DeepSEA Language Reference

Preview release, July 27, 2020.

DeepSEA/EVM version 1.0

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 1.0?

The language described in this manual is "version 1.0". 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. In the immediate future (late 2020/early 2021) we will improve the language to expose all the functionality of the Ethereum network and provide better support for general-purpose programming. The details will be worked out in more detail, but it 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 modelling 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.

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. For now, this is for debugging only; the MiniC itself is not executable. This will be added in a future update.
- `coq` : Create a directory with a gallina model of the contract.
- `ewasm`: Compile to Ethereum WebAssembly

B The coq option is not supported in the preview release.

Running Unit Tests

The directory *unittests* contains a few small programs which test particular features of DeepSEA. Each of them also comes with a small javascript program which runs them. There are two slightly larger examples in the *contracts* directory.

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 and some packages from npm:

- ethers.js
- ganache-cli

Ubuntu / Debian The following will configure npm to install global packages in ~/.npm-global

```
sudo apt install nodejs npm
mkdir ~/.npm-global
npm config set prefix "~/.npm-global"
export PATH=$PATH:~/.npm-global/bin
```

You'll probably want to configure your shell to add ~/.npm-global/bin to your $PATH on startup.

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
```

Once the npm package manager is installed, the command

```
npm install ethers ganache-cli -g
```

will install the required packages.

To exercise a test, first start running `ganache-cli` in a separate terminal, and then compile the .ds program and give the resulting bytecode and abi file to the test script, e.g.:

```
$ dsc forloop.ds bytecode > forloop.bytecode
$ dsc forloop.ds abi > forloop.abi
$ ./forloop_test.js
Testing function calls...
multiply: pass
multiply: pass
multiply: pass
$
```

The test script will use ethers.js to connect to the ganache server to run the bytecode.

To compile and run all the tests:

```
./run_unittests.sh
```

Running a contract locally in ganache-cli

The infrastructure for the unit test scripts are also a convenient way to experiment with programs written in DeepSEA. When the computer is set up to run the test scripts, it is also easy to modify the script to run another program. The javascript contracts/token/test.js is a convenient starting point. In order to adapt it to run another program, there are two changes to make:

- Change the body of the `deploy` function to call methods in your program.
- Change the line `const abi = ...` to specify the type of the functions in your program. This value can be generated by the DeepSEA compiler's `abi` mode.

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 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

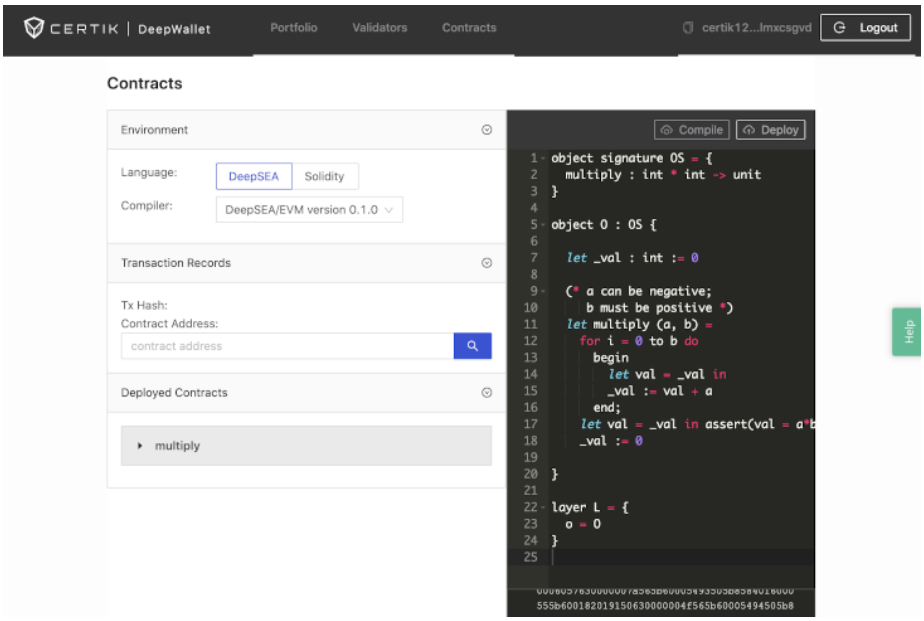

Figure 1: Contract deployment via DeepWallet

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 seperately.

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 statemetns 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 teh contract. The scipt 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
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

Configuration item	Description	Example value
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

B The DeepSEA WebAssembly support is currently experimental and under development. In particular, arrays, hashmappings, and keccak-256 hashes will not compile correctly.

To use the WebAssembly backend of DeepSEA, you can simply use the DeepSEA binary `dsc` with the option `ewasm` (or `ewasm-runtime` if you want to generate the runtime version of contracts).

```
dsc example.ds ewasm > example.wat && wat2wasm example.wat
```

This will compile the DeepSEA contract `example.ds` into `example.wasm`. The assembler we use, `wat2wasm`, is in the WebAssembly toolkit `wabt` which is maintained by the WebAssembly foundation (required for running DeepSEA WebAssembly)

Currently DeepSEA supports Ethereum-flavored WebAssembly ewasm, which is used by Ethereum 2.0 and Hyperledger Burrow.

Here is a simple demo (running in a Javascript VM):

```
cd contracts/wasmtest/for
dsc ./for.ds ewasm > ./for.wat #generate human readable WebAssembly text representation
wat2wasm ./for.wat             #assemble the text representation to binary
cd ..
./for/for.js ./for/for.wasm multiply 4 8 #gives result 32, using ethereum storage to compute
```

DeepSEA by example

To give the flavor of the language, we show a contract which implements a simple auction system. (Or parts of it, the function to withdraw ether has been omitted for now.)

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 = {
  initialize  : uint -> unit;
  getbid      : unit -> uint;
  getbidder   : unit -> address;
  getchair    : unit -> address;
  getdeadline : unit -> uint;
  bid         : 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

  (* placeholder for constructor *)
  let initialize 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
}

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 *) ddd
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 seperate 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 In the current 1.0 version of the compiler, there is a limitation that mapping types can only have a single key. This will be fixed in upcoming releases, but in the meantime it can by worked around by adding an extra struct. For example, in Solidity a mapping with two keys can be defined asa

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

In the current DeepSEA version, one can get the same effect by writing

```
type Wrapper = {
  val: mapping[address] uint
}

let _allowed : mapping [address] Wrapper := mapping_init
```

One of our example contracts, `Olive.ds`, uses this method.

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 interactiving 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.

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 distinguished 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 *)
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
```

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);
```

Release History and Changes

- Version 1.0
  - More complete language implemention
    - Declaring and emitting ethereum events
    - Function calls to lower layers
    - Expose the ethereum builtin opcodes (callvalue etc)
    - Variable initializers and contract constructors
    - assert statements (to revert contract execution)
    - address type as a primitive
  - More example contracts (Olive.ds, ...)
  - Better error messages (line numbers, informative errors from the backend)
  - Partial support for compilation to eWasm.
  - Support for deployment on Ant Blockchain

- Version 0.1
  - Initial preview release