Technical Details of the Solidity compiler

Current Developments and Future Plans for Solidity

Daniel

Solidity Team, Ethereum Foundation
contract Example {
    address immutable owner;

    constructor() {
        owner = msg.sender;
    }

    function withdraw(uint256 amount) public {
        require(msg.sender == owner); // Requires no sload!
        ...
    }

    ...
}
Immutables

What about immutable arrays and structs?

```solidity
IERC20[] immutable _tokens;
constructor(IERC20[] memory tokens) {
    _tokens.length = tokens.length;
    for (uint256 i = 0; i < tokens.length; ++i)
        _tokens[i] = tokens[i];
}
function getTokenIndex(IERC20 token) public view returns (uint256) {
    for (uint256 i = 0; i < _tokens.length; ++i)
        if (_tokens[i] == token)
            return i;
    revert("invalid token");
}
```
Immutables

What about immutable arrays and structs?

```solidity
IERC20[] immutable _tokens;
constructor(IERC20[] memory tokens) {
    _tokens.length = tokens.length;
    for (uint256 i = 0; i < tokens.length; ++i)
        _tokens[i] = tokens[i]; // Is this still "immutable"?
}

function getTokenIndex(IERC20 token) public view returns (uint256) {
    for (uint256 i = 0; i < _tokens.length; ++i)
        if (_tokens[i] == token)
            return i;
    revert("invalid token");
}
```
contract C {
  uint[] immutable x;
  uint[] immutable y;

  constructor() {
  ...
  }

  function f() public {
    uint[] immutable z = x;
    z = y; // z is “immutable”? Really?
    ...
  }
}
contract Example {
    address immutable owner;
    constructor () {
        owner = msg.sender; // `owner` is a memory variable
    }
    function withdraw (uint256 amount) public {
        // `owner` is a literal filled into the deployed code
        require(msg.sender == owner);
        ... 
    }
    ... 
}
Filling literal values into the bytecode is not an option for dynamic types.

Instead we need to rely on `codecopy`.

Why not pass dynamic immutables around by reference?

Why not slice them?

Immutables will become a new data location!
Immutables -> code data location

```solidity
bytes code _data;

// In the constructor `_data` behaves similar to a `memory` variable.
constructor() { _data = new bytes(32); _data[0] = ...; ... }

// In runtime code `_data` behaves similar to a `calldata` reference.
function f(bytes code partOfData) { ... }
function g(bool which) {
    bytes code firstHalfOfData = _data[0:_data.length/2];
    bytes code secondHalfOfData = _data[_data.length/2:_data.length];
    f(which ? firstHalfOfData : secondHalfOfData);
}
```

- A bit tricky to type-check (creation + runtime pass).
- Still needs some gas considerations (no `codeload` opcode).
User-Defined Value Types

// Introduce a new type without any properties or implicit conversions, based on an underlying built-in value type. T.wrap / T.unwrap to convert from/to underlying type. Uses the underlying type in the ABI.

type Fixed is uint128;

uint128 constant FixedMultiplier = 10**18;

function uintToFixed(uint128 a) pure returns (Fixed) {
    return Fixed.wrap(a * FixedMultiplier);
}

...
User-Defined Value Types

...  
// Add functions to the new type globally (only for types defined in same file).
// No need to repeat "using".
using {add, mul} for Fixed global;

function add(Fixed a, Fixed b) pure returns (Fixed) {
    return Fixed.wrap(Fixed.unwrap(a) + Fixed.unwrap(b));
}

function mul(Fixed a, Fixed b) pure returns (Fixed) {
    uint result = (uint(Fixed.unwrap(a)) * uint(Fixed.unwrap(b))) /
        uint(FixedMultiplier);
    require(result <= type(uint128).max);
    return Fixed.wrap(uint128(result));
}

// In a different file:
function square(Fixed x) pure returns (Fixed) {
    return x.mul(x);
}

Soon: User-Defined Operators and Literals

using {add as +, mul as *} for Fixed global;
function square(Fixed value) pure returns (Fixed) {
    return value * value;
}

// special "literal suffix" function
function f(uint128 val, uint8 exp) pure returns (Fixed) {
    return Fixed.wrap(val * 10**(18 - exp));
}

function addVAT(Fixed value) pure returns (Fixed) {
    // Same as mul(value, f(115, 2))
    return value * 1.15 f;
}
User-Defined Data Types

- So far only user-defined **value** types.
- What about arrays, structs, dynamic types?
- Algebraic data types?
- What about data locations?
- Option: Tie data locations to types instead of variables.

```solidity
type EncapsulatedMemoryArray is uint256[] memory;
type EncapsulatedCalldataStruct is S calldata;
```
User-Defined Data Types

- All this, user-defined container types, etc., increases the need for generics.
- Can currently built-in types be “user”-defined instead?
Standard Library

- Move manually hard-coded compiler implementations to user-code.
- Ship as a compiler-integrated standard library.

```solidity
pragma stdlib;
import {addmod} from "std/math.sol";
... w = addmod(x, y, z); ...

// File: "std/math.sol"
function addmod(uint x, uint y, uint modulus) pure returns (uint result) {
    require(modulus != 0);
    assembly { result := addmod(x, y, modulus) }
}
```
Standard Library

- Limited by being restricted to monomorphic functions.
- Full potential only unleashed with generics.
- Not only move builtin functions, but also builtin types. (then defined as “user”-defined data types in the standard library)
- End goal:

Reduce solidity to a small, simple core language with most of the current feature set implemented in a Solidity-written standard library.
Generics

- Logically grounded type system with products, sums, function types (cartesian closed category).
- System of type classes (Haskell), resp. traits (Rust).
- Polymorphic functions and ad-hoc polymorphism using type classes.
- General algebraic data types.
- Compile-time constant expression evaluation.
- Maybe linear types (basis for Rust’s borrow checker).
Generics

```cpp
struct ResizableArray<T> {
  uint size;
  T[] data;
}

function append(ResizableArray<T> array, T value) {
  if (array.size >= array.data.length) {
    // resize
    T[] newData = new T[](array.data.length * 2);
    for (uint i = 0; i < array.data.length; ++i)
      newData[i] = array.data[i];
    array.data = newData;
  }
  array.data[array.size++] = value;
}

function index_access(ResizableArray<T> array, uint256 index) {
  require(index < array.size);
  return array.data[index];
}

using {append, index_access as []} for ResizableArray;
```
Generics

struct ResizableArray<T::CanLiveInMemory> {
    uint size;
    T[] memory data;
}

function append(ResizableArray<T> memory array, T value) {
    if (array.size >= array.data.length) {
        // resize
        T[] memory newData = new T[](array.data.length * 2);
        for (uint i = 0; i < array.data.length; ++i)
            newData[i] = array.data[i];
        array.data = newData;
    }
    array.data[array.size++] = value;
}

function index_access(ResizableArray<T> memory array, uint256 index) {
    require(index < array.size);
    return array.data[index];
}

using {append, index_access as []} for ResizableArray;
// Memory array as pointer to memory offset storing the size followed by the data.

type<T> T[] memory is StackSlot;

function index_access(T[] memory x, uint256 index) returns (T result)
{
    StackSlot mptr = (T[] memory).unwrap(x);
    uint256 offset = 32 + index * 32;
    assembly ("memory-safe") {
        let size := mload(mptr)
        if iszero(lt(index, size)) { revert(0, 0) /* out of bounds */ }
        result := mload(add(mptr, offset))
    }
}

using {index_access as []} for T[] memory global;
// Memory array as tuple of memory offset of data and size (slicable!).

type<T> T[] memory is (StackSlot, StackSlot);

function index_access(T[] memory x, uint256 index) returns (T[] result) {
    (StackSlot mptr, StackSlot size) = (T[] memory).unwrap(x);
    uint256 offset = index * 32;
    assembly ("memory-safe") {
        if iszero(lt(index, size)) { revert(0, 0) /* out of bounds */ }
        result := mload(add(mptr, offset))
    }
}

using {index_access as []} for T[] memory global;
Generics

type\<N\> \ uint\<N\> \ = \ StackSlot;
using \{ add\<N\> \ as \ +, \ mul\<N\> \ as \ *, \ ... \} \ for \ uint\<N\>;
type\_alias \ uint8 \ = \ uint\<8\>;
type\_alias \ uint16 \ = \ uint\<16\>;
...
type\_alias \ uint256 \ = \ uint\<256\>;
Generics

- Still in early design phase.
- Several conceptual iterations away from a final semantic design.
- No concrete syntax yet.
- Tradeoff between generality and fixed semantic properties usable for optimization.
● Allow more precomputation.
  (code data location; compile-time constant expression evaluation)
● Make the language extensible and self-defining.
  (improved user-defined data types; standard library; generics)

also:
● Finally stop wasting memory.
  (Life-time analysis, potentially on the Solidity instead of the Yul level)
● Move completely towards via-IR codegen.
  (increase performance; more debugging data for better tooling support)
Thank you!

To participate in language design or for any feedback reach out to us:

- [https://docs.soliditylang.org/en/latest/contributing.html](https://docs.soliditylang.org/en/latest/contributing.html)
- Forum ([https://forum.soliditylang.org/](https://forum.soliditylang.org/))
- Chat ([https://matrix.to/#/#ethereum_solidity-dev:gitler.im](https://matrix.to/#/#ethereum_solidity-dev:gitler.im))

@solidity_lang