Compiling a Higher-Order Smart Contract Language to LLVM

Vaivaswatha Nagaraj, Jacob Johannsen, Anton Trunov, George Pîrlea, Amrit Kumar and Ilya Sergey

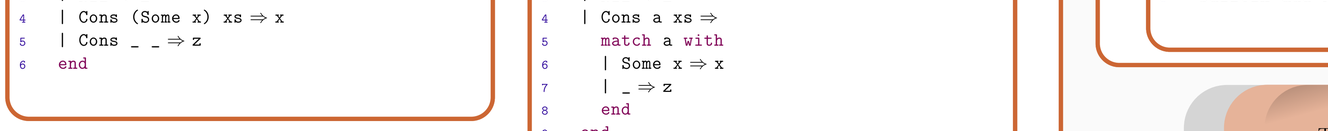
Zilliqa Research

Scilla - Smart Contract Intermediate Level Language

- DSL for blockchains: Used to write smart-contracts.
- Typed, higher-order, polymorphic language in the ML family.
- Currently interpreted in production. This work explores its compilation.

The Compiler and Runtime

- The compiler, written in OCaml (using LLVM's OCaml bindings), shares the parser and type-checker with the existing interpreted implementation.
- The runtime implements language built-in operations and provides an interface b/w the JIT'ed code and the blockchain.

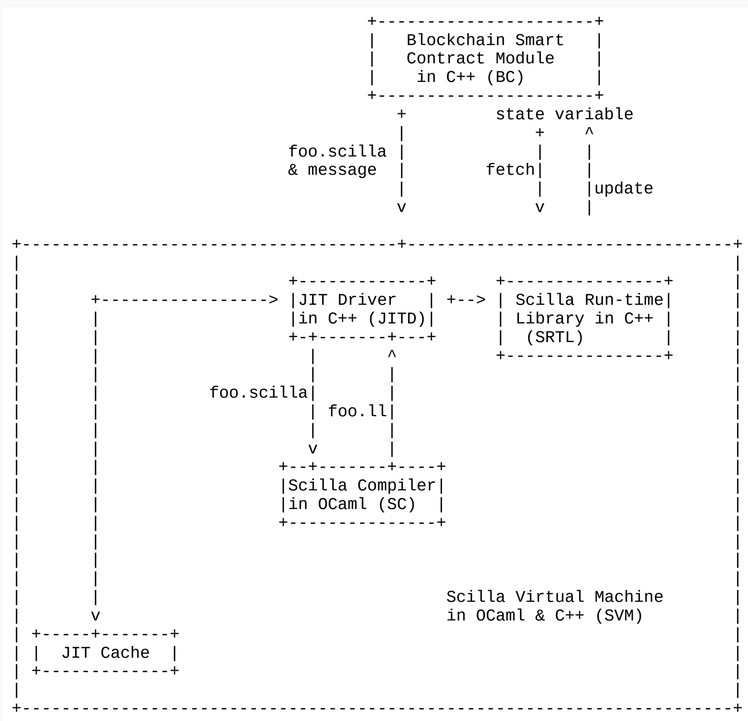


Experimental Results

In comparison to the reference interpreter:

- Synthetic tests like Ackermann function computation and Church numerals show improvements from 47x to more than 100x.
- Production benchmarks show improvements ranging from 5x to 10x.

System Architecture



Part I: Front-End Transformations

Pattern Match Flattening

- Nested patterns are flattened into nested pattern matches.
- Enables direct translation to efficient LLVM switch statements.

Before

```
fun (p : List (Option Int32)) ⇒
  match p with
  | Nil ⇒ z
  | Cons (Some x) xs ⇒ x
  | Cons _ _ ⇒ z
  end
```

After

```
fun (p : List (Option Int32)) ⇒
  match p with
  | Nil ⇒ z
  | Cons a xs ⇒
    match a with
    | Some x ⇒ x
    | _ ⇒ z
    end
  end
```

Uncurrying

- Scilla uses curried semantics. Transform the AST to be uncurried.
- Transforms nested sequence of functions to a single function with multiple arguments.

Before

```
fun (x : Int32) ⇒
fun (y : Int32) ⇒
  builtin add x y
```

After

```
fun (x : Int32, y : Int32) ⇒
  builtin add x y
```

The syntax used in these examples is not always legal Scilla syntax.
We don't have a textual syntax for every intermediate form.

Monomorphization

- Parametric polymorphism is implemented via monomorphization (as against type erasure).
- A higher-order type-flow analysis (conservatively) computes the possible ground types that may flow into a type variable.
- Polymorphic expressions are specialized with all types computed by the analysis.

Before

```
let option_to_bool =
  tfun 'A ⇒
    fun (a : Option 'A) ⇒
    match a with
    | Some _ ⇒ True
    | None ⇒ False
in
let otb_i32 = @option_to_bool Int32 in
let otb_string = @option_to_bool String in ...
```

After

```
let option_to_bool =
  [ Int32 →
    fun (a : Option Int32) ⇒
    match a with
    | Some _ ⇒ True
    | None ⇒ False
  ; String →
    fun (a : Option String) ⇒
    match a with
    | Some _ ⇒ True
    | None ⇒ False ]
in
let otb_i32 = option_to_bool[Int32] in
let otb_string = option_to_bool[String] in ...
```

Closure Conversion

- In Scilla, functions can be nested: inner functions have access to the **free** variables that are defined outside their bodies.
- Closure conversion eliminates free variables by collecting them in **environments** taken as a function's additional argument.
- The modified function with no free variable accesses is lifted to global level (thus translatable to an LLVM function).
- Every reference to the function is now a pair, of the function's pointer and its environment.

Before

```
let f1 =
  let y = Int32 1 in
  fun (x : Int32) ⇒
    builtin add x y
in ...
```

After

```
let f2 =
  fun (e : {y : Int32}, x : Int32) ⇒
  let y = e.y in
  builtin add x y
in
let f1 =
  let y = Int32 1 in
  let e = new {y : Int32} in
  e.y ← y;
  {f2, e}
in ...
```

Part II: Mapping Scilla to LLVM-IR

Type Descriptors

- Type Descriptors are data structures defined in the runtime library, objects of which are created in the JIT'ed code.
- A global array of type descriptors, with an entry for every type that occurs in the Scilla program, with each type identified by its index in the array, is inserted by the compiler.
- Some functions in the runtime library (such as the JSON (de)serializer) that can operate on different Scilla types, take in a type descriptor argument.

Primitive Types

- Integer types are wrapped with an LLVM struct type to differentiate signedness. For example the Scilla type `Int32` becomes the LLVM type `%Int32 = type { i32 }`.
- `String` and `ByStr` (bytes of arbitrary length) types are translated to LLVM types `%String = { i8*, i32 }` and `%Bystr = { i8*, i32 }`.
- Fixed sized byte sequences `ByStr`X (i.e., X is known statically) are expressed as LLVM array types `[` $X \times$ `i8 ]`.

Maps

- Map values are boxed, handled through an opaque pointer.
- The runtime library creates and operates on Map values.
- Internally, the runtime library uses `unordered_map<string, any>`. `any` is used to enable representing values of any Scilla type, including nested maps and user-defined types.

Algebraic Data Types

- Algebraic data types, aka variants or tagged unions, are composite types that have one or more "constructors" (sum type), each defining a tuple (product type).
- A canonical example of `List` defined in Scilla, with two constructors `Cons` and `Nil`:

```
type MyList =
| Nil
| Cons of Int32 MyList
```

- ADT objects are boxed, *i.e.* represented using a pointer.

The compiler generates three LLVM types for the `List` ADT:

- `%tname = type { i8, ... }`
- `%nil = type <{ i8 }>`
- `%cons = type <{ i8, Int32, %tname* }>`

The `i8` field (called the tag) in `%tname` determines the constructor used. No object of type `%tname` is built. Only pointers to it are type-casted to the right constructor before use. Packed LLVM structs are used to simplify access in the runtime library.

Closures

- A closure is represented by an anonymous struct `type { fundef_sig*, void* }`.
- `fundef_sig` is the signature of the LLVM function definition. The `void*` represents the environment pointer.
- All Scilla functions are represented as closures, with their first argument being the environment pointer.
- If the function's return type cannot be "by value", then the second argument will be a stack pointer ("sret") where the return value must be stored.
- To avoid ABI complexities, generated LLVM functions and the hand written functions in the runtime library that they may call, all follow the simple rule that if the value size is larger than two eightbytes, we define that parameter (or return value) to be passed by reference (*i.e.* via a stack pointer).