Declared via `__default__` syntax

```
@external
@payable
def __default__():
    self.funds = self.funds + msg.value
```

- Constructor function
  - Syntax similar to Python

```
# Setup global variables
beneficiary: address
deadline: public(uint256)
goal: public(uint256)
timelimit: public(uint256)

@public
def __init__(_beneficiary: address, _goal: uint256, _timelimit: uint256):
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    self.goal = _goal
```

- Control flow
  - `if-else` as in Python
  - `for` as in Python (with fixed range)

```
for i in range(len(self.funders)):
    if self.funders[i].value >= 0:
        send(self.funders[i].sender, self.funders[i].value)
        clear(self.funders[i])
```

- Events to send to UI (e.g. web browser)
  - Syntax similar to `structs`
  - Use indexed arguments that can be searched for by listeners
  - Sent via `log` command

```
# Declare event
event Transfer:
    sender: indexed(address)
    receiver: indexed(address)
    value: uint256

# Transfer some tokens from message sender to another address
def transfer(_to : address, _value : uint256) -> bool:
    # Do transfer here

    # Then generate event for listeners to update UI
    log Transfer(msg.sender, _to, _amount)
```

- Within Web3.js front-end

```
var abi = /* abi as generated by the compiler */;
var MyToken = web3.eth.contract(abi);
var myToken = MyToken.at("0x1234...ab67" /* address */);

// watch for changes in the callback
var event = myToken.Transfer(function(error, result) {
    if (!error) {
        var args = result.returnValues;
        console.log('value transferred = ', args._amount)
;
    }
});
```

Fe

Fe

- Vyper spin-off
  - https://decrypt.co/44961/ethereum-blockchain-gets-new-language-called-fe
  - Syntactic properties from Rust typing added
  - Burgdorf: "It's likely that Fe will begin to more closely resemble Rust"
  - Note: Vyper compiler written in Rust with Python bindings

Final projects

Portland State
Computer Science

DApp of your own in Vyper

- Games
- Auctions
- Parking meter
- Stock market trading application
- Ticket application
- See https://codelabs.cs.pdx.edu for specification