PyTeal

Feb 20, 2020

Getting Started

PyTeal is a Python language binding for Algorand Smart Contracts (ASC1s).

Algorand Smart Contracts are implemented using a new language that is stack-based, called Transaction Execution Approval Language (TEAL). This a non-Turing complete language that allows branch forwards but prevents recursive logic to maximize safety and performance.

However, TEAL is essentially an assembly language. With PyTeal, developers can express smart contract logic purely using Python. PyTeal provides high level, functional programming style abstactions over TEAL and does type checking at construction time.

PyTeal **hasn't been security audited**. Use it at your own risk.

CHAPTER 1

Overview

With PyTeal, developers can easily write Algorand Smart Contracts (ASC1s) in Python.

Below is the example of writing *Hashed Time Locked Contract* in Pyteal:

```
from pyteal import *

""" Hash Time Locked Contract
"""
alice = Addr("6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUYDHO57VTCMJOBGY")
bob = Addr("7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M")
secret = Bytes("base32", "23232323232323")

fee_cond = Txn.fee() < Int(1000)

type_cond = Txn.type_enum() == Int(1)

recv_cond = And(Txn.close_remainder_to() == Global.zero_address(),
                Txn.receiver() == alice,
                Sha256(Arg(0)) == secret)

esc_cond = And(Txn.close_remainder_to()  == Global.zero_address(),
               Txn.receiver() == bob,
               Txn.first_valid() > Int(3000))

atomic_swap = And(fee_cond,
                  type_cond,
                  Or(recv_cond, esc_cond))

print(atomic_swap.teal())
```

As shown in this exmaple, the logic of smart contract is expressed using PyTeal expressions constructed in Python. PyTeal overloads Python's arithmetic operators such as < and == (more overloaded operators can be found in *Arithmetic Operators*), allowing Python developers express smart contract logic more naturally.

Last, `teal()` is called to convert an PyTeal expression to a TEAL program, consisting a sequence of TEAL opcodes.

The output of the above example is:

```
txn Fee
int 1000
<
txn TypeEnum
int 1
==
&&
txn CloseRemainderTo
global ZeroAddress
==
txn Receiver
addr 6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUYDHO57VTCMJOBGY
==
&&
arg 0
sha256
byte base32 23232323232323
==
&&
txn CloseRemainderTo
global ZeroAddress
==
txn Receiver
addr 7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M
==
&&
txn FirstValid
int 3000
>
&&
||
&&
```

CHAPTER 2

Install PyTeal

The easiest way of installing PyTeal is using `pip` :

```
$ pip3 install pyteal
```

Alternatively, choose a distribution file, and run

```
$ pip3 install [file name]
```

CHAPTER 3

PyTeal Examples

We showcase some example PyTeal programs:

3.1 Split Payment

Split Payment splits payment between `tmpl_rcv1` and `tmpl_rcv2` on the ratio of `tmpl_ratn / tmpl_ratd`

```
from pyteal import *

"""Split
"""

# template variables
tmpl_fee = Int(1000)
tmpl_rcv1 = Addr("6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUYDHO57VTCMJOBGY")
tmpl_rcv2 = Addr("7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M")
tmpl_own = Addr("5MK5NGBRT5RL6IGUSYDIX5P7TNNZKRVXKT6FGVI6UVK6IZAWTYQGE4RZIQ")
tmpl_ratn  = Int(1)
tmpl_ratd = Int(3)
tmpl_min_pay = Int(1000)
tmpl_timeout = Int(3000)

split_core = (Txn.type_enum() == Int(1)).And(Txn.fee() < tmpl_fee)

split_transfer = And(Gtxn.sender(0) == Gtxn.sender(1),
                     Txn.close_remainder_to() == Global.zero_address(),
                     Gtxn.receiver(0) == tmpl_rcv1,
                     Gtxn.receiver(1) == tmpl_rcv2,
                     Gtxn.amount(0) == ((Gtxn.amount(0) + Gtxn.amount(1)) * tmpl_
↪ratn) / tmpl_ratd,
                     Gtxn.amount(0) == tmpl_min_pay)

split_close = And(Txn.close_remainder_to() == tmpl_own,
```

(continues on next page)

(continued from previous page)

```
                  Txn.receiver() == Global.zero_address(),
                  Txn.first_valid() == tmpl_timeout)

split = And(split_core,
            If(Global.group_size() == Int(2),
               split_transfer,
               split_close))

print(split.teal())
```

3.2 Periodic Payment

Periodic Payment allows some account to execute periodic withdrawal of funds. This PyTeal program creates an contract account that allows `tmpl_rcv` to withdraw TMPL_AMT every `tmpl_period` rounds for `tmpl_dur` after every multiple of `tmpl_period`.

After `tmpl_timeout`, all remaining funds in the escrow are available to `tmpl_rcv`

```
from pyteal import *

tmpl_fee = Int(1000)
tmpl_period = Int(50)
tmpl_dur = Int(5000)
tmpl_x = Bytes("base64", "023sdDE2")
tmpl_amt = Int(2000)
tmpl_rcv = Addr("6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUYDHO57VTCMJOBGY")
tmpl_timeout = Int(30000)

periodic_pay_core = And(Txn.type_enum() == Int(1),
                        Txn.fee() < tmpl_fee,
                        Txn.first_valid() % tmpl_period == Int(0),
                        Txn.last_valid() == tmpl_dur + Txn.first_valid(),
                        Txn.lease() == tmpl_x)


periodic_pay_transfer = And(Txn.close_remainder_to() ==  Global.zero_address(),
                            Txn.receiver() == tmpl_rcv,
                            Txn.amount() == tmpl_amt)

periodic_pay_close = And(Txn.close_remainder_to() == tmpl_rcv,
                         Txn.receiver() == Global.zero_address(),
                         Txn.first_valid() == tmpl_timeout,
                         Txn.amount() == Int(0))

periodic_pay_escrow = periodic_pay_core.And(periodic_pay_transfer.Or(periodic_pay_
↪close))

print(periodic_pay_escrow.teal())
```

CHAPTER 4

Data Types and Constants

A PyTeal expression has one of the following two data types:

- `TealType.uint64`, 64 bit unsigned integer
- `TealType.bytes`, a slice of bytes

For example, all the transaction arguments (e.g. `Arg(0)`) are of type `TealType.bytes`. The first valid round of current transaction (`Txn.first_valid()`) is typed `TealType.uint64`.

`Int(n)` creates a `TealType.uint64` constant, where `n >= 0 and n < 2 ** 64`.

`Bytes(encoding, value)` creates a `TealType.bytes` constant, where `encoding` could be either of the following:

- `"base16"`: its paired `value` needs to be a RFC 4648 base16 encoded string, e.g. `"0xA21212EF"` or `"A21212EF"`
- `"base32"`: its paired `value` needs to be a RFC 4648 base32 encoded string **without padding**, e.g. `"7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M"`
- `"base64"`: its paired `value` needs to be a RFC 4648 base64 encoded string, e.g. `"Zm9vYmE="`

All PyTeal expressions are type checked at construction time, for example, running the following code triggers a `TealTypeError`:

```
Int(0) < Arg(0)
```

Since `<` (overloaded Python operator, see *Arithmetic Operators* for more details) requires both operands of type `TealType.uint64`, while `Arg(0)` is of type `TealType.bytes`.

Converting a value to its corresponding value in the other data type is supported by the following two operators:

- `Itob(n)`: generate a `TealType.bytes` value from a `TealType.uint64` value `n`
- `Btoi(b)`: generate a `TealType.uint64` value from a `TealType.bytes` value `b`

CHAPTER 5

Arithmetic Operators

An arithmetic expression is an expression that results in a `TealType.uint64` value. In PyTeal, arithmetic expressions include integer arithmetics operators and boolean operators. We overloaded all integer arithmetics operator in Python.

Operator	Overloaded	Semantics	Example
`Lt(a, b)`	`<`	*1* if a is less than b, *0* otherwise	`Int(1) < Int(5)`
`Gt(a, b)`	`>`	*1* if a is greater than b, *0* otherwise	`Int(1) > Int(5)`
`Le(a, b)`	`<=`	*1* if a is no greater than b, *0* otherwise	`Int(1) <= Int(5)`
`Ge(a, b)`	`>=`	*1* if a is no less than b, *0* otherwise	`Int(1) >= Int(5)`
`Add(a, b)`	`+`	*a* + *b*, error (panic) if overflow	`Int(1) + Int(5)`
`Minus(a, b)`	`-`	*a* - *b*, error if underflow	`Int(5) - Int(1)`
`Mul(a, b)`	`*`	*a* * *b*, error if overflow	`Int(2) * Int(3)`
`Div(a, b)`	`/`	*a* / *b*, error if devided by zero	`Int(3) / Int(2)`
`Mod(a, b)`	`%`	*a* % *b*, modulo operation	`Int(7) % Int(3)`
`Eq(a, b)`	`==`	*1* if a equals b, *0* otherwise	`Int(7) == Int(7)`
`And(a, b)`		*1* if *a* > *0* && *b* > *0*, *0* otherwise	`And(Int(1), Int(1))`
`Or(a, b)`		*1* if *a* > *0* || *b* > *0*, *0* otherwise	`Or(Int(1), Int(0))`

All these operators takes two `TealType.uint64` values. In addition, `Eq(a, b)` (==) is polymorphic: it also takes two *TealType.bytes* values. For example, `Arg(0) == Arg(1)` is a valid PyTeal expression.

Both `And` and `Or` also support more than 2 arguements:

- `And(a, b, ...)`
- `Or(a, b, ...)`

The associativity and precedence of the overloaded Python arithmatic operators are the same as the original python operators . For example:

- `Int(1) + Int(2) + Int(3)` is equivalent to `Add(Add(Int(1), Int(2)), Int(3))`
- `Int(1) + Int(2) * Int(3)` is equivalent to `Add(Int(1), Mul(Int(2), Int(3)))`

CHAPTER 6

Transaction Fields and Global Parameters

A PyTeal expression can be the value of a field of the current transaction or the value of a global parameter. Blow are the PyTeal expressions that refer to transaction fields:

Operator	Type	Notes
`Txn.sender()`	`TealType.bytes`	32 byte address
`Txn.fee()`	`TealType.uint64`	in microAlgos
`Txn.first_valid()`	`TealType.uint64`	round number
`Txn.first_valid_time()`	`TealType.uint64`	causes program to fail, reserved for future use
`Txn.last_valid()`	`TealType.uint64`	round number
`Txn.note()`	`TealType.bytes`	
`Txn.lease()`	`TealType.bytes`	
`Txn.receiver()`	`TealType.bytes`	32 byte address
`Txn.amount()`	`TealType.uint64`	in microAlgos
`Txn.close_remainder_to()`	`TealType.bytes`	32 byte address
`Txn.vote_pk()`	`TealType.bytes`	32 byte address
`Txn.selection_pk()`	`TealType.bytes`	32 byte address
`Txn.vote_first()`	`TealType.uint64`	
`Txn.vote_last()`	`TealType.uint64`	
`Txn.vote_key_dilution()`	`TealType.uint64`	
`Txn.type()`	`TealType.bytes`	
`Txn.type_enum()`	`TealType.uint64`	see table below
`Txn.xfer_asset()`	`TealType.uint64`	asset ID
`Txn.asset_amount()`	`TealType.uint64`	value in Asset's units
`Txn.asset_sender()`	`TealType.bytes`	32 byte address, causes clawback of all value if sender is the Clawback
`Txn.asset_receiver()`	`TealType.bytes`	32 byte address
`Txn.asset_close_to()`	`TealType.bytes`	32 byte address
`Txn.group_index()`	`TealType.uint64`	position of this transaction within a transaction group
`Txn.tx_id()`	`TealType.bytes`	the computed ID for this transaction, 32 bytes

`Txn.type_enum()` values:

Value	Type String	Description
`Int(0)`	unkown	unknown type, invalid
`Int(1)`	pay	payment
`Int(2)`	keyreg	key registration
`Int(3)`	acfg	asset config
`Int(4)`	axfer	asset transfer
`Int(5)`	afrz	asset freeze

PyTeal expressions that refer to global parameters:

Operator	Type	Notes
`Global.min_txn_fee()`	`TealType.uint64`	in microAlgos
`Global.min_balance()`	`TealType.uint64`	in mircoAlgos
`Global.max_txn_life()`	`TealType.uint64`	number of rounds
`Global.zero_address()`	`TealType.bytes`	32 byte address of all zero bytes
`Global.group_size()`	`TealType.uint64`	number of txns in this atomic transaction group, At least 1

CHAPTER 7

Atomic Transfer

Atomic Transfer are irreducible batch transactions that allow groups of transactions to be submitted at one time. If any of the transactions fail, then all the transactions will fail. PyTeal uses `Gtxn` operator to access transactions in an atomic transfer. For example:

```
Gtxn.sender(1)
```

gets the sender of the second (Atomic Transfers are 0 indexed) transaction in the atomic transaction group.

List of `Gtxn` operators:

Operator	Type	Notes
`Gtxn.sender(n)`	`TealType.bytes`	32 byte address
`Gtxn.fee(n)`	`TealType.uint64`	in microAlgos
`Gtxn.first_valid(n)`	`TealType.uint64`	round number
`Gtxn.first_valid_time(n)`	`TealType.uint64`	causes program to fail, reserved for future use
`Gtxn.last_valid(n)`	`TealType.uint64`	round number
`Gtxn.note(n)`	`TealType.bytes`	
`Gtxn.lease(n)`	`TealType.bytes`	
`Gtxn.receiver(n)`	`TealType.bytes`	32 byte address
`Gtxn.amount(n)`	`TealType.uint64`	in microAlgos
`Gtxn.close_remainder_to(n)`	`TealType.bytes`	32 byte address
`Gtxn.vote_pk(n)`	`TealType.bytes`	32 byte address
`Gtxn.selection_pk(n)`	`TealType.bytes`	32 byte address
`Gtxn.vote_first(n)`	`TealType.uint64`	
`Gtxn.vote_last(n)`	`TealType.uint64`	
`Gtxn.vote_key_dilution(n)`	`TealType.uint64`	
`Gtxn.type(n)`	`TealType.bytes`	
`Gtxn.type_enum(n)`	`TealType.uint64`	see table below
`Gtxn.xfer_asset(n)`	`TealType.uint64`	asset ID
`Gtxn.asset_amount(n)`	`TealType.uint64`	value in Asset's units
`Gtxn.asset_sender(n)`	`TealType.bytes`	32 byte address, causes clawback of all value if sender is the Clawback
`Gtxn.asset_receiver(n)`	`TealType.bytes`	32 byte address
`Gtxn.asset_close_to(n)`	`TealType.bytes`	32 byte address
`Gtxn.group_index(n)`	`TealType.uint64`	position of this transaction within a transaction group
`Gtxn.tx_id(n)`	`TealType.bytes`	the computed ID for this transaction, 32 bytes

where `n >= 0 && n < 16`.

CHAPTER 8

Cryptographic Primitives

Algorand Smart Contracts support 4 cryptographic primitives, including 3 cryptographic hash functions and 1 digital signature verification. Each of these cryptographic primitives is associated with a cost, which is a number indicating its relative performance overhead comparing with simple teal operations such as addition and substraction. All TEAL opcodes except crypto primitives have cost *1*. Below is how you express cryptographic primitives in PyTeal:

Operator	Cost	Description
`Sha256(e)`	*7*	*SHA-256* hash function, produces 32 bytes
`Keccak256(e)`	*26*	*Keccak-256* hash funciton, produces 32 bytes
`Sha512_256(e)`	*9*	*SHA512-256* hash function, produces 32 bytes
`Ed25519verify(d, s, p)`	*1900*	*1* if `s` is the signature of `d` signed by `p` (PK), else *0*

These cryptographic primitives cover the most used ones in blockchains and cryptocurrencies. For example, Bitcoin uses *SHA-256* for creating Bitcoin addresses; Alogrand uses *ed25519* signature scheme for authorization and uses *SHA512-256* hash function for creating contract account addresses from TEAL bytecode.

CHAPTER 9

Conditionals

PyTeal provides two conditional expressions, the simple branching `If` expression, and `Cond` expression for chaining tests.

9.1 Simple Branching: `If`

In an `If` expression,

```
If(test-expr, then-expr, else-expr)
```

the `test-expr` is always evaludated and needs to be typed `TealType.uint64`. If it results in a value greater than *0*, then the `then-expr` is evaluated. Otherwise, `else-expr` is evaluated.

An `If` expression must contain a `then-expr` and an `else-expr`; the later is not optional.

9.2 Chaining Tests: `Cond`

A code:*Cond* expression chians a series of tests to select a result expression. The syntax of *Cond* is:

```
Cond([test-expr body],
     . . . )
```

Each `test-expr` is evaluated in order. If it produces *0*, the paired `body` is ignored, and evaluation proceeds to the next `test-expr`. As soon as a `test-expr` produces a true value (*> 0*), its `body` is evaluated to produce the value for this `Cond` expression. If none of `test-expr` s evaluates to a true value, the `Cond` expression will be evaluated to `err`, a TEAL opcode that causes the runtime panic.

In a `Cond` expression, each `test-expr` needs to be typed `TealType.uint64`. A `body` could be typed either `TealType.uint64` or `TealType.bytes`. However, all `body` s must have the same data type. Otherwise, a `TealTypeError` is triggered.

Example:

```
Cond([Global.group_size() == Int(5), bid],
     [Global.group_size() == Int(4), redeem],
     [Global.group_size() == Int(1), wrapup])
```

This PyTeal code branches on the size of the atomic transaction group.

CHAPTER 10

Using PyTeal With Algorand Python SDK

CHAPTER 11

Indices and tables

- genindex