Introducing Core Solidity

Argot Collective

Solidity, today

1. Most used smart contract language
2. Successfully secures billions of dollars
3. However, it has some limitations:
   - Type system lacks expressiveness
   - Does not support compile time eval.
4. Extending the current impl. is **hard**.

What is Core Solidity?

1. A complete rebuild of the language type system and compiler pipeline.
2. Introduce new features
3. A formal semantics to enable analysis and verification.

Core Solidity New features

1. Generics and type classes (traits)
2. Algebraic data types and pattern matching
3. Type inference
4. High-order and anonymous functions.
5. Compile time evaluation.

Algebraic data types

```
data wad = wad(uint256)
```

- Data type for representing a 18 decimal fixed point

Algebraic data types

Type name

```
data wad = wad(uint256)
```

Constructor name

- Data type for representing a 18 decimal fixed point
  - Constructors and type names live in different namespaces

Algebraic data types

```
data AuctionState =
    NotStarted(uint256)
  | Active(uint256, address)
  | Ended(uint256, address)
  | Cancelled(uint256, address);
```

- Allows data modelling using sum and product types.

Algebraic data types

Type name

```
data AuctionState =
    NotStarted(uint256)
  | Active(uint256, address)
  | Ended(uint256, address)
  | Cancelled(uint256, address);
```

Sum types/Alternatives

- Sums allows the definition of exclusive alternatives in a type.

Algebraic data types

Type name

```
data AuctionState =
    NotStarted(uint256)
  | Active(uint256, address)
  | Ended(uint256, address)
  | Cancelled(uint256, address);
```

Product type

- Products combine values into structured tuples.

Pattern Matching

```
data wad = wad(uint256)

let WAD = 10 ** 18;

function wmul(lhs : wad, rhs : wad) -> wad {
    match (lhs, rhs) {
    | (wad(l), wad(r)) => return wad((l * r) / WAD);
    }
}
```

- Pattern matching allows changing control flow based on value shape.

Pattern Matching

```
let WAD = 10 ** 18;

function wmul(lhs : wad, rhs : wad) -> wad {
    match (lhs, rhs) {
    | (wad(l), wad(r)) => return wad((l * r) / WAD);
    }
}
```

Pattern match block

Pattern Matching

Discriminee

```
let WAD = 10 ** 18;

function wmul(lhs : wad, rhs : wad) -> wad {
    match (lhs, rhs) {
    | (wad(l), wad(r)) => return wad((l * r) / WAD);
    }
}
```

Pattern Matching

```
let WAD = 10 ** 18;

function wmul(lhs : wad, rhs : wad) -> wad {
    match (lhs, rhs) {
    | (wad(l), wad(r)) => return wad((l * r) / WAD);
    }
}
```

Wrapped uint256 values

Pattern Matching

```
let WAD = 10 ** 18;

function wmul(lhs : wad, rhs : wad) -> wad {
    match (lhs, rhs) {
    | (wad(l), wad(r)) => return wad((l * r) / WAD);
    }
}
```

Wrapping result in data constructor

Generics

```
forall T . function identity(x : T) -> T {
    return x;
}
```

- Generics allows functions that work in a uniform way for all types.
- Example: polymorphic identity function.

Generics

Type variable definition.

```
forall T . function identity(x : T) -> T {
    return x;
}
```

Parameter type.

- Generics allows functions that work in a uniform way for all types.
- Example: polymorphic identity function.

Generics

Type variable definition.

```
forall T . function identity(x : T) -> T {
    return x;
}
```

Function return type.

- Generics allows functions that work in a uniform way for all types.
- Example: polymorphic identity function.

Generics

```
data Result(T) = Ok | Err(T);
```

- Core supports the definition of generic types
  - Type parameter for the payload used in Err constructor

Generics

Type parameter definition

```
data Result(T) = Ok | Err(T);
```

- Core supports the definition of generic types
  - Type parameter for the payload used in Err constructor

Generics

Type parameter use

```
data Result(T) = Ok | Err(T);
```

- Core supports the definition of generic types
  - Type parameter for the payload used in Err constructor

Generics and Type classes

1. Generics are a bit limited.
   - Few functionality works **for all** types.

2. We need functionality which works for **some** types.

3. Type classes are the solution!
   - Allow the systematic combination of generics and overloading.

Type classes

```
forall T . class T:Mul {
    function mul(lhs : T, rhs : T) -> T;
}
```

- Type classes define a set of operations with their types.
  - Example: class for defining a multiplication operation

Type classes

Class name

```
forall T . class T:Mul {
    function mul(lhs : T, rhs : T) -> T;
}
```

- Type classes define a set of operations with their types.
    - Example: class for defining a multiplication operation

Type classes

Type argument

```
forall T . class T:Mul {
    function mul(lhs : T, rhs : T) -> T;
}
```

Function signature

Type classes

Type argument

```
instance wad:Mul {
    function mul(lhs : wad, rhs : wad) -> wad {
        return wmul(lhs, rhs);
    }
}
```

- Instances provide an implementation for a specific type.
  - Function types should “replace” the class type argument for the corresponding concrete type.

Type classes

Type class constraint

```
forall T . T:Mul => function square(val : T) -> T {
    return Mul.mul(val, val);
}
```

- Type class constraints allow us to restrict polymorphic types.
  - Function square can be called on all types which are instances of Mul.

High-order and anonymous functions

1. Functions are first-class in Core Solidity.

2. This means that they can be used:
   - Function parameters
   - Return values

Anonymous functions

```
forall T U . function count_calls(fn : (T) -> U) -> (memory(word), (T) -> U) {
    let counter : memory(word) = allocate(32);
    return (counter, lam (a : T) -> {
        counter += 1;
        return fn(a);
    });
}
```

- Anonymous functions are defined using the **lam** keyword.
  - Can capture values of their definition scope.
  - Example: count the number of times which an argument function is called.

Anonymous functions

```
forall T U . function count_calls(fn : (T) -> U) -> (memory(word), (T) -> U) {
    let counter : memory(word) = allocate(32);
    return (counter, lam (a : T) -> {
        counter += 1;
        return fn(a);
    });
}
```

Anonymous function definition

- Anonymous functions are defined using the **lam** keyword.

Anonymous functions

```
forall T U . function count_calls(fn : (T) -> U) -> (memory(word), (T) -> U) {
    let counter : memory(word) = allocate(32);
    return (counter, lam (a : T) -> {
        counter += 1;
        return fn(a);
    });
}
```

Captured value

- Anonymous functions are defined using the **lam** keyword.
    - Can capture values of their definition scope.

High-order functions

```
forall T U . function count_calls(fn : (T) -> U) -> (memory(word), (T) -> U) {
    let counter : memory(word) = allocate(32);
    return (counter, lam (a : T) -> {
        counter += 1;
        return fn(a);
    });
}
```

Function type

- Type (T) → U represents a function
  - That takes a value of type T as argument.
  - Returns a value of type U.

Type inference

1. Core Solidity infers types in functions and local variables.
2. Annotations are required only to solve ambiguities.
   - Occur in very rare situations.
3. Can help readability by omitting unnecessary type annotations.

Type inference

```
uint256[3] memory a = [1, 2, 3];
```

- Consider this simple Classic Solidity definition.

Type inference

```
uint256[3] memory a = [1, 2, 3];
```

- It is rejected with the following error message:

Error: Type uint8[3] is not implicitly convertible to type uint256[3].

Type inference

```
uint256[3] memory a = [1, 2, 3];
```

- Classic Solidity has a limited support for inferring the type of array literals.
  - Elements have the type of the first expression such that all other elements can be casted to it.
- Inferred type for the first element: uint8

Type inference

```
uint256[3] memory a = [uint256(1), 2, 3];
```

- Solution: use a type coercion on the first array element.

Type inference

```
uint256[3] memory a = [1, 2, 3];
```

- In Core Solidity, this definition is accepted
  - Thanks to type inference, which uses a more general strategy which eliminates the need of the type coercion.

Extended example

1. Core Solidity introduces several new features.

2. How those features help?

Console.log

```
function logUint(uint256 p0) internal pure {
    _sendLogPayload(abi.encodeWithSignature("log(uint256)", p0));
}

function logString(string memory p0) internal pure {
    _sendLogPayload(abi.encodeWithSignature("log(string)", p0));
}
```

- Library forge-std implementation has a combinatorial explosion of overloaded functions.
  - More than 30 different functions, one for each argument type.

Console.log

```
function _sendLogPayloadImplementation(bytes memory payload) internal view {
    address consoleAddress = CONSOLE_ADDRESS;
    /// @solidity memory-safe-assembly
    assembly {
        pop(
            staticcall(
                gas(), consoleAddress,
                add(payload, 32),
                mload(payload), 0, 0
            )
        )
    }
}
```

- Log functions just redirect to _sendLogPayloadImplementation.

Console.log - Core Solidity

```
forall T . T:ABIEncode => function log(val : T) {
   let CONSOLE_ADDRESS : word = 0x000000000000000000636F6e736F6c652e6c6f67;
   let payload = abi_encode(val);
   // extract the underlying word representation of the payload
   let ptr = Typedef.rep(payload);
   assembly {
       pop(
           staticcall(
               gas(), CONSOLE_ADDRESS,
               add(ptr, 32),
               mload(ptr), 0, 0
           )
       )
   }
}
```

- Generics and type classes eliminate all repetitive definitions!

Console.log - Core Solidity

ABI encode constraint

```
forall T . T:ABIEncode => function log(val : T) {
   let CONSOLE_ADDRESS : word = 0x000000000000000000636F6e736F6c652e6c6f67;
   let payload = abi_encode(val);
   // extract the underlying word representation of the payload
   let ptr = Typedef.rep(payload);
   assembly {
       pop(
           staticcall(
               gas(), CONSOLE_ADDRESS,
               add(ptr, 32),
               mload(ptr), 0, 0
           )
       )
   }
}
```

ABI encode argument

- Generics and type classes eliminate all repetitive definitions!

Console.log - Core Solidity

```
forall T . T:ABIEncode => function log(val : T) {
   let CONSOLE_ADDRESS : word = 0x000000000000000000636F6e736F6c652e6c6f67;
   let payload = abi_encode(val);
   // extract the underlying word representation of the payload
   let ptr = Typedef.rep(payload);
   assembly {
       pop(
           staticcall(
               gas(), CONSOLE_ADDRESS,
               add(ptr, 32),
               mload(ptr), 0, 0
           )
       )
   }
}
```

Convert to assembly level representation.

- Generics and type classes eliminate all repetitive definitions!

SAIL, desugaring and standard library

- SAIL is a new mid-level IR for Core.
  - Solidity Algebraic Intermediate Language.

- Features
  - Functions and contracts
  - Algebraic types and pattern matching
  - Assembly blocks
  - Type classes and generics
  - Variable introduction and assignment.
  - Short-circuiting if-then-else expression.

SAIL, desugaring and standard library

- Core Solidity high-level constructs will be implemented by a combination of desugaring steps and std-lib definitions.
- Similar approach used in proof assistants like Lean.
- Objective: RFC style for language and std-lib changes

Compatibility and interoperability

- Introducing major reviews to a language is challenging.
- We plan a smooth transition to avoid language split.
  - ABI compatibility will be maintained.
  - We’ll try to minimize syntax changes.
  - Investigate tooling for automated migration.

Simple Contract - Classic Solidity

```
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Simple Contract - Core Solidity

```
import std;
contract SimpleStorage {
    storedData : uint256;

    function set(x : uint256) -> () {
        storedData = x;
    }

    function get() -> uint256 {
        return storedData;
    }
}
```

- Minimal changes: postfix types, import of standard library.

The road to production

- Prototype implementation available.
- Standard library has implementations of:
  - ABI compatible contracts
  - ABI encoding / decoding
  - Dispatch
  - Storage access.
- Current prototype is able to compile a ERC20 contract.

The road to production

- Next steps (finish type system):
  - Define compile time evaluation.
  - Module system.
- Prototype stable:
  - Start a production implementation.
  - Mechanize the meta-theory using a proof assistant.

Beyond 1.0

- Our focus: deliver the language described so far.
- Possible future iterations:
  - Linear types.
  - Refinement types.
  - Macros
  - Theorem proving

Conclusion

- Core Solidity is a foundational re-imagining of the language.

- Our objective is to build a language:
  - More expressive.
  - More secure.
  - Mathematically sound.

Thanks!

Core Solidity repository

Core Solidity feedback forum thread