DeepSEA Language Reference

DeepSEA/EVM version 1.3.0, preview release on January 26, 2021.

This document will be continually updated as the language evolves.

Contents

Introduction

DeepSEA is a language designed to allow interactive formal verification of smart contracts with foundational security guarantees. From an input program, the compiler generates executable code, but also a model of the program that can be loaded into the Coq proof assistant, and an automatically generated Coq proof that the model matches the behaviour of the compiled assembly code.

This document describes the programming part of the DeepSEA language: it shows how to write smart contracts similar to other EVM-based languages like Solidity and Vyper. In future updates we will also explain proof-related features like invariants and abstraction refinement.

What's missing in language version 1.x?

The language described in this manual is "version 1.x". It has features to write nontrivial contracts (e.g. token contracts), but it will still be undergoing further development, and the language manual will be continuously updated as the language evolves. The details will be worked out in more detail, but additional features could include e.g.

- Built-ins to query contract sender, send ether to addresses, probe if a contract is installed on the other side,
- Better support for arrays and structs.
- Checking cryptographic signatures.
- Support for building libraries which can be imported into Solidity.
- More efficient loops that update local variables.

In the long run the DeepSEA language will evolve further, but it may continue to lack several more "advanced" features, such as

- Dynamic memory allocation
- Pointers into memory and storage
- Function pointers and calling a contract address dynamically at runtime
- ...

There is always a tradeoff between advanced programming language features and the easy of modeling the language formally, and it will require further experimentation to see what is best.

Using the compiler

Installation

The preview release comes with `dsc` binary executables for MacOS (in binaries/MacOS/dsc) and Linux (in binaries/linux/dsc). Either add this directory to your PATH environment variable, or copy the file to a directory which is in your `PATH`. If you want to use the Ant Blockchain, instead copy and rename the `dsc_ant` binary.

The Linux binaries are tested on Ubuntu Linux. They may not work on some other distributions, because they are dynamically linked to system C library. It is also possible to build the project from source, following the instructions in the `src` directory.

Synopsis

```
usage: dsc <source file> <mode>
```

Modes

- `bytecode` : Compile to EVM bytecode.
- `abi` : Output a JSON-format description of the function types
- `assembly` : Compile to EVM mnemonics.
- `minic` : Compile to the MiniC IR.
- `coq` : Create a directory with a Gallina model of the contract.
- `ewasm`: Compile to Ethereum WebAssembly

Running Unit Tests

The directory `unittests` contains a few small programs which test particular features of DeepSEA. There are some TypeScript files that deploy these contracts and test their method calls.

If you just want to try compiling some programs, all you need is to download the binary. If you want to run the unit tests yourself, you'll need `node.js`, `tsc` (the TypeScript compiler) and some packages from `npm`.

For more info on some of the package dependencies:

- ethers.js: Ethereum API.
- ganache-cli: For running a local Ethereum chain.
- js-conflux-sdk Conflux chain API.

Installing `node.js` and `npm`

Ubuntu / Debian

```
sudo apt install nodejs npm
```

MacOS Downloading `node.js` from the website https://nodejs.org/en/ will install both `node.js` and `npm` at the same time. It is recommended to upgrade `npm` immediately after installing:

```
npm install npm@latest -g
```

Installing Dependencies Once `node.js` and `npm` are installed, the following will download all dependencies (as specified in `unittests/package.json`) to a local `node_modules` folder within this repository.

```
cd unittests
npm install
```

Finally, clone the conflux-rust repo, and within `start_conflux.sh`, set `$conflux` to the location of your repo clone. Otherwise, if you only want to run the test on Ethereum, open `tests.ts` and comment out the line containing "`new ConfluxChain`":

```
const chains = [
  new EthereumChain,
  // new ConfluxChain
]
```

Before running the tests, you need to start `ganache-cli` and `conflux` in separate terminals (or, run them as background processes in the same terminal). To do so, `cd` to the `unittests` directory and run `npm run ganache` and `start_conflux.sh`, respectively. Then, in a third terminal, run `./run_unittests.sh`.

Deploying a contract to the CertiK chain

The CertiK Foundation DeepWallet has built-in support for DeepSEA, making it very convenient to deploy contracts. Log in to DeepWallet (https://wallet.certik.foundation/), and make sure you have some CTK tokens in your account. (During the testing period you can request tokens from the faucet. Then click the "contracts" tap, and then click the DeepSEA button. Paste the source code of your contract into the text box, click "Compile" to check that there are no compilation errors, and then click the "Deploy" button).

Deploying a contract to Ethereum (or an Ethereum testnet)

DeepSEA contracts can be deployed to the Ethereum network, or since the the real Ethereum network costs money, for experimental purposes one can use a free

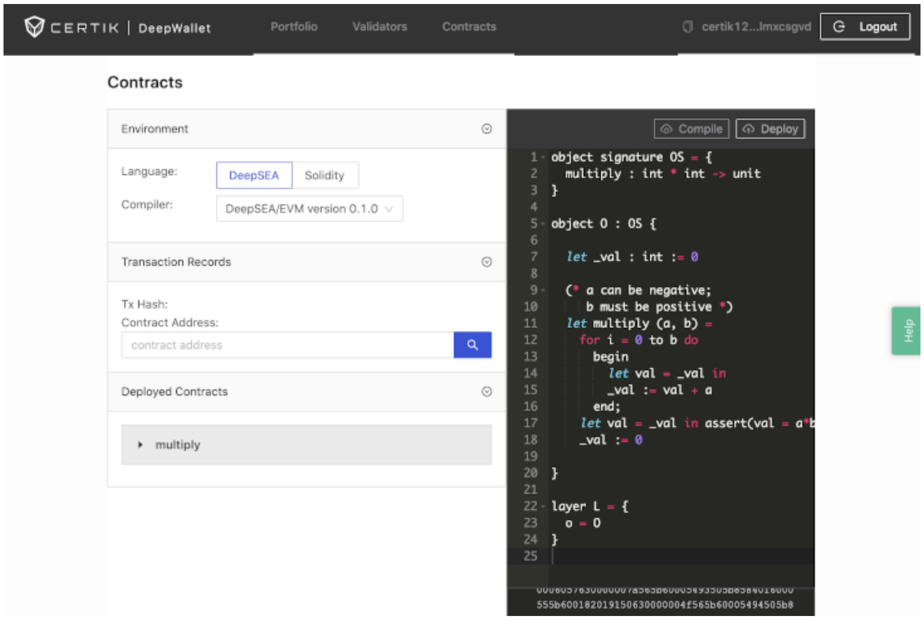

Figure 1: Contract deployment via DeepWallet

testnet (e.g. Ropsten, Kovan, and Rinkeby). The same version of the DeepSEA compiler works for both CertiK Chain and Ethereum.

There are many different tools and clients that can be used to deploy contracts to Ethereum. Generally speaking, this involves compiling using the "bytecode" and "abi" options, and then uploading both. For illustrative purposes, here is how to do it using MetaMask and MyEtherWallet, so the entire process can be done from a web browser.

1. Install the MetaMask browser extension for Chrome or Firefox, and create a new identity when prompted.
2. Click on the metamask button and from the "network" button on the top, select your preferred testnet, e.g. Kovan.
3. Go to a Kovan faucet and request some ETH.
4. Go to https://www.myetherwallet.com, click "access my wallet" -> Using a Browser extension.
5. In the myetherwallet interface, click on Contract -> Deploy Contract. Copy and paste the output of "dsc contract.ds bytecode" into the Bytecode box (you also need to add "0x" to the beginning of the text), and the output of "dsc contract.ds abi" into the abi box. Click deploy, and note what address the contract was given.

6. In the myetherwallet interface, click on Contract -> Interact with a contract. Fill in the contract address, and again paste the abi into the box. You get a screen where you can call the methods of the contract.

Deploying a contract to Ant Chain

In order to deploy contract to Ant Chain, you need to use the AntBlockchain version of the DeepSEA compiler, which can be downloaded separately.

Access to the Ant Chain testnet requires an enterprise level Alipay account. More details about how to apply for the testnet can be found in Ant Financial's documentation (in Chinese). After registering for the testnet, users can create their public and private keys and use one of Ant's SDKs to deploy the contract. In the DeepSEA release we provide an example using the javascript SDK, and in the rest of this section we describe how to run it and how it works.

Install the `dsc` binary following the instructions in the previous section. Make sure that you use the binary that is configured for Ant. Go to the `contracts/token_ant` folder, and compile the example contract:

```
dsc token.ds bytecode > token.bytecode
dsc token.ds abi > token.abi
```

Next, download the javascript SDK from the Ant website. We provide an example script in `contracts/token_ant/token.js` which demonstrates how to use SDK. The script begins with a require statements to load the SDK, then it creates a connection to the blockchain and deplos the contract (using the bytecode and abi files). Finally, a `console.log` statement prints out information such as msg_type, txhash, receipt and so on. We can record the txhash and later use that in Ant chain explorer to look up the contract. The script can also be customized by making calls to the contract inside the callback function.

In order to run the script, the user must first edit it to fill in information using their account information, so they can access the testnet. The table below shows the required information.

Table 1: Required configuration items in deploy script (fill in using your own account)

Configuration item	Description	Example value
host	Node's IP or host name. When using TLS, it is IP; when using https, it is host name	127.0.0.1, https://www.ali.com
port	Connection port number	18130

Configuration item	Description	Example value
timeouts	Timeout configuration	3000000
ca	Root certificate of target chain	fs.readFileSync(“./ca.crt”,{encoding:”utf8”})
cert	Client certificate file	fs.readFileSync(“./client.crt”,{encoding:”utf8”})
key	The private key file generated by the client.	fs.readFileSync(“./client.key”,{encoding:”utf8”})
userPublicKey	Account public key, the content is hexadecimal	0xswa4abce19c45fa9a69dcd39c40e95ce35af7ec1be2187d9cas221fd8e6f54c3acf3280599b33b0d20477e687b1ce3f310d1935bc7b4c427ff8f87f7132ba153
userPrivateKey	Account private key, the content is hexadecimal	0x12r4abce19c45cv1469dcd39c40e95ce35af7ec1be2187d9cas221fd8e6f54c3acf3280599b33b0d20477e687b1ce3f310d1935bc7b4c427ff8f87f7132ba153
userRecover PublicKey	Account recovery public key, the content is hexadecimal	0x12r4abce19c45cv1469dcd39c40e95ce35af7ec1be2187d9cas221fd8e6f54c3acf3280599b33b0d20477e687b1ce3f310d1935bc7b4c427ff8f87f7132ba153
userRecover PrivateKey	Account recovery private key the content is hexadecimal	0x76gtsce19c45cv1469dcd39c40e95ce35af7ec1be2187d9cas221fd8e6f54c3acf3280599b33b0d20477e687b1vf31410d1935bc7b4c427ff8f87f7132ba153
passphrase	Passphrase for client.key	pass123

Compiling contracts to WebAssembly

Currently DeepSEA supports Ethereum-flavored WebAssembly ewasm, which is used by Ethereum 2.0, CertiK Chain, and Hyperledger Burrow. The main difference between DeepSEA WebAssembly and other (blockchain) WebAssembly variants is that DeepSEA WebAssembly is 32-bit word sized while most other Webassemly (e.g., Ethereum Foundation ewasm) are 256-bit word sized. However, this does not mean ewasm cannot do 256-bit operations, it is just users are not allowed to define 256bit constants, write addresses that are larger than 2ˆ32, or have a DeepSEA function return a value that is larger than 2ˆ32. 256-bit values like Keccak256 hash function results works in DeepSEA through the use of DeepSEA local variables (256-bit addressable) and maneuvering of WebAssembly memory.

To use the WebAssembly backend of DeepSEA, you can simply use the DeepSEA binary `dsc` with the option `ewasm`.

```
dsc example.ds ewasm > example.wat
```

This will compile the DeepSEA contract `example.ds` into `example.wat`, the WebAssembly text format. Then, we will use the script `wasm_init.sh` to assembly our text format into `example-r.wasm` and `example-i.wasm`, namely, the runtime version of the example contract and the deployable version of the example contract (i.e., with initializers). The script depends on installing WebAssembly assembler `wat2wasm`, which is in the WebAssembly toolkit `wabt`, maintained by the WebAssembly foundation (required for running DeepSEA WebAssembly)

Here is a simple demo to run the contract in a Javascript VM:

```
cd deepsea/
cd contracts/wasmtest/for
dsc ./for.ds ewasm > ./for.wat #generate human readable WebAssembly text representation
../../../wasm_init.sh for            #assemble the text representation to binary
cd ..
./for/for.js ./for/for-r.wasm multiply 4 8 #gives result 32, using ethereum storage to compu
```

Or, to execute the example contracts in `deepsea/contracts/wasmtest`, you can simply:

```
cd deepsea/
cd contracts/wasmtest
./run_test.sh <CONTRACTNAME> <FUNCTIONNAME> <FUNCTIONARG1> <FUNCTIONARG2>
```

You can also deploy and play around with the above contract on CertiK's shentu chain, note that CertiK chain depends on Hyperledger Burrow VM, which does not support Keccak256 system contracts yet, so currently deployed contracts cannot use any hash-related operations (e.g., arrays, structs, storage):

```
#assume already used dsc to generate contract abi and wasm init version, i.e., example-i.was
cd deepsea
init-single.sh #initializes a local CertiK shentu chain node
cp example-i.wasm shentu/tests/example-i.wasm
cp example.abi shentu/tests/example.abi
#change testEWASM to use example contract
testEWASM.sh #test and deploy the eWasm contract specified in this script to shentu testnet
```

DeepSEA by example

To give the flavor of the language, we show a contract which implements a simple auction system. A DeepSEA program is written as a set of objects holding state. The auction contract is simple enough that we only need one object, which keeps the table of allowed withdrawals and the highest bidder.

Each object also provides a set of methods. The programmer has to first declare an *object signature* giving the type of the methods, and then the *object definition* which defines the state variables and the methods.

The final line collects all the objects of the program into a single "layer", which represents the complete contract. In this case there is only a single object involved.

```
object signature AuctionInterface = {
  constructor   : uint -> unit;
  const getbid      : unit -> uint;
  const getbidder   : unit -> address;
  const getchair    : unit -> address;
  const getdeadline : unit -> uint;
  bid         : unit -> bool;
  withdraw    : unit -> bool;
}

object OpenAuction : AuctionInterface {
  (* initial values don't get used *)
  let _bid         : uint := 0u0
  let _bidder      : address := address(0)
  let _chair       : address := address(0)
  let _deadline    : uint := 0u0
  let withdrawals : mapping[address] uint := mapping_init

  let constructor deadline =
    _bid          := 0u0;
    _bidder       := msg_sender;
    _chair        := msg_sender;
    _deadline     := deadline;
    ()

  (* getter functions *)
  let getbid      () = _bid
  let getbidder   () = _bidder
  let getchair    () = _chair
  let getdeadline () = _deadline

  (* place bid *)
  let bid () =
    let bidder     = _bidder             in
    let bid        = _bid                in
    let deadline   = _deadline           in
    let withdrawal = withdrawals[bidder] in

    assert ((msg_sender   <> bidder ) /\
            (msg_value    >  bid    ) /\
            (block_number < deadline));

    withdrawals[bidder] := withdrawal + bid;
    _bidder := msg_sender;
    _bid    := msg_value;
    true

  (* losers can withdraw their money *)
  let withdraw () =
    let winner = _bidder in
```

```
    assert (msg_sender <> winner);
    let balance = withdrawals[msg_sender] in
    withdrawals[msg_sender] := 0u0;
    transferEth(msg_sender, balance);
    true

}

layer AUCTION = {
  auction = OpenAuction
}
```

Structure of a DeepSEA source file

We now begin a more formal description of the language.

A source file consists of a number of comments, type definitions, type signatures, object definitions, and layer definitions. The source file should end with a layer definition which represents the entire program.

Comments

```
(* Comments are written like this. *)
```

Datatype declarations

Primitive types These primitive types are built-in:

```
int             (* 256-bit unsigned integer *)
uint            (* 256-bit unsigned integer, overflow allowed *)
address         (* 160-bit ethereum address, no arithmetic operations supported. *)
bool
unit            (* dummy value *)
array[42] int   (* e.g.: array of integers, length 42 *)
mapping[int] int (* e.g.: hash mapping from ints to ints. *)
int*bool*bool   (* tuple types, e.g. a triple. *)
hashvalue       (* the result of the keccak256 function. *)
identity        (* 256-bit Ant-blockchain address, no arithmetic operations supported. *)
```

The run-time representation of `hashvalue` is the same as `uint`, and when interoperating with Solidity program it can be treated as a synonym. However, we make it a separate type, because when generating Coq specifications for programs we treat hashing symbolically (implicitly assuming that the hash functions are injective).

Note: the `identity` type is only available in DeepSEA version for Ant Chain, but not in DeepSEA for Ethereum. It can be considered as a replacement of the `address` type. Conversely, `address` is not supported in that version of DeepSEA. More information about Ant blockchain technology and the identity type can be found in the Ant documentation (written in Chinese, but e.g. Google Translate works well on it).

Type aliases A type alias declares a new name for a type, in order to help readability. The generated Coq specifications will also declare a type alias.

```
type timeunit := uint (* time in seconds *)
```

Struct types A struct (a.k.a. record) type defines a set of fields.

```
type Usage = {
  u_quota : int;
  u_usage : int
}
```

Fields can be accessed by dot-notation. This is useful as a component of a larger type, e.g. we can define a mapping from integers to Usage,

```
let usages : mapping[int] Usage
```

and then access it as `usages[i].u_usage`.

Multidimensional mapping types It is possible to create mappings or arrays indexed on multiple keys by making the return type of a mapping be a mapping itself. This is similar to how they are handled in Solidity. For example, in Solidity a mapping with two keys can be defined as

```
mapping (address => mapping (address => uint256)) private _allowances;
```

In DeepSEA one can get the same effect by writing

```
let allowances : mapping[address] mapping[address] int := mapping_init
```

Algebraic types It is also possible to declare the algebraic datatypes, similar to datatypes declarations in functional languages like ML and Haskell, but in order to avoid runtime overhead the implementation is quite restricted. It is possible to write down quite general datatype declarations, which will get copied into the generated Coq specifications, but the compiler will not work on code that uses them.

There is also a smaller subset of datatype declarations which can be used for code as well as for writing specifications. In the future this will be extended to cover more idioms, but for now algebraic types can be used as enums, i.e. to give names to a set of integer constants. For example, one can write

```
type Status [[int]] =
  | Ok        [[= 0]]
  | Triggered [[= 1]]
  | Finalized [[= 2]]
```

The annotations in double square brackets mean that values of the type will be realized as an int, and gives the corresponding values. After this declaration, the constructor Ok, Triggered, etc can be used as values, and one can write a match statement

```
match x with
| Ok =>
   ...
| Triggered =>
   ...
| Finalized =>
   ...
end
```

The match statement is translated into a series of if-statements comparing x with the given values, but unlike using integers directly the typechecker will enforce that all possible values of the type are handled.

Event types The syntax for declaring Ethereum event types is similar to the syntax for algebraic types, except it used the keyword "event", there is no name for the datatype, and each constructor argument can optionally be tagged as "indexed":

```
event
    Transfer(from : address indexed) (to: address indexed) (value: uint)
  | Approval(owner: address indexed) (spender: address indexed) (value: uint)
```

You can also use a bar before the first line, with no change in meaning, so the above declaration could equivalently be written:

```
event
  | Transfer(from : address indexed) (to: address indexed) (value: uint)
  | Approval(owner: address indexed) (spender: address indexed) (value: uint)
```

The fact that several lines are grouped into a single declaration is not significant, there is only a single name space for event names so the same declaration could equivalently be written:

```
event
  Transfer(from : address indexed) (to: address indexed) (value: uint)

event
  Approval(owner: address indexed) (spender: address indexed) (value: uint)
```

Currently, only word-sized arguments to events are supported. Also, currently all events are named, there is no syntax to declare what Solidity calls "anonymous" events.

Object and Layer signatures

A DeepSEA program consists of a collection of *objects* with methods, and collections of objects are further grouped into *layers*. The layers are composed vertically, i.e. a layer can be put "on top" of another layer.

In order to make proofs about programs more tractable, methods can not call arbitrary other methods; they can **only** call methods from the layers below them. (This means that there can be no recursive or mutually recursive functions). To keep track of what methods are available to call, DeepSEA uses *object signatures* and *layer signatures*. An object signature specifies the argument types and return type of each method, while a layer signature lists the objects in the layer.

A signature can be given as a list of method declarations between curly braces, separated by semicolons.It is also possible to define a name for a signature with the syntax

```
object signature ERC20Interface = {
  transfer : address * int -> bool;
  const totalSupply : unit -> int;
  const balanceOf : addr -> int
}
```

In the rest of the file, the name ERC20Interface can be used wherever the curly-brace expression could. In addition to curly-brace lists and names, signature expression can also use minus-expressions to delete an item from a signature.

```
object signature ERC20Interface2 = ERC20Interface - totalSupply

object signature ERC20Interface3 = ERC20Interface - {totalSupply; balanceOf}
```

As shown above, a method can be annotated as `const`, and if so the typechecker enforces that it does not modify the contract state. When generating contract ABI descriptions, the compiler marks const functions as "view", which means they can run them on a local node without creating a transaction, so this feature is useful for interacting with contracts in practice.

(The full compiler will also support a different annotation, "ghost", but this is only useful when writing proofs.)

A layer signature is a curly-brace list of the object in the layer, separated by semicolons. Each item in the list specifies a name for the object, and its signature. Similar to object signatures, we can define names for signatures with the syntax

```
layer signature <name> = <layer_signature_expression>
```

For example,

```
layer signature FIXEDSUPPLYTOKENSig = {
  fixedSupplyToken : ERC20Interface
}
```

Objects

Most of a DeepSEA program consists of object definitions. Each object consists of a collection of fields (i.e. state variables) and a set of methods acting on those fields. Just like in typical object-oriented languages, the only way to change the fields is by calling the methods. However, note that DeepSEA objects are not dynamically allocated; there is only a single instance of each object, and each field corresponds to a fixed location in contract storage. That being said, there are ways to avoid boiler-plate code when writing multiple objects with the same signature and (almost) same implementation, you can use the keyword `clone` in front an object declaration in a layer definition to indicate that your object is the same as another. For example, the `erc20Token0` object is an instantiation (on address 0x10033) of the `FixedSupplyToken` object definition, and `ercToken1` is a clone of `erc20Token0` (on address 0x10032):

```
layer AMMLIB : [{}] AMMLibSig = {
  erc20Token0 = address(0x10033) <: FixedSupplyToken;
  erc20Token1 = address(0x10032) <: (clone FixedSupplyToken); (* cloned an object *)
  liquidityToken = address(0x10031) <: LiquidityToken;
}
```

Here is an example object definition.

```
object Pausable (ownable : OwnableInterface) : PausableInterface  {
  let paused : bool  = false;

  let whenPaused () =
    assert (paused = true)

  let whenNotPaused () =
    assert (paused = false)

  let pause () =
    ownable.onlyOwner();
    pause := true;
    emit Pause

  let unpause () =
    ownable.onlyOwner();
    pause := false;
    emit Unpause
}
```

The first line gives the name of the object, Pausable, followed by a declaration of what objects we assume that the layer below provides. Inside the parenthesis can be either a layer signature expression (e.g. a layer signature name), or we can give a comma-separated list of objects and their object signatures (as above, with a single object). Finally, after the colon we specify what object signature this object satisfies.

Methods can call methods from the objects in the layer below, e.g. ownable.onlyOwner().

The body of the method definition consists of a set of field definition and a set of method definitions, with the syntax

```
let <field_name> = <initializer>
let <method_name> arg1 ... argn = <statement>
```

If a method does not need any arguments, it can be declared as having type `unit-><ret_type>`, and the dummy argument can be written as `()`. The syntax of statements and initializers are discussed below.

Layers

Layers can be declared with the syntax

```
layer <name> = <layer_expression>
```

or

```
layer <name>: [<signature>] <signature> = <layer_expression>
```

The signature inside the square brackets is the signature of the layer underneath, and the second signature is the signature of the layer currently being defined. If the layer does not depend on anything else, this signature can be an empty set of square brackets.

There are two kinds of commonly used layer expressions. First, a semicolon-separated list in curly braces which links each object name specified in the signature with an implementing objects, like so:

```
layer OWNABLE : [ { } ] { ownable : OwnableInterface }  = {
  ownable = Ownable
}

layer PAUSABLE : [{ ownable : OwnableInterface }]
                 { pausable : PausableInterface }
= {
  pausable = Pausable
}
```

Second, the syntax the L1@L2 means putting L1 on top of L2. The signature of L2 should match the signature that L1 expects (but it can provide additional items). A DeepSEA program typically ends by composing all the layers together into a single object:

```
layer CONTRACT = TOKEN @ PAUSABLE @ OWNABLE
```

The final layer in the file will be compiled into the contract.

Initializers and Constructors

The DeepSEA compiler generates code to initialize the object fields when a contract is uploaded. When you declare a field in an object, the syntax is

let fieldname : type := initializer

and the right hand side specifies the initial value. There is also a provision for adding arbitrary code to the contract constructor, in order to do more complicated forms of initialization.

Initializers The initial values that can be written on the right-hand-side depends on the type of the field. For booleans and integers one can write an ordinary literal expression:

```
let a : bool := true
let b : int := 42
let c : uint := 0u42
let d : address := address(0xd115bffabbdd893a6f7cea402e7338643ced44a6)
```

For arrays and hash mappings, the special (dummy) initializers `array_init` and `mapping_init` are used. These leave the field with the EVM default value, e.g. an array or mapping of zeros for an integer array.

```
let apool : array[64] int := array_init
let balances : mapping[addr] int := mapping_init
```

Struct types can be initialized using curly-braces notation:

```
type Token = {
  totalSupply : int;
  owner : addr
}

object O : OS {
  let token : Token := { totalSupply=0; owner=0 }
}
```

Keccak-256 can be left as an all-zero-bit word using the keyword `null_hash`.

```
let _hashlock : hashvalue := null_hash
```

Constructors If an object method is named `constructor` it is treated specially. The code in the method gets executed when the contract is deployed. It can be used to initialize data in the storage.

```
object signature LowerInterface = {
  constructor : uint -> unit;
  const foo : unit -> int;
  const bar : unit -> uint
```

```
}

object Lower : LowerInterface {
  let x : int := 6
  let y : uint := 0u6

  let constructor new_val =
     x := 100;  (* This overwrites the previous value 6 *)
     y := new_val

  ...
```

The constructor is not present after the contract has been deployed, so it cannot be called by other functions. It also does not support calls to other methods.

Multiple Constructors Each object can declare its own constructor. If the entire program contains multiple objects with nontrivial constructors, all the constructors will be appended together into a single constructor for the final contract, in the order they appear in the .ds file. The arguments of that constructor is the list of all arguments for the object constructors, appended in the order they appear in the file, so when you deploy the contract you need to be careful of order of the arguments. It is possible to examine the `abi` file to confirm what the type of the final constructor is. For example, if the source file has

```
object Lower : LowerInterface {
  let constructor (_x, _y) = ...
}

object Upper (lower : LowerInterface) : UpperInterface  {
  let constructor (_z, _w) = ...
}
```

Then the generated .abi file may contain

```
[ {"type":"constructor",
   "name":"constructor",
   "inputs":[{"name":"_x", "type":"uint256"},
             {"name":"_y", "type":"uint256"},
             {"name":"_z", "type":"uint256"},
             {"name":"_w", "type":"uint256"}],
   "outputs":[],
...
```

Expressions and statements

Like many programming languages, the grammar distinguishes between expressions and statements. Expressions in DeepSEA have no side effects and in the generated Coq specifications they are pure Coq expressions, while the generated Coq specifications for statements are monadic commands.

Integer literals Integers of type int can be written in decimal or hexadecimal, e.g.

```
x := 42;
y := 0x2a
```

Integers of type uint can be written with a leading 0u, e.g.

```
z := 0u42
```

(Recall that both int and uint are unsigned, the difference is whether numbers are allowed to overflow or not.)

Ethereum addresses are written as integer literals with the `address` keyword:

```
w := address(0xd115bffabbdd893a6f7cea402e7338643ced44a6)
```

Boolean/Arithmetic/Bitwise expressions Arithmetic operations are supported on int and uint expressions:

```
-e        (* negate a number *)
e1 + e2   (* addition *)
e1 - e2   (* subtraction *)
e1 * e2   (* multiplication *)
e1 / e2   (* division *)
e1 mod e2 (* remainder *)
```

Bitwise operations are supported only on uints:

```
~e       (* bitwise not *)
e1 || e2 (* bitwise or *)
e1 ^ e2  (* bitwise xor *)
e1 & e2  (* bitwise and *)
e1 << e2 (* shift left *)
e1 >> e2 (* shift right *)
```

Comparison operators produce booleans:

```
e1 = e2  (* equality test *)
e1 <> e2 (* not equal *)
e1 < e2  (* less than *)
e1 <= e2 (* less than or equal *)
e1 > e2  (* greater than *)
e1 >= e2 (* greater than or equal *)
```

Logical operators supported on boolean expressions:

```
!e       (* not *)
e1 /\ e2 (* and *)
e1 \/ e2 (* or *)
```

Integer overflow One caveat is that **the programmer is responsible** for making sure that `int` expressions do not overflow. Values of type `int` are represented as unsigned integers, so for example the expression (3 - 5) will wrap around and evaluate to a huge positive 256-bit number. In the full system, the programmer must prove that no over/underflows occur, as a side condition for all other proofs. However, if one only compiles programs and don't write any formal proofs in Coq, then the programmers will need to informally convince themselves that the operations are safe. Later versions of the language might provide an option for "safe arithmetic" which reverts the program on overflow, making the side condition trivial.

Expressions of type `uint` are allowed to overflow, which is suitable for e.g. bitwise operations but not for storing money balances.

Ethereum-specific built-in expressions The following expressions expose information about the current Ethereum transaction. As the language develops, we will extend this functionality to be more complete.

```
this_address    (* like address(this) in Solidity, address of this contract *)
tx_origin       (* like tx.origin in Solidity. *)
msg_sender      (* like msg.sender in Solidity *)
msg_value       (* like msg.value in Solidity, the value of this message in Wei. *)
block_coinbase  (* like block.coinbase in Solidity, address of the current block's miner *)
block_timestamp (* like block.timestamp in Solidity, current block's Unix timestamp in seconds *)
block_number    (* like block.number in Solidity, current block's number *)
chain_id        (* like chainid (assembly) in Solidity, ID of the EVM network *)
self_balance    (* like selfbalance (assembly) in Solidity, balance of this contract *)
balance(addr)   (* like addr.balance in Solidity, balance of the address in Wei. *)
blockhash(n)    (* like blockhash(n) in Solidity, hash of the given block *)
```

Keccak-256 hashing The built-in `keccak256` function computes a cryptographic hash.

```
keccak256(x)    (* like uint256(keccak256(abi.encodePacked(x))) in Solidity. *)
keccak256(x, y) (* like uint256(keccak256(abi.encodePacked(x,y))) in Solidity. *)
```

The resulting expression has type `hashvalue`, and the arguments `x` and `y` can be either of type `uint` or `hashvalue`. Values of type `hashvalue` are represented the same as `uint` at runtime, but they support a more limited set of operations, they only thing you can do with them is compare them for equality or use them as inputs for further hashing.

B It is also possible to hash values of `address` type. However, these are treated as zero-padded 256-bit integers (32 bytes), which is different from hashing addresses in Solidity (which hashes a 20 byte string).

```
keccak256(x) (* like uint256(keccak256(abi.encodePacked(uint(x)))) in Solidity. *)
```

L-expressions Structs, arrays, and hashtables can be read and assigned to using "dot" and "square bracket" notation:

```
let x = in strct.fld1 in
strct.fld2 := 3;
let y = in arr[3] in
arr[4] := 879;
let z = in tbl[3] in
tbl[4] := 879;
```

These expressions are called l-expressions (the L means "left", since they can be on the left side of an assignment.) It is a quirk of the language that these expressions can not be directly used as part of larger expression, instead the programmer needs to use a let-statement to give a name to the value (as above).

State variables in an object are also considered l-values, so accessing them similarly needs a let-statement:

```
let _totalSupply :  int := 10000

let getTotalSupply () =
  let totalSupply = _totalSupply in
  totalSupply
```

Future versions of the language may omit the l-value restriction and allow them to be used freely as r-values.

Simple control structures Commands can be separated by **semicolons**. They can be grouped by **begin-end** pairs (similar to curly braces in curly-brace languages). Because the semicolon separates (rather than ends) commands, it can be omitted for the final command in a method, but it does no harm to have one there.

Another frequently used construction is **let-statements**, which bind a value to a local variable. The syntax is

```
let <name> = <expression> in <statement>
```

where the expression can be either an ordinary expression or an l-expression. The variable can not be assigned to, and in the generated Coq specifications the statement becomes a let-expression binding an immutable variable. The EVM backend puts the variables on the stack, and there is no support for spilling to memory, so compilation can fail if there are more than 16 local variables in a function. For the same reason, the let-bound variables can only be atomic values like integers, not e.g. structs.

The **return value of a method** is the value of the last expression, e.g. the following method returns either true or false. (There is no provision for “early return” from a method.)

```
let decrement n =
  if (current >= n)
  then begin
    current := current - n;
    true
  end else
    false
```

Assertions Assertions in DeepSEA work like assert/require statements in Solidity: if the assertion is not satisfied it reverts the transaction. An assert statement takes boolean expression as an argument, for example

```
let onlyOwner () =
  assert (msg_sender = owner)
```

It is also possible to use the keywords `deny` and `fail`:

```
deny e (* equivalent to (assert (!e)) *)
fail   (* equivalent to (assert false) *)
```

In the generated specifications, asserts becomes a failure in the option monad; also asserted expressions are added as known assumptions when proving datatype verification conditions.

Loops In order to keep the generated specifications simple, DeepSEA only provides bounded loops. The means that methods always terminate, so they can be directly modeled by a terminating coq function. (In Ethereum smart contracts it is not possible to write unbounded loops anyway, because they will run out of gas.)

For-loops The simplest kind of loop is a for-loop over a given range, which has the syntax

```
for <var> = <exp1> to <exp2> do <cmd>
```

The variable ranges from exp1 (inclusive) to exp2 (exclusive). The type of the variable, and the expressions, is int. For example,

```
for i = 0 to 16 do
   a[i] := 2*i;
```

Fold/First-loops Although for-loops are fairly general, there is a limitation because DeepSEA statements can only affect storage variables. For efficiency, DeepSEA also provides special purpose support for certain idioms which can be computed using only stack variables.

B The preview release only implements for-loops, and other loop types will be added in the future.

Calls to lower layers If one of the objects in the lower layer is named `o`, and that object contains a method named f, then we can call the method with the syntax `o.f(arg1, ... argn)`.

```
object Upper (lower : LowerInterface) : UpperInterface  {
  let f () =
    lower.incr();
    lower.incr()
}
```

The method call may have side effects, and it may also return a return value. In the latter case it counts as an l-expression, so it also needs a let-expression.

```
 let n = lower.get() in
    n
```

Calls to external contract methods

The same syntax as calling lower layers is used for calling methods of external contracts. A value and gas parameter may optionally be specified with the syntax `o(value, gas).f(arg1, ... argn)`.

```
object Upper (extern_contract : ExternalContractInterface) : UpperInterface  {
  let f () =
    let value = 5 in
    let gas = 50 in
    extern_contract.incr();
    extern_contract(value).incr(); (* Just value specified *)
    let n = extern_contract(0, gas).get() in (* Value and gas specified *)
    n
}
```

Value can be specified without gas, but if gas is given than value must be too. When not provided, value defaults to zero and gas defaults to all available gas.

The address of external contracts has to be hard-coded in the layer definitions, so they must be known at compile-time. Future versions of DeepSEA may support calling external contracts with addresses provided at runtime. The following snippet, based on `contracts/amm/amm.ds`, demonstrates the syntax for specifying a contract's address. `erc20Token0`, `erc20Token1`, and `liquidityToken` are all contracts called by `amm`.

```
layer signature AMMLibSig = {
  erc20Token0 : FixedERC20Interface;
  erc20Token1 : FixedERC20Interface;
  liquidityToken : ERC20Interface;
}

layer AMMLIB : [{}] AMMLibSig = {
  erc20Token0 = address(0x102319b50F5BAC2B105a214aCFAdec7B96e61c96) <: FixedSupplyToken;
  erc20Token1 = address(0x4D9fe8921b52a316B6Bb9796e07AFF6b98F4528a) <: (clone FixedSupplyToken);
  liquidityToken = address(0x6D7CbbD06cA6c47b34A3C6811592Cb3c64aB87a2) <: LiquidityToken;
}

layer signature AMMSig = {
  amm : AMMInterface
}

layer AMM : [AMMLibSig] AMMSig = {
  amm = address(0x24558f5eEA3D0e550698f093Ea43bAce27Ad1a47) <: AutomatedMarketMaker
}

layer COMPLETE = AMM @ AMMLIB
```

Constructing/matching values of algebraic data types Once one has declared an algebraic datatype (see above), it is possible to create values of it by mentioning the datatype constructors. E.g. given the declaration

```
type Status [[int]] =
  | Ok        [[= 0]]
  | Triggered [[= 1]]
  | Finalized [[= 2]]
```

we can assign it to a field, or make a value using

```
let x = Ok in ...
```

Note that constructors are considered L-expressions, so they need a let-statement to be embedded in other expressions.

Similarly, we can do a case statement on values using the `match...with...end` command:

```
let v = match x with
| Ok =>
    40
| Triggered =>
    41
| Finalized =>
    42
end
```

Generating Ethereum events Once an event has been declared (see above), it can be generated with the "emit" command.

```
emit OwnershipTransferred(owner, newOwner);
```

Generating Specifications

In addition to executable bytecode, a DeepSEA program can be compiled into Coq specifications. These specifications are a model that provide a framework for formally proving facts about the program. The compiler also generates a proof that the model matches the behavior of the bytecode.

To generate the specifications for a contract, run the DeepSEA compiler with the `coq` option:

```
dsc <contract.ds> coq
```

This will create several new files in a subdirectory with the contract's name.

Example: `token.ds`

We'll walk through how to generate the Coq specifications for `contracts/token/token.ds` and run the Coq compiler `coqc` to check the spec. First, `cd` to the `contracts/token` directory. Before we generate the Coq files for this program, take a look in the `token` subdirectory (that is, in `contracts/token/token`). Notice that there are several Coq source files, with a `*.v` extension. These files were either entirely written by hand, for example `FunctionalCorrectness.v`, or previously generated by the DeepSEA compiler and completed manually, as in the case of `ObjFixedSupplyTokenCodeProofs.v`. They'll fail when checked by Coq, because they depend on some specifications we have yet to generate.Let's compile them now–in the `contracts/token` folder, run:

```
dsc token.ds coq
```

This creates several new files Coq in the `token` subdirectory, as well as a `_CoqProject` file, which tells Coq which files to check and where to look for dependencies. Because the DeepSEA compiler doesn't know that there are additional hand-written files to check, there are a few lines we need to append to `_CoqProject`:

```
# These lines are generated by dsc
-R ../../../ DeepSpec
-R . token
./EdsgerIdents.v
./DataTypes.v
./DataTypeOps.v
./DataTypeProofs.v
./LayerFIXEDSUPPLYTOKEN.v
./ObjFixedSupplyTokenCodeProofs.v
```

```
# Add these lines to tell Coq to look at our hand-written files:
./Invariant.v
./FunctionalCorrectness.v
```

We're almost ready to run Coq, but there's one more wrinkle: the DeepSEA compiler wanted to generate the file `ObjFixedSupplyTokenCodeProofs.v`, but it recognized that it already existed. So, instead of overwriting it, it appended to it. This is good, because this file contains manually filled-in proofs for admitted lemmas that the compiler had previously generated. However, in this case, it appended identical lemmas to the file which we need to remove to avoid a redefinition error from Coq. So, open this file and remove all the admitted lemmas (alternatively, if you cloned this repo, checkout the file with `git checkout ObjFixedSupplyTokenCodeProofs.v`).

We can now check the proofs:

```
coq_makefile -f _CoqProject -o Makefile
make COQEXTRAFLAGS='-w none'
```

The first line generates a `Makefile` to check the files we listed in `_CoqProject`; the second line runs `coqc` with a flag to suppress warnings (can be omitted).

Example: `amm.ds`

As a bigger example, we have implemented an constant-product automated market maker contract, similar to e.g. Uniswap, and proven a nontrivial theorem about it. The files are in `contracts/amm` directory, and the the `cst_man.v` file contains a formal proof of the fact that the cost of manipulating the market price of coins of type alpha is bounded away from zero. Hence this result disincentivizes any external attempts at manipulating prices in the Uniswap market setup. This theorem was previous proven using pen-and-paper by Angeris et al. (2019, https://arxiv.org/abs/1911.03380, the proof is in Appendix E), and now we give a formalized proof checked by Coq.

This section describes the instructions to download the executables of the Coq Interval library in order to successfully build the files containing the cost of manipulation proof for the AMM contract. Depending on the way your Coq installation works, follow the instructions given on the following page to install the Coq-interval library : http://coq-interval.gforge.inria.fr .

Note that the page includes a list of libraries that you would need in case you do not use OPAM to install Coq-Interval. The file containing the proof for the AMM contract additionally imports modules from the Coquelicot library.

The proof proceeds via a case analysis on the value of eps (the fraction by which the market price is manipulated). Corresponding to the two cases, we have the lemmas: `cst_func_0_1` (the proof for the case when eps is bounded by 1) and `cst_func_ge_1` (the proof for the case when eps is greater than or equal to 1).

Note that the constants K and kappa as used in the original paper have been approximated to account for algebraic operations involving irrational numbers. The proof for the first case uses the Taylor-Lagrange theorem as formalized in the Coq-Interval library (it is not a part of the treatment of Reals in the standard library). The proof for the second case involves an algebraic manipulation of the cost function to apply monotonicity properties established using properties of it's derivative.

Understanding the Specifications

The files `EdsgerIdents.v`, `DataTypes.v`, `DataTypeOps.v`, and `DataTypeProofs.v` should pass Coq's checking as-is and are intended to be left untouched, whereas the `Layer*.v` and `Obj*CodeProofs.v` files are meant to be filled in, though they should also pass unchanged.

The lemmas in `Obj*CodeProofs.v` combine several important statements about the behavior of each function of a DeepSEA program. For example, some state that integer overflow integer overflow doesn't occur, or that array indices are always in-bounds. These lemmas are assumed to be true in the other `core/*` and `backend/*` proof files, so filling in their proofs in an essential step in proving compilation correctness for a DeepSEA program.

The generated specifications do some of the heavy lifting of formally verifying a program, but of course they won't contain all the facts that one would want to prove about a program. Additional Coq files should be written that describe the expected behavior of methods. For the `token.ds` contract, look at `FunctionalCorrectness.v` for an example of this. The theorem `constant_balances_sum` asserts that the total number of tokens is fixed by asserting that the `transfer` method can neither create nor destroy tokens. Notice that this theorem uses some machinery from the generated specs: namely, `FixedSupplyToken_transfer_opt` from `LayerFIXEDSUPPLYTOKEN.v`.

The DeepSEA Coq generation currently doesn't include a "whole program semantics"–that is, a description of how the program state will change by calling any of the program's functions. See `contracts/amm/amm/prf.v` for an example of how such a specification could be written–in particular, the inductive definition `mstep`.

MiniC

MiniC is DeepSEA's intermediate representation, complete with its own compiler and pretty-printer. Developers can write their own frontend languages that use the DeepSEA backend by compiling to MiniC. MiniC is based on CompCert's Clight and largely follows C syntax.

Using the compiler

Installation

Add the appropriate `minicc` binary to a directory in your `PATH`. As with `dsc`, the Linux binaries have been tested on Ubuntu Linux and my not run properly on other distributions.

Synopsis

```
usage: minicc <source file> <mode>
```

Modes

- `print` : Print to MiniC source.
- `print-verbose` : Print to MiniC source with commented annotations.
- `assembly` : Compile to EVM mnemonics.
- `bytecode` : Compile to EVM bytecode.
- `bytecode-runtime` : Compile to EVM bytecode runtime.
- `ewasm`: Compile to Ethereum WebAssembly.
- `ewasm-runtime`: Compile to Ethereum WebAssembly runtime.

Running Unit Tests

The directory `minic/unittests` has two test scripts. The first, `test_parse_and_print.sh`, compiles some MiniC source files, prints back to MiniC, and then compiles and prints again. It then confirms that the two printed MiniC files are identical. Note that due to some variable renaming, whitespace, and comments, the source MiniC files likely won't match the printed ones exactly.

The second test, `test_bytecode`, requires that `dsc` is in your `PATH`. It compiles DeepSEA source files to bytecode and MiniC. Then, it compiles the MiniC to bytecode with `minicc` and compares it with bytecode generated by `dcs`.

MiniC by example

The following shows the example from the section DeepSEA by example as translated to MiniC. This, up to variable renaming, is nearly the output of compiling the DeepSEA example to MiniC using `dsc`.

```
int256 _bid;
int256 _bidder;
int256 _chair;
int256 _deadline;
int256#[int256] withdrawals;
```

```
/* initial values don't get used */
void constructor() {
    _bid = 0;
    _bidder = 0;
    _chair = 0;
    _deadline = 0;
}

void initialize(int256 deadline) {
    _bid = 0;
    _bidder = caller;
    _chair = caller;
    _deadline = deadline;
}

/* getter functions */
int256 getbid() {
    int256 bid;

    bid = _bid;
    return bid;
}

int256 getbidder() {
    int256 bidder;

    bidder = _bidder;
    return bidder;
}

int256 getchair() {
    int256 chair;

    chair = _chair;
    return chair;
}

int256 getdeadline() {
    int256 deadline;

    deadline = _deadline;
    return deadline;
}

/* place bid */
int256 bid() {
    int256 condition;
    int256 bidder;
    int256 bid;
```

```
    int256 deadline;
    int256 bidder_withdrawals;

    bidder = _bidder;
    bid = _bid;
    deadline = _deadline;
    bidder_withdrawals = withdrawals[bidder];
    condition = ((caller != bidder) && ((callvalue > bid) && (number < deadline)));

    if (condition) {
    } else {
        revert;
    }

    withdrawals[bidder] = (bidder_withdrawals + bid);
    _bidder = caller;
    _bid = callvalue;
    condition = 1;
    return condition;
}
```

Structure of a MiniC source file

A source file is made up of a series of comments, variable declarations, a constructor definition, and method/function definitions. These can occur in any order and need not be grouped together.

For a look at the OCaml data structure that represents a MiniC file, see the next section.

Comments

```
/* Comments follow C syntax,
   and can be multiline */

// Or single line
```

Variable declarations

Variables are defined with the syntax

`<type> <ident>;`

where `<ident>` is a C-identifier, and `<type>` is one of the following:

- `int8` : An unsigned 8-bit integer.
- `int16` : An unsigned 16-bit integer.
- `int32` : An unsigned 32-bit integer.
- `int256` : An unsigned 256-bit integer.

- bool : A boolean.
- void : A dummy type.

There are also four composite types:

- <type>[<int>] : An array of <type> of size <int>.
- <type1>#[<type2>] : A hashmap from <type1> to <type2>.
- struct <ident> { <type> <field_ident>; ... } : A struct with a list of named fields.
- union <ident> { <type> <field_ident>; ...} : A union with a list of named fields.

Function definitions

Constructor, method, and function definitions all follow the same syntax, except that the name of the constructor must be constructor, and functions are marked with a private keyword before their return type. Optionally, methods can be marked with a public keyword.

Parameters are declared after the function name and before the function body. The body begins with a list of declarations of temp variables, followed by a series of statements. The temp declarations must precede all other statements; this differs from C syntax, where variables can be declared mid-function.

```
private int256 addNums(int256 a, int256 b){
  int256 sum;  // temp variable declarations

  sum = a + b; // statements
  return sum;
}
```

Expressions

Integer literals MiniC currently only supports decimal literals.

```
myint = 42;
```

Binary Operations MiniC has the following binary operations that take an expression on either side:

```
e1 + e2    // addition
e1 - e2    // subtraction
e1 * e2    // multiplication
e1 / e2    // division
e1 % e2    // modulo
e1 ** e2   // exponentiation
e1 & e2    // bit and
```

```
e1 | e2    // bit or
e1 ^ e2    // bit xor
e1 << e2   // bit shift left
e1 >> e2   // bit shift right
e1 == e2   // equality
e1 != e2   // not equal
e1 < e2    // less than
e1 > e2    // greater than
e1 <= e2   // less than or equal
e1 >= e2   // greater than or equal
e1 >= e2   // greater than or equal
e1 sha2 e2 // Keccak-256 hash
```

Unary Operations These operations precede an expression:

```
!e     // bool not
~e     // int not
-e     // negative
sha1 e // Keccak-256 hash
```

Note that whitespace is required around `sha1` and `sha2` if they directly precede or follow an identifier or integer literal. Whitespace is optional after other binary and unary operations.

Ethereum-specific built-in expressions The following built-in keywords correspond to the DeepSEA ones as discussed earlier.

```
                // Solidity equivalent:
address         // address(this)
origin          // tx.origin
caller          // msg.sender
callvalue       // msg.value
coinbase        // block.coinbase
timestamp       // block.timestamp
number          // block.number
chainid         // chainid (assembly)
selfbalance     // selfbalance (assembly)
balance(<addr>) // addr.balance
blockhash(<n>)  // blockhash(n)
```

Statements

The smallest type of statement is a semicolon-terminated statement:

```
e1 = e2;                          // assignment
break;                            // break out of loop
return;                           // return from function
revert;                           // revert function
transferEth(e1, e2);              // transfer e2 Wei of Ether to address e1
```

```
emit(e11, e12, ... ; e21, e22, ...); // emit events
<func>(e1, e2, ...);                 // call function <func>
<var> <- <func>(e1, e2, ...);        // call function <func> and
                                     //   store return value in <var>
callmethod(addr;                     // Call an external contract at addr
           var_1, var_2, ... ;       //   store return vals in vars
           sig;                      //   function signature
           value;                    //   Amount to send to contract, in Wei
           arg_1, arg_2, ... ;       //   function arguments
          )
```

These can be grouped together to create larger statements of the following forms:

```
if (e)                      // If-then-else conditional
  { then_statements }       // with expression e
  else { else_statements }
while { stmt1; stmt2; ... } // Infinte loop
```

A while loop will go on until it reaches a `break` statement. As in C, if-then-else constructs don't need braces around the if-clause or else-clause if they're followed by a single statement. Likewise, the else-clause is optional.

The MiniC Datatype

A MiniC file is stored in the datatype `genv` as defined in `backend/Language.ml`. The full type expression is `(coq_function, coq_type) Genv.t`. Expanding this down further, we have the following typedefs:

```
(* backend/Language.ml *)
type coq_function = { fn_return : coq_type;
                      fn_params : (ident, coq_type) prod list;
                      fn_temps : (ident, coq_type) prod list;
                      fn_body : statement }

(* backend/Ctypes.ml *)
type coq_type =
| Tvoid
| Tint of intsize * signedness
| Tpointer of coq_type
| Tarray of coq_type * coq_Z
| Thashmap of coq_type * coq_type
| Tfunction of typelist * coq_type
| Tstruct of ident * fieldlist
| Tunion of ident * fieldlist
| Tcomp_ptr of ident
and typelist =
| Tnil
| Tcons of coq_type * typelist
and fieldlist =
```

```
| Fnil
| Fcons of ident * coq_type * fieldlist
```

```
(* backend/Globalenvs.ml, in module Genv *)
type ('f, 'v) t = { genv_vars : ident list; genv_funcs : ident list;
                    genv_methods : Int.int list; genv_defs : 'v PTree.t;
                    genv_fundefs : 'f PTree.t;
                    genv_methoddefs : 'f option IntMap.t;
                    genv_constructor : 'f option }
```

Even without a deep understanding of this data structure, we can see that it closely mirrors the grammar as described in the previous section. A MiniC file consists of global variables, functions, methods, and a constructor. Global variables and functions are stored by an identifier `ident`, which is of type `BinNums.positive`. The fields `genv_vars` and `genv_funcs` keep track of these `idents`. Methods are cataloged by a signed integer `Int.int`, and `genv_methods` lists these integers. The fields `genv_defs`, `genv_fundefs`, and `genv_methoddefs` store information about global variables, functions, and methods, respectively, in tree structures.

The `prod` and `list` types in the code above functionally equivalent to OCaml's product and list types; however, they need to be translated into their OCaml counterparts before being used as such. The functions `caml_prod` and `caml_list` in `DatatypesExt.ml` will convert them to the OCaml types.

A `coq_function` has a return type, param and temp variables, and a statement. The statement actually likely encapsulates multiple statements, because the statement `Ssequence` is the product of two other statements. So, a statement is actually a tree of statements.

```
(* backend/Language.ml *)
type expr =
| Econst_int of Int.int * coq_type
| Econst_int256 of Int256.int * coq_type
| Evar of ident * coq_type
| Etempvar of ident * coq_type
| Ederef of expr * coq_type
| Eunop of unary_operation * expr * coq_type
| Ebinop of binary_operation * expr * expr * coq_type
| Efield of expr * ident * coq_type
| Earrayderef of expr * expr * coq_type
| Ehashderef of expr * expr * coq_type
| Ecall0 of builtin0 * coq_type
| Ecall1 of builtin1 * expr * coq_type

type statement =
| Sskip
| Sassign of expr * expr
```

```
| Sset of ident * expr
| Scall of ident option * label * expr list
| Ssequence of statement * statement
| Sifthenelse of expr * statement * statement
| Sloop of statement
| Sbreak
| Sreturn of expr option
| Stransfer of expr * expr
| Scallmethod of expr * ident list * Int.int * expr * expr list
| Slog of expr list * expr list
| Srevert
```

Writing a frontend for MiniC

Language developers can write a frontend language that compiles to MiniC and the DeepSEA backend. To do so, input files should be parsed into the `genv` datatype. Then, the following functions are available to compile `genv`:

- `LanguageExt.show_genv v nt genv` : Print to MiniC. Set verbose `v` to `true` to print type and name annotations as comments. Optionally, supply a name tables struct `nt` of type `NameTablesExt.name_tables` so that original function and variable names can be printed in verbose output. Otherwise, pass an empty name tables struct `NameTablesExt.empty_name_tables`.
- `BytecodeExt.bytecode runtime genv` : Compile to EVM bytecode. Set `runtime` to `true` to generate the runtime version.
- `BytecodeExt.ewasm runtime genv` : Compile to Ethereum WebAssembly. Set `runtime` to `true` to generate the runtime version.
- `BytecodeExt.assembly genv` : Compile to EVM mnemonics.

All the functions above return `string`.

See the directory `opensc` for an example of how to parse a frontend language to MiniC.