Cardano Goguen | IELE Testnet

IELE Virtual Machine Design

The design of IELE was based on Runtime Verification experience with formally defining dozens of languages in K, but especially on recent experience and lessons learned while formally defining two other virtual machines in K, namely:

- KEVM, our semantics of the Ethereum virtual machine (EVM); and
- KLLVM, our semantics of LLVM. the latest version of the LLVM semantics will be made public when complete and published, but an earlier version is available.

Unlike the EVM, which is a stack-based machine, IELE is a register-based machine, like LLVM. IELE also directly supports functions, like LLVM. It has an unbounded number of registers and also supports unbounded integers. There are some tricky but manageable aspects with respect to gas calculation, a critical part of the design.

We first discuss the rationale behind the design of IELE, then the major design decisions from the perspective of differences from EVM and LLVM, respectively.

Design Rationale

Here are the forces that drove the design of IELE:

1. To serve as a uniform, lower-level platform for translating and executing smart contracts from higher-level languages. The contracts can also interact with each other by means of an ABI (application binary interface). The ABI is a core element of IELE, and not just a convention on top of it. The unbounded integers and unbounded number of registers should make compilation from higher-level languages more straightforward and elegant and, looking at the success of LLVM, more efficient in the long term. Indeed, many of the LLVM optimizations are expected to carry over. For that reason, IELE followed the design choices and

representation of LLVM as much as possible. Of course, our partnership with the University of Illinois, where LLVM was created, was a great advantage in creating IELE.

2. To provide a uniform gas model, across all languages. The general design philosophy of gas calculation in IELE is 'no limitations, but pay for what you consume'. For example, the more registers an IELE program uses, the more gas it consumes. Or the larger the numbers computed at runtime, the more gas it consumes. The more memory it uses, in terms of both locations and size of data stored at locations, the more gas it consumes. And so on.
3. To make it easier to write secure smart contracts. This includes writing requirements specifications that smart contracts must obey as well as making it easier to develop automated techniques that mathematically verify smart contracts correct with respect to such specifications. For example, pushing a possibly computed number on to the stack and then jumping to it regarded as an address makes verification hard, and thus security weaker, with current smart contract paradigms. IELE has named labels, like LLVM, and jump statements can only jump to those labels. Also, avoiding the use of a bounded stack and not having to worry about stack or arithmetic overflow makes specification and verification of smart contracts significantly easier.

Like KEVM, the formal semantics of EVM that we previously defined, validated and evaluated using the K framework, the design of IELE was also done in a semantics-based style, using K. Together with a fast (LLVM-based) execution backend for K that is still under development, it is expected that the interpreter obtained automatically from the semantics of IELE will be sufficiently efficient to serve as a reference implementation of IELE.

Design Changes Relative to EVM

Here are the IELE design decisions described from the perspective of differences from EVM.

Registers

IELE is a register-based bytecode language, unlike EVM, which is a stack-based bytecode. As a result, the following alterations have been made:

- Each instruction in the bytecode takes register operands representing the arguments and result of the instruction.
- There is an unlimited number of registers available and they can be referred to by user-defined names, similar to the virtual registers of LLVM. The more registers a contract uses, the more gas it consumes.
- Because the stack no longer exists, it is no longer possible for the VM to throw an exception due to stack underflow or overflow.
- Because the stack no longer exists, the POP, DUP, SWAP, and PUSH instructions no longer exist. They are replaced with an assignment instruction that can be used to load a constant value into a register (see section on arbitrary-precision integers for more details), and/or copy a value from one register to another.
- Registers are visible only within the function in which they are used.
- Registers have a lifetime equal to the lifetime of a function call to the function that contains them.

Program Structure

A IELE program consists of a list of contracts, where contracts later in the list can create accounts with deployed contracts found earlier in the list. Unlike EVM, in which a contract is a sequence of instructions, an IELE contract consists of a header giving the contract a name, followed by one or more function definitions and, optionally, one or more external contract declarations.

- An external contract declaration simply declares the name of another contract. The contract being defined will only be able to create accounts with deployed code that is a copy of one of the contracts that have been declared as external, or a copy of an existing account's code (see section on contract creation for details). Each externally declared contract should have been defined in the same file and above the contract that externally declares it.
- A function definition includes the function signature, the function body and whether or not the function is public. A function signature includes a function name (which is prefixed by the @ symbol) and names of formal arguments (which are represented as local registers, as they are only visible within the function body).

- A public function can be called by other accounts, while a non-public one can only be called by other functions within the same contract.
- A special private function named `@init` should be defined for any contract and will be called when an account is created with this contract. If this function is not defined the contract is malformed.
- An account to which code has never been deployed contains an implicit public function `@deposit` which takes no arguments, returns no values, and does nothing. This function exists to allow accounts to receive payment even if they do not have a contract deployed to them, and functions analogously to calling an account with no code in EVM. Note that a contract can forbid payments by refusing to declare the `@deposit` function, and explicitly raising an exception if any of its entry points are invoked with a balance transfer. A contract can also define code to be executed when it is paid by defining its own deposit function.
- A global definition defines a global name and its constant value (which is an unbounded signed integer). Globals are accessible from within any function of the contract and their value cannot be modified.
- The contract is malformed if it contains function and/or global definitions with the same name, or if it references a global that has not been declared.

Static Jumps

IELE no longer has dynamic jump instructions. They are replaced with jumps that take a named label as an argument, and with a call/return mechanism to allow function calls that return to the caller.

- The code within a function contains named labels. Any instruction can be prefixed by a named label that makes this instruction a valid target of a jump.
- The `br` instruction has two variants. It can be used with a named label as a single argument for unconditional jumps within a function body, or with an additional register holding a condition value for conditional jumps when the condition value is not zero. The instruction prefixed by the named label is the target of the jump. Therefore all jump targets can be known statically.
- The contract is malformed if labels within the same function have the same name.

- The contract is malformed if the label argument of the `br` instruction is not found as a label in the function containing the instruction. This is unlike the corresponding behavior in EVM, where an exception is thrown if the argument of JUMP or JUMPI is not the program counter of a JUMPDEST instruction.

Function Call/Return

IELE function calls take an arbitrary number of register arguments and return values instead of a memory range.

- IELE has a `call` instruction used for local calls to other functions within the contract. `call` takes a function name, an arbitrary number of arguments, and an arbitrary number of register return values, and jumps to the start of the function in the current contract with the corresponding name. It pushes the current instruction position, the values of all local registers, and the register operands of the return values onto the local call stack. The arguments of the call are copied into local registers corresponding to the formal arguments of the function being called. A contract is malformed if the number of results in a local `call` does not match the number of results returned by the target function.
- IELE has a `call .. at` and a `staticcall .. at` instruction used for account calls to public functions of other contracts. The target account address is a register operand of these instructions, as is the amount of gas to be spent during the call, and `call .. at` has an additional operand for the balance to be transferred with the call. Also, `call .. at` and `staticcall .. at` take an arbitrary number of arguments and return values in the form of register operands instead of memory ranges. The arguments are copied into local registers corresponding to the formal arguments of the function being called. The first register operand in the list of return values is the status register. This register will hold the exit status code upon returning from the call. This means that the list of return value registers given to a `call .. at` or `staticcall .. at` instruction should be long enough to accept the correct number of values expected to be returned by the callee plus the exit status code.
- The `ret` instruction takes an arbitrary number of register operands holding returned values. `ret` will return to the instruction position on the top of the local call stack if one exists, restoring the values of local registers and copying the returned values into the return value operands of the call instruction. If the local call stack is empty, it returns from the contract to the callsite of the caller account code with an exit status code of `0` in

addition to the given returned values. A contract is malformed if any function contains `ret` instructions with different numbers of arguments.

- The `revert` instruction takes a single register operand. `revert` always returns from the contract to the callsite of the caller account code, independently of the size of the local stack. It refunds unused gas and returns a value in the first register return value of the call (that is the exit status register), but does not write to the other return values, and it also rolls back the state changes of the contract.
- A static account call, done with `staticcall .. at`, similarly to EVM, should not attempt to change any state in the network (eg, log, account storage, account creation/deletion) during its execution. If it does, an exception is thrown. (Avoiding state changes also means that no balance can be transferred with the `staticcall .. at`, which is why `call .. at` takes one additional parameter.)
- If the name of the function being called does not correspond to a public function of the contract being called, an exception is thrown.
- In all cases, if a mismatch occurs between the number of arguments or return values of a function and the matching `call` or `ret` instruction, an exception is thrown.
- When an exception is thrown during the lifetime of an account call, the call is terminated and control returns to the callsite of the caller account code, similarly to a `revert`. Also in a similar fashion to `revert`, an exit status code is returned in the first register return value of the call (that is the exit status register), while nothing is written to the other return values, and the state changes of the contract are rolled back. However, unlike `revert`, no unused gas is refunded.
- Following is a comprehensive list of exit status codes:
    - 0: success
    - 1: function does not exist
    - 2: function has wrong signature
    - 3: function does not exist on empty account
    - 4: execution of instructions led to failure
    - 5: out of gas

    - 6: deploying to an account that already exists
    - 7: insufficient balance to transfer
    - 8: negative balance or gas limit or call depth exceeded
    - 9: contract being uploaded to blockchain is not well formed
- Because they are no longer needed, we remove EVM's CALLDATA* instructions and RETURNDATA* instructions.
- Because of security concerns, we remove EVM's CALLCODE and DELEGATECALL instructions. We provide a way to create many contracts that share code more cheaply.

Arbitrary Precision Words

Unlike EVM, which uses 32-byte unsigned words, IELE has arbitrary-precision signed words.

- Because words are now explicitly signed, the SDIV, SMOD, SLT, and SGT instructions are removed, and DIV, MOD, LT, and GT are replaced with signed versions.
- Added the `expmod` instruction to calculate the modulo exponentiation natively. So the old MODEXP contract is no longer necessary.
- a `twos` instruction is added to convert a signed integer into an `N`-byte twos-complement representation of the number, where `N` is passed as an argument to the instruction.
- `sext` (corresponding to EVM's SIGNEXTEND) is changed to convert an N-byte twos-complement representation of a signed number into its signed value, where `N` is passed as an argument to the instruction.
- Because integers are unbounded, the index operand of `byte` (corresponding to EVM's BYTE) now counts from the least-significant byte instead of the most-significant.
- In order to reduce gas costs for multiplication and division by powers of two, we introduce an explicit bitwise `shift` operator. It takes two arguments; the second is positive for left shift and negative for right shift.

Local Execution Memory

Unlike EVM, which uses a 2^256-cell array of bytes as local execution memory, IELE's local execution memory is a 2^256-cell array of arbitrary-length byte buffers.

- Unbounded signed integers are stored with their most-significant byte first (at offset 0 within the cell).
- There are two variations of the `load` and `store` instructions, one that accesses the whole contents of a cell and interprets them as an unbound signed integer; and one that accesses a subrange of the bytes stored in a cell.
- Similar to EVM, the local execution memory is cleared when the currently executing contract returns.

Account Storage

IELE's account storage is an unbound array of unbound signed integers, unlike EVM's storage, which is a 2^256 cell array of 256-bit words. Similar to EVM, IELE's account storage (unlike the local execution memory) is persistent over contract calls and returns, and is only cleared when the corresponding account is destroyed.

Instructions and Precompiled Contracts

Unlike EVM, where each instruction is just an opcode operating on the values found on top of the stack, IELE instructions follow a syntax similar to LLVM: each instruction is an opcode followed by some arguments in the form of registers. If the instruction produces a result, it is formed as an assignment where the LHS contains the register(s) to hold the result(s) and the RHS contains the opcode and the arguments. Other changes with respect to EVM include:

- IELE has one integer comparison instruction, namely `cmp`, that accepts various predicates. In addition to the predicates `lt`, `gt`, and `eq`, which have corresponding EVM instructions, the predicates `le`, `ge`, and `ne` are also supported.
- The IELE instructions for storing in the local execution memory (`store`) and account storage (`sstore`) accept two arguments, where the first argument is the value to be stored and the second argument is the

index of the memory cell to be updated. This is opposite from the order expected on the stack in the corresponding EVM instructions, but similar to the order used in the corresponding LLVM instruction.

- Various EVM instructions that query local and/or network state (eg, GAS, BALANCE, etc) have been replaced by IELE built-ins that can be called using the `call` instruction (eg, `%balance = call @iele.balance(%bank.account)`). The names of all IELE built-ins start with the prefix `@iele`, following a convention similar to the LLVM intrinsics. All names for global functions starting with this prefix are reserved: A contract is malformed if it contains user-defined registers and/or functions with names starting with `@iele`.
- Precompiled contracts in IELE are also invoked using corresponding built-ins (eg, `@iele.ecpairing`), but should be called using the `call .. at` instruction and targeting the account with address `1` (eg, `%res = call @iele.sha256 at 1 ( %len, %data ) send %val, gaslimit %gas`).

Contract Creation

Unlike EVM, both for ease of verification and due to security concerns, we do not allow smart contracts to create accounts from dynamically computed code. Accounts can be created only in two ways, either dynamically from code that is part of a previously validated contract or by a top-level blockchain transaction.

- The `create` instruction is modified from its EVM counterpart. Instead of taking a memory start address and a memory width containing initialization code, the `create` instruction takes a contract name, and an arbitrary number of register arguments to be passed to the `@init` function of the contract to be created. If the contract name is not an externally declared contract name, the contract is malformed.
- Each externally declared contract should have been defined in the same file and above the contract that externally declares it, otherwise the contract is malformed. As a result of this requirement, the exact code that will be deployed by a `create` instruction is statically known and available.
- The `create` instruction copies the corresponding contract code into the code storage of the account that contains the created contract. The copied code includes the code of the contract to be deployed along with the code of every contract listed above it in the file (in order to allow the new account to create contracts on

its own in the same way). Then the `@init` function is invoked in the context of the created account to initialize the contract. It is passed the values in the registers as arguments in the same fashion as a `call .. at` or `staticcall .. at` instruction. If the `@init` function of a contract returns a value, the contract is malformed.

- A separate instruction, `copycreate`, is introduced for the case of copying an entire contract, or duplicating your own contract. The `copycreate` instruction behaves like `create`, except that instead of taking a contract name, it takes a register operand containing an account address, and deploys the entire contract deployed to said account to the newly created account.
- Both instructions return an exit status code and the address of the created contract on success or zero on failure.
- Because we no longer actually need to address portions of code by program counter when creating contracts, we remove EVM's CODECOPY and EXTCODECOPY instructions. Because these are also the only remaining instructions that require the code to be addressed by a particular byte offset, we also remove EVM's PC instruction.
- Transactions that create a contract now explicitly provide the binary data of the contract instead of the binary data of the initialization function. Contracts created in this way still invoke the `@init` function to initialize the contract. If the contract is malformed, the transaction fails.

Because accounts created by `create` and `copycreate` only use code that already exists on the blockchain, they do not have a charge based on code size. This allows for cheap code reuse, without unsafe operations like DELEGATECALL or an explicit notion of shared library code.

Gas Model

IELE provides a uniform gas model across all programming languages. And unlike EVM, the amount of used memory in IELE can decrease when memory cells are deallocated or resized.

Our general design philosophy for gas calculations is 'no limitations, but pay for what you consume'. Therefore, gas is consumed either when you increase the amount of memory being used, or by the computational effort to execute instructions.

So, the gas cost of an instruction is the cost incurred from the memory used by the instruction plus the computational cost of executing the instruction.

Choices in IELE are similar to those for EVM:

- Each contract has a default amount of memory within which to execute.
- Memory over the limit incurs a quadratic allocation cost.
- Memory is charged only at allocation time.

Memory consumed is tracked to determine the appropriate amount of gas to charge for each operation.

Learn more about the gas model at iele-gas.md

Design Changes Relative to LLVM

Here are the IELE design decisions described from the perspective of differences from LLVM.

Type System

Unlike LLVM, which uses finite-precision words as values of its various integer and floating point types, IELE values are arbitrary-precision words. Moreover, we decided to support initially only one type, `int`, describing arbitrary-precision integer values. This suffices to capture the functionality of EVM in terms of values. In the future we may add a type system supporting aggregate types (such as arrays and structures) as well as (higher-order) function types, similar to the LLVM type system.

Cardano Goguen | IELE Testnet

Static Single Assignment (SSA) Form

IELE registers are not in static single assignment (SSA) form, meaning that they can be (statically) assigned more than once and that there is no need for IELE phi functions. This departure from the LLVM SSA style was dictated by the fact that IELE, as the target interpreted language for smart contracts, should allow programs to minimize the number of registers they use and so reduce the consequent gas costs. A compiler that generates IELE can still use an SSA IELE representation internally to ease optimization, but we do not enforce SSA in the final IELE program. This means compiler and/or hand-written code can be used to optimize their register use.

Control Flow

We decided to relax the LLVM restrictions about basic block structure, where code is organized as a control-flow graph of maximal basic blocks with explicit and statically defined successors/predecessors. In IELE, we maintain static labels as the only allowed targets of jumps, but also allow execution to fall through from one block to the next, whereas in LLVM the first instruction of a basic block can only be reached through a jump, and every LLVM block must end with a control flow instruction. We made this decision to minimize the size of IELE programs, because the block structure restrictions in LLVM programs usually result in additional branch instructions.

Bitwise Shift

Because IELE has arbitrary-precision signed integers, bitwise shifts are expressed in terms of a signed shift amount instead of separate shr and shl instructions.