Gas Accounting on ZILLIQA

[Draft version 0.1]
Subject to changes

The ZILLIQA Team
www.zilliqa.com gitter.im/zilliqa

August 31, 2019

1 Storage

Gas consumed for storage (presented in Table 1) depends on the type of literal being stored. In the table below, ℓ denotes the literal and $litGas(\ell)$, the gas consumed for storing it. Note that for certain literal types such as **Messsage**, **Map** and **ADT**, $litGas(\ell)$ is recursively computed.

Table 1: Gas for storage of literals.

Literal Type	$litGas(\ell)$	**Remarks**
`String`	$\begin{cases} 20, & \text{if } len(\ell) \leq 20, \\ len(\ell), & \text{else} \end{cases}$	$len(\cdot)$ computes the length of an input string, e.g., $len($`"Hello"`$) = 5$.
`Int32`, `Uint32`	4	Gas consumed is equal to the size of integer in bytes.
`Int64`, `Uint64`	8	
`Int128`, `Uint128`	16	
`Int256`, `Uint256`	32	
`BNum`	64	Internally represented as an unsigned BigNum.
`ByStrX`, `ByStr`	$width(\ell)$	$width(\cdot)$ returns the number of bytes needed to represent the hexadecimal input, e.g., $width($`0x1cd2`$) = 2$.
`Messsage`	For $\ell = \{s_1 : v_1; s_2 : v_2; \ldots, s_n : v_n\}$: $\begin{cases} 0, & \text{if } n = 0, \\ litGas(\{s_2 : v_2; \ldots, s_n : v_n\}) \\ +litGas(s_1) + litGas(v_1), & \text{else} \end{cases}$	As a **Messsage** literal is an associative array between a **String** literal (denoted s_i) and a value literal (denoted v_i), $litGas$ is recursively computed by summing up over each $s_i : v_i$.
`Map`	For $\ell = \{(k_1 \rightarrow v_1), \ldots, (k_n \rightarrow v_n)\}$: $\begin{cases} 0, & \text{if } n = 0, \\ \sum_{(k \rightarrow v) \in \ell} litGas(k) + litGas(v), & \text{else} \end{cases}$	As a **Map** literal maps a key literal (k_i) to a value literal (v_i), $litGas$ is recursively computed by summing up over each $k_i \rightarrow v_i$.
`ADT` (e.g., `Bool`, `Option`, `List`)	For $\ell = \texttt{cname}\{\texttt{types}\}t_1 t_2 \ldots t_n$: $\begin{cases} 1, & \text{if } n = 0, \\ \sum_i litGas(t_i), & \text{else} \end{cases}$	An **ADT** literal takes a constructor name (`cname`), types of the arguments (`types`) and arguments denoted by $t_1, t_2, \ldots, t_n$. $litGas$ is recursively computed by summing up the gas required over each t_i.

2 Computation

In this section, we present the gas required to perform computations in a contract using SCILLA *expressions*. Expressions handle purely mathematical computations and do not have any side-effects.

Gas consumed for a computation via an expression can be divided into two parts: the static part associated with employing a specific expression and the dynamic part that takes into account the cost that can only be estimated at run time. Some expressions do not entail any dynamic gas consumption.

2.1 Static Cost for Expressions

Every expression has a static gas associated with it. Table 2 lists the gas for each expression supported in SCILLA. In the table below, e denotes the expression and $statExprCost(e)$ the static gas associated with using e.

Table 2: Static gas for expressions.

Expression	$statExprGas(e)$	Remarks
`Literal`	1	
`Var`		
`Let`		
`Message`		
`Fun`		A function declaration, e.g., `fun (x : T) => e`.
`App`		A function application, e.g., `f <x_i>`.
`TFun`		A type function of the form: `tfun` α `=> e`.
`TApp`		A type instantiation: `@x T`.
`Constr`		Constructors.
`MatchExpr`	$nbClauses(e)$	$nbClauses(\cdot)$ returns the number of clauses in the pattern match.
`Fixpoint`	1	Rest of the gas is accounted during recursive evaluation.
`Builtin`	0	Purely dynamic gas accounting. For more details see Table 4.

2.2 Executing Statements

State changes and other operations that entail side-effects such as reading the current state of the blockchain or invoking message calls to other contracts are performed via SCILLA *statements*. Table 3 presents the gas consumed for statements. In the table, l is the literal being handled and $stateGas(\ell)$ is the required gas. Note that the $litGas(\cdot)$ function used below comes from Table 1.

2.3 Builtins

In Table 4, we present the dynamic gas associated with `Builtin` operators in SCILLA. The gas consumed often depends on the operator and operand types, etc. In the table below, we group `Builtin` in categories which are self-explanatory.

Table 4: Gas required for Builtin operations.

Category	Operation	$builtinGas$	Remarks
`String`	`eq`	For `eq` s_1 s_2: $\min\{len(s_1), len(s_2)\}$	$len(\cdot)$ computes the length of an input string, e.g., $len($`"Hello"`$) = 5$.
	`concat`	For `concat` s_1 s_2: $len(s_1) + len(s_2)$	

(To be continued)

Category	Operation	*builtinGas*	Remarks
	substr	For substr $s\ i_1\ i_2$: $i_1 + \min\{len(s) - i_1, i_2\}$	
Hashing	eq	For eq $d_1\ d_2$: $width(d_i)$	$width(\cdot)$ returns the size in number of bytes of the input. Note that the operator expects two inputs of the same size.
	dist	32	dist computes the distance between two ByStr32 values.
	ripemd160hash	For ripemd160hash x: $10 \times \left\lceil \frac{size(x)}{64} \right\rceil$	$size(\cdot)$ returns the size of the serialized input in bytes. Gas consumed is dependent on the input size and the block size of the hash function. Block sizes of ripemd160hash, sha256hash and keccak256hash are 64, 64 and 136 bytes respectively.
	sha256hash	For sha256hash x: $15 \times \left\lceil \frac{size(x)}{64} \right\rceil$	
	keccak256hash	For keccak256hash x: $15 \times \left\lceil \frac{size(x)}{136} \right\rceil$	
Signing	schnorr_gen_key_pair	20	
	schnorr_sign	For schnorr_sign $k_{priv}\ k_{pub}\ m$: $350 + 15 \times \left\lceil \frac{66 + size(m)}{64} \right\rceil$	Signing requires computing hash of the input message and other parameters (of size 66 bytes). The constant cost of 350 is for elliptic curve operations and other base field operations.
	schnorr_verify	For schnorr_verify $k_{pub}\ m\ sig$: $250 + 15 \times \left\lceil \frac{66 + size(m)}{64} \right\rceil$	Verification also requires computing hash of the input message and other parameters (of size 66 bytes). The constant cost of 250 is for elliptic curve operations and other base field operations. Verification is cheaper than signing.
ByStrX	to_bystr	$width(a)$	to_bystr is a conversion utility to convert from ByStrX to ByStr. $width(\cdot)$ returns the number of bytes represented by the input.
Map	contains, get	1	Requires constant time.
	put, remove, to_list, size	$1 + lenOfMap$	$lenOfMap$ is the number of keys in the map.

(To be continued)

Category	Operation	*builtinGas*	Remarks
`Nat`	`to_nat`	For `to_nat` i: i	
`Int32`, `Uint32`	`mul`, `div`, `rem`	20	
	`eq`, `lt`, `add`, `sub`, `to_int32`, `to_uint32`, `to_int64`, `to_uint64`, `to_int128`, `to_uint128`, `to_int256`, `to_uint256`, `to_nat`	4	
`Int64`, `Uint64`	`mul`, `div`, `rem`	20	
	`eq`, `lt`, `add`, `sub`, `to_int32`, `to_uint32`, `to_int64`, `to_uint64`, `to_int128`, `to_uint128`, `to_int256`, `to_uint256`, `to_nat`	4	
`Int128`, `Uint128`	`mul`, `div`, `rem`	40	
	`eq`, `lt`, `add`, `sub`, `to_int32`, `to_uint32`, `to_int64`, `to_uint64`, `to_int128`, `to_uint128`, `to_int256`, `to_uint256`, `to_nat`	8	
`Int256`, `Uint256`	`mul`, `div`, `rem`	80	
	`eq`, `lt`, `add`, `sub`, `to_int32`, `to_uint32`, `to_int64`, `to_uint64`, `to_int128`, `to_uint128`, `to_int256`, `to_uint256`, `to_nat`	16	
`BNum`	`eq`, `blt`, `badd`	32	

3 Contract Deployment and Transition Invocation

In addition to the above gas, there is an upfront cost for contract deployment and transition invocations. The goal of these costs is to prevent spam attacks.

At the time of contract deployment, an end user will provide two input files: one containing the contract (a `.scilla` file) and the other containing the value of the immutable parameters (a JSON file). The upfront gas consumed for contract deployment is equal to the size (in bytes) of the SCILLA file plus that of the JSON file.

In a similar manner, at the time of contract invocation, and end user (as a part of a transaction) or a contract (as a part of a message call) will supply a JSON file that will contain information on which transition needs to be invoked and the parameters to pass. Gas associated with such transition calls is equal to the size of this JSON file.

Table 3: Gas for statements.

Statement (l)	$stateGas(l)$	**Remarks**
G_Load	$litGas(l)$	
G_Store	$\max\{litGas(l_{old}), litGas(l_{new})\} + litGas(l_{new}) - litGas(l_{old})$	ℓ_{old} is the existing literal, while, ℓ_{new} is the new literal to be stored. Note that gas for the store statement can be 0.
G_Bind	1	
G_MatchStmt	$nbclauses(l)$	
G_ReadFromBC	1	Gas to read current blockchain values such as BLOCKNUMBER (previous block number).
G_AcceptPayment	1	
G_SendMsgs	For $t = [\{s_1^1 : v_1^1; \ldots, s_n^1 : v_n^1\}, \{s_1^2 : v_1^2; \ldots, s_n^2 : v_n^2\}, \ldots, \{s_1^m : v_1^m; \ldots, s_n^2 : v_n^m\}]$: $$\sum_{\ell \in t} litGas(l)$$	G_SendMsgs takes a list of Messsage as an input. Hence, gas required is the sum of $litGas(\cdot)$ for each individual Messsage in the list.
G_CreateEvent	$litGas(\ell)$	