The Evolution of Solidity

Once upon a time, Solidity

- Solidity was designed/built to compile to EVM bytecode

- EVM bytecode was designed to execute smart-contracts

Source code

```
function withdraw(address account, uint amount) {
  balances[account] -= amount;
  payable(account).transfer(amount);
}
```

Compile

Bytecode

```
SSTORE
CALLER
PUSH2 0x08fc
CALL
```

Deploy

On-chain contract

```
0x303503......8aC8602
```

Solidity "evolution"

Solidity has is evolving to fulfill more use-cases

- Frameworks such as Foundry use Solidity for
  - Testing
  - Scripting
  - Interactive environment (REPL)

- Solidity/EVM doesn't support many uses cases out of the box
  - Access external environment (e.g. filesystem, env vars)
  - Deploy/call a contract from an EOA

Solidity "devolution"

- In most cases, we start from a general purpose programming language and implement domain-specific languages on top of it
  - Testing frameworks in many languages
  - Rust to implement smart contracts

- Solidity is a domain-specific language trying (moderately hard) to become a general purpose programming language

- Slightly awkward situation that makes many things a bit quirky

Extending Solidity

To support more features, Solidity needs to do more than it was designed to.
Some approaches include:

- Using a custom VM (anvil)
  - Can change the behaviour of calls (use cases: filesystem, env vars)
  - Can instrument execution (use cases: sending transactions from EOAs)

- Using a mini-language (chisel)
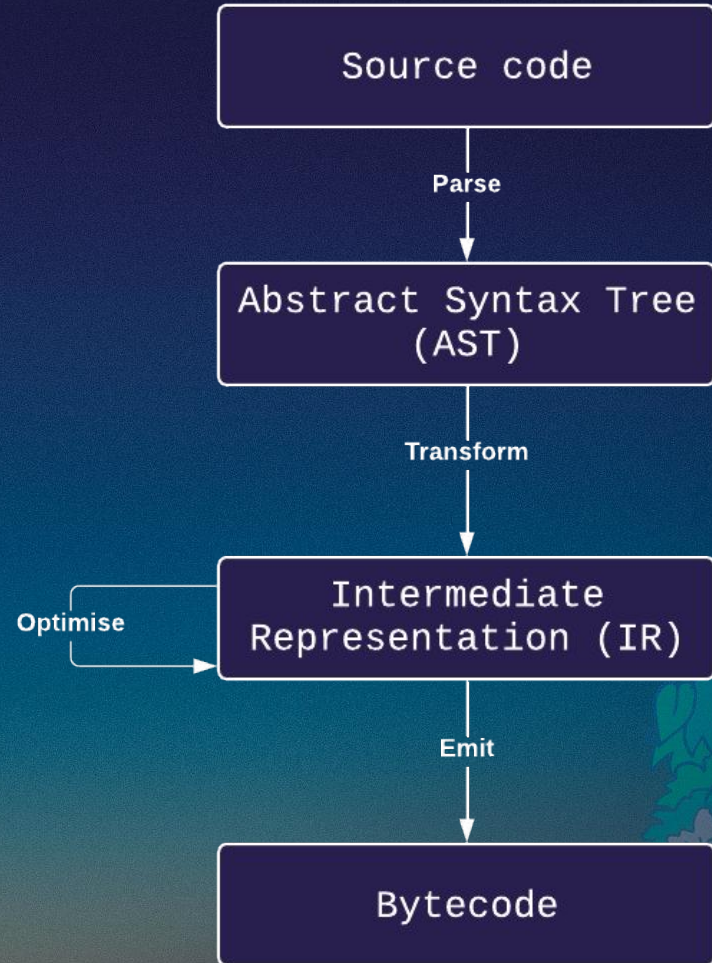 - Can use a separate interpreter (e.g. `!fetch DAI 0x6B17…1d0F`)

Changing Solidity

- Some tasks require precise semantics (e.g. testing), some don't (e.g. scripting)

- Current approach indirectly changes Solidity semantics by changing the VM semantics

- If semantics don't matter, we can change the actual language

- Changing the language would make it easier to offer better UX for some use-cases
  - General-purpose scripting
  - REPL

(not) Compiling Solidity

Compiler 101

1. Parse the source code (code -> AST)

2. Transform/optimise the AST (AST -> IR)

3. Emit bytecode (IR -> bytecode)

4. Execute the bytecode

```
Source code
```

Parse

```
Abstract Syntax Tree
(AST)
```

Transform

Optimise →
```
Intermediate
Representation (IR)
```

Emit

```
Bytecode
```

Semantics desiderata

We want to keep things similar to Solidity semantics but replace on-chain specific semantics

- Arithmetic and other basic operations should stay the same
  - `1 + 2 -> 3`
  - `[1, 2, 3][1] -> 2`

- Contract calls and other on-chain semantics should be replaced
  - `Contract("0x12..34").func()` -> RPC call

"Ideal" approach

The most flexible way would be:

- Replace the parser to allow flexibility on the syntax if desired

- Design a new bytecode to allow for general-purpose use-cases

- Implement the transformation and bytecode generation steps

This means re-implementing a language from scratch, which is a lot of work.
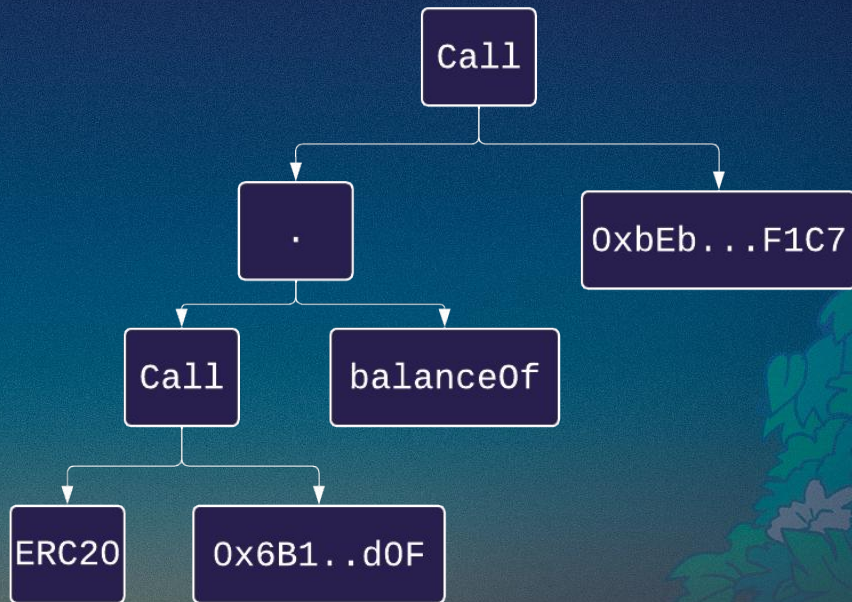
Mitigating the amount of effort

- Use existing Solidity parser
    - Almost no work to implement
    - No flexibility on the syntax

- Interpret the AST rather than interpreting bytecode
    - Easy to implement, no generation step
    - Much (much) slower

Interpreting the AST

2 * (4 + 1)
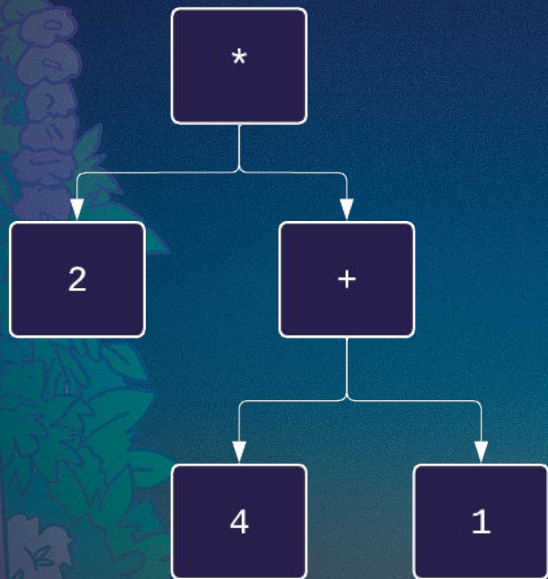
ERC20(0x6b1...d0f).balanceOf(0xbEb...F1C7)

Eclair, a Solidity Interpreter

https://eclair.so

Introducing Eclair

- Eclair is a Solidity REPL

- It executes the Solidity AST

- Its main goal is to allow easy interaction with smart contracts

- Built in Rust

```
>> repl.rpc("https://mainnet.optimism.io")
>> repl.loadLedger(5)
0x2Ed58a93c5Daf1f7D8a8b2eF3E9024CB6BFa9a77
>> usdc = ERC20(0x0b2C639c533813f4Aa9D7837CAf62653d097Ff85)
>> usdc.balanceOf(repl.account).format(usdc.decimals())
"5.00"
>> swapper = repl.fetchAbi("Swapper",
0x956f9d69Bae4dACad99fF5118b3BEDe0EED2abA2)
>> usdc.approve(swapper, 2e6)
Transaction(0xed2cfee9d712fcaeb0bf42f98e45d09d9b3626a0ee93dfc
730a3fb7a0cda8ff0)
>> target = 0xC013551A4c84BBcec4f75DBb8a45a444E2E9bbe7
>> trx = swapper.mint(usdc, target, 2e6, 0.5e18)
>> receipt = trx.getReceipt()
>> receipt.txHash
0xbdbaddb66c696afa584ef93d0d874fcba090e344aa104f199ecb6827170
09691
```

Main features

- Most common Solidity features

- Interaction with smart contracts using any RPC

- Loading ABIs from existing projects (Foundry, Hardhat, Brownie) and from Etherscan

- Loading accounts from raw private key, ledger, or encrypted keystore

```
>> for (uint256 i; i < 2; i++) {
    console.log((i + 1) * 10); }
0
10
>> repl.rpc("optimism")
>> repl.rpc()
"https://mainnet.optimism.io/"
>> router = repl.fetchAbi("SwapRouter",
0x68b3465833fb72A70ecDF485E0e4C7bD8665Fc45)
>> repl.loadKeystore("account-name")
Enter password:
0x559822cf7213bC2DDa0aeCffA0b66Bd083C169CD
>> router.swapTokensForExactTokens(...)
```

Differences to Solidity

- Dynamically typed

  ```
  a = 1; a = "foo"; console.log(a.length);
  ```

- First-class functions and types

  ```
  getBalance = token.balanceOf; getBalance(addr)
  ```

- More syntax sugar
  - Concatenation: `[1, 2] + [3, 4] -> [1, 2, 3, 4]`
  - Anonymous functions: `((a) >> a * 2)(3) -> 6`
  - Functionalish programming

    ```
    [1, 2, 3].filter((v) >> (v % 2 == 0)).map((v) >> v + 1)
    ```

State of the project

- Most desired features implemented

- Have been "tested in prod" for a few months
  - Used for most live debugging tasks
  - Used to generate/verify all kinds of transactions
  - Used to interact with wide range of contracts

- Only moderately tested
  - Few unit and integration tests but very (very) far from prod-level

Going forward

Short term

- Improve UX (completions, error messages, etc.)
- Add features
  - Execute files
  - Read/write files
  - JSON (de)serialization
- More testing

Longer term

- Move to a custom-built parser
  - Experiment with new syntaxes
  - More syntax sugar
- Move from interpreting the AST to interpreting bytecode
  - Much faster execution

Thank you!

Daniel Perez

Co-founder at Gyroscope & TLX

@danhper

https://eclair.so