PREDA Language Specification

Version 2.0

Nov 2022

Table of Contents

- PREDA Language Specification
  - Table of Contents
  - 1 Introduction
  - 2 Data Types
    - 2.1 Type Categories
    - 2.2 Value Types
      - 2.2.1 Built-in Boolean and Integer Types
      - 2.2.2 Built-in Floating Point Types
      - 2.2.3 Enumerations
      - 2.2.4 Other Built-in Value Types
    - 2.3 Reference Types
      - 2.3.1 Built-in Generic Containers
        - 2.3.1.1 Dynamic Array : array
        - 2.3.1.2 Key-Value Map: map
      - 2.3.2 string
      - 2.3.3 token
      - 2.3.4 Structures
    - 2.4 Type Conversion
    - 2.4.1 Implicit Conversion
    - 2.4.2 Explicit Conversion
  - 3 Statements
    - 3.1 Code Block
    - 3.2 Variable Declaration Statement
    - 3.2.1 'auto' Keyword
    - 3.2.2 'const' Keyword
    - 3.3 if Statement
    - 3.4 for, do-while and while Statements
    - 3.5 continue and break Statements
    - 3.6 return Statement
    - 3.7 relay Statement
      - 3.7.1 relay Statement with Lambda Function

- 3.8 deploy Statement
- 4 Expressions
  - 4.1 Literals
    - 4.1.1 Number Literals
    - 4.1.2 Other Literals
  - 4.2 Unary Operators
  - 4.3 Binary Operators
  - 4.4 The Conditional Operator
  - 4.5 Operator Precedence
- 5 Smart Contract
  - 5.1 Contract Definition
- 5.2 State Variables
  - 5.3 Functions
  - 5.4 Sharding Schemes and Scopes
    - 5.4.1 Built-in Scopes
    - 5.4.2 Accessing State Variables and Functions across scopes
  - 5.5 System Reserved Functions
    - 5.5.1 on_deploy()
    - 5.5.2 on_scaleout()
  - 5.6 Execution Context
  - 5.7 Working with Multiple Contracts
    - 5.7.1 Importing Contracts
      - 5.7.1.1 Explicit Import and Implicit Import
    - 5.7.2 Using Types and Scopes Defined in Other Contracts
    - 5.7.3 Calling Functions Defined in Other Contracts
  - 5.8 Interfaces
    - 5.8.1 Defining an Interface
    - 5.8.2 Implementing an Interface
    - 5.8.3 Using Interfaces
- 5.9 Deploy Unnamed contract
  - 5.10 Supply Tokens from a Contract
- 6 Runtime Environment
  - 6.1 Contexts
    - 6.1.1 Transaction Context
    - 6.1.2 Block Context
    - 6.1.3 Debug Context

1 Introduction

Parallel Relay-and-Execution Distributed Architecture (PREDA) is the smart contract language for defining states and transaction logic that are executed on a parallel multi-chain architecture with relayed execution.

2 Data Types

2.1 Type Categories

In PREDA types could be mainly classified into 2 categories: **value type** and **reference type**.

Value types store the data right in the variable, when they are assigned, **the data is copied**.

```
bool a = true;
bool b;
b = a;       // b get a copy of a's value
b = false;   // only b's copy of value is modified, value of a is still "true"
```

Typical value types are numbers, boolean and enumeration types.

Reference types store a reference to the actual data, when they are assigned, **the reference is copied and the data is shared**.

```
struct S{
  bool a;
}
S s0;
s0.a = true;
S s1;
s1 = s0;        // s1 now holds a reference to the same data as s0's
s1.a = false;  // the shared copy of data is modified, hence s0.a is now also
"false"
```

Typical reference types are string, array, map, token and user-defined structures.

2.2 Value Types

2.2.1 Built-in Boolean and Integer Types

PREDA has the following built-in boolean and integer types:

type	size in bytes	value range
bool	1	true / false
int8	1	[-2^7, 2^7-1]
int16	2	[-2^15, 2^15-1]
int32	4	[-2^31, 2^31-1]
int64	8	[-2^63, 2^63-1]

type	size in bytes	value range
int128	16	[-2^127, 2^127-1]
int256	32	[-2^255, 2^255-1]
int512	64	[-2^511, 2^511-1]
uint8	1	[0, 2^8-1]
uint16	2	[0, 2^16-1]
uint32	4	[0, 2^32-1]
uint64	8	[0, 2^64-1]
uint128	16	[0, 2^128-1]
uint256	32	[0, 2^256-1]
uint512	64	[0, 2^512-1]
bigint	varying	[-2^8128 + 1, 2^8128 - 1]

Supported operators

type	symbols	bool	int types	uint types	bigint
assignment	=	X	X	X	X
logical	&&, ||, !	X			
equality comparison	==, !=	X	X	X	X
generic comparison	<, >, <=, >=		X	X	X
increment and decrement	++, --		X	X	X
arithmetic	+, -, *, /, %, +=, -=, *=, /=, %=		X	X	X
negation	-		X		X
bitwise	~, &, ^, |, <<, >>, &=, ^=, |=, <<=, >>=			X	

> Note: when uint types are shifted more than its bit-width, result would be 0.

2.2.2 Built-in Floating Point Types

PREDA has three types of build-in floating point types: **float256**, **float512** and **float1024**, with the corresponding bit-width. They support the following operators

type	symbols	float256 / float512 / float1024
assignment	=	X
logical	&&, ||, !	
equality comparison	==, !=	X
generic comparison	<, >, <=, >=	X
increment and decrement	++, --	
arithmetic	+, -, *, /, %, +=, -=, *=, /=, %=	X (except modulo)
negation	-	X
bitwise	~, &, ^, |, <<, >>, &=, ^=, |=, <<=, >>=	

2.2.3 Enumerations

Enumeration types are defined with a list of enumerators which the value is restricted to.

```
enum MyEnum{
  EnumValueA,
  EnumValueB,
  ...
}
MyEnum e = MyEnum.EnumValueA;    // enumerators must be accessed through the
enumeration type name
```

An enumeration type cannot have more than 65535 enumerators.

2.2.4 Other Built-in Value Types

In addition to the types above, PREDA also provides the following built-in fundamental types:

type	size in bytes	description
blob	36	digest of a data block
hash	32	SHA256 hash value
address	36	an address on the chain

Supported operators

type	symbols	blob	hash	address
assignment	=	X	X	X
logical	&&, ||, !			
equality comparison	==, !=	X	X	X

type	symbols	blob	hash	address
generic comparison	<, >, <=, >=	X	X	X
increment and decrement	++, --			
arithmetic	+, -, *, /, %, +=, -=, *=, /=, %=			
negation	-			
bitwise	~, &, ^, |, <<, >>, &=, ^=, |=, <<=, >>=			

The type **address** has the following built-in member functions:

function	return type	arguments	is const	description
is_user	bool	None	Yes	if the address is a user address
is_delegated	bool	None	Yes	if the address is a delegated address
is_dapp	bool	None	Yes	if the address is a dapp address
is_asset	bool	None	Yes	if the address is an asset address
is_name	bool	None	Yes	if the address is a name address
is_contract	bool	None	Yes	if the address is a contract address
is_custom	bool	None	Yes	if the address is a custom address

2.3 Reference Types

2.3.1 Built-in Generic Containers

type	size in bytes	description
array	varying	a dynamic array of elements of the same type
map	varying	a mapping from keys to values

Supported operators

type	symbols	array	map
assignment	=	X	X
logical	&&, ||, !		

type	symbols	array	map
equality comparison	==, !=		
generic comparison	<, >, <=, >=		
increment and decrement	++, --		
arithmetic	+, -, *, /, %, +=, -=, *=, /=, %=		
negation	-		
bitwise	~, &, ^, |, <<, >>, &=, ^=, |=, <<=, >>=		

2.3.1.1 Dynamic Array : array

An **array** is an array of dynamic size containing elements of the same type. It supports the bracket operator **"[]"**, the index type must be **uint32**. The corresponding value type is the first template parameter given at definition, e.g. **array<int64>**. **array** has the following built-in member functions that could be access through the dot operator **"."**:

function	return type	arguments	is const	description
length	uint32	None	Yes	returns the number of elements in the array
set_length	None	uint32 newLength	No	resize the array to newLength. Existing elements are kept. If newLength is larger than current length, elements with default value of valueType are appended.
push	None	valueType newElement	No	append a new element to the end of the array
pop	None	None	No	remove the last element from the array

2.3.1.2 Key-Value Map: map

A **map** is a mapping from keys to values. It supports the bracket operator **"[]"**. The key type and value type are the first and second template parameter given at definition, e.g. **map<address, string>**. **map** has the following built-in member functions that could be access through the dot operator **"."**:

function	return type	arguments	is const	description
has	bool	keyType key	Yes	if the key exists in the map
erase	None	keyType key	No	remove the element that has the given key, if it exists

2.3.2 string

string holds an array of characters in UTF-8 format. Besides assignment ("="), it also supports equality comparison ("==", "!=") and generic comparison ("<", ">", "<=", ">=") operators. **string** has the following built-in member functions that could be access through the dot operator **"."**:

function	return type	arguments	is const	description
set	None	string str	No	set the content of the string to str.
append	None	string str	No	append str to the end of the current string
length	uint16	None	Yes	get the length of the string

A **string** can have up to 65535 characters.

2.3.3 token

token is the built-in type for carrying certain amount of tokens that can be also stored in contracts states or carried around in transactions. It doesn't support any operator besides assignment "=" (to copy the reference, since it's a reference type). It has the following built-in functions

function	return type	arguments	is const	description
get_id	uint64	None	Yes	returns the id of the token stored in token
get_amount	bigint	None	Yes	returns the amount of token stored in type
transfer	bool	token recipient, bigint transfer_amount	No	transfers a certain amount of token to another token
transfer_all	bool	token recipient	No	transfers all token in the current token to another one

transfer() and transfer_all() would fail if:

1. the token id is 0, or
2. the recipient is already holding some token of a different id, or

In addition, transfer() could fail if:

3. the amount to transfer is negative, or
4. the token doesn't have sufficient amount to transfer.

2.3.4 Structures

Users can define custom **struct** types in their code.

A **struct** is a collection of data grouped together under one name. The members of a struct can be of any built-in type and other user-defined structs.

A struct cannot have member functions.

```
struct MyStruct{
 TypeA memberA;
 TypeB memberB;
 TypeC memberC;
 ...
}
```

The members of a **struct** can be accessed using the dot operator **"."**, for example:

```
// following the definition above
MyStruct myStruct;
myStruct.memberA = ...
```

A **struct** cannot have more than 255 members.

2.4 Type Conversion

2.4.1 Implicit Conversion

PREDA allows implicit conversion from integers to another integer type with wider range, if both types are signed or unsigned, e.g.

```
int16 x;
x = 3;  // Error: integer literals default to type "int32", which has wider range
than int16 . Write x = 3i16 instead.
int8 y;
y = x;  // Error: int8 is not a subset of int16. Write y = int16(x) instead.
uint64 v;
int256 u = v; // Error: there is no implicit conversion between signed / unsigned
integer types. Write u = int256(v) instead.
uint128 w = v; // Ok: Converting from unsigned 64-bit to unsigned 128-bit, which
has a wider range
```

2.4.2 Explicit Conversion

Explicit type conversion is allowed between specific types:

From	To	Explanation
any integer type	any integer type	A runtime check will be performed to verify that the source value is inside range of the target type. Otherwise a runtime error will be generated and execution ends immediately
any integer / float type	string	Convert the number to a readable string

From	To	Explanation
string	address	Convert a string in the format of address literal to a string
string	hash	Calculate the hash value of a string
hash	address	Create a custom type address with value equal to the hash
hash	string	Convert the hash to a string in the same format as a hash literal, but without ":hash" suffix
address	string	Convert the address to a string in the same format as a address literal

A couple examples:

```
int16 x = 1000i16;
uint8 y = uint8(x);    // Runtime error: 1000 is not within value range of uint8,
which is [0, 255]
float256 f = 100.1324;
string s = string(f);  // s is now "100.1324"
address a = vffgwr07yq323axszgxbr2qp9azzbyjjm844s90z8ack63s6hrch683z48:ed25519
s = string(a);         // s is now
"vffgwr07yq323axszgxbr2qp9azzbyjjm844s90z8ack63s6hrch683z48:ed25519"
hash h = 36nwe8x9sig7gb98zkb6gh@qarffhvf6c3ok9433@tz9ne4mb6qi:hash
s = string(h);         // s is now
"36nwe8x9s1g7gb98zkb6hzzqarffhvf6c30k9433ztz9ne4mb6q0"
```

3 Statements

3.1 Code Block

Statements could be grouped in paired curly brackets. Local variables defined in a code block are not visible outside the block.

```
{
  int32 i;
}
// i no longer defined
```

if-, for- and some other statements are always followed by a code block as part of the statement. The body of a function definition can also be regarded as a code block.

3.2 Variable Declaration Statement

PREDA is a statically-typed language and all variables must be assigned a static type at definition.

```
MyType myVariable = initializer;
```

The initializer is optional but its type must match the type used if it's present.

3.2.1 'auto' Keyword

Another way to specify the type is to use the '**auto**' keyword. In this case, an initializer must be provided.

```
auto x = 1u16;   // x is defined as uint16
auto y = "123";  // y is defined as string
auto z;          // compile error
```

The scope of the defined variable is the innermost block that contains it. A variable cannot shadow another one with the same name defined in an outer scope. Instead, it will generate a compile error.

```
{
  int32 i;
}
int32 i;     // the i defined above is no longer available here. Hence a new definition of i is possible.
int32 j;
{
  int32 j;   // re-defining j here will not shadow the definition in the outer scope. Instead, it gets a compile error
}
```

3.2.2 'const' Keyword

A variable can also be declared as a constant using the '**const**' keyword. The data of a constant variable cannot be changed once it's initialized from the initializer in its declaration statement.

```
const int32 i = 3;   // declaring i as constant
i = 4;               // Compile error: a constant value cannot be modified after initialization
```

Constant variables of **reference types** behave a bit differently with the assignment operator '**=**', because it shares the underlying data instead of making a copy. Therefore, for reference types, assigning a constant variable to a non-constant variable would generate a compile error. Otherwise the shared data would be modifiable through the non-constant variable.

```
struct S{
  bool a;
}
const S s0;
S s1;
s1 = s0;        // Compile error: Cannot assign a constant reference type to non-constant.
```

3.3 if Statement

if statement has the following syntax

```
if (condition) {
  // statements when condition is satisfied
}
else {
  // statements when condition is not satisfied
}
```

if and **else** must always be followed by a block, even if there's only one statement in it. The only exception is when **else** is immediately followed by an **if**, so they can be chained together like:

```
if (...) {
}
else if (...) {
}
else if (...) {
}
else {
}
```

3.4 for, do-while and while Statements

for, **do-while** and **while** statements has the following syntax

```
for (init-statement; condition; iteration-expression){
  // loop body
}
```

```
do{
  // loop body, executed at least once
} while (condition);
```

```
while (condition){
  // loop body
}
```

for, **do-while** and **while** must always be followed by a block, even if there's only one statement in it.

3.5 continue and break Statements

continue statement is used to skip the rest of loop body in **for**, **do-while** or **while** statements for the current loop. **break** statement is used to terminate the corresponding loop statement.

3.6 return Statement

return statement is used to end the execution in current function and return to the caller. If the current function has a return type, it must be followed by an expression of the same type.

3.7 relay Statement

A relay statement is similar to a function call, except that the call is asynchronous. The call data is packaged in a so-called "relay transaction" and relayed to the target for execution. The relay statement itself returns immediately.

```
relay@TargetExpression functionName(params);
relay@shards functionName(params);
relay@global functionName(params);
```

There are 3 types of relay targets, as shown in the above example. The first type is the general form, where **TargetExpression** is an expression that evaluates to a type that matches the function's scope.

The second type is a broadcast relay, which uses the 'shards' keyword. It relays to all the non-global shards, like a broadcast. In this case, the called function must be defined in the shard scope.

The last type is a global relay, which uses the 'global' keyword. It relays to the global shard and must be called from a shard- or address- function. The function must be defined in the global scope.

In all types, the function being called must be from the same contract.

3.7.1 relay Statement with Lambda Function

Alternatively, relay statement define a lambda function inline and relay to it.

```
relay@TargetExpression|'shards'|'global' (['const'] parameterType parameterName =
argumentExpression, ...) ['const']{
  // function body
}
```

The format is quite similar to defining a function except that:

1. A function name is not needed. The compiler automatically generates a name for it.
2. The scope of the anonymous function is *TargetExpression*, *shard* or *global*, based on the relay type.
3. For each parameter, an argument must be provided as well.
4. It is possible to use the 'auto' keyword as parameter type. In this case, the type is taken from the corresponding argument expression.

Be aware that the relay function body is executed on the per-address context of TargetAddress, the per-shard context of the target shards, or the global context. It is not to be mixed with the current context on which the relay statement is invoked.

To simplify the code, there's another way to specify a parameter in the relay lambda:

```
relay@someAddress (..., ^identifier, ...){
}
```

This is exactly the same as

```
relay@someAddress (..., auto identifier = identifier, ...){
}
```

3.8 deploy Statement

The **deploy** statement is used to programmatically create a new contract on chain from within a contract.

```
deploy contractName(parameters);
```

For more details check Deploy Unnamed Contract.

4 Expressions

4.1 Literals

4.1.1 Number Literals

Integer literals are by default regarded as of type **int32**.

To force a specific type, append a suffix of "**u**" (for unsigned types) or "**i**" (for signed types), plus a bit-width. e.g., 100u8, 1000i64.

Hex literals can also be followed by the same suffices.

Bigint literals ends with suffix 'ib', without a bit-width.

Floating point literals uses the suffix of 'f', followed by the bit-width.

4.1.2 Other Literals

String literals are characters surrounded by pair of quotation marks "" and are of type string.

Address literals are base32 characters of length 58 followed by :ed25519.

Hash literals are base32 characters of length 52 followed by :hash.

```
"Hello world!"  // string literal
vffgwr07yq323axszgxbr2qp9azzbyjjm844s90z8ack63s6hrch683z48:ed25519 // address
literal
ccnwe8x9sig7gb98zkb6gh@qarffhvf6c3ok9433@tz9ne4mb6qi:hash          // hash
literal
```

4.2 Unary Operators

PREDA supports the following unary operators: increment (**++**), decrement (**--**), negation (**-**), bitwise negation (**~**) and logical negation (**!**).

4.3 Binary Operators

PREDA supports the following binary operators: addition (**+**), subtraction (**-**), multiplication (*****), division (**/**), modulo (**%**), left-shift (**<<**), right-shift (**>>**), bitwise and (**&**), bitwise or (**|**), bitwise exclusive or (**^**).

The above operator can be combine with assignment operator (**=**) to form compound assignment operators, like addition assignment (**+=**), left shift assignment (**<<=**), etc.

Besides, there are also logical binary operators less than(**<**), greater than (**>**), less than or equal(**<=**), greater than or equal(**>=**), equal(**==**), not equal(**!=**), logical and (**&&**) and logical or (**||**)。

4.4 The Conditional Operator

The conditional operator is a ternary operator used in expressions in the format:

```
condition ? expression1 : expression2
```

It takes the result of expression1 if condition is satisfied, otherwise result of expression2. expression1 and expression2 must have the same result type.

4.5 Operator Precedence

The following table lists all operators sorted from higher to lower precedence.

Operator	Symbol	Format	constraints	result type	result type is const
PostIncrement	++	x++	x is not const	/	/
PostDecrement	--	x--	x is not const	/	/
Bracket	[]	x[y]		varies	Yes if x is const
Parentheses	()	x(y, z, ...)	x is a function or type	varies	varies
Dot	.	x.y		varies	Yes if x is const
WithParentheses	()	(x)		same as x	if x is const
PreIncrement	++	++x	x is not const	/	/
PreDecrement	--	--x	x is not const	/	/
UnaryPlus	+	+x		same as x	Yes
UnaryMinus	-	-x		same as x	Yes
LogicalNot	!	!x		same as x	Yes
BitwiseNot	~	~x		same as x	Yes

Operator	Symbol	Format	constraints	result type	result type is const
Multiply	*	x * y	x and y of the same type	same as x	Yes
Divide	/	x / y	x and y of the same type	same as x	Yes
Modulo	%	x % y	x and y of the same type	same as x	Yes
Add	+	x + y	x and y of the same type	same as x	Yes
Subtract	-	x - y	x and y of the same type	same as x	Yes
ShiftLeft	<<	x << y	x and y of the same type	same as x	Yes
ShiftRight	>>	x >> y	x and y of the same type	same as x	Yes
LessThan	<	x < y	x and y are bool	bool	Yes
GreaterThan	>	x > y	x and y are bool	bool	Yes
LessThanOrEqual	<=	x <= y	x and y are bool	bool	Yes
GreaterThanOrEqual	>=	x >= y	x and y are bool	bool	Yes
Equal	==	x == y	x and y are bool	bool	Yes
NotEqual	!=	x != y	x and y are bool	bool	Yes
BitwiseAnd	&	x & y	x and y of the same type	same as x	Yes
BitwiseXor	^	x ^ y	x and y of the same type	same as x	Yes
BitwiseOr	|	x | y	x and y of the same type	same as x	Yes
LogicalAnd	&&	x && y	x and y are bool	bool	Yes

Operator	Symbol	Format	constraints	result type	result type is const
LogicalOr	||	x || y	x and y are bool	bool	Yes
TernaryConditional	? :	x ? y : z	x is bool y and z of the same type	same as y	Yes if either y or z is const
Assignment	=	x = y	x is not const x and y of the same type	/	/
AssignmentAdd	+=	x += y	x is not const x and y of the same type	/	/
AssignmentSubtract	-=	x -= y	x is not const x and y of the same type	/	/
AssignmentMultiply	*=	x *= y	x is not const x and y of the same type	/	/
AssignmentDivide	/=	x /= y	x is not const x and y of the same type	/	/
AssignmentModulo	%=	x %= y	x is not const x and y of the same type	/	/
AssignmentShiftLeft	<<=	x <<= y	x is not const x and y of the same type	/	/
AssignmentShiftRight	>>=	x >>= y	x is not const x and y of the same type	/	/
AssignmentBitwiseAnd	&=	x &= y	x is not const x and y of the same type	/	/
AssignmentBitwiseXor	^=	x ^= y	x is not const x and y of the same type	/	/
AssignmentBitwiseOr	|=	x |= y	x is not const x and y of the same type	/	/

5 Smart Contract

In the PREDA model, all contract deployed on the chain have a unique name in the format of "DAppName.ContractName". The dapp name is given as a parameter when deploying the contract. The contract name is defined in the contract's source code.

5.1 Contract Definition

The main part of PREDA source code is the definition of the contract, which usually looks like:

```
contract MyContract {                // here, the contract name is defined as
"MyContract"
 // contract code here:
 //   enumeration type definition
 //   structure type definition
 //   user-scope definition
 //   interface definition
 //   state variable definition
 //   function definition
}
```

Enumeration and structure definition have already been covered in previous sections, scopes and interfaces will be introduced later in this section.
These definitions don't have to strictly follow the order shown above and can be interleaved, although it's recommended to keep them structured to allow for easier reading.

5.2 State Variables

A state variable is defined similarly way to regular variables, except that it does not have an initializer and an optional scope could be added before the type (discussed later).

```
[scope] TypeName variableName;
```

Similar to regular variables, a state variable declaration statement can also be prefixed by the '**const**' keyword. In this case, it is not actually stored in the contract state storage, but rather used as a compile-time constant. Therefore, `scope` is no longer necessary and an initializer should be provided.

```
'const' TypeName variableName = initializer;
```

Since this variable must be constant at compile-time, the `initializer` can only reference other constant state variables or literals.

```
contract c{
    const hash hhh = ccnwe8x9sig7gb98zkb6gh@qarffhvf6c3ok9433@tz9ne4mb6qi:hash;
 // Ok: initialized as a literal
    const string sss = string(hhh);
 // Ok: Only referencing another constant
    int32 i;
    const int32 j = i;
// Compile error: 'i' is not constant
}
```

5.3 Functions

A function is defined as follows:

```
[scope] 'function' [returnValueType] functionName(parameterList)
[accessSpecifier] ['const'] {
 //function body
}
```

If **returnValueType** is not given, the function does not return any value.

When specified after the parameter list. '**const**' makes the function constant, which means that it cannot modifier any state variable and cannot call other non-const functions (whether in the same or another contract). Constant functions also cannot issue a relay call.

By default, all functions of a contract can only be accessed from within the contract itself. To make a function accessible from other places, access specifiers need to be added to the function definition.

There are two access specifiers available, to enable a function to be invocable from a transaction or another contract:

specifier	accessibility	constraints on function
export	can be invoked by a transaction	Cannot have move-only parameters
public	can be called from another contract	no constrains

A function can also have both specifier so that it's available for both contracts and transactions.

5.4 Sharding Schemes and Scopes

On conventional non-sharding blockchains, each smart contract's state can be seen as a single global instance, which is accessible across the chain. On sharding blockchains, the state of a contract be distributed across multiple shard based on its sharding scheme to achieve parallelism. In the PREDA model, contract developers have the flexibly to freely define how the state of a contract is structured on a shading blockchain by using **scopes**.

Each state variable or function, as shown in the previous section, can include a scope in its definition. In general, the scope of a state variable defines how many copies of that variable are there on the chain and how they are indexed; and the scope of a function defines which state variables it has access to.

5.4.1 Built-in Scopes

PREDA has the following built-in scopes **global**, **shard**, **address**, **uint32**, **uint64**, **uint96**, **uint128**, **uint160**, **uint256** and **uint512**.

The **global** scope is the equivalent of a conventional smart contract, everything defined in the **global** scope has only one single instance globally.

The **shard** scope defines states that has one instance for each shard on the chain.

The **address** scope defines states that has one instance for each valid address on chain.

The **uint** scopes are similar to **address**, except that the defined state has one instance for each valid value of the corresponding uint type.

```
contract MyContract {
  @global uint32 numTotalAccounts;     // only one instance globally
  @shard uint32 numAccountsInShard;    // one instance per shard
  @address uint512 addressBalance;     // one instance per address
  @uint256 string str;                 // one instance for each
                                       // valid value of uint256
}
```

> Note: When a state variable or function is defined without specifying a scope, it defaults to @global.

In the above case, *str* can have up to *2^256 - 1* instances, indexable by a uint256 value. In which shard each of these instances resides, is decided by the underlying blockchain system and transparent to the contract.

5.4.2 Accessing State Variables and Functions across scopes

A state variable defined in any scope other than **global** can have multiple instances stored across the blockchain. What a function is executed, it has access to state variables defined with the same scope but limited to one instance. This instance is indexed by the so-called scope target. Access to variables in another target of the same scope is only possible with an asynchronous relay.

```
contract MyContract {
  @address string s;

  // SetS() has scope address and is always executed with a scope target of type
address
  @address function SetS(string newS) {
    s = newS;                        // the accessed state variable s is from the
current scope target
  }

  @address function SetRemoteS(address otherAddr, string newS) {
    relay@otherAddr SetS(newS);      // other instances of the same scope only
accessible via relay, here it calls SetS() with newS as the scope target
  }
}
```

A function cannot access state variables defined in another scope directly but only via relaying to a function of that scope.

```
contract MyContract {
  @address string s;

  @address function SetS(string newS) {
    s = newS;
  }

  @shard function SetAddressS(address addr, string newS) {
    relay@addr SetS(newS);          // relay to a function in address scope with
newS as the scope target
  }
}
```

global and **shard** scopes are two special cases. Since the **global** scope has only instance, it is readable in any scope but only modifiable inside the **global** scope. Its const functions can also be called directly from any scope.

```
contract C {
  // state variables and function defined without a scope defaults to @global
  int32 i;
  function int32 Get() const {
    return i;
  }
  function Set(int32 newValue) const {
    i = newValue;
  }

  @address int32 j;
  @address function CopyValue() {
    j = i;       // read-only access to i defined in global scope
    j = Get();   // call a const function defined in global scope
  }
  @address function SetGlobalValue(int32 newValue) {
    relay@global Set(newValue);      // non-const global function only accessible
via relay, like functions in other scopes
  }
}
```

State variables in the **shard** scope has one instance in each shard of the blockchain. Any other scope other than the **global** scope has read-write access to the instance in the current shard.

```
contract C {
  @shard bool b;
  @shard function Enable() {
    b = true;
  }
  @address function f() {
    b = !b;         // direct read write access to shard scope instance in the
current shard
    Enable();       // call a function in the shard scope, it is executed in the
context of the current shard
  }
  @global function g() {
    relay@shards Enable();       // global functions are not executed in any
shard, it can only use relay@shards statement to broadcast to all shards
  }
}
```

> Note: Relaying to a specific shard using a shard index is not possible.

5.5 System Reserved Functions

System-reserved functions are a group of special functions with the names reserved by PREDA for special purposes. They don't always have to be defined by a contract. But when they are, the definition must match a certain signature and will be invoked by the system at certain points.

5.5.1 on_deploy()

on_deploy is a global function that is automatically invoked when a contract is deployed. It works like a constructor and can be used to do some initialization of the contract state. The signature is:

```
function on_deploy(parameterList)
```

5.5.2 on_scaleout()

on_scaleout is a shard function that is invoked when a scaleout happens, i.e. when the shard order of the blockchain system is increased by 1 and the total number of shards doubles from 2^(shard_order-1) to 2^shard_order.

On scaleout, each of the old 2^(shard_order-1) shards is forked to two new shards: shard[i] -> shard[i] and shard[i + 2^(shard_order-1)], where 0 <= i < 2^(shard_order-1).
on_scaleout is called 2^shard_order times, once per shard. It can be used to split the old per-shard contract state to the into the two new shards. The signature is:

```
function on_scaleout(bool)
```

The boolean parameter tells whether the current shard is forked in place (when false), or with offset 2^(shard_order-1) (when true). Its value is basically **block.get_shard_index() >= 1u32 <<** (block.get_shard_order() - 1u32).

5.6 Execution Context

During the execution of contract code, the runtime provides with some built-in data and interfaces called the execution context.

An execution context includes:

1. Contract state context, including states variables defined in the contract. If the function is defined as const, the access is read-only, otherwise it's read-write. These variables can be directly accessed using their name.
2. Transaction context, containing metadata of the transaction that directly / indirectly triggered the function call. Typical data in the transaction context are sender address (who authorized the transaction), current address (in the case of a relay call), transaction parameters, etc. These data can be accessed through built-in functions.
3. Block context, containing metadata of the block, in which the transaction is about to be included. Typical data in the block context are shard index, block height, block timestamp, etc. These data can be accessed through built-in functions.

Check the Runtime Environment section for a detailed list of available data in these contexts.

5.7 Working with Multiple Contracts

In PREDA, a contract could interact with other contracts that are already deployed on the chain.

5.7.1 Importing Contracts

To interact with another contract, that contract must first be imported to the current contract.

import DAppName.ContractName [as AliasName];

DAppName and ContractName are the corresponding names assigned when deploying that contract. AliasName is an optional arbitrary identifier to reference it in the current contract. If AliasName is not given, ContractName will be used instead for referencing.

import must be declared before contract definition.

5.7.1.1 Explicit Import and Implicit Import

When a contract is imported by an import directive, it is **explicitly imported**. Besides that, a contract could also be **implicitly imported** if it is indirectly imported, like in the following example.

```
contract ContractA{
}
```

```
import MyDApp.ContractA as A;    // ContractA is explicitly imported
contract ContractB{
}
```

```
import MyDApp.ContractB as B;    // ContractB is explicitly imported
// ContractB imports ContractA, therefore ContractA is implicitly imported here
contract ContractC{
}
```

An implicitly-imported contract doesn't have a user-defined alias and can be reference by its contract name by the compiler. In the above example, MyDApp.ContractA is referenced as ContractA in Contract C. To have a specific alias, it could be explicitly imported again. For example:

```
import MyDApp.ContractB as B;    // ContractB is explicitly imported
import MyDApp.ContractA as A;    // now ContractA is explicitly imported as A,
this overrides the implicit import via contractB
contract ContractC{
}
```

5.7.2 Using Types and Scopes Defined in Other Contracts

After importing a contract, all user-defined types from it could be accessed under the contract alias.

```
contract ContractA{
  struct S{
    int32 i;
  }
  enum E{
    E0,
    E1
  }
}
```

```
import MyDApp.ContractA as A;
contract ContractB{
  @address A.S s;
  @address A.E e;
  @address function f(){
    s.i = 1i32;
    e = A.E.E0;
  }
}
```

5.7.3 Calling Functions Defined in Other Contracts

Similar to user-defined types, public functions defined in other contracts could also be directly referenced via the alias.

```
contract ContractA{
  struct S{
    int32 i;
  }
  enum E{
    E0,
    E1
  }
  // must be defined as public to be callable from other contracts
  @address function f(S s, E e) public{
  }
}
```

```
import MyDApp.ContractA as A;
contract ContractB{
  @address A.S s;
  @address A.E e;
  @address function f(){
    A.f(s, e);      // call public function f from MyDApp.ContractA
  }
}
```

> The basic scope visibility rules hold for cross-contract calls, i.e. each scope can only call function in the same scope, in the shard scope and const functions in the global scope

5.8 Interfaces

Interfaces provide another way to work with multiple contracts. While only known contracts can be imported, interfaces enables interaction with arbitrary contracts that implements it, thus achieving runtime polymorphism.

5.8.1 Defining an Interface

Interfaces are defined at the contract level. Each interface is a set of function definitions with empty bodies. Similar to regular functions, the functions of an interface must also reside in scopes:

```
contract A {
  // defining an interface
  interface Addable {
    // The interface has two functions, each in a different scope
    @address function Add(uint64 value);
    @global function uint64 GetTotal() const;
  }
}
```

The above contract defines an interface *Addable* with 2 functions, each in a different scope. Interfaces can use scopes freely like scopes in contracts, including user-defined scopes and imported scopes from other contracts.

5.8.2 Implementing an Interface

Contracts can choose to implement interfaces using the **implements** keyword at definition. A contract can choose to implement arbitrary number of interfaces, which can either be those defined in the same contract, or imported interfaces from other contracts.

```
import A;
contract B implements A.Addable, Printable {        // use "implements" to
implement interfaces
  interface Printable {
    @global function Print() const;
  }

  uint64 total;
  function uint64 GetTotal() public const {         // GetTotal() for A.Addable
    return total;
  }
```

```
  function Print() public const {                    // Print() for Printable
    __debug.print(globalTotal);
  }
  @address function Add(uint64 value) public {       // Add() for A.Addable
    relay@global (^value) {                          // global scope is read only
in other scopes, must use relay to modify its state
      total += value;
    }
  }
}
```

The above contract implements two interface: *Printable* defined in the contract itself, and *Addable* defined in contract *A* from the previous section.

To implement an interface, a contract must implement all the functions defined in that interface, and the signature of the implemented function must match exactly the definition in the interface, i.e. same function name, parameter list and type, return type, const-ness and scope. In addition, interface function must be implemented as public, since they used for cross-contract calls.

5.8.3 Using Interfaces

When a contract implements an interface, other contracts can interact with it via the interface. For example:

```
import B;                                          // A is implicitly imported via B
contract C {
    @address function test() {
      A.Addable addable = A.Addable(B.__id());  // define a variable of interface
A.Addable and initialize it with contract B's id
      addable.Add(100u64);                         // Calls B.Add() via the
interface
    }
}
```

Here a variable of interface type *A.Addable* is defined. Interface types can be initialized with a contract id. Here, it is initialized with a *B*'s id using the build-in function *__id()* that is automatically generated for each contract. Once a interface variable is initialized, it can be used to call any function defined in the interface and routed to the corresponding implementation in contract B.

With interfaces, a contract can interact with any other contract that implements the interface without knowing them. For example:

```
import A;       // No need to import any other contract other than A, where the
interface is defined
contract Adder {
  @address function Add(A.Addable addable, uint64 value) public {
    addable.Add(value);
  }
}
```

Here the function *Add* accepts an *A.Addable* interface as parameter, which could possibly be initialized by the id of any other contract that implements A.Addable.

> Note: If calling a function on an interface variable that is uninitialized, or initialized with the id of a contract that actually doesn't implement the interface, an error would occur and contract execution will stop immediately.

5.9 Deploy Unnamed Contract

Contracts can be deployed via a transaction or from within a contract. A contract deployed via a transaction is already **named**, that it could be imported using its name, as shown in Importing Contracts. A contract deployed programmatically inside a contract, on the other hand, is **unnamed**, that it cannot be referenced through a name but rather only through its contract id.

An unnamed contract is deployed using the deploy statement:

```
deploy contractName(parameters);
```

Here `contractName` is the alias name of an imported contract and `parameters` should match the argument list of the imported contract's `on_deploy` function. It can be thought of as and create a new contract using the code of `contractName` but with fresh new contract state.
The deploy statement returns a **uint64** value as the contract id of the newly deployed contract, which could be used to reference it using a contract type variable.

```
contract ContractA{
    int32 value;
    function on_deploy(int32 v) {
        value = v;
    }
    function int32 get_value() {
        return value;
    }
}
```

```
import MyDApp.ContractA as A;
contract ContractB {
    function f() {
        uint64 cid0 = deploy A(42);     // deploy a new contract using A's code
        A a0 = A(cid0);                 // Reference the newly created contract
using a contract type variable
        uint64 cid1 = deploy A(100);    // deploy another new contract using A's
code
        A a1 = A(cid1);

        int32 v0 = a0.get_value();      // v0 is 42
        int32 v1 = a1.get_value();      // v1 is 100
        int32 v = A.get_value();        // Referencing the named contract A, the
value of v depends on the argument
                                        // passed by the transaction that deployed
A
    }
}
```

Deploy statements are only allowed inside a non-constant global scope function.

5.10 Supply Tokens from a Contract

A contract can supply its own type of token using built-in functions *__mint* and *__burn* that are automatically generated for each contract.

function	return type	arguments	is const	description
__mint	token	bigint amount	Yes	mint the amount of token
__burn	None	token tk	Yes	burn the tokens stored in tk

The *id* of the token returned by *__mint* is the same as the *id* of the contract. *tk* passed to *__burn* must contain token with the same *id* of the contract, otherwise the function would do nothing.

6 Runtime Environment

6.1 Contexts

- `__block`: the block context
- `__transaction` the transaction context
- `__debug` the debug context (debug only)

6.1.1 Transaction Context

All variables and functions are const.

```
enum transaction_type{
  normal_type,    // invoked by a normal transaction
  relay_type,     // invoked by a relay call
  system_type,    // invoked by the system (the reserved on_xxx() functions)
  scheduled_type // invoked by a scheduled transaction
}
```

Name	Type	Description	N	R	S	S
get_type()	`() -> transaction_type`	get the type of transaction	X	X	X	X
get_self_address()	`() -> address`	The address of `this` (not accessible from shard functions)	X	X	X	X
get_sender()	`() -> address`	Returns the first signer of the transaction or the contract that called the current contract	X	X	X	X
get_timestamp()	`() -> uint64`	Timestamp of the transaction	X	X	X	X

Name	Type	Description	N	R	S	S
get_signers()	`() -> array<address>`	the number of signers	X			
verify_signer()	`(uint32) -> bool`	check the signature of a signer	X			
verify_signer()	`(address) -> bool`	check the signature of a signer	X			
get_originated_shard_index()	`() -> uint32`	Index of the originate shard		X		
get_originated_shard_order()	`() -> uint32`	Order of the originate shard		X		
get_initiator_address()	`() -> address`	Target address of originate transaction		X		

6.1.2 Block Context

All variables and functions are const.

Name	Type	Description
get_height()	`() -> uint64`	Height of the block
get_shard_index()	`() -> uint32`	Index of the shard
get_shard_order()	`() -> uint32`	Order of the shard
get_timestamp()	`() -> uint64`	Timestamp of the block
get_random_number()	`() -> uint64`	get random number based on block metadata
get_miner_address()	`() -> address`	get address of the block miner

6.1.3 Debug Context

All variables and functions are const.

Name	Type	Description
assert()	`(bool) -> void`	if false: raise assertion failure exception and terminate execution
assert()	`(bool, string) -> void`	if false: raise assertion failure exception and terminate execution, display the string in log
print()	`(arbitrary) -> void`	print informational message