Bootstrapping a Compiler with Yul

Grant Wuerker
Fe-lang

Fe overview

- High-level smart contract language
- Compiler is implemented in Rust
- Targets EVM
- Currently uses Yul IR

```
use std::context::Context

# The `contract` keyword defines a new contract type
contract GuestBook:
    # Strings are generic over a constant number
    # that restricts its maximum size
    messages: Map<address, String<100>>

    # Events can be defined on contract or module level
    event Signed:
        book_msg: String<100>

    pub fn sign(self, ctx: Context, book_msg: String<100>):
        # All storage access is explicit using `self.<some-key>`
        self.messages[ctx.msg_sender()] = book_msg

        # Emit the `Signed` event
        emit Signed(ctx, book_msg)

    pub fn get_msg(self, addr: address) -> String<100>:
        # Copying data from storage to memory
        # has to be done explicitly via `to_mem()`
        return self.messages[addr].to_mem()
```

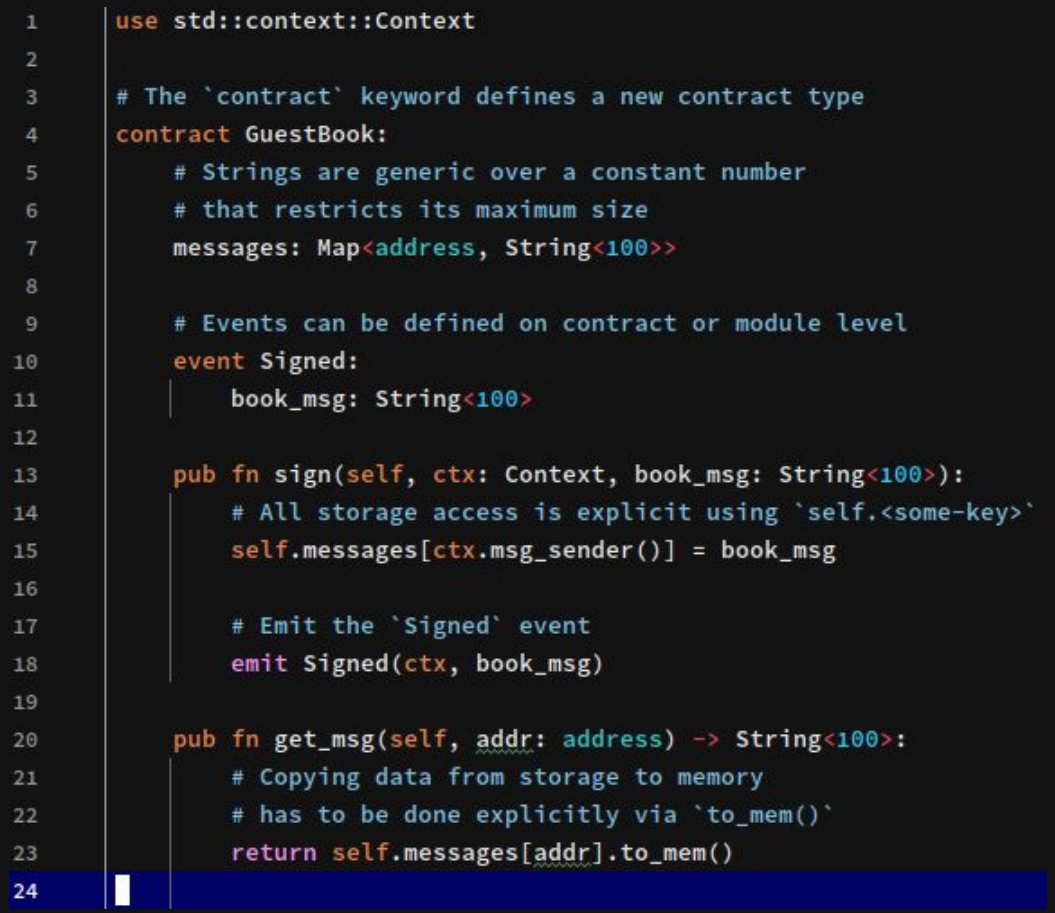
Yul overview

- Developed by the Solidity team
- Human readable
- Provides functions and data objects
- Basic control flow

```
object "Token" {
    code {
        /* ---------- contract deployment ---------- */
        datacopy(0, dataoffset("runtime"), datasize("runtime"))
        return(0, datasize("runtime"))
    }
    object "runtime" {
        code {
            /* ---------- dispatcher statement ---------- */
            switch selector()
            case 0x70a08231 /* "balanceOf(address)" */ { ... }
            case 0x18160ddd /* "totalSupply()" */ { ... }
            ...
            default { revert(0, 0) }
            /* ---------- user-defined functions ---------- */
            function mint(account, amount) { ... }
            function transfer(to, amount) { ... }
            ...
            /* ---------- calldata decoding functions ---------- */
            function selector() -> s { ... }
            function decodeAsAddress(offset) -> v { ... }
            /* ---------- calldata encoding functions ---------- */
            function returnUint(v) { ... }
            function returnTrue() { ... }
            ...
            /* -------- events ---------- */
            function emitTransfer(from, to, amount) { ... }
            function emitApproval(from, spender, amount) { ... }
            ...
            /* -------- storage layout ---------- */
            function ownerPos() -> p { p := 0 }
            function totalSupplyPos() -> p { p := 1 }
            ...
            /* -------- storage access ---------- */
            function owner() -> o { }
            function totalSupply() -> supply { }
            ...
            /* ---------- utility functions ---------- */
            function lte(a, b) -> r { ... }
            function gte(a, b) -> r { ... }
            ...
        }
    }
}
```

Smart contract language features

- ABI encoding/decoding
- Call dispatching
- Special statements
- User defined functions
- Type specific functions
- Storage fields



```
use std::context::Context

# The `contract` keyword defines a new contract type
contract GuestBook:
    # Strings are generic over a constant number
    # that restricts its maximum size
    messages: Map<address, String<100>>

    # Events can be defined on contract or module level
    event Signed:
        book_msg: String<100>

    pub fn sign(self, ctx: Context, book_msg: String<100>):
        # All storage access is explicit using `self.<some-key>`
        self.messages[ctx.msg_sender()] = book_msg

        # Emit the `Signed` event
        emit Signed(ctx, book_msg)

    pub fn get_msg(self, addr: address) -> String<100>:
        # Copying data from storage to memory
        # has to be done explicitly via `to_mem()`
        return self.messages[addr].to_mem()
```

Building the Yul runtime

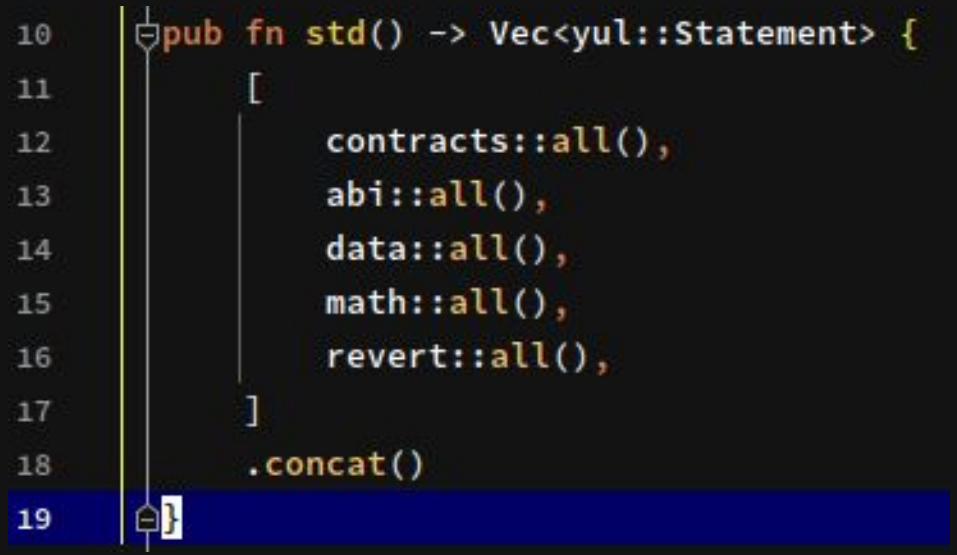
- Add standard runtime functions (memory/storage access, math,...)
- Add runtime functions from analysis (ABI encoding/decoding, events,...)
- Add user defined functions and dispatching

Building the Yul runtime (yultsur overview)

- https://github.com/g-r-a-n-t/yultsur
- AST
- Printer
- Shorthand macros

```
256  fn checked_div_unsigned() -> yul::Statement {
257      function_definition! {
258          function checked_div_unsigned(val1, val2) -> result {
259              (if (iszero(val2)) { [revert_with_div_or_mod_by_zero()] })
260              (result := div(val1, val2))
261          }
262      }
263  }
```

Building the Yul runtime (standard functions)



Building the Yul runtime (analysis based functions)

```
227  pub fn decode_component_uint(size: usize, location: AbiDecodeLocation) -> yul::Statement {
228      let func_name = abi_names::decode_component_uint(size, location);
229      let decode_expr = load_word(expression! { ptr }, location);
230      let check_padding = check_left_padding(
231          literal_expression! { ((32 - size) * 8) },
232          expression! { return_val },
233      );
234
235      function_definition! {
236          function [func_name](head_start, offset) -> return_val {
237              (let ptr := add(head_start, offset))
238              (return_val := [decode_expr])
239              [check_padding]
240          }
241      }
242  }
```

Building the Yul runtime (dispatcher and user defined functions)

```
26      let dispatcher = if arms.is_empty() {
27          statement! { revert(0, 0) }
28      } else {
29          switch! {
30              switch (cloadn(0, 4))
31              [arms...]
32              (default { (revert(0, 0)) })
33          }
34      };
35
36      let call_fn_ident = identifier! { ("$$__call__") };
37
38      function_definition! {
39          function [call_fn_ident]() {
40              [dispatcher]
41          }
42      }
```

Contract initialization

```
65      code! {
66          // add init function and dependencies to scope
67          [init_callgraph...]
68          [decode_fns...]
69
70          // copy the encoded parameters to memory
71          (let params_start_code := datasize([contract_name]))
72          (let params_end_code := codesize())
73          (let params_size := sub(params_end_code, params_start_code))
74          (let params_start_mem := alloc(params_size))
75          (let params_end_mem := add(params_start_mem, params_size))
76          (codecopy(params_start_mem, params_start_code, params_size))
77
78          // decode the parameters from memory
79          [maybe_decode_params...]
80
81          // call the init function defined above
82          (pop([init_call]))
83
84          // deploy the contract
85          [deployment...]
86      }
```

Running solc

- https://github.com/g-r-a-n-t/solc-rust
- Pass printed Yul into solc-rust via JSON interface
- Enable optimizations

Remarks

- Yul has made it easy to implement a high-level language.
- Yul is well documented and easy to use.
- Solc-rust adds significant build overhead.
- We are gradually replacing the Yul runtime code with Fe code.
- The Yul optimization step makes codegen much simpler.