Scilla: Syntax and Semantics

[Version 0.1]

April 29, 2018

Zilliqa Team
Singapore and United Kingdom
zilliqa.com

Abstract

Ths document describes the language grammar and runtime semantics of Scilla, an intermediate-level language for smart contracts executed on top of Zilliqa blockchain. In addition to the key language components, we also outline the static typing discipline for Scilla, as well as its translation to Coq for machine-assisted formal verification of smart contracts.

1 Introduction

Scilla programming language has been proposed to tackle the challenge of constructing provably correct smart contracts on Zilliqa blockchain [11]. In this manuscript we present its syntax, static, and dynamic semantics, as well as describe its model of interaction with a blockchain targeting multi-shard execution. We split the description of Scilla's syntax into several fragments, focusing on several orthogonal programming aspects, outlined in the corresponding sections: types and pure *expressions* (Section 2), effectful *computations* (Section 3), and, *communication* primitives (Section 4), culminating with the definition of the top-level contract structure in Section 4.3. The remaining sections explain Scilla's lexical syntax (Section 5) and serialisation for data types and messages 6.

2 λ_{SC}: Pure Fragment of Scilla

We start by presenting the language λ_{SC} of the *pure* expression fragment of Scilla, wich is very much inspired by the Girard-Reynolds' System F [3, 9] (*aka* polymorphic lambda-calculus) with elements of Standard ML [5], the Core language of the Glasgow Haskell Compiler [4, 10] and Coq's Calculus of Inductive Constructions [2]. We have chosen System F as our expression language for the it features parametric polymorphism (*i.e.*, allows one to construct reusable definitions) and also enjoys *strong normalisation* (*i.e.*, evaluation of expressions written in it *always* terminates). A limited support for structural (primitive) recursion in λ_{SC} for a number of embedded algebraic data types is provided via built-in *recursion principles* (*cf.* Section 2.2.4).

2.1 Types

Every expression in λ_{SC} has a type, capturing its structural properties. Every well-formed expression has a type, which can be statically checked at the compilation type, such a type determines a set of values the expression can be evaluated to at run-time. Figure 1 presents basic data types of the language, which are either primitive (P) or parametric, *i.e.*, generic, T, S, which might include type variables α, β. The standard notation $\langle T \rangle$ denotes a possibly empty sequence of (possibly similar) occurrences of T, *i.e.*, $T_1, \ldots, T_n$. We denote the union of primitive and fully instantiated (*i.e.*, containing no type variables) types as *ground*.

Notational conventions. In Figure 1 and further in this document, P ranges over built-in primitive types, T, S range over arbitrary

Primitive type	P	::=	`int`	Integers
			`char`	Character
			`vhash`	Value hash
			`bnum`	Block number
			`btime`	Block time
			`thash`	Transaction hash
			`address`	Account address
Type	T, S	::=	P	primitive type
			`map` $P\ T$	map
			T `->` S	value function
			$\mathcal{D}\ \langle T_k \rangle$	instantiated data type
			α	type variable
			`forall` $\alpha\,.\,T$	polymorphic function

Figure 1. Syntax of λ_{SC} types.

types, α, β range over type variables, $\mathcal{D}$ ranges over type constructors. The notation $\langle x_k \rangle$ stands for a sequence of one or more occurrences of x, indexed by k. The notation $\langle x \rangle$ is a shortcut of zero or one occurrence of x. In the actual program syntax, parentheses `( ... )` are used to disambiguate nested applications of type constructors.

2.1.1 Primitive data types

A selection of primitive data types is standard for a functional ML-style language. Integers are signed and range from $\mathsf{MININT} = -2^{31}$ to $\mathsf{MAXINT} = 2^{31} - 1$. In addition to that, the `int` data type includes two special values, Inf, $-\mathsf{Inf}$, and NaN that make basic operations on it totally defined, as, *e.g.*, in the case $0/0 = \mathsf{Inf}$, and $\mathsf{Inf} + -\mathsf{Inf} = \mathsf{NaN}$. The datatype of characters uses two bytes, similarly to integers, and, thus, can encode UTF-16 character set. Other primitive types include block and transaction hashes, `bnum` and `btime` for block number and time, correspondingly,[1] `thash` for transaction hashes and `hash` for general-purpose hash values, obtained by means of a standard SHA3 256 implementation.

2.1.2 Parametric types

In addition to primitive types, we provide a a fixed number of parametric (higher-order) types that come with a number of constructors and can be used to construct a variety of data structure to be operated in a purely functional style [6].

The initial language proposal includes ML-style pairs (product type) and choices (tagged sum type), as well lists and options, encoded as a syntactic sugar on top of the former two higher-order types. Each of such types is parameterised by either one

[1] Their precise implementation is to be defined later, although they come as opaque types with a fixed set of operations, such as comparison $\leq$ for ordering.

or more type variables (referred to as T, S, R), which are all assumed to be eventually instantiated via some is a ground types. This way, one can, for instance, define a lsit of lists of natural numbers. Lastly, Scilla does not feature character strings as built-in primitive datatype, as they are encoded via lists of characters `char`.

Partial maps. For the convenience of programming in Scilla, we also provide the type of partial finite maps, from keys P, where P is a primitive type, to values of type T, where T is a ground type. Partial finite maps are immutable, meaning that altering the map will create a new map, which is an updated instance of the former.

Function type. λ_{SC} defines the type of first-class functions (implemented as lambda-expressions described furhter) as a parametric type `T -> S`, where both T and S are either primitive or ground types (including type variables described below). There is no explicit recursion operator, so all lambda-expressions, when applied are strictly terminating [7], if applied.

Built-in algebraic data types. At the moment Scilla provides a small number of parametric built-in algebraic data types (ADTs), allowing one to program with structured data by means of constructing and decomposing its values. Each ADT also comes with a predefined recursion principle, making it possible to implement primitively-recursive functions on this data in the spirit of a *visitor* pattern.[2] In Figure 1, types induced by ADTs are denoted by an application of the type constructor $\mathcal{D}$ to a list of type parameters.

Unit data type `unit` contains just one element tt and is typically used for an operation or a command whose result does not matter for the future computation. Boolean data type `bool` is represented by the two constructors, `True` and false. Peano-style natural numbers (`nat`) are defined via two constructors Zero and Succ, and, unlike "flat" integers are used for primitive-recursive operations (described in Section 2.2.4).

The following ADTs are polymorphic, *i.e.*, they are parameterised by other types. In the future, we are planning to add support for user-defined data types. The built-in data types in the current version of Scilla include:

- pairs (`pair` T S)
- choice type (`either` T S)
- list type (`list` T)
- option type (`option` T)
- exception type (`excn` T)

All but the last one should be familiar from the functional programming languages, such as OCaml and Haskell. The last type for exceptions serves to denote the results of failed executions of Scilla commands.

Polymorphic function type. Types in λ_{SC} can contain variables, which can be instantiated with other types. For instance, the identity function `fun x => x` can be used with a value of any type, hence its type should be parametric in the type of its parameter `x`.[3] In System F, a type of the identity can be expressed via a polymorphic function type `forall` α . α `->` α, where α is a type variable, which can be instantiated with any type, *e.g.*, `int`, thus elaborating, in this particular case, the polymorphic type of the identity function

[2] In the further version, we are planning to add support for user-defined ADTs.
[3] The λ_{SC} syntax for polymorphic functions, such as this one, is a bit different and is described below.

to the type `Int -> Int` At the moment, λ_{SC} features no higher-rank polymorphism, which means that a type variable can only be instantiated with a monomorphic, *i.e.*, ground type.

2.2 Expressions

Types of λ_{SC} are inhabited by expressions, whose syntax and the informal meaning we describe in this section. The full abstract syntax of λ_{SC} expressions is given in Figure 2.

Notational conventions. (Ilya: Explain what ranges over what and give pointers to the lexical spec.)

2.2.1 Variables and A-normal form

(Ilya: Explain why everything is in a-normal form)

2.2.2 Built-in primitive data types and their operations

(Ilya: TODOs)

1. Add syntax for built-ins
2. All in A-normal form
3. Statements: syntax for imported primitives

The full list of data type operations for primitive data types (Ilya: TODO: provide a big table, outlining specific operations.)

- Boolean operations
- Arithmetics
- Operations on hashes only include comparison for equality, which returns a boolean value.
- Characters: comparing for equality
- Blockchain-related data types, hashes, addresses: comparing for equality
- hashes: compute hash/check;

(Ilya: A subset of ML/Haskell's Core comes here, programs throw no exceptions.)

2.2.3 Lambda-expressions and applications

- Ordinary applications
- type applications

(Add polymorphism and type functions).

2.2.4 Working with structured data

(Ilya: We provide a built- primitive recursors to work with the structured data. Most of them are higher-order)

- a table for constructors and their types;
- syntax for patterns in pattern matching (including tuple projections);
- recursors and how to work with them [1];

(Ilya: Emphasize that recursors always terminate!)

(Ilya: In the future, we consider extending this with user-defined data types with predefined recursion principles.)

The exception data type comes with just one constructor Exn that takes a single argument. Throwing and catching an exception constitutes a *computation* effect and never occurs when evaluating expressions.

(Ilya: Show how to implement pair getters via recursors.)

(Ilya: Emphasize that the recursors are eager, while pattern matching is lazy.)

2.2.5 Working with maps

- making an empty map;

			Term	Meaning	Description
Expression	e	::=	f	simple expression	
			`let` x ⟨: T⟩ = f `in` e	let-form	
Simple expression	f	::=	i	primitive integer	
			a	address	
			h	hash	
			c	UTF-16 character	
			`Emp`	empty map literal	
			x	variable	
			`fun` (x : T) => e	function	
			`builtin` b $\langle x_k \rangle$	built-in application	
			x $\langle x_k \rangle$	application	
			`tfun` α => e	type function	
			@x T	type instantiation	
			C ⟨ {$\langle T_k \rangle$} ⟩ $\langle x_k \rangle$	constructor instantiation	
			`match` x `with` ⟨ | sel_k ⟩ `end`	pattern matching	
Selector	sel	::=	pat => e		
Pattern	pat	::=	x	variable binding	
			C $\langle pat_k \rangle$	constructor pattern	
			(pat)	paranthesized pattern	
			_	wildcard pattern	
Built-in name	b			identifier	

Figure 2. Syntax of λ_{SC} expressions.

Operation	Symbol	Parameters	Result type	Result	Remarks
Structural equality	`eq`	$(x : T)\ (y : T)$	`bool`	$x = y$	T is any ground type
Integer addition	`add`	$(x :$ `int`$)\ (y :$ `int`$)$	`int`	$x \mathbin{\widehat{+}} y$	*cf.* details in §2.2.2
Integer subtraction	`sub`	$(x :$ `int`$)\ (y :$ `int`$)$	`int`	$x \mathbin{\widehat{-}} y$	*cf.* details in §2.2.2
Integer multiplication	`mult`	$(x :$ `int`$)\ (y :$ `int`$)$	`int`	$x \mathbin{\widehat{\times}} y$	*cf.* details in §2.2.2
Integer division	`div`	$(x :$ `int`$)\ (y :$ `int`$)$	`int`	$x \mathbin{\widehat{/}} y$	*cf.* details in §2.2.2
Integer remainder	`mod`	$(x :$ `int`$)\ (y :$ `int`$)$	`int`	$x \mathbin{\widehat{\mathrm{mod}}} y$	*cf.* details in §2.2.2
Integer comparison	`lt`	$(x :$ `int`$)\ (y :$ `int`$)$	`bool`	$x < y$	*cf.* details in §2.2.2
Hashing	`hash`	$(x : T)$	`vhash`	SHA3 256 hash	
Time comparison	`tlt`	$(x :$ `btime`$)\ (y :$ `btime`$)$	`bool`	$x < y$	
Block # comparison	`blt`	$(x :$ `bnum`$)\ (y :$ `bnum`$)$	`bool`	$x < y$	
Type conversions					
`nat` to `int` conversion	`toint`	$(x :$ `nat`$)$	`option int`	Some x as `int` None	if $x \leq$ MAXINT otherwise
`int` to `nat` conversion	`tonat`	$(x :$ `int`$)$	`option nat`	Some x as `nat` None	if $x \geq 0$ otherwise

Figure 3. Built-in operations and conversions on primitive data types.

- get/put/contains;
- delete;
- recursion over a map's entries;

2.3 Static Semantics

(Ilya: Standard typing rules for System F)

2.4 Operational Semantics for λ_{SC} Expressions

(Ilya: TODO: CEK machine comes here)

2.5 Examples

Let us now see several examples of actual programs written in λ_{SC}.
(Ilya: TODO: provide example programs.)

3 Computations and Commands

The following categories are present, all commands are in the CPS style, ending via either send or return.

- Modifying contract fields;

Data type	Constructors and their types	Recursor and its type
unit	U : unit	unit_rec : forall $\alpha . \alpha$ -> unit -> α
bool	True : bool False : bool	bool_rec : forall $\alpha . \alpha$ -> α -> bool -> α
nat	Zero : nat Succ : nat -> nat	nat_rec : forall $\alpha . \alpha$ -> (nat -> α -> α) -> nat -> α
pair T S	And : forall $\alpha\ \beta . \alpha$ -> β -> pair α β	pair_rec : forall $\alpha\ \beta\ \gamma . (\alpha$ -> β -> γ) -> pair α β -> γ
either T S	Left : forall $\alpha .$ α -> either α Right : forall $\beta .$ β -> either β	either_rec : forall $\alpha\ \beta\ \gamma . (\alpha$ -> γ) -> (β -> γ) -> either α β -> γ
option T	Some : forall $\alpha .$ α -> option α None : forall $\alpha .$ option α	option_rec : forall $\alpha .$ forall $\beta . (\alpha$ -> β) -> β -> option α -> β
list T	Nil : forall $\alpha .$ list α Cons : forall $\alpha .$ α -> list α -> list α	list_rec : forall $\alpha\ \beta . \beta$ -> (α -> list α -> β -> β) -> list β

Figure 4. Built-in Algebraic Data Types (ADTs), their constructors and recursion principles.

Figure 5. Runtime semantics of λ_{SC}.

```
let list_product =
fun (ls : List Int) => fun (acc : Int) =>
    let iter =
      fun (h : Int) => fun (t : List Int) => fun (res : Int) =>
        let zero = 0 in
        let b = builtin eq h zero in
        match b with
        | True => 0
        | False => builtin mult h res
        end
    in
    let rec_nat = @ list_rec nat in
    let rec_nat_nat = @ rec_nat nat in
    rec_nat_nat acc iter ls
```

Figure 6. Product of integer numbers in a list `ls`.

```
let fib = fun (n : nat) =>
  let iter_nat = @ nat_rec (pair nat nat) in
  let iter_fun =
    fun (n: nat) => fun (res : pair nat nat) =>
      match res with
      | And x y => let z = add x y in
                   And {Nat Nat} z x
      end
    in
  let zero = Zero in
  let one = Succ zero in
  let init_val = And {Nat Nat} one zero in
  let res = iter_nat init_val iter_fun n in
  fst res
```

Figure 7. Fibonacci numbers.

- Interacting with the blockchain (what are the primitives)?
- try/catch
- Exceptions;
- Events;
- Accepting funds (inverse of payable);
- Sending funds;

All this will be manifested in a type-and-effect system, keeping track of the funny things.

(Ilya: Do we need anything else?)

```
let insert_sort =
  fun (ls : List Int) =>
  let true = True in
  let false = False in
  let rec_int = @ list_rec Int in
  let rec_int_int = @ rec_int Int in
  let rec_int_pair = @ rec_int (Pair Bool (List Int)) in
  let nil_int = Nil {Int} in
  let sink_down =
    fun (e : Int) => fun (ls : List Int) =>
      let init = And {Bool (List Int)} false nil_int in
      let iter1 =
        fun (h : pair bool (list int)) =>
        fun (t : list int) =>
        fun (res : pair bool (list int)) =>
          let b    = fst res in
          let rest = snd res in
          match b with
          | True =>
            let z = Cons {Int} h rest in
            And {Bool (List Int)} true z
          | False =>
              let bl = builtin lt h e in
              match bl with
              | True =>
                let z = Cons {Int} e rest in
                let z2 = Cons {Int} h z in
                And {Bool (List Int)} true z2
              | False =>
              let z = Cons {Int} h rest in
              And {Bool (List Int)} false z
              end
          end
      in
      let res1 = rec_int_pair init iter1 ls in
      let b0 = fst res1 in
      let ls1 = snd res1 in
      match b0 with
      | True => ls1
      | False => Cons {int} e ls1
      end
  in
  let iter2 = fun (h : Int) => fun (t : List Int) =>
              fun (res : Int) => sink_down h res
  in
  rec_int_int iter2 nil_int ls
```

Figure 8. Insertion sort of a list in λ_{SC}.

(Ilya: Here comes the precise grammar of scilla components)

Figure 9. Lexical grammar of Scilla

4 Communication and Transitions

4.1 Messages

(Ilya: Emphasize the sending of several messages)

(Ilya: Messages come with typed components, so there might be no in-contract failure due to the ill-typed message! This is how we keep expressions pure.)

4.2 Transitions

(Ilya: Just a wrapper around expressions)

(Ilya: No continuations at this point.)

4.3 Top-Level Definitions

4.4 Named Functions and Standard Library

Functions are non-(mutually) recursive: defined in the order they appear in the contract. They might encapsulate traversals. They also do not have side-effects.

(Ilya: Say how we provide all functions to program with Scilla functional data types and how other types.)

4.5 Contract Transitions

5 Lexical Grammar

5.1 Reserved keywords

The following keywords are reserved in Scilla:

`builtin`, **`fun`**, **`tfun`**, **`let`**, **`in`**, **`match`**, **`with`**, **`end`**, **`contract`**, **`transition`**, **`send`**, **`log`**, **`import`**, **`as`**, **`if`**, **`then`**, **`else`**.

5.2 Special symbols

The following multi-character symbols are considered special and treated as indivisible tokens: ->, =>, <-. Other single-character symbols occur precisely as in Figure 2.

5.3 Other tokens

Figure **??** provides description of the lexical tokens occurring in Scilla expressions, statements, and contracts, with their textual explanation. The remaining

(Ilya: TODO: provide a table with tokens)

(Provide)

We have implemented the reference lexer and parser in OCaml using ocamllex and Menhir tools [8].

(Ilya: Do not forget about comments!)

6 Serialisation

(Ilya: Tell how values are serialized into the messages, packed with the typing information)

So far, we serialize into JSON. Here's the format: (Ilya: TODO)

References

[1] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[2] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[3] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

[4] Simon L. Peyton Jones. Type-Directed Compilation in the Wild: Haskell and Core. In *TLCA*, volume 7941 of *LNCS*. Springer, 2013.

[5] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML.* MIT Press, 1990.

[6] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999. ISBN 978-0-521-66350-2.

[7] Benjamin C. Pierce. *Types and programming languages.* MIT Press, 2002.

[8] François Pottier and Yann Régis-Gianas. Menhir Reference Manua, 2017. INRIA. Version 20171222.

[9] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *LNCS*, pages 408–423. Springer, 1974.

[10] Ilya Sergey, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Modular, higher-order cardinality analysis in theory and practice. In *POPL*, pages 335–348. ACM, 2014.

[11] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage, 2018. https://arxiv.org/abs/1801.00687.