ErgoTree Specification

authors

September 12, 2019

Abstract

In this document we consider typed abstract syntax of the language called ErgoTree which defines semantics of a condition which protects a closed box in the Ergo Platform blockchain. Serialized graph is written into a box. Most of Ergo users are unaware of the graph since they are developing contracts in higher-level languages, such as ErgoScript. However, for developers of alternative higher-level languages, client libraries and clients knowledge of internals would be highly useful. This document is providing the internals, namely, the following data structures and algorithms:

- Serialization to a binary format and graph deserialization from the binary form.
- When a graph is considered to be well-formed and when not.
- Type system and typing rules.
- How graph is transformed into an execution trace.
- How execution trace is costed.
- How execution trace is reduced into a Sigma-expression.
- How Sigma-expression is proven and verified.

kushti : Please note that the document is intended for general high-skilled tech audience, so avoid describing Scala classes etc.

Contents

1 Introduction

The design space of programming languages is very broad ranging from general-purpose languages like C,Java,Python up to specialized languages like SQL, HTML, CSS, etc.

Since Ergo's goal is to provide a platform for contractual money, the choice of the language for writing contracts is very important.

First of all the language and contract execution environment should be *deterministic*. Once created and stored in Ergo blockchain, smart contract should always behave predictably and deterministically, it should only depend on well-defined data context and nothing else. As long as data context doesn't change, any execution of the contract should return the same value any time it is executed, on any execution platform, and even on any *compliant* language implementation. No general purpose programming language is deterministic because all of them provide non-deterministic operations. ErgoScript doesn't have non-deterministic operations.

Second, the language should be *spam-resistant*, meaning it should facilitate in defending against attacks when malicious contracts can overload network nodes and bring the blockchain down. To fullfill this goal ErgoScript support *ahead-of-time cost estimation*, the fast check performed before contract execution to ensure the evaluation cost is within acceptable bounds. In general, such cost prediction is not possible, however if the language is simple enough (which is the case of ErgoScript) and if operations are carefully selected, then costing is possible and doesn't require usage of Gas Morphic : cite etherium and allow to avoid related problems Morphic : cite Gas related problems.

Third, being simple, the contracts language should be *expressive enough*. It should be possible to implement most of the practical scenarios, which is the case of ErgoScript. In our experience expressivity of contracts language comes hand in hand with design and capabilities of Ergo blockchain platform itself, making the whole system *turing-complete* as we demonstrated in Morphic : cite TuringPaper.

Forth, simplicity and expressivity are often characteristics of domain-specific languages Morphic : cite DSL. From this perspective ErgoScript is a DSL for writing smart contracts. The language directly captures the Ubiquites Language [Ubi] of smart contracts domain directly manipulating with first-class Boxes, Tokens, Zero-Knowledge Sigma-Propostions etc., these are the novel features Ergo aims to provide as a platform/service for custom user applicatons. Domain-specific nature nature of ErgoScript also fasilitates spam-resistance, because operations of ErgoScript are all carefully selected to be *costing friendly*.

And last, but not the least, we wanted our new language to be, nevertheless, *familiar to the most* since we aim to address as large audience of programmers as possible with minimum surprise and WTF ratio [WTF]. The syntax of ErgoScript is inspired by Scala/Kotlin, but in fact it shares a common subset with Java and C#, thus if you are proficient in any of these languages you will be right at home with ErgoScript as well.

Guided by this requirements we designed ErgoScript as a new yet familiar looking language which directly support all novel features of Ergo blockchain. We also implemented reference implementation of the specification described in this document.

2 Language

Here we define abstract syntax for ErgoTree language. It is a typed functional language with tuples, collections, optional types and `val` binding expressions. The semantics of ErgoTree is specified by first translating it to a core calculus (`Core-`λ) and then by giving its evaluation semantics. Typing rules is given in Section 3 and evaluation semantics is given in Section 4.

ErgoTree is defined here using abstract syntax notation as shown in Figure 1. This corresponds to `ErgoTree` data structure, which can be serialized to an array of bytes. The mnemonics shown in the figure correspond to classes of `ErgoTree` reference implementation.

Set Name		Syntax	Mnemonic	Description
$\mathcal{T} \ni T$	::=	`P`	`SPredefType`	predefined types (see Appendix A)
	|	τ	`STypeVar`	type variable
	|	$(T_1, \dots, T_n)$	`STuple`	tuple of n elements (see `Tuple` type)
	|	$(T_1, \dots, T_n) \to T$	`SFunc`	function of n arguments (see `Func` type)
	|	`Coll`$[T]$	`SCollection`	collection of elements of type T
	|	`Option`$[T]$	`SOption`	optional value of type T
$Term \ni e$	::=	$C(v, T)$	`Constant`	typed constants
	|	x	`ValUse`	variables
	|	$\lambda(\overline{x_i : T_i}).e$	`FuncExpr`	lambda expression
	|	$e_f\langle\overline{e_i}\rangle$	`Apply`	application of functional expression
	|	$e.m\langle\overline{e_i}\rangle$	`MethodCall`	method invocation
	|	$(e_1, \dots, e_n)$	`Tuple`	constructor of tuple with n items
	|	$\delta\langle\overline{e_i}\rangle$		primitive application (see Appendix B)
	|	`if` (e_{cond}) e_1 `else` e_2	`If`	if-then-else expression
	|	$\{$`val` $x_i = e_i;$ $e\}$	`BlockExpr`	block expression
cd	::=	`trait` I $\{\overline{ms_i}\}$	`STypeCompanion`	interface declaration
ms	::=	`def` $m[\overline{\tau_i}](\overline{x_i : T_i}) : T$	`SMethod`	method signature declaration

Figure 1: Abstract syntax of ErgoScript language

We assign types to the terms in a standard way following typing rules shown in Figure 3.

Constants keep both the type and the data value of that type. To be well-formed the type of the constant should correspond to its value.

Variables are always typed and identified by unique id, which refers to either lambda bound variable of `val` bound variable. The encoding of variables and their resolution is described in Section **??**.

Lambda expressions can take a list of lambda-bound variables which can be used in the body expression, which can be *block expression*.

Function application takes an expression of functional type (e.g. $T_1 \to T_n$) and a list of arguments. The reason we do not write it $e_f(\overline{e})$ is that this notation suggests that $(\overline{e})$ is a subterm, which it is not.

Method invocation allows to apply functions defined as methods of *interface types*. If expression e has interface type I and and method m is declared in the interface I then method invocation $e.m(args)$ is defined for the appropriate $args$.

Conditional expressions of ErgoTree are strict in condition and lazy in both of the branches. Each branch is an expression which is executed depending on the result of condition. This laziness of branches specified by lowering to `Core-`λ (see Figure 2).

Block expression contains a list of `val` definitions of variables. To be wellformed each subsequent definition can only refer to the previously defined variables. Result of block execution is the result of the resulting expression e, which can use any variable of the block.

Each type may be associated with a list of method declarations, in which case we say that *the type has methods*. The semantics of the methods is the same as in Java. Having an instance of some type with methods it is possible to call methods on the instance with some additional arguments. Each method can be parameterized by type variables, which can be used in method signature. Because ErgoTree supports only monomorphic values each method call is monomorphic and all type variables are assigned to concrete types (see `MethodCall` typing rule in Figure 3).

The semantics of ErgoTree is specified by translating all its terms to a somewhat lower and simplified language, which we call `Core-`λ. This *lowering* translation is shown in Figure 2.

$Term_{ErgoTree}$		$Term_{Core}$
$\mathcal{L}[\![\lambda(\overline{x_i : T_i}).e]\!]$	$\mapsto$	$\lambda x : (T_0, \ldots, T_n).\mathcal{L}[\![\{\overline{\texttt{val}\ x_i : T_i = x.\_i;}\ e\}]\!]$
$\mathcal{L}[\![e_f\langle\overline{e_i}\rangle]\!]$	$\mapsto$	$\mathcal{L}[\![e_f]\!]\langle\mathcal{L}[\![(\overline{e_i})]\!]\rangle$
$\mathcal{L}[\![e.m\langle\overline{e_i}\rangle]\!]$	$\mapsto$	$\mathcal{L}[\![e]\!].m\langle\overline{\mathcal{L}[\![e_i]\!]}\rangle$
$\mathcal{L}[\![(e_1, \ldots, e_n)]\!]$	$\mapsto$	$(\mathcal{L}[\![e_1]\!], \ldots, \mathcal{L}[\![e_n]\!])$
$\mathcal{L}[\![e_1\ \texttt{\|\|}\ e_2]\!]$	$\mapsto$	$\mathcal{L}[\![\texttt{if}\ (e_1)\ \texttt{true else}\ e_2]\!]$
$\mathcal{L}[\![e_1\ \texttt{\&\&}\ e_2]\!]$	$\mapsto$	$\mathcal{L}[\![\texttt{if}\ (e_1)\ e_2\ \texttt{else false}]\!]$
$\mathcal{L}[\![\texttt{if}\ (e_{cond})\ e_1\ \texttt{else}\ e_2]\!]$	$\mapsto$	$(\texttt{if}(\mathcal{L}[\![e_{cond}]\!],\ \lambda(\_ : Unit).\mathcal{L}[\![e_1]\!],\ \lambda(\_ : Unit).\mathcal{L}[\![e_2]\!]))\langle\rangle$
$\mathcal{L}[\![\{\overline{\texttt{val}\ x_i : T_i = e_i;}\ e\}]\!]$	$\mapsto$	$(\lambda(x_1 : T_1).(\ldots(\lambda(x_n : T_n).\mathcal{L}[\![e]\!])\langle\mathcal{L}[\![e_n]\!]\rangle \ldots))\langle\mathcal{L}[\![e_1]\!]\rangle$
$\mathcal{L}[\![\delta\langle\overline{e_i}\rangle]\!]$	$\mapsto$	$\delta\langle\overline{\mathcal{L}[\![e_i]\!]}\rangle$
$\mathcal{L}[\![e]\!]$	$\mapsto$	e

Figure 2: Lowering to `Core-`λ

All n-ary lambdas when $n > 1$ are transformed to single arguments lambdas using tupled arguments. Note that `if` (e_{cond}) e_1 `else` e_2 term of ErgoTree has lazy evaluation of its branches whereas right-hand-side `if` is a primitive operation and have strict evaluation of the arguments. The laziness is achieved by using lambda expressions of `Unit` $\to$ `Boolean` type.

We translate logical operations (`||`, `&&`) of ErgoTree, which are lazy on second argument to `if` term of ErgoTree, which is recursively translated to the corresponding `Core-`λ term.

Syntactic blocks of ErgoTree are completely eliminated and translated to nested lambda expressions, which unambiguously specify evaluation semantics of blocks. The `Core-`λ is specified in Section 4.

3 Typing

ErgoTree is a strictly typed language, in which every term should have a type in order to be wellformed and evaluated. Typing judgement of the form $\Gamma \vdash e : T$ say that e is a term of type T in the typing context Γ.

$$\frac{}{\Gamma \vdash C(\_,T) : T}\ \texttt{(Const)} \quad \frac{}{\Gamma, x : T \vdash x : T}\ \texttt{(Var)} \quad \frac{\overline{\Gamma \vdash e_i\colon T_i} \quad ptype(\delta,\overline{T_i})\colon (T_1,\ldots,T_n) \to T}{\delta\langle\overline{e_i}\rangle\colon T}\ \texttt{(Prim)}$$

$$\frac{\Gamma \vdash e_1\colon T_1 \quad \ldots \quad \Gamma \vdash e_n\colon T_n}{\Gamma \vdash (e_1,\ldots,e_n) : (T_1,\ldots,T_n)}\ \texttt{(Tuple)}$$

$$\frac{\Gamma \vdash e : I,\ e_i\colon T_i \quad mtype(m,I,\overline{T_i}) : (I,T_1,\ldots,T_n) \to T}{e.m\langle\overline{e_i}\rangle\colon T}\ \texttt{(MethodCall)}$$

$$\frac{\Gamma,\overline{x_i\colon T_i} \vdash e : T}{\Gamma \vdash \lambda(\overline{x_i{:}T_i}).e : (T_0,\ldots,T_n) \to T}\ \texttt{(FuncExpr)} \quad \frac{\Gamma \vdash e_f\colon (T_1,\ldots,T_n) \to T \quad \overline{\Gamma \vdash e_i\colon T_i}}{\Gamma \vdash e_f\langle\overline{e_i}\rangle : T}\ \texttt{(Apply)}$$

$$\frac{\Gamma \vdash e_{cond}\colon \texttt{Boolean} \quad \Gamma \vdash e_1\colon T \quad \Gamma \vdash e_2\colon T}{\Gamma \vdash \texttt{if}\ (e_{cond})\ e_1\ \texttt{else}\ e_2 : T}\ \texttt{(If)}$$

$$\frac{\Gamma \vdash e_1\colon T_1 \wedge \forall k{\in}\{2,\ldots,n\}\ \Gamma,x_1\colon T_1,\ldots,x_{k-1}\colon T_{k-1} \vdash e_k\colon T_k \wedge \Gamma,x_1\colon T_1,\ldots,x_n\colon T_n \vdash e\colon T}{\Gamma \vdash \{\overline{\texttt{val}\ x_i{=}e_i;}\ e\} : T}\ \texttt{(BlockExpr)}$$

Figure 3: Typing rules of ErgoTree

Note that each well-typed term has exactly one type hence we assume there exists a funcion $termType : Term \to \mathcal{T}$ which relates each well-typed term with the corresponding type.

Primitive operations can be parameterized with type variables, for example addition (Table **??**) has the signature $+ : (T,T) \to T$ where T is numeric type (Table 3). Function *ptype*, defined in Appendix B returns a type of primitive operation specialized for concrete types of its arguments, for example $ptype(+,\texttt{Int},\texttt{Int}) = (\texttt{Int},\texttt{Int}) \to \texttt{Int}$.

Similariy, the function *mtype* returns a type of method specialized for concrete types of the arguments of the `MethodCall` term.

`BlockExpr` rule defines a type of well-formed block expression. It assumes a total ordering on `val` definitions. If a block expression is not well-formed than is cannot be typed and evaluated.

The rest of the rules are standard for typed lambda calculus.

4 Evaluation Semantics

Evaluation of ErgoTree is specified by its translation to `Core-`λ, whose terms form a subset of ErgoTree terms. Thus, typing rules of `Core-`λ form a subset of typing rules of ErgoTree.

Here we specify evaluation semantics of `Core-`λ, which is based on call-by-value (CBV) lambda calculus. Evaluation of `Core-`λ is specified using denotational semantics. To do that, we first specify denotations of types, then typed terms and then equations of denotational semantics.

Definition 1 *(values, producers)*

- *The following CBV terms are called values:*

$$V :== x \mid C(d, T) \mid \lambda x.M$$

- *All CBV terms are called producers. (This is because, when evaluated, they produce a value.)*

We now describe and explain a denotational semantics for the `Core-`λ language. The key principle is that each type A denotes a set $[\![A]\!]$ whose elements are the denotations of values of the type A.

Thus the type `Boolean` denotes the 2-element set $\{\mathtt{true}, \mathtt{false}\}$, because there are two values of type `Boolean`. Likewise the type $(T_1, \ldots, T_n)$ denotes $([\![T_1]\!], \ldots, [\![T_n]\!])$ because a value of type $(T_1, \ldots, T_n)$ must be of the form $(V_1, \ldots, V_n)$, where each V_i is value of type T_i.

Given a value V of type A, we write $[\![V]\!]$ for the element of A that it denotes. Given a close term M of type A, we recall that it produces a value V of type A. So M will denote an element $[\![M]\!]$ of $[\![A]\!]$.

A value of type $A \to B$ is of the form $\lambda x.M$. This, when applied to a value of type A gives a value of type B. So $A \to B$ denotes $[\![A]\!] \to [\![B]\!]$. It is true that the syntax appears to allow us to apply $\lambda x.M$ to any term N of type A. But N will be evaluated before it interracts with $\lambda x.M$, so $\lambda x.M$ is really only applied to the value that N produces.

Definition 2 *A* context Γ *is a finite sequence of identifiers with value types* $x_1 : A_1, \ldots, x_n : A_n$*. Sometimes we omit the identifiers and write* Γ *as a list of value types.*

Given a context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, an environment (list of bindings for identifiers) associates to each x_i as value of type A_i. So the environment denotes an element of $([\![A_1]\!], \ldots, [\![A_n]\!])$, and we write $[\![\Gamma]\!]$ for this set.

Given a `Core-`λ term $\Gamma \vdash M : B$, we see that M, together with environment, gives a closed term of type B. So M denotes a function $[\![M]\!]$ from $[\![\Gamma]\!]$ to $[\![B]\!]$.

In summary, the denotational semantics is organized as follows.

- A type A denotes a set $[\![A]\!]$
- A context $x_1 : A_1, \ldots, x_n : A_n$ denotes the set $([\![A_1]\!], \ldots, [\![A_n]\!])$
- A term $\Gamma \vdash M : B$ denotes a function $[\![M]\!] : [\![\Gamma]\!] \to [\![B]\!]$

The denotations of types and terms is given in Figure 4.

The denotations of `Core-`λ types

$$
\begin{array}{lcl}
[\![\texttt{Boolean}]\!] & = & \{\texttt{true}, \texttt{false}\} \\
[\![\texttt{P}]\!] & = & \text{see Appendix A} \\
[\![(T_1, \ldots, T_n)]\!] & = & ([\![T_1]\!], \ldots, [\![T_n]\!]) \\
[\![A \rightarrow B]\!] & = & [\![A]\!] \rightarrow [\![B]\!]
\end{array}
$$

The denotations of `Core-`λ terms

$$
\begin{array}{lcl}
[\![\texttt{x}]\!]\langle(\rho, \texttt{x} \mapsto x, \rho')\rangle & = & x \\
[\![C(d, T)]\!]\langle\rho\rangle & = & d \\
[\![(\overline{M_i})]\!]\langle\rho\rangle & = & (\overline{[\![M_i]\!]\langle\rho\rangle}) \\
[\![\delta\langle N\rangle]\!]\langle\rho\rangle & = & ([\![\delta]\!]\langle\rho\rangle)\langle v\rangle \textit{ where } v = [\![N]\!]\langle\rho\rangle \\
[\![\lambda \texttt{x}.M]\!]\langle\rho\rangle & = & \lambda x.[\![M]\!]\langle(\rho, \texttt{x} \mapsto x)\rangle \\
[\![M_f\langle N\rangle]\!]\langle\rho\rangle & = & ([\![M_f]\!]\langle\rho\rangle)\langle v\rangle \textit{ where } v = [\![N]\!]\langle\rho\rangle \\
[\![M_I.\texttt{m}\langle\overline{N_i}\rangle]\!]\langle\rho\rangle & = & ([\![M_I]\!]\langle\rho\rangle).m\langle\overline{v_i}\rangle \textit{ where } \overline{v_i = [\![N_i]\!]\langle\rho\rangle}
\end{array}
$$

Figure 4: Denotational semantics of `Core-`λ

5 Serialization

This section defines a binary format, which is used to store ErgoTree contracts in persistent stores, to transfer them over wire and to enable cross-platform interoperation.

Terms of the language described in Section 2 can be serialized to array of bytes to be stored in Ergo blockchain (e.g. as Box.propositionBytes).

When the guarding script of an input box of a transaction is validated the `propositionBytes` array is deserialized to an ErgoTree IR (called ErgoTree), which can be evaluated as it is specified in Section 4.

Here we specify the serialization procedure in general. The serialization format of ErgoTree terms and types is specified in Appendix C and **??** correspondingly.

Table 2 shows size limits which are checked during contract deserialization.

Name	Value	Description
VLQ_{max}	10	Maximum size of VLQ encoded byte sequence (See VLQ formats)
$Type_{max}$	100	Maximum size of serialized type term (see Type format)
$Data_{max}$	$10Kb$	Maximum size of serialized data instance (see Data format)
$Const_{max}$	$= Type_{max} + Data_{max}$	Maximum size of serialized data instance (see Const format)
$Expr_{max}$	$1Kb$	Maximum size of serialized ErgoTree term (see Expr format)
$ErgoTree_{max}$	$24Kb$	Maximum size of serialized ErgoTree contract (see ErgoTree format)

Table 1: Serialization limits

All serialization formats which are uses and defined thoughout this section are listed in Table 2.

Format	#bytes	Description
`Byte`	1	8-bit signed two's-complement integer
`Short`	2	16-bit signed two's-complement integer (big-endian)
`Int`	4	32-bit signed two's-complement integer (big-endian)
`Long`	8	64-bit signed two's-complement integer (big-endian)
`UByte`	1	8-bit unsigned integer
`UShort`	2	16-bit unsigned integer (big-endian)
`UInt`	4	32-bit unsigned integer (big-endian)
`ULong`	8	64-bit unsigned integer (big-endian)
`VLQ(UShort)`	[1..3]	Encoded unsigned `Short` value using VLQ. See [VLQa, VLQb] and E.1
`VLQ(UInt)`	[1..5]	Encoded unsigned 32-bit integer using VLQ.
`VLQ(ULong)`	$[1..VLQ_{max}]$	Encoded unsigned 64-bit integer using VLQ.
`Bits`	$[1..Bits_{max}]$	A collection of bits packed in a sequence of bytes.
`Bytes`	$[1..Bytes_{max}]$	A sequence (block) of bytes. The size of the block should either stored elsewhere or wellknown.
`Type`	$[1..Type_{max}]$	Serialized type terms of ErgoTree. See 5.1
`Data`	$[1..Data_{max}]$	Serialized ErgoTree values. See 5.2
`GroupElement`	33	Serialized elements of eliptic curve group. See 5.2.1
`SigmaProp`	$[1..SigmaProp_{max}]$	Serialized sigma propositions. See 5.2.2
`Box`	$[1..Box_{max}]$	Serialized box data. See 5.2.3
`AvlTree`	44	Serialized dynamic dictionary digest. See 5.2.4
`Const`	$[1..Const_{max}]$	Serialized ErgoTree constants (values with types). See 5.3
`Expr`	$[1..Expr_{max}]$	Serialized expression terms of ErgoTree. See 5.4
`ErgoTree`	$[1..ErgoTree_{max}]$	Serialized instances of ErgoTree contracts. See 5.5

Table 2: Serialization formats

Table 2 introduce a name for each format and also shows the number of bytes each format may occupy in the byte stream. We use $[1..n]$ notation when serialization may produce from 1 to n bytes depending of actual data instance.

Serialization format of ErgoTree is optimized for compact storage. In many cases serialization procedure is data dependent and thus have branching logic. To express this complex serialization logic we use *pseudo-language operators* like `for, match, if, optional` which allow to specify a *structure* on *simple serialization slots*. Each *slot* specifies a fragment of serialized stream of bytes, whereas *operators* specifiy how the slots are combined together to form the stream of bytes.

5.1 Type Serialization

In this section we describe how the types (like `Int`, `Coll[Byte]`, etc.) are serialized, then we define serialization of typed data. This will give us a basis to describe serialization of Constant nodes of ErgoTree. From that we proceed to serialization of arbitrary ErgoTree trees.

For motivation behind this type encoding please see Appendix D.1.

5.1.1 Distribution of type codes

The whole space of 256 codes is divided as the following:

Interval	**Distribution**
`0x00`	special value to represent undefined type (`NoType` in ErgoTree)
`0x01 - 0x6F(111)`	data types including primitive types, arrays, options aka nullable types, classes (in future), 111 = 255 - 144 different codes
`0x70(112) - 0xFF(255)`	function types `T1 => T2`, 144 = 12 x 12 different codes

Figure 5: Distribution of type codes

5.1.2 Encoding Data Types

There are 9 different values for primitive types and 2 more are reserved for future extensions. Each primitive type has an id in a range 1,...,11 as the following.

Id	**Type**
1	Boolean
2	Byte
3	Short (16 bit)
4	Int (32 bit)
5	Long (64 bit)
6	BigInt (java.math.BigInteger)
7	GroupElement (org.bouncycastle.math.ec.ECPoint)
8	SigmaProp
9	reserved for Char
10	reserved for Double
11	reserved

For each type constructor like `Coll` or `Option` we use the encoding schema defined below. Type constructor has associated *base code* (e.g. 12 for `Coll[_]`, 24 for `Coll[Coll[_]]` etc.), which is multiple of 12. Base code can be added to primitive type id to produce code of constructed type, for example $12 + 1 = 13$ is a code of `Coll[Byte]`. The code of type constructor (12 in this example) is used when type parameter is non-primitive type (e.g. `Coll[(Byte, Int)]`). In this case the code of type constructor is read first, and then recursive descent is performed to read bytes of the parameter type (in this case `(Byte, Int)`) This encoding allows very simple and quick decoding by using div and mod operations.

The interval of codes for data types is divided as the following:

Interval	**Type constructor**	**Description**
0x01 - 0x0B(11)		primitive types (including 2 reserved)
0x0C(12)	`Coll[_]`	Collection of non-primivite types (`Coll[(Int,Boolean)]`)
0x0D(13) - 0x17(23)	`Coll[_]`	Collection of primitive types (`Coll[Byte]`, `Coll[Int]`, etc.)
0x18(24)	`Coll[Coll[_]]`	Nested collection of non-primitive types (`Coll[Coll[(Int,Boolean)]]`)
0x19(25) - 0x23(35)	`Coll[Coll[_]]`	Nested collection of primitive types (`Coll[Coll[Byte]]`, `Coll[Coll[Int]]`)
0x24(36)	`Option[_]`	Option of non-primitive type (`Option[(Int, Byte)]`)
0x25(37) - 0x2F(47)	`Option[_]`	Option of primitive type (`Option[Int]`)
0x30(48)	`Option[Coll[_]]`	Option of Coll of non-primitive type (`Option[Coll[(Int, Boolean)]]`)
0x31(49) - 0x3B(59)	`Option[Coll[_]]`	Option of Coll of primitive type (`Option[Coll[Int]]`)
0x3C(60)	`(_,_)`	Pair of non-primitive types (`((Int, Byte), (Boolean,Box))`, etc.)
0x3D(61) - 0x47(71)	`(_, Int)`	Pair of types where first is primitive (`(_, Int)`)
0x48(72)	`(_,_,_)`	Triple of types
0x49(73) - 0x53(83)	`(Int, _)`	Pair of types where second is primitive (`(Int, _)`)
0x54(84)	`(_,_,_,_)`	Quadruple of types
0x55(85) - 0x5F(95)	`(_, _)`	Symmetric pair of primitive types (`(Int, Int)`, `(Byte,Byte)`, etc.)
0x60(96)	`(_,...,_)`	`Tuple` type with more than 4 items `(Int, Byte, Box, Boolean, Int)`
0x61(97)	`Any`	Any type
0x62(98)	`Unit`	Unit type
0x63(99)	`Box`	Box type
0x64(100)	`AvlTree`	AvlTree type
0x65(101)	`Context`	Context type
0x65(102)	`String`	String
0x66(103)	`IV`	TypeIdent
0x67(104)- 0x6E(110)		reserved for future use
0x6F(111)		Reserved for future `Class` type (e.g. user-defined types)

5.1.3 Encoding Function Types

We use 12 different values for both domain and range types of functions. This gives us $12 * 12 = 144$ function types in total and allows to represent $11 * 11 = 121$ functions over primitive types using just single byte.

Each code F in a range of function types can be represented as $F = D * 12 + R + 112$, where $D, R \in \{0, \dots, 11\}$ - indices of domain and range types correspondingly, 112 - is the first code in an interval of function types.

If $D = 0$ then domain type is not primitive and recursive descent is necessary to write/read domain type.

If $R = 0$ then range type is not primitive and recursive descent is necessary to write/read range type.

5.1.4 Recursive Descent

When an argument of a type constructor is not a primitive type we fallback to the simple encoding schema.

In such a case we emit the special code for the type constructor according to the table above and descend recursively to every child node of the type tree.

We do this descend only for those children whose code cannot be embedded in the parent code. For example, serialization of `Coll[(Int,Boolean)]` proceeds as the following:

1. emit `0x0C` because element of collection is not primitive
2. recursively serialize `(Int, Boolean)`
3. emit `0x3D` because first item in the pair is primitive
4. recursivley serialize `Boolean`
5. emit `0x02` - the code for primitive type `Boolean`

Examples

Type	D	R	Bytes	#Bytes	Comments
`Byte`			1	1	
`Coll[Byte]`			12 + 1 = 13	1	
`Coll[Coll[Byte]]`			24 + 1 = 25	1	
`Option[Byte]`			36 + 1 = 37	1	register
`Option[Coll[Byte]]`			48 + 1 = 49	1	register
`(Int,Int)`			84 + 3 = 87	1	fold
`Box=>Boolean`	7	2	198 = 7*12+2+112	1	exist, forall
`(Int,Int)=>Int`	0	3	115=0*12+3+112, 87	2	fold
`(Int,Boolean)`			60 + 3, 2	2	
`(Int,Box)=>Boolean`	0	2	0*12+2+112, 60+3, 7	3	

5.2 Data Serialization

In ErgoTree all runtime data values have an associated type also available at runtime (this is called *type reification*[Rei]). However serialization format separates data values from its type descriptors. This allows to save space when for example a collection of items is serialized.

The contents of a typed data structure can be fully described by a type tree. For example having a typed data object `d: (Int, Coll[Byte], Boolean)` we can tell that `d` has 3 items, the first item contain 32-bit integer, the second - collection of bytes, and the third - logical true/false value.

To serialize/deserialize typed data we need to know its type descriptor (type tree). Serialization procedure is recursive over type tree and the corresponding subcomponents of an object. For primitive types (the leaves of the type tree) the format is fixed. The data values of ErgoTree types are serialized using predefined function shown in Figure 6.

5.2.1 GroupElement serialization

5.2.2 SigmaProp serialization

5.2.3 Box serialization

5.2.4 AvlTree serialization

5.3 Constant Serialization

`Constant` format is simple and self sufficient to represent any data value in ErgoTree. Every data block of `Constant` format contains both type and data, such it can be stored or wire transfered and then later unambiguously interpreted. The format is shown in Figure 11

5.4 Expression Serialization

Expressions of ErgoTree are serialized as tree data structure using recursive procedure described here.

5.5 ErgoTree serialization

The root of a serializable ErgoTree term is a data structure called ErgoTree which serialization format shown in Figure **??**

Serialized instances of ErgoTree are self sufficient and can be stored and passed around. ErgoTree format defines top-level serialization format of ErgoTree scripts. The interpretation of the byte array depend on the first *header* bytes, which uses VLQ encoding up to 30 bits. Currently we define meaning for only first byte, which may be extended in future versions.

Currently we don't specify interpretation for the second and other bytes of the header. We reserve the possibility to extend header by using Bit 7 == 1 and chain additional bytes as in VLQ. Once the new bytes are required, a new version of the language should be created and implemented via soft-forkability. That new language will give an interpretation for the new bytes.

The default behavior of ErgoTreeSerializer is to preserve original structure of ErgoTree and check consistency. In case of any inconsistency the serializer throws exception.

If constant segregation bit is set to 1 then *constants* collection contains the constants for which there may be `ConstantPlaceholder` nodes in the tree. If is however constant segregation bit is 0, then `constants` collection should be empty and any placeholder in the tree will lead to exception.

5.6 Constant Segregation

Slot	Format	#bytes	Description
`def serializeData`(t, v)			
`match` (t, v)			
`with` $(Unit, v \in [\![Unit]\!])$ // nothing serialized			
`with` $(Boolean, v \in [\![Boolean]\!])$			
v	`Byte`	1	0 or 1 in a single byte
`with` $(Byte, v \in [\![Byte]\!])$			
v	`Byte`	1	in a single byte
`with` $(N, v \in [\![Short]\!]), N \in Short, Int, Long$			
v	`VLQ(ZigZag(`N`))`	[1..3]	16,32,64-bit signed integer encoded using ZigZag and then using VLQ
`with` $(BigInt, v \in [\![BigInt]\!])$			
$bytes = v$`.toByteArray`			
$numBytes$	`VLQ(UInt)`		number of bytes in *bytes* array
$bytes$	`Bytes`		serialized *bytes* array
`with` $(GroupElement, v \in [\![GroupElement]\!])$			
v	`GroupElement`		serialization of GroupElement data. See 5.2.1
`with` $(SigmaProp, v \in [\![SigmaProp]\!])$			
v	`SigmaProp`		serialization of SigmaProp data. See 5.2.2
`with` $(Box, v \in [\![Box]\!])$			
v	`Box`		serialization of Box data. See 5.2.3
`with` $(AvlTree, v \in [\![AvlTree]\!])$			
v	`AvlTree`		serialization of AvlTree data. See 5.2.4
`with` $(Coll[T], v \in [\![Coll[T]]\!])$			
len	`VLQ(UShort)`	[1..3]	length of the collection
`match` (T, v)			
`with` $(Boolean, v \in [\![Coll[Boolean]]\!])$			
$items$	`Bits`	[1..1024]	boolean values packed in bits
`with` $(Byte, v \in [\![Coll[Byte]]\!])$			
$items$	`Bytes`	[1..len]	items of the collection
`otherwise`			
`for` $i = 1$ `to` len			
`serializeData`(T, v_i)			
`end for`			
`end match`			
`end match`			
`end serializeData`			

Figure 6: Data serialization format

Slot	Format	#bytes	Description
`def serialize`(ge)			
`if` $ge.isIdentity$ `then`			
	`Const`		
`else`			
`end if`			
`end def`			

Figure 7: GroupElement serialization format

Slot	Format	#bytes	Description

Figure 8: SigmaProp serialization format

Slot	Format	#bytes	Description

Figure 9: Box serialization format

Slot	Format	#bytes	Description

Figure 10: AvlTree serialization format

Slot	Format	#bytes	Description
type	`Type`	$[1..Type_{max}]$	type of the data instance (see 5.1)
value	`Data`	$[1..Data_{max}]$	serialized data instance (see 5.2)

Figure 11: Constant serialization format

Slot	Format	#bytes	Description
`def serializeExpr(`e`)`			
$e.opCode$	`Byte`	1	opcode of ErgoTree node, used for selection of an appropriate node serializer from Appendix C
`if` $opCode <= LastConstantCode$ `then`			
c	`Const`	$[1..Const_{max}]$	Constant serializaton slot
`else`			
body	Op	$[1..Expr_{max}]$	serialization of operation arguments depending on $e.opCode$ as defined in Appendix C
`end if`			
`end serializeExpr`			

Figure 12: Expression serialization format

Slot	Format	#bytes	Description
header	`VLQ(UInt)`	[1, *]	the first bytes of serialized byte array which determines interpretation of the rest of the array
numConstants	`VLQ(UInt)`	[1, *]	size of *constants* array
`for` $i = 1$ `to` $numConstants$			
$const_i$	`Const`	[1, *]	constant in i-th position
`end for`			
root	`Expr`	[1, *]	If constantSegregationFlag is true, the contains ConstantPlaceholder instead of some Constant nodes. Otherwise may not contain placeholders. It is possible to have both constants and placeholders in the tree, but for every placeholder there should be a constant in *constants* array.

Figure 13: ErgoTree serialization format

Bits	Default Value	Description
Bits 0-2	0	language version (current version == 0)
Bit 3	0	reserved (should be 0)
Bit 4	0	== 1 if constant segregation is used for this ErgoTree (see Section 5.6
Bit 5	0	== 1 - reserved for context dependent costing (should be = 0)
Bit 6	0	reserved for GZIP compression (should be 0)
Bit 7	0	== 1 if the header contains more than 1 byte (should be 0)

Figure 14: ErgoTree *header* bits

6 The Graph

7 Costing

This is how the file name is specified

```
val env: ScriptEnv = Map(
  ScriptNameProp -> s"filename_verify",
...
```

The file should be in `test-out` directory. The graph should have explicit nodes like `CostOf(...)`, which represent access to CostTable entries. The actual cost is counted in the nodes like this `s1340: Int = OpCost(2, List(s1361, s1360), s983)`. Each such node is handled like `costAccumulator.add(` See `CostAccumulator`

How much cost is represented by OpCost node?

1. Symbols s1361, s1360 are dependencies. They represent cost that should be accumulated before s983.
2. If upon handling of OpCost, the dependencies are not yet accumulated, then they are accumulated first, and then s983 is accumulated.
3. the values of s1340 is the value of s983.
4. Thus execution of OpCost, consists of 2 parts: a) data flow b) side effect on CostAccumulator
5. OpCost is special node, interpreted in a special way. See method evaluate in Evaluation.

References

[Rei] Reification. https://en.wikipedia.org/wiki/Reification_(computer_science).

[Ubi] Ubiquitous language. https://www.itworld.com/article/2833252/the-most-wtf-y-programming-languages.html.

[VLQa] Variable-length quantity. https://en.wikipedia.org/wiki/Variable-length_quantity.

[VLQb] Variable-length quantity. https://rosettacode.org/wiki/Variable-length_quantity.

[WTF] The most wtf-y programming languages. https://www.itworld.com/article/2833252/the-most-wtf-y-programming-languages.html.

A Predefined types

Name	Code	IsConstSize	isPrim[1]	isEmbed	isNum	Set of values
Boolean	1	true	true	true	false	$\{\texttt{true}, \texttt{false}\}$
Byte	2	true	true	true	true	$\{-2^7 \dots 2^7 - 1\}$ A.2
Short	3	true	true	true	true	$\{-2^{15} \dots 2^{15} - 1\}$ A.3
Int	4	true	true	true	true	$\{-2^{31} \dots 2^{31} - 1\}$ A.4
Long	5	true	true	true	true	$\{-2^{63} \dots 2^{63} - 1\}$ A.5
BigInt	6	true	true	true	true	$\{-2^{255} \dots 2^{255} - 1\}$ A.6
GroupElement	7	true	true	true	false	$\{p \in \texttt{SecP256K1Point}\}$
SigmaProp	8	true	true	true	false	Sec. A.8
Box	99	false	false	false	false	Sec. A.9
AvlTree	100	true	false	false	false	Sec. A.10
Context	101	false	false	false	false	Sec. A.13
Header	104	true	false	false	false	Sec. A.11
PreHeader	105	true	false	false	false	Sec. A.12
Global	106	true	false	false	false	Sec. A.14

Table 3: Predefined types of ErgoTree

The following subsections are autogenerated from type descriptors of ErgoTree reference implementation.

A.1 Boolean type

A.2 Byte type

A.2.1 `Byte.toByte` method (Code 106.1)

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Parameters	
Result	`Byte`
Serialized as	`PropertyCall`

A.2.2 `Byte.toShort` method (Code 106.2)

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Parameters	
Result	`Short`
Serialized as	`PropertyCall`

A.2.3 `Byte.toInt` method (Code 106.3)

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Parameters	
Result	`Int`
Serialized as	`PropertyCall`

A.2.4 `Byte.toLong` method (Code 106.4)

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Parameters	
Result	`Long`
Serialized as	`PropertyCall`

A.2.5 `Byte.toBigInt` method (Code 106.5)

Description	Converts this numeric value to `BigInt`
Parameters	
Result	`BigInt`
Serialized as	`PropertyCall`

A.2.6 `Byte.toBytes` method (Code 106.6)

Description	Returns a big-endian representation of this numeric value in a collection of bytes. For example, the `Int` value `0x12131415` would yield the collection of bytes `[0x12, 0x13, 0x14, 0x15]`.
Parameters	
Result	`Coll[Byte]`
Serialized as	`PropertyCall`

A.2.7 `Byte.toBits` method (Code 106.7)

Description	Returns a big-endian representation of this numeric in a collection of Booleans. Each boolean corresponds to one bit.
Parameters	
Result	`Coll[Boolean]`
Serialized as	`PropertyCall`

A.3 Short type

A.3.1 `Short.toByte` method (Code 106.1)

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Parameters	
Result	`Byte`
Serialized as	`PropertyCall`

A.3.2 `Short.toShort` method (Code 106.2)

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Parameters	
Result	`Short`
Serialized as	`PropertyCall`

A.3.3 `Short.toInt` method (Code 106.3)

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Parameters	
Result	`Int`
Serialized as	`PropertyCall`

A.3.4 `Short.toLong` method (Code 106.4)

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Parameters	
Result	`Long`
Serialized as	`PropertyCall`

A.3.5 `Short.toBigInt` method (Code 106.5)

Description	Converts this numeric value to `BigInt`
Parameters	
Result	`BigInt`
Serialized as	`PropertyCall`

A.3.6 `Short.toBytes` method (Code 106.6)

Description	Returns a big-endian representation of this numeric value in a collection of bytes. For example, the `Int` value `0x12131415` would yield the collection of bytes `[0x12, 0x13, 0x14, 0x15]`.
Parameters	
Result	`Coll[Byte]`
Serialized as	`PropertyCall`

A.3.7 `Short.toBits` method (Code 106.7)

Description	Returns a big-endian representation of this numeric in a collection of Booleans. Each boolean corresponds to one bit.
Parameters	
Result	`Coll[Boolean]`
Serialized as	`PropertyCall`

A.4 Int type

A.4.1 `Int.toByte` method (Code 106.1)

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Parameters	
Result	`Byte`
Serialized as	`PropertyCall`

A.4.2 `Int.toShort` method (Code 106.2)

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Parameters	
Result	`Short`
Serialized as	`PropertyCall`

A.4.3 `Int.toInt` method (Code 106.3)

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Parameters	
Result	`Int`
Serialized as	`PropertyCall`

A.4.4 `Int.toLong` method (Code 106.4)

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Parameters	
Result	`Long`
Serialized as	`PropertyCall`

A.4.5 `Int.toBigInt` method (Code 106.5)

Description	Converts this numeric value to `BigInt`
Parameters	
Result	`BigInt`
Serialized as	`PropertyCall`

A.4.6 `Int.toBytes` method (Code 106.6)

Description	Returns a big-endian representation of this numeric value in a collection of bytes. For example, the `Int` value `0x12131415` would yield the collection of bytes `[0x12, 0x13, 0x14, 0x15]`.
Parameters	
Result	`Coll[Byte]`
Serialized as	`PropertyCall`

A.4.7 `Int.toBits` method (Code 106.7)

Description	Returns a big-endian representation of this numeric in a collection of Booleans. Each boolean corresponds to one bit.
Parameters	
Result	`Coll[Boolean]`
Serialized as	`PropertyCall`

A.5 Long type

A.5.1 `Long.toByte` method (Code 106.1)

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Parameters	
Result	`Byte`
Serialized as	`PropertyCall`

A.5.2 `Long.toShort` method (Code 106.2)

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Parameters	
Result	`Short`
Serialized as	`PropertyCall`

A.5.3 `Long.toInt` method (Code 106.3)

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Parameters	
Result	`Int`
Serialized as	`PropertyCall`

A.5.4 `Long.toLong` method (Code 106.4)

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Parameters	
Result	`Long`
Serialized as	`PropertyCall`

A.5.5 `Long.toBigInt` method (Code 106.5)

Description	Converts this numeric value to `BigInt`
Parameters	
Result	`BigInt`
Serialized as	`PropertyCall`

A.5.6 `Long.toBytes` method (Code 106.6)

Description	Returns a big-endian representation of this numeric value in a collection of bytes. For example, the `Int` value `0x12131415` would yield the collection of bytes `[0x12, 0x13, 0x14, 0x15]`.
Parameters	
Result	`Coll[Byte]`
Serialized as	`PropertyCall`

A.5.7 `Long.toBits` method (Code 106.7)

Description	Returns a big-endian representation of this numeric in a collection of Booleans. Each boolean corresponds to one bit.
Parameters	
Result	`Coll[Boolean]`
Serialized as	`PropertyCall`

A.6 BigInt type

A.6.1 `BigInt.toByte` method (Code 106.1)

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Parameters	
Result	`Byte`
Serialized as	`PropertyCall`

A.6.2 `BigInt.toShort` method (Code 106.2)

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Parameters	
Result	`Short`
Serialized as	`PropertyCall`

A.6.3 `BigInt.toInt` method (Code 106.3)

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Parameters	
Result	`Int`
Serialized as	`PropertyCall`

A.6.4 `BigInt.toLong` method (Code 106.4)

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Parameters	
Result	`Long`
Serialized as	`PropertyCall`

A.6.5 `BigInt.toBigInt` method (Code 106.5)

Description	Converts this numeric value to `BigInt`
Parameters	
Result	`BigInt`
Serialized as	`PropertyCall`

A.6.6 `BigInt.toBytes` method (Code 106.6)

Description	Returns a big-endian representation of this numeric value in a collection of bytes. For example, the `Int` value `0x12131415` would yield the collection of bytes `[0x12, 0x13, 0x14, 0x15]`.
Parameters	
Result	`Coll[Byte]`
Serialized as	`PropertyCall`

A.6.7 `BigInt.toBits` method (Code 106.7)

Description	Returns a big-endian representation of this numeric in a collection of Booleans. Each boolean corresponds to one bit.
Parameters	
Result	`Coll[Boolean]`
Serialized as	`PropertyCall`

A.7 GroupElement type

A.7.1 `GroupElement.getEncoded` method (Code 7.2)

Description	Get an encoding of the point value.
Parameters	
Result	`Coll[Byte]`
Serialized as	`PropertyCall`

A.7.2 `GroupElement.exp` method (Code 7.3)

Description	Exponentiate this `GroupElement` to the given number. Returns this to the power of k
Parameters	`k : BigInt` // The power
Result	`GroupElement`
Serialized as	`Exponentiate`

A.7.3 `GroupElement.multiply` method (Code 7.4)

Description	Group operation.
Parameters	`other : GroupElement` // other element of the group
Result	`GroupElement`
Serialized as	`MultiplyGroup`

A.7.4 `GroupElement.negate` method (Code 7.5)

Description	Inverse element of the group.
Parameters	
Result	`GroupElement`
Serialized as	`PropertyCall`

A.8 SigmaProp type

Values of `SigmaProp` type hold sigma propositions, which can be proved and verified using Sigma protocols. Each sigma proposition is represented as an expression where sigma protocol primitives such as `ProveDlog`, and `ProveDHTuple` are used as constants and special sigma protocol connectives like `&&`,`||` and `THRESHOLD` are used as operations.

The abstract syntax of sigma propositions is shown in Figure 15.

Set		Syntax	Mnemonic	Description
$Tree \ni t$	:=	`Trivial(b)`	`TrivialProp`	boolean value `b` as sigma proposition
	|	`Dlog(ge)`	`ProveDLog`	knowledge of discrete logarithm of `ge`
	|	`DHTuple(g,h,u,v)`	`ProveDHTuple`	knowledge of Diffie-Hellman tuple
	|	`THRESHOLD`$(k, t_1, \ldots, t_n)$	`THRESHOLD`	knowledge of k out of n secrets
	|	`OR`$(t_1, \ldots, t_n)$	`OR`	knowledge of any one of n secrets
	|	`AND`$(t_1, \ldots, t_n)$	`AND`	knowledge of all n secrets

Figure 15: Abstract syntax of sigma propositions

Every well-formed tree of sigma proposition is a value of type `SigmaProp`, thus following the notation of Section 4 we can define denotation of `SigmaProp`

$$[\![\texttt{SigmaProp}]\!] = \{t \in Tree\}$$

The following methods can be called on all instances of `SigmaProp` type.

A.8.1 `SigmaProp.propBytes` method (Code 8.1)

Description	Serialized bytes of this sigma proposition taken as ErgoTree.
Parameters	
Result	`Coll[Byte]`
Serialized as	`SigmaPropBytes`

A.8.2 `SigmaProp.isProven` method (Code 8.2)

Description	Verify that sigma proposition is proven. (FRONTEND ONLY)
Parameters	
Result	`Boolean`

For a list of primitive operations on `SigmaProp` type see Appendix B.

A.9 Box type

A.9.1 `Box.value` method (Code 99.1)

Description	Mandatory: Monetary value, in Ergo tokens (NanoErg unit of measure)
Parameters	
Result	`Long`
Serialized as	`ExtractAmount`

A.9.2 `Box.propositionBytes` method (Code 99.2)

Description	Serialized bytes of guarding script, which should be evaluated to true in order to open this box. (aka spend it in a transaction)
Parameters	
Result	`Coll[Byte]`
Serialized as	`ExtractScriptBytes`

A.9.3 `Box.bytes` method (Code 99.3)

Description	Serialized bytes of this box's content, including proposition bytes.
Parameters	
Result	`Coll[Byte]`
Serialized as	`ExtractBytes`

A.9.4 `Box.bytesWithoutRef` method (Code 99.4)

Description	Serialized bytes of this box's content, excluding transactionId and index of output.
Parameters	
Result	`Coll[Byte]`
Serialized as	`ExtractBytesWithNoRef`

A.9.5 `Box.id` method (Code 99.5)

Description	Blake2b256 hash of this box's content, basically equals to `blake2b256(bytes)`
Parameters	
Result	`Coll[Byte]`
Serialized as	`ExtractId`

A.9.6 `Box.creationInfo` method (Code 99.6)

Description	If `tx` is a transaction which generated this box, then `creationInfo._1` is a height of the tx's block. The `creationInfo._2` is a serialized transaction identifier followed by box index in the transaction outputs.
Parameters	
Result	`(Int,Coll[Byte])`
Serialized as	`ExtractCreationInfo`

A.9.7 `Box.getReg` method (Code 99.7)

Description	Extracts register by id and type. Type param `T` expected type of the register. Returns `Some(value)` if the register is defined and has given type and `None` otherwise
Parameters	`regId : Int` // zero-based identifier of the register.
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.8 `Box.tokens` method (Code 99.8)

Description	Secondary tokens
Parameters	
Result	`Coll[(Coll[Byte],Long)]`
Serialized as	`PropertyCall`

A.9.9 `Box.R0` method (Code 99.9)

Description	Monetary value, in Ergo tokens
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.10 `Box.R1` method (Code 99.10)

Description	Guarding script
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.11 `Box.R2` method (Code 99.11)

Description	Secondary tokens
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.12 `Box.R3` method (Code 99.12)

Description	Reference to transaction and output id where the box was created
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.13 `Box.R4` method (Code 99.13)

Description	Non-mandatory register
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.14 `Box.R5` method (Code 99.14)

Description	Non-mandatory register
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.15 `Box.R6` method (Code 99.15)

Description	Non-mandatory register
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.16 `Box.R7` method (Code 99.16)

Description	Non-mandatory register
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.17 `Box.R8` method (Code 99.17)

Description	Non-mandatory register
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.9.18 `Box.R9` method (Code 99.18)

Description	Non-mandatory register
Parameters	
Result	`Option[T]`
Serialized as	`ExtractRegisterAs`

A.10 `AvlTree` type

A.10.1 `AvlTree.digest` method (Code 100.1)

Description	Returns digest of the state represented by this tree. Authenticated tree `digest` = `root hash bytes` ++ `tree height`
Parameters	
Result	`Coll[Byte]`
Serialized as	`PropertyCall`

A.10.2 `AvlTree.enabledOperations` method (Code 100.2)

Description	Flags of enabled operations packed in single byte. `isInsertAllowed == (enabledOperations & 0x01) != 0` `isUpdateAllowed == (enabledOperations & 0x02) != 0` `isRemoveAllowed == (enabledOperations & 0x04) != 0`
Parameters	
Result	`Byte`
Serialized as	`PropertyCall`

A.10.3 `AvlTree.keyLength` method (Code 100.3)

Description	
Parameters	
Result	`Int`
Serialized as	`PropertyCall`

A.10.4 `AvlTree.valueLengthOpt` method (Code 100.4)

Description	
Parameters	
Result	`Option[Int]`
Serialized as	`PropertyCall`

A.10.5 `AvlTree.isInsertAllowed` method (Code 100.5)

Description	
Parameters	
Result	`Boolean`
Serialized as	`PropertyCall`

A.10.6 `AvlTree.isUpdateAllowed` method (Code 100.6)

Description	
Parameters	
Result	`Boolean`
Serialized as	`PropertyCall`

A.10.7 `AvlTree.isRemoveAllowed` method (Code 100.7)

Description	
Parameters	
Result	`Boolean`
Serialized as	`PropertyCall`

A.10.8 `AvlTree.updateOperations` method (Code 100.8)

Description	
Parameters	
Result	`AvlTree`
Serialized as	`MethodCall`

A.10.9 `AvlTree.contains` method (Code 100.9)

Description	
Parameters	
Result	`Boolean`
Serialized as	`MethodCall`

A.10.10 `AvlTree.get` method (Code 100.10)

Description	
Parameters	
Result	`Option[Coll[Byte]]`
Serialized as	`MethodCall`

A.10.11 `AvlTree.getMany` method (Code 100.11)

Description	
Parameters	
Result	`Coll[Option[Coll[Byte]]]`
Serialized as	`MethodCall`

A.10.12 `AvlTree.insert` method (Code 100.12)

Description	
Parameters	
Result	`Option[AvlTree]`
Serialized as	`MethodCall`

A.10.13 `AvlTree.update` method (Code 100.13)

Description	
Parameters	
Result	`Option[AvlTree]`
Serialized as	`MethodCall`

A.10.14 `AvlTree.remove` method (Code 100.14)

Description	
Parameters	
Result	`Option[AvlTree]`
Serialized as	`MethodCall`

A.10.15 `AvlTree.updateDigest` method (Code 100.15)

Description	
Parameters	
Result	`AvlTree`
Serialized as	`MethodCall`

A.11 Header type

A.11.1 `Header.id` method (Code 104.1)

Description	
Parameters	`arg0 : Header //`
Result	`Coll[Byte]`

A.11.2 `Header.version` method (Code 104.2)

Description	
Parameters	`arg0 : Header //`
Result	`Byte`

A.11.3 `Header.parentId` method (Code 104.3)

Description	
Parameters	`arg0 : Header //`
Result	`Coll[Byte]`

A.11.4 `Header.ADProofsRoot` method (Code 104.4)

Description	
Parameters	`arg0 : Header //`
Result	`Coll[Byte]`

A.11.5 `Header.stateRoot` method (Code 104.5)

Description	
Parameters	`arg0 : Header //`
Result	`AvlTree`

A.11.6 `Header.transactionsRoot` method (Code 104.6)

Description	
Parameters	`arg0 : Header //`
Result	`Coll[Byte]`

A.11.7 `Header.timestamp` method (Code 104.7)

Description	
Parameters	`arg0 : Header //`
Result	`Long`

A.11.8 `Header.nBits` method (Code 104.8)

Description	
Parameters	`arg0 : Header //`
Result	`Long`

A.11.9 `Header.height` method (Code 104.9)

Description	
Parameters	`arg0 : Header //`
Result	`Int`

A.11.10 `Header.extensionRoot` method (Code 104.10)

Description	
Parameters	`arg0 : Header //`
Result	`Coll[Byte]`

A.11.11 `Header.minerPk` method (Code 104.11)

Description	
Parameters	`arg0 : Header //`
Result	`GroupElement`

A.11.12 `Header.powOnetimePk` method (Code 104.12)

Description	
Parameters	`arg0 : Header //`
Result	`GroupElement`

A.11.13 `Header.powNonce` method (Code 104.13)

Description	
Parameters	`arg0 : Header //`
Result	`Coll[Byte]`

A.11.14 `Header.powDistance` method (Code 104.14)

Description	
Parameters	`arg0 : Header //`
Result	`BigInt`

A.11.15 `Header.votes` method (Code 104.15)

Description	
Parameters	`arg0 : Header //`
Result	`Coll[Byte]`

A.12 PreHeader type

A.12.1 `PreHeader.version` method (Code 105.1)

Description	
Parameters	`arg0 : PreHeader //`
Result	`Byte`

A.12.2 `PreHeader.parentId` method (Code 105.2)

Description	
Parameters	`arg0 : PreHeader //`
Result	`Coll[Byte]`

A.12.3 `PreHeader.timestamp` method (Code 105.3)

Description	
Parameters	`arg0 : PreHeader //`
Result	`Long`

A.12.4 `PreHeader.nBits` method (Code 105.4)

Description	
Parameters	`arg0 : PreHeader //`
Result	`Long`

A.12.5 `PreHeader.height` method (Code 105.5)

Description	
Parameters	`arg0 : PreHeader //`
Result	`Int`

A.12.6 `PreHeader.minerPk` method (Code 105.6)

Description	
Parameters	`arg0 : PreHeader //`
Result	`GroupElement`

A.12.7 `PreHeader.votes` method (Code 105.7)

Description	
Parameters	`arg0 : PreHeader //`
Result	`Coll[Byte]`

A.13 Context type

A.13.1 `Context.dataInputs` method (Code 101.1)

Description	
Parameters	`arg0 : Context //`
Result	`Coll[Box]`

A.13.2 `Context.headers` method (Code 101.2)

Description	
Parameters	`arg0 : Context //`
Result	`Coll[Header]`

A.13.3 `Context.preHeader` method (Code 101.3)

Description	
Parameters	`arg0 : Context //`
Result	`PreHeader`

A.13.4 `Context.INPUTS` method (Code 101.4)

Description	
Parameters	`arg0 : Context //`
Result	`Coll[Box]`

A.13.5 `Context.OUTPUTS` method (Code 101.5)

Description	
Parameters	`arg0 : Context //`
Result	`Coll[Box]`

A.13.6 `Context.HEIGHT` method (Code 101.6)

Description	
Parameters	`arg0 : Context //`
Result	`Int`

A.13.7 `Context.SELF` method (Code 101.7)

Description	
Parameters	`arg0 : Context //`
Result	`Box`

A.13.8 `Context.selfBoxIndex` method (Code 101.8)

Description	
Parameters	`arg0 : Context //`
Result	`Int`

A.13.9 `Context.LastBlockUtxoRootHash` method (Code 101.9)

Description	
Parameters	`arg0 : Context //`
Result	`AvlTree`

A.13.10 `Context.minerPubKey` method (Code 101.10)

Description	
Parameters	`arg0 : Context //`
Result	`Coll[Byte]`

A.13.11 `Context.getVar` method (Code 101.11)

Description	
Parameters	`arg0 : Context //` `arg1 : Byte //`
Result	`Option[T]`

A.14 Global type

A.14.1 `SigmaDslBuilder.groupGenerator` method (Code 106.1)

Description	
Parameters	
Result	`GroupElement`
Serialized as	`GroupGenerator`

A.14.2 `SigmaDslBuilder.xor` method (Code 106.2)

Description	

Parameters	`arg0 : SigmaDslBuilder //`
	`arg1 : Coll[Byte] //`
	`arg2 : Coll[Byte] //`

Result	`Coll[Byte]`

A.15 Coll type

A.15.1 `SCollection.size` method (Code 12.1)

Description	The size of the collection in elements.
Parameters	
Result	`Int`
Serialized as	`SizeOf`

A.15.2 `SCollection.getOrElse` method (Code 12.2)

Description	Return the element of collection if `index` is in range `0 .. size-1`
Parameters	`index : Int` // index of the element of this collection `default : IV` // value to return when `index` is out of range
Result	`IV`
Serialized as	`ByIndex`

A.15.3 `SCollection.map` method (Code 12.3)

Description	Builds a new collection by applying a function to all elements of this collection. Returns a new collection of type `Coll[B]` resulting from applying the given function `f` to each element of this collection and collecting the results.
Parameters	`f : (IV) => OV` // the function to apply to each element
Result	`Coll[OV]`
Serialized as	`MapCollection`

A.15.4 `SCollection.exists` method (Code 12.4)

Description	Tests whether a predicate holds for at least one element of this collection. Returns `true` if the given predicate `p` is satisfied by at least one element of this collection, otherwise `false`
Parameters	`p : (IV) => Boolean` // the predicate used to test elements
Result	`Boolean`
Serialized as	`Exists`

A.15.5 `SCollection.fold` method (Code 12.5)

Description	Applies a binary operator to a start value and all elements of this collection, going left to right.
Parameters	`zero : OV` // a starting value `op : (OV,IV) => OV` // the binary operator
Result	`OV`
Serialized as	`Fold`

A.15.6 `SCollection.forall` method (Code 12.6)

Description	Tests whether a predicate holds for all elements of this collection. Returns `true` if this collection is empty or the given predicate `p` holds for all elements of this collection, otherwise `false`.
Parameters	`p : (IV) => Boolean` // the predicate used to test elements
Result	`Boolean`
Serialized as	`ForAll`

A.15.7 `SCollection.slice` method (Code 12.7)

Description	Selects an interval of elements. The returned collection is made up of all elements `x` which satisfy the invariant: `from <= indexOf(x) < until`
Parameters	`from : Int` // the lowest index to include from this collection `until : Int` // the lowest index to EXCLUDE from this collection
Result	`Coll[IV]`
Serialized as	`Slice`

A.15.8 `SCollection.filter` method (Code 12.8)

Description	Selects all elements of this collection which satisfy a predicate. Returns a new collection consisting of all elements of this collection that satisfy the given predicate `p`. The order of the elements is preserved.
Parameters	`p : (IV) => Boolean` // the predicate used to test elements.
Result	`Coll[IV]`
Serialized as	`Filter`

A.15.9 `SCollection.append` method (Code 12.9)

Description	Puts the elements of other collection after the elements of this collection (concatenation of 2 collections)
Parameters	`other : Coll[IV]` // the collection to append at the end of this
Result	`Coll[IV]`
Serialized as	`Append`

A.15.10 `SCollection.apply` method (Code 12.10)

Description	The element at given index. Indices start at `0`; `xs.apply(0)` is the first element of collection `xs`. Note the indexing syntax `xs(i)` is a shorthand for `xs.apply(i)`. Returns the element at the given index. Throws an exception if `i < 0` or `length <= i`
Parameters	`i : Int` // the index
Result	`IV`
Serialized as	`ByIndex`

A.15.11 `SCollection.indices` method (Code 12.14)

Description	Produces the range of all indices of this collection as a new collection containing [0 .. length-1] values.
Parameters	
Result	`Coll[Int]`
Serialized as	`PropertyCall`

A.15.12 `SCollection.flatMap` method (Code 12.15)

Description	Builds a new collection by applying a function to all elements of this collection and using the elements of the resulting collections. Function `f` is constrained to be of the form `x => x.someProperty`, otherwise it is illegal. Returns a new collection of type `Coll[B]` resulting from applying the given collection-valued function `f` to each element of this collection and concatenating the results.
Parameters	`f : (IV) => Coll[OV]` // the function to apply to each element.
Result	`Coll[OV]`
Serialized as	`MethodCall`

A.15.13 `SCollection.patch` method (Code 12.19)

Description	
Parameters	
Result	`Coll[IV]`
Serialized as	`MethodCall`

A.15.14 `SCollection.updated` method (Code 12.20)

Description	
Parameters	
Result	`Coll[IV]`
Serialized as	`MethodCall`

A.15.15 `SCollection.updateMany` method (Code 12.21)

Description	
Parameters	
Result	`Coll[IV]`
Serialized as	`MethodCall`

A.15.16 `SCollection.indexOf` method (Code 12.26)

Description	
Parameters	
Result	`Int`
Serialized as	`MethodCall`

A.15.17 `SCollection.zip` method (Code 12.29)

Description	
Parameters	
Result	`Coll[(IV,OV)]`
Serialized as	`MethodCall`

A.16 Option type

A.16.1 `SOption.isDefined` method (Code 36.2)

Description	Returns `true` if the option is an instance of `Some`, `false` otherwise.
Parameters	
Result	`Boolean`
Serialized as	`OptionIsDefined`

A.16.2 `SOption.get` method (Code 36.3)

Description	Returns the option's value. The option must be nonempty. Throws exception if the option is empty.
Parameters	
Result	`T`
Serialized as	`OptionGet`

A.16.3 `SOption.getOrElse` method (Code 36.4)

Description	Returns the option's value if the option is nonempty, otherwise return the result of evaluating `default`.
Parameters	`default : T` // the default value
Result	`T`
Serialized as	`OptionGetOrElse`

A.16.4 `SOption.map` method (Code 36.7)

Description	Returns a `Some` containing the result of applying `f` to this option's value if this option is nonempty. Otherwise return `None`.
Parameters	`f : (T) => R` // the function to apply
Result	`Option[R]`
Serialized as	`MethodCall`

A.16.5 `SOption.filter` method (Code 36.8)

Description	Returns this option if it is nonempty and applying the predicate `p` to this option's value returns true. Otherwise, return `None`.
Parameters	`p : (T) => Boolean` // the predicate used for testing
Result	`Option[T]`
Serialized as	`MethodCall`

B Predefined global functions

Code	Mnemonic	Signature	Description

115	`ConstantPlaceholder`	`placeholder:`
		`(Int)`
		`=> T`
			Create special ErgoTree node which can be replaced by constant with given id.

116	`SubstConstants`	`substConstants:`
		`(Coll[Byte], Coll[Int], Coll[T])`
		`=> Coll[Byte]`
			...

122	`LongToByteArray`	`longToByteArray:`
		`(Long)`
		`=> Coll[Byte]`
			Converts `Long` value to big-endian bytes representation.

123	`ByteArrayToBigInt`	`byteArrayToBigInt:`
		`(Coll[Byte])`
		`=> BigInt`
			Convert big-endian bytes representation (Coll[Byte]) to BigInt value.

124	`ByteArrayToLong`	`byteArrayToLong:`
		`(Coll[Byte])`
		`=> Long`
			Convert big-endian bytes representation (Coll[Byte]) to Long value.

125	`Downcast`	`downcast:`
		`(T)`
		`=> R`
			Cast this numeric value to a smaller type (e.g. Long to Int). Throws exception if overflow.

126	`Upcast`	`upcast:`
		`(T)`
		`=> R`
			Cast this numeric value to a bigger type (e.g. Int to Long)

140	`SelectField`	`selectField:`
		`(T, Byte)`
		`=> R`
			Select tuple field by its 1-based index. E.g. `input._1` is transformed to `SelectField(input, 1)`

143	`LT`	`<:`
		`(T, T)`
		`=> Boolean`
			Returns `true` is the left operand is less then the right operand, `false` otherwise.

144	`LE`	`<=:`
		`(T, T)`
		`=> Boolean`
			Returns `true` is the left operand is less then or equal to the right operand, `false` otherwise.

145	`GT`	`>:`
		`(T, T)`
		`=> Boolean`
			Returns `true` is the left operand is greater then the right operand, `false` otherwise.

146	`GE`	`>=:`
		`(T, T)`
		`=> Boolean`
			Returns `true` is the left operand is greater then or equal to the right operand, `false` otherwise.

147	`EQ`	`==:`
		`(T, T)`
		`=> Boolean`
			Compare equality of `left` and `right` arguments

148	`NEQ`	`!=:`
		`(T, T)`
		`=> Boolean`
			Compare inequality of `left` and `right` arguments

149	`If`	`if:`
		`(Boolean, T, T)`
		`=> T`
			Compute condition, if true then compute trueBranch else compute falseBranch

150	`AND`	`allOf:`
		`(Coll[Boolean])`
		`=> Boolean`
			Returns true if *all* the elements in collection are `true`.

151	`OR`	`anyOf:`
		`(Coll[Boolean])`
		`=> Boolean`
			Returns true if *any* the elements in collection are `true`.

152	`AtLeast`	`atLeast:`
		`(Int, Coll[SigmaProp])`
		`=> SigmaProp`
			...

153	`Minus`	`-:`
		`(T, T)`
		`=> T`
			Returns a result of subtracting second numeric operand from the first.

154	`Plus`	`+:`
		`(T, T)`
		`=> T`
			Returns a sum of two numeric operands

155	`Xor`	`binary_|:`
		`(Coll[Byte], Coll[Byte])`
		`=> Coll[Byte]`
			Byte-wise XOR of two collections of bytes

156	`Multiply`	`*:`
		`(T, T)`
		`=> T`
			Returns a multiplication of two numeric operands

157	`Division`	`/:`
		`(T, T)`
		`=> T`
			Integer division of the first operand by the second operand.

158	`Modulo`	`%:`
		`(T, T)`
		`=> T`
			Reminder from division of the first operand by the second operand.

161	`Min`	`min:`
		`(T, T)`
		`=> T`
			Minimum value of two operands.

162	`Max`	`max:`
		`(T, T)`
		`=> T`
			Maximum value of two operands.

182	`CreateAvlTree`	`avlTree:`
		`(Byte, Coll[Byte], Int, Option[Int])`
		`=> AvlTree`
			Construct a new authenticated dictionary with given parameters and tree root digest.

183	`TreeLookup`	`treeLookup:`
		`(AvlTree, Coll[Byte], Coll[Byte])`
		`=> Option[Coll[Byte]]`	

203	`CalcBlake2b256`	`blake2b256:`
		`(Coll[Byte])`
		`=> Coll[Byte]`
			Calculate Blake2b hash from `input` bytes.

204	`CalcSha256`	`sha256:`
		`(Coll[Byte])`
		`=> Coll[Byte]`
			Calculate Sha256 hash from `input` bytes.

205	CreateProveDlog	proveDlog: (GroupElement) => SigmaProp	ErgoTree operation to create a new SigmaProp value representing public key of discrete logarithm signature protocol.
206	CreateProveDHTuple	proveDHTuple: (GroupElement, GroupElement, GroupElement, GroupElement) => SigmaProp	ErgoTree operation to create a new SigmaProp value representing public key of Diffie Hellman signature protocol. Common input: (g,h,u,v)
209	BoolToSigmaProp	sigmaProp: (Boolean) => SigmaProp	...
212	DeserializeContext	executeFromVar: (Byte) => T	...
213	DeserializeRegister	executeFromSelfReg: (Byte, Option[T]) => T	...
218	Apply	apply: ((T) => R, T) => R	Apply the function to the arguments.
227	GetVar	getVar: (Byte) => Option[T]	Get context variable with given varId and type.
234	SigmaAnd	allZK: (Coll[SigmaProp]) => SigmaProp	Returns sigma proposition which is proven when *all* the elements in collection are proven.
235	SigmaOr	anyZK: (Coll[SigmaProp]) => SigmaProp	Returns sigma proposition which is proven when *any* of the elements in collection is proven.
236	BinOr	||: (Boolean, Boolean) => Boolean	Logical OR of two operands
237	BinAnd	&&: (Boolean, Boolean) => Boolean	Logical AND of two operands
238	DecodePoint	decodePoint: (Coll[Byte]) => GroupElement	Convert Coll[Byte] to GroupElement using GroupElementSerializer
239	LogicalNot	unary_!: (Boolean) => Boolean	Logical NOT operation. Returns true if input is false and false if input is true.
240	Negation	unary_-: (T) => T	Negates numeric value x by returning -x.
241	BitInversion	unary_~: (T) => T	Invert every bit of the numeric value.
242	BitOr	bit_|: (T, T) => T	Bitwise OR of two numeric operands.
243	BitAnd	bit_&: (T, T) => T	Bitwise AND of two numeric operands.
244	BinXor	^: (Boolean, Boolean) => Boolean	Logical XOR of two operands
245	BitXor	bit_^: (T, T) => T	Bitwise XOR of two numeric operands.
246	BitShiftRight	bit_>>: (T, T) => T	Right shift of bits.
247	BitShiftLeft	bit_<<: (T, T) => T	Left shift of bits.
248	BitShiftRightZeroed	bit_>>>: (T, T) => T	Right shift of bits.
255	XorOf	xorOf: (Coll[Boolean]) => Boolean	Similar to allOf, but performing logical XOR operation between all conditions instead of &&

Morphic : This table is autogenerated from sigma operation descriptors. See SigmaPredef.scala

B.0.1 placeholder method (Code 115)

Description	Create special ErgoTree node which can be replaced by constant with given id.
Parameters	index : Int // index of the constant in ErgoTree header
Result	T
Serialized as	ConstantPlaceholder

B.0.2 `substConstants` method (Code 116)

Description	Transforms serialized bytes of ErgoTree with segregated constants by replacing constants at given positions with new values. This operation allow to use serialized scripts as pre-defined templates. The typical usage is "check that output box have proposition equal to given script bytes, where minerPk (constants(0)) is replaced with currentMinerPk". Each constant in original scriptBytes have SType serialized before actual data (see ConstantSerializer). During substitution each value from newValues is checked to be an instance of the corresponding type. This means, the constants during substitution cannot change their types. Returns original scriptBytes array where only specified constants are replaced and all other bytes remain exactly the same.
Parameters	`scriptBytes : Coll[Byte]` // serialized ErgoTree with ConstantSegregationFlag set to 1. `positions : Coll[Int]` // zero based indexes in ErgoTree.constants array which shoul `newValues : Coll[T]` // new values to be injected into the corresponding positions i
Result	`Coll[Byte]`
Serialized as	`SubstConstants`

B.0.3 `longToByteArray` method (Code 122)

Description	Converts `Long` value to big-endian bytes representation.
Parameters	`input : Long` // value to convert
Result	`Coll[Byte]`
Serialized as	`LongToByteArray`

B.0.4 `byteArrayToBigInt` method (Code 123)

Description	Convert big-endian bytes representation (Coll[Byte]) to BigInt value.
Parameters	`input : Coll[Byte]` // collection of bytes in big-endian format
Result	`BigInt`
Serialized as	`ByteArrayToBigInt`

B.0.5 `byteArrayToLong` method (Code 124)

Description	Convert big-endian bytes representation (Coll[Byte]) to Long value.
Parameters	`input : Coll[Byte]` // collection of bytes in big-endian format
Result	`Long`
Serialized as	`ByteArrayToLong`

B.0.6 `downcast` method (Code 125)

Description	Cast this numeric value to a smaller type (e.g. Long to Int). Throws exception if overflow.
Parameters	`input : T` // value to cast
Result	`R`
Serialized as	`Downcast`

B.0.7 `upcast` method (Code 126)

Description	Cast this numeric value to a bigger type (e.g. Int to Long)
Parameters	`input : T` // value to cast
Result	`R`
Serialized as	`Upcast`

B.0.8 `selectField` method (Code 140)

Description	Select tuple field by its 1-based index. E.g. `input._1` is transformed to `SelectField(input, 1)`
Parameters	`input : T` // tuple of items `fieldIndex : Byte` // index of an item to select
Result	`R`
Serialized as	`SelectField`

B.0.9 `<` method (Code 143)

Description	Returns `true` is the left operand is less then the right operand, `false` otherwise.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`Boolean`
Serialized as	`LT`

B.0.10 `<=` method (Code 144)

Description	Returns `true` is the left operand is less then or equal to the right operand, `false` otherwise.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`Boolean`
Serialized as	`LE`

B.0.11 `>` method (Code 145)

Description	Returns `true` is the left operand is greater then the right operand, `false` otherwise.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`Boolean`
Serialized as	`GT`

B.0.12 `>=` method (Code 146)

Description	Returns `true` is the left operand is greater then or equal to the right operand, `false` otherwise.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`Boolean`
Serialized as	`GE`

B.0.13 `==` method (Code 147)

Description	Compare equality of `left` and `right` arguments
Parameters	`left : T` // left operand `right : T` // right operand
Result	`Boolean`
Serialized as	`EQ`

B.0.14 `!=` method (Code 148)

Description	Compare inequality of `left` and `right` arguments
Parameters	`left : T` // left operand `right : T` // right operand
Result	`Boolean`
Serialized as	`NEQ`

B.0.15 `if` method (Code 149)

Description	Compute condition, if true then compute trueBranch else compute falseBranch

Parameters	`condition : Boolean` // condition expression
	`trueBranch : T` // expression to execute when `condition == true`
	`falseBranch : T` // expression to execute when `condition == false`

Result	`T`
Serialized as	`If`

B.0.16 `allOf` method (Code 150)

Description	Returns true if *all* the elements in collection are `true`.
Parameters	`conditions : Coll[Boolean]` // a collection of conditions
Result	`Boolean`
Serialized as	`AND`

B.0.17 `anyOf` method (Code 151)

Description	Returns true if *any* the elements in collection are `true`.
Parameters	`conditions : Coll[Boolean]` // a collection of conditions
Result	`Boolean`
Serialized as	`OR`

B.0.18 `atLeast` method (Code 152)

Description	Logical threshold. AtLeast has two inputs: integer `bound` and `children` same as in AND/OR. The result is true if at least `bound` children are proven.
Parameters	`bound : Int` // required minimum of proven children `children : Coll[SigmaProp]` // proposition to be proven/validated
Result	`SigmaProp`
Serialized as	`AtLeast`

B.0.19 `-` method (Code 153)

Description	Returns a result of subtracting second numeric operand from the first.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`Minus`

B.0.20 `+` method (Code 154)

Description	Returns a sum of two numeric operands
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`Plus`

B.0.21 `binary_|` method (Code 155)

Description	Byte-wise XOR of two collections of bytes
Parameters	`left : Coll[Byte]` // left operand `right : Coll[Byte]` // right operand
Result	`Coll[Byte]`
Serialized as	`Xor`

B.0.22 `*` method (Code 156)

Description	Returns a multiplication of two numeric operands
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`Multiply`

B.0.23 / method (Code 157)

Description	Integer division of the first operand by the second operand.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`Division`

B.0.24 % method (Code 158)

Description	Reminder from division of the first operand by the second operand.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`Modulo`

B.0.25 `min` method (Code 161)

Description	Minimum value of two operands.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`Min`

B.0.26 `max` method (Code 162)

Description	Maximum value of two operands.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`Max`

B.0.27 `avlTree` method (Code 182)

Description	Construct a new authenticated dictionary with given parameters and tree root digest.
Parameters	`operationFlags : Byte` // flags of available operations `digest : Coll[Byte]` // hash of merkle tree root `keyLength : Int` // length of dictionary keys in bytes `valueLengthOpt : Option[Int]` // optional width of dictionary values in bytes
Result	`AvlTree`
Serialized as	`CreateAvlTree`

B.0.28 `treeLookup` method (Code 183)

Description	
Parameters	`tree : AvlTree` // tree to lookup the key `key : Coll[Byte]` // a key of an item in the `tree` to lookup `proof : Coll[Byte]` // proof to perform verification of the operation
Result	`Option[Coll[Byte]]`
Serialized as	`TreeLookup`

B.0.29 `blake2b256` method (Code 203)

Description	Calculate Blake2b hash from `input` bytes.
Parameters	`input : Coll[Byte]` // collection of bytes
Result	`Coll[Byte]`
Serialized as	`CalcBlake2b256`

B.0.30 `sha256` method (Code 204)

Description	Calculate Sha256 hash from `input` bytes.
Parameters	`input : Coll[Byte]` // collection of bytes
Result	`Coll[Byte]`
Serialized as	`CalcSha256`

B.0.31 `proveDlog` method (Code 205)

Description	ErgoTree operation to create a new `SigmaProp` value representing public key of discrete logarithm signature protocol.
Parameters	`value : GroupElement` // element of elliptic curve group
Result	`SigmaProp`
Serialized as	`CreateProveDlog`

B.0.32 `proveDHTuple` method (Code 206)

Description	ErgoTree operation to create a new SigmaProp value representing public key of Diffie Hellman signature protocol. Common input: (g,h,u,v)
Parameters	`g : GroupElement` // `h : GroupElement` // `u : GroupElement` // `v : GroupElement` //
Result	`SigmaProp`
Serialized as	`CreateProveDHTuple`

B.0.33 `sigmaProp` method (Code 209)

Description	Embedding of `Boolean` values to `SigmaProp` values. As an example, this operation allows boolean experessions to be used as arguments of `atLeast(..., sigmaProp(boolExpr), ...)` operation. During execution results to either `TrueProp` or `FalseProp` values of `SigmaProp` type.
Parameters	`condition : Boolean` // boolean value to embed in SigmaProp value
Result	`SigmaProp`
Serialized as	`BoolToSigmaProp`

B.0.34 `executeFromVar` method (Code 212)

Description	Extracts context variable as `Coll[Byte]`, deserializes it to script and then executes this script in the current context. The original `Coll[Byte]` of the script is available as `getVar[Coll[Byte]](id)`. Type parameter `V` result type of the deserialized script. Throws an exception if the actual script type doesn't conform to T. Returns a result of the script execution in the current context
Parameters	`id : Byte` // identifier of the context variable
Result	`T`
Serialized as	`DeserializeContext`

B.0.35 `executeFromSelfReg` method (Code 213)

Description	Extracts SELF register as `Coll[Byte]`, deserializes it to script and then executes this script in the current context. The original `Coll[Byte]` of the script is available as `SELF.getReg[Coll[Byte]](id)`. Type parameter `T` result type of the deserialized script. Throws an exception if the actual script type doesn't conform to `T`. Returns a result of the script execution in the current context
Parameters	`id : Byte` // identifier of the register `default : Option[T]` // optional default value, if register is not available
Result	`T`
Serialized as	`DeserializeRegister`

B.0.36 `apply` method (Code 218)

Description	Apply the function to the arguments.
Parameters	`func : (T) => R` // function which is applied `args : T` // list of arguments
Result	`R`
Serialized as	`Apply`

B.0.37 `getVar` method (Code 227)

Description	Get context variable with given `varId` and type.
Parameters	`varId : Byte` // `Byte` identifier of context variable
Result	`Option[T]`
Serialized as	`GetVar`

B.0.38 `allZK` method (Code 234)

Description	Returns sigma proposition which is proven when *all* the elements in collection are proven.
Parameters	`propositions : Coll[SigmaProp]` // a collection of propositions
Result	`SigmaProp`
Serialized as	`SigmaAnd`

B.0.39 `anyZK` method (Code 235)

Description	Returns sigma proposition which is proven when *any* of the elements in collection is proven.
Parameters	`propositions : Coll[SigmaProp]` // a collection of propositions
Result	`SigmaProp`
Serialized as	`SigmaOr`

B.0.40 `||` method (Code 236)

Description	Logical OR of two operands
Parameters	`left : Boolean` // left operand `right : Boolean` // right operand
Result	`Boolean`
Serialized as	`BinOr`

B.0.41 `&&` method (Code 237)

Description	Logical AND of two operands
Parameters	`left : Boolean` // left operand `right : Boolean` // right operand
Result	`Boolean`
Serialized as	`BinAnd`

B.0.42 `decodePoint` method (Code 238)

Description	Convert `Coll[Byte]` to `GroupElement` using `GroupElementSerializer`
Parameters	`input : Coll[Byte]` // serialized bytes of some `GroupElement` value
Result	`GroupElement`
Serialized as	`DecodePoint`

B.0.43 `unary_!` method (Code 239)

Description	Logical NOT operation. Returns `true` if input is `false` and `false` if input is `true`.
Parameters	`input : Boolean` // input `Boolean` value
Result	`Boolean`
Serialized as	`LogicalNot`

B.0.44 `unary_-` method (Code 240)

Description	Negates numeric value `x` by returning `-x`.
Parameters	`input : T` // value of numeric type
Result	`T`
Serialized as	`Negation`

B.0.45 `unary_~` method (Code 241)

Description	Invert every bit of the numeric value.
Parameters	`input : T` // value of numeric type
Result	`T`
Serialized as	`BitInversion`

B.0.46 `bit_|` method (Code 242)

Description	Bitwise OR of two numeric operands.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`BitOr`

B.0.47 `bit_&` method (Code 243)

Description	Bitwise AND of two numeric operands.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`BitAnd`

B.0.48 `^` method (Code 244)

Description	Logical XOR of two operands
Parameters	`left : Boolean` // left operand `right : Boolean` // right operand
Result	`Boolean`
Serialized as	`BinXor`

B.0.49 `bit_^` method (Code 245)

Description	Bitwise XOR of two numeric operands.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`BitXor`

B.0.50 `bit_>>` method (Code 246)

Description	Right shift of bits.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`BitShiftRight`

B.0.51 `bit_<<` method (Code 247)

Description	Left shift of bits.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`BitShiftLeft`

B.0.52 `bit_>>>` method (Code 248)

Description	Right shift of bits.
Parameters	`left : T` // left operand `right : T` // right operand
Result	`T`
Serialized as	`BitShiftRightZeroed`

B.0.53 `xorOf` method (Code 255)

Description	Similar to `allOf`, but performing logical XOR operation between all conditions instead of `&&`
Parameters	`conditions : Coll[Boolean]` // a collection of conditions
Result	`Boolean`
Serialized as	`XorOf`

C Serialization format of ErgoTree nodes

Morphic : These subsections are autogenerated from instrumented ValueSerializers

C.0.1 `ConcreteCollection` operation (OpCode 131)

Slot	Format	#bytes	Description
numItems	`VLQ(UShort)`	[1, *]	number of item in a collection of expressions
elementType	`Type`	[1, *]	type of each expression in the collection
`for` $i = 1$ `to` *numItems*			
$item_i$	`Expr`	[1, *]	expression in i-th position
`end for`			

C.0.2 `ConcreteCollectionBooleanConstant` operation (OpCode 133)

Slot	Format	#bytes	Description
numBits	`VLQ(UShort)`	[1, *]	number of items in a collection of Boolean values
bits	`Bits`	[1, 1024]	Boolean values encoded as as bits (right most byte is zero-padded on the right)

C.0.3 `Tuple` operation (OpCode 134)

Slot	Format	#bytes	Description
numItems	`UByte`	1	number of items in the tuple
`for` $i = 1$ `to` *numItems*			
$item_i$	`Expr`	[1, *]	tuple's item in i-th position
`end for`			

C.0.4 `SelectField` operation (OpCode 140)

Select tuple field by its 1-based index. E.g. `input._1` is transformed to `SelectField(input, 1)` See `selectField`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	tuple of items
fieldIndex	`Byte`	1	index of an item to select

C.0.5 `LT` operation (OpCode 143)

Returns `true` is the left operand is less then the right operand, `false` otherwise. See `<`

Slot	Format	#bytes	Description
`match` $(left, right)$			
`otherwise`			
left	`Expr`	[1, *]	left operand
right	`Expr`	[1, *]	right operand
`end match`			

C.0.6 LE operation (OpCode 144)

Returns `true` is the left operand is less then or equal to the right operand, `false` otherwise. See `<=`

Slot	Format	#bytes	Description
`match` $(left, right)$			
`otherwise`			
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand
`end match`			

C.0.7 GT operation (OpCode 145)

Returns `true` is the left operand is greater then the right operand, `false` otherwise. See `>`

Slot	Format	#bytes	Description
`match` $(left, right)$			
`otherwise`			
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand
`end match`			

C.0.8 GE operation (OpCode 146)

Returns `true` is the left operand is greater then or equal to the right operand, `false` otherwise. See `>=`

Slot	Format	#bytes	Description
`match` $(left, right)$			
`otherwise`			
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand
`end match`			

C.0.9 EQ operation (OpCode 147)

Compare equality of `left` and `right` arguments See `==`

Slot	Format	#bytes	Description
`match` $(left, right)$			
`otherwise`			
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand
`end match`			

C.0.10 NEQ operation (OpCode 148)

Compare inequality of `left` and `right` arguments See `!=`

Slot	Format	#bytes	Description
`match` $(left, right)$			
`otherwise`			
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand
`end match`			

C.0.11 `If` operation (OpCode 149)

Compute condition, if true then compute trueBranch else compute falseBranch See `if`

Slot	Format	#bytes	Description
condition	`Expr`	[1, *]	condition expression
trueBranch	`Expr`	[1, *]	expression to execute when `condition == true`
$falseBranch$	`Expr`	[1, *]	expression to execute when `condition == false`

C.0.12 `AND` operation (OpCode 150)

Returns true if *all* the elements in collection are `true`. See `allOf`

Slot	Format	#bytes	Description
conditions	`Expr`	[1, *]	a collection of conditions

C.0.13 `OR` operation (OpCode 151)

Returns true if *any* the elements in collection are `true`. See `anyOf`

Slot	Format	#bytes	Description
conditions	`Expr`	[1, *]	a collection of conditions

C.0.14 `AtLeast` operation (OpCode 152)

Logical threshold. AtLeast has two inputs: integer `bound` and `children` same as in AND/OR. The result is true if at least `bound` children are proven. See `atLeast`

Slot	Format	#bytes	Description
bound	`Expr`	[1, *]	required minimum of proven children
children	`Expr`	[1, *]	proposition to be proven/validated

C.0.15 `Minus` operation (OpCode 153)

Returns a result of subtracting second numeric operand from the first. See `-`

Slot	Format	#bytes	Description
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand

C.0.16 `Plus` operation (OpCode 154)

Returns a sum of two numeric operands See `+`

Slot	Format	#bytes	Description
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand

C.0.17 `Xor` operation (OpCode 155)

Byte-wise XOR of two collections of bytes See `binary_|`

Slot	Format	#bytes	Description
left	`Expr`	[1, *]	left operand
right	`Expr`	[1, *]	right operand

C.0.18 `Multiply` operation (OpCode 156)

Returns a multiplication of two numeric operands See `*`

Slot	Format	#bytes	Description
left	`Expr`	[1, *]	left operand
right	`Expr`	[1, *]	right operand

C.0.19 `Division` operation (OpCode 157)

Integer division of the first operand by the second operand. See `/`

Slot	Format	#bytes	Description
left	`Expr`	[1, *]	left operand
right	`Expr`	[1, *]	right operand

C.0.20 `Modulo` operation (OpCode 158)

Reminder from division of the first operand by the second operand. See `%`

Slot	Format	#bytes	Description
left	`Expr`	[1, *]	left operand
right	`Expr`	[1, *]	right operand

C.0.21 `Exponentiate` operation (OpCode 159)

Exponentiate this `GroupElement` to the given number. Returns this to the power of k See `GroupElement.exp`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
k	`Expr`	[1, *]	The power

C.0.22 `MultiplyGroup` operation (OpCode 160)

Group operation. See `GroupElement.multiply`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
other	`Expr`	[1, *]	other element of the group

C.0.23 `Min` operation (OpCode 161)

Minimum value of two operands. See `min`

Slot	Format	#bytes	Description
left	`Expr`	[1, *]	left operand
right	`Expr`	[1, *]	right operand

C.0.24 `Max` operation (OpCode 162)

Maximum value of two operands. See `max`

Slot	Format	#bytes	Description
left	`Expr`	[1, *]	left operand
right	`Expr`	[1, *]	right operand

C.0.25 `MapCollection` operation (OpCode 173)

Builds a new collection by applying a function to all elements of this collection. Returns a new collection of type `Coll[B]` resulting from applying the given function `f` to each element of this collection and collecting the results. See `SCollection.map`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
f	`Expr`	[1, *]	the function to apply to each element

C.0.26 `Exists` operation (OpCode 174)

Tests whether a predicate holds for at least one element of this collection. Returns `true` if the given predicate `p` is satisfied by at least one element of this collection, otherwise `false` See `SCollection.exists`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
p	`Expr`	[1, *]	the predicate used to test elements

C.0.27 `ForAll` operation (OpCode 175)

Tests whether a predicate holds for all elements of this collection. Returns `true` if this collection is empty or the given predicate `p` holds for all elements of this collection, otherwise `false`. See `SCollection.forall`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
p	`Expr`	[1, *]	the predicate used to test elements

C.0.28 `Fold` operation (OpCode 176)

Applies a binary operator to a start value and all elements of this collection, going left to right. See `SCollection.fold`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
zero	`Expr`	[1, *]	a starting value
op	`Expr`	[1, *]	the binary operator

C.0.29 `SizeOf` operation (OpCode 177)

The size of the collection in elements. See `SCollection.size`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance

C.0.30 `ByIndex` operation (OpCode 178)

Return the element of collection if `index` is in range `0 .. size-1` See `SCollection.getOrElse`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
index	`Expr`	[1, *]	index of the element of this collection
`optional` *default*			
tag	`Byte`	1	0 - no value; 1 - has value
`when` *tag* == 1			
default	`Expr`	[1, *]	value to return when `index` is out of range
`end optional`			

C.0.31 `Append` operation (OpCode 179)

Puts the elements of other collection after the elements of this collection (concatenation of 2 collections) See `SCollection.append`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
other	`Expr`	[1, *]	the collection to append at the end of this

C.0.32 `Slice` operation (OpCode 180)

Selects an interval of elements. The returned collection is made up of all elements `x` which satisfy the invariant: `from <= indexOf(x) < until` See `SCollection.slice`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
from	`Expr`	[1, *]	the lowest index to include from this collection
until	`Expr`	[1, *]	the lowest index to EXCLUDE from this collection

C.0.33 `ExtractAmount` operation (OpCode 193)

Mandatory: Monetary value, in Ergo tokens (NanoErg unit of measure) See `Box.value`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance

C.0.34 `ExtractScriptBytes` operation (OpCode 194)

Serialized bytes of guarding script, which should be evaluated to true in order to open this box. (aka spend it in a transaction) See `Box.propositionBytes`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance

C.0.35 `ExtractBytes` operation (OpCode 195)

Serialized bytes of this box's content, including proposition bytes. See `Box.bytes`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance

C.0.36 `ExtractBytesWithNoRef` operation (OpCode 196)

Serialized bytes of this box's content, excluding transactionId and index of output. See `Box.bytesWithoutRef`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance

C.0.37 `ExtractId` operation (OpCode 197)

Blake2b256 hash of this box's content, basically equals to `blake2b256(bytes)` See `Box.id`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance

C.0.38 `ExtractRegisterAs` operation (OpCode 198)

Extracts register by id and type. Type param `T` expected type of the register. Returns `Some(value)` if the register is defined and has given type and `None` otherwise See `Box.getReg`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance
regId	`Byte`	1	zero-based identifier of the register.
type	`Type`	[1, *]	expected type of the value in register

C.0.39 `ExtractCreationInfo` operation (OpCode 199)

If `tx` is a transaction which generated this box, then `creationInfo._1` is a height of the tx's block. The `creationInfo._2` is a serialized transaction identifier followed by box index in the transaction outputs. See `Box.creationInfo`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance

C.0.40 `CalcBlake2b256` operation (OpCode 203)

Calculate Blake2b hash from `input` bytes. See `blake2b256`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	collection of bytes

C.0.41 `CalcSha256` operation (OpCode 204)

Calculate Sha256 hash from `input` bytes. See `sha256`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	collection of bytes

C.0.42 `CreateProveDlog` operation (OpCode 205)

ErgoTree operation to create a new `SigmaProp` value representing public key of discrete logarithm signature protocol. See `proveDlog`

Slot	Format	#bytes	Description
value	`Expr`	[1, *]	element of elliptic curve group

C.0.43 `CreateProveDHTuple` operation (OpCode 206)

ErgoTree operation to create a new SigmaProp value representing public key of Diffie Hellman signature protocol. Common input: (g,h,u,v) See `proveDHTuple`

Slot	Format	#bytes	Description
g	`Expr`	[1, *]	
h	`Expr`	[1, *]	
u	`Expr`	[1, *]	
v	`Expr`	[1, *]	

C.0.44 `SigmaPropBytes` operation (OpCode 208)

Serialized bytes of this sigma proposition taken as ErgoTree. See `SigmaProp.propBytes`

Slot	Format	#bytes	Description
this	`Expr`	[1, *]	this instance

C.0.45 `BoolToSigmaProp` operation (OpCode 209)

Embedding of `Boolean` values to `SigmaProp` values. As an example, this operation allows boolean experessions to be used as arguments of `atLeast(..., sigmaProp(boolExpr), ...)` operation. During execution results to either `TrueProp` or `FalseProp` values of `SigmaProp` type. See `sigmaProp`

Slot	Format	#bytes	Description
condition	`Expr`	[1, *]	boolean value to embed in SigmaProp value

C.0.46 `DeserializeContext` operation (OpCode 212)

Extracts context variable as `Coll[Byte]`, deserializes it to script and then executes this script in the current context. The original `Coll[Byte]` of the script is available as `getVar[Coll[Byte]](id)`. Type parameter `V` result type of the deserialized script. Throws an exception if the actual script type doesn't conform to T. Returns a result of the script execution in the current context See `executeFromVar`

Slot	Format	#bytes	Description
type	`Type`	[1, *]	expected type of the deserialized script
id	`Byte`	1	identifier of the context variable

C.0.47 `DeserializeRegister` operation (OpCode 213)

Extracts SELF register as `Coll[Byte]`, deserializes it to script and then executes this script in the current context. The original `Coll[Byte]` of the script is available as `SELF.getReg[Coll[Byte]](id)`. Type parameter `T` result type of the deserialized script. Throws an exception if the actual script type doesn't conform to `T`. Returns a result of the script execution in the current context See `executeFromSelfReg`

Slot	Format	#bytes	Description
id	`Byte`	1	identifier of the register
$type$	`Type`	[1, *]	expected type of the deserialized script
`optional` $default$			
tag	`Byte`	1	0 - no value; 1 - has value
`when` $tag == 1$			
$default$	`Expr`	[1, *]	optional default value, if register is not available
`end optional`			

C.0.48 `ValDef` operation (OpCode 214)

Slot	Format	#bytes	Description

C.0.49 `FunDef` operation (OpCode 215)

Slot	Format	#bytes	Description

C.0.50 `BlockValue` operation (OpCode 216)

Slot	Format	#bytes	Description
$numItems$	`VLQ(UInt)`	[1, *]	number of block items
`for` $i = 1$ `to` $numItems$			
$item_i$	`Expr`	[1, *]	block's item in i-th position
`end for`			
$result$	`Expr`	[1, *]	result expression of the block

C.0.51 `FuncValue` operation (OpCode 217)

Slot	Format	#bytes	Description
$numArgs$	`VLQ(UInt)`	[1, *]	number of function arguments
`for` $i = 1$ `to` $numArgs$			
id_i	`VLQ(UInt)`	[1, *]	identifier of the i-th argument
$type_i$	`Type`	[1, *]	type of the i-th argument
`end for`			
$body$	`Expr`	[1, *]	function body, which is parameterized by arguments

C.0.52 `Apply` operation (OpCode 218)

Apply the function to the arguments. See `apply`

Slot	Format	#bytes	Description
$func$	`Expr`	[1, *]	function which is applied
$\#items$	`VLQ(UInt)`	[1, *]	number of items in the collection
`for` $i = 1$ `to` $\#items$			
$args_i$	`Expr`	[1, *]	i-th item in the list of arguments
`end for`			

C.0.53 `PropertyCall` operation (OpCode 219)

Slot	Format	#bytes	Description
$typeCode$	`Byte`	1	type of the method (see Table 3)
$methodCode$	`Byte`	1	a code of the property
obj	`Expr`	[1, *]	receiver object of this property call

C.0.54 `MethodCall` operation (OpCode 220)

Slot	Format	#bytes	Description
$typeCode$	`Byte`	1	type of the method (see Table 3)
$methodCode$	`Byte`	1	a code of the method
obj	`Expr`	[1, *]	receiver object of this method call
$\#items$	`VLQ(UInt)`	[1, *]	number of items in the collection
`for` $i = 1$ `to` $\#items$			
$args_i$	`Expr`	[1, *]	i-th item in the arguments of the method call
`end for`			

C.0.55 `GetVar` operation (OpCode 227)

Get context variable with given `varId` and type. See `getVar`

Slot	Format	#bytes	Description
$varId$	`Byte`	1	`Byte` identifier of context variable
$type$	`Type`	[1, *]	expected type of context variable

C.0.56 `OptionGet` operation (OpCode 228)

Returns the option's value. The option must be nonempty. Throws exception if the option is empty. See `SOption.get`

Slot	Format	#bytes	Description
$this$	`Expr`	[1, *]	this instance

C.0.57 `OptionGetOrElse` operation (OpCode 229)

Returns the option's value if the option is nonempty, otherwise return the result of evaluating `default`. See `SOption.getOrElse`

Slot	Format	#bytes	Description
$this$	`Expr`	[1, *]	this instance
$default$	`Expr`	[1, *]	the default value

C.0.58 `OptionIsDefined` operation (OpCode 230)

Returns `true` if the option is an instance of `Some`, `false` otherwise. See `SOption.isDefined`

Slot	Format	#bytes	Description
$this$	`Expr`	[1, *]	this instance

C.0.59 `SigmaAnd` operation (OpCode 234)

Returns sigma proposition which is proven when *all* the elements in collection are proven. See `allZK`

Slot	Format	#bytes	Description
#items	`VLQ(UInt)`	[1, *]	number of items in the collection
`for` $i = 1$ `to` *#items*			
$propositions_i$	`Expr`	[1, *]	i-th item in the a collection of propositions
`end for`			

C.0.60 `SigmaOr` operation (OpCode 235)

Returns sigma proposition which is proven when *any* of the elements in collection is proven. See `anyZK`

Slot	Format	#bytes	Description
#items	`VLQ(UInt)`	[1, *]	number of items in the collection
`for` $i = 1$ `to` *#items*			
$propositions_i$	`Expr`	[1, *]	i-th item in the a collection of propositions
`end for`			

C.0.61 `BinOr` operation (OpCode 236)

Logical OR of two operands See `||`

Slot	Format	#bytes	Description
`match` $(left, right)$			
`otherwise`			
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand
`end match`			

C.0.62 `BinAnd` operation (OpCode 237)

Logical AND of two operands See `&&`

Slot	Format	#bytes	Description
`match` $(left, right)$			
`otherwise`			
$left$	`Expr`	[1, *]	left operand
$right$	`Expr`	[1, *]	right operand
`end match`			

C.0.63 `DecodePoint` operation (OpCode 238)

Convert `Coll[Byte]` to `GroupElement` using `GroupElementSerializer` See `decodePoint`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	serialized bytes of some `GroupElement` value

C.0.64 `LogicalNot` operation (OpCode 239)

Logical NOT operation. Returns `true` if input is `false` and `false` if input is `true`. See `unary_!`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	input `Boolean` value

C.0.65 `Negation` operation (OpCode 240)

Negates numeric value `x` by returning `-x`. See `unary_-`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	value of numeric type

C.0.66 `BinXor` operation (OpCode 244)

Logical XOR of two operands See `^`

Slot	Format	#bytes	Description
`match` $(left, right)`			
`otherwise`			
left	`Expr`	[1, *]	left operand
right	`Expr`	[1, *]	right operand
`end match`			

C.0.67 `XorOf` operation (OpCode 255)

Similar to `allOf`, but performing logical XOR operation between all conditions instead of `&&` See `xorOf`

Slot	Format	#bytes	Description
conditions	`Expr`	[1, *]	a collection of conditions

C.0.68 `SubstConstants` operation (OpCode 116)

Transforms serialized bytes of ErgoTree with segregated constants by replacing constants at given positions with new values. This operation allow to use serialized scripts as pre-defined templates. The typical usage is "check that output box have proposition equal to given script bytes, where minerPk (constants(0)) is replaced with currentMinerPk". Each constant in original scriptBytes have SType serialized before actual data (see ConstantSerializer). During substitution each value from newValues is checked to be an instance of the corresponding type. This means, the constants during substitution cannot change their types.

Returns original scriptBytes array where only specified constants are replaced and all other bytes remain exactly the same. See `substConstants`

Slot	Format	#bytes	Description
scriptBytes	`Expr`	[1, *]	serialized ErgoTree with ConstantSegregationFlag set to 1.
positions	`Expr`	[1, *]	zero based indexes in ErgoTree.constants array which should be replaced with new values
newValues	`Expr`	[1, *]	new values to be injected into the corresponding positions in ErgoTree.constants array

C.0.69 `LongToByteArray` operation (OpCode 122)

Converts `Long` value to big-endian bytes representation. See `longToByteArray`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	value to convert

C.0.70 `ByteArrayToBigInt` operation (OpCode 123)

Convert big-endian bytes representation (Coll[Byte]) to BigInt value. See `byteArrayToBigInt`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	collection of bytes in big-endian format

C.0.71 `ByteArrayToLong` operation (OpCode 124)

Convert big-endian bytes representation (Coll[Byte]) to Long value. See `byteArrayToLong`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	collection of bytes in big-endian format

C.0.72 `Downcast` operation (OpCode 125)

Cast this numeric value to a smaller type (e.g. Long to Int). Throws exception if overflow. See `downcast`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	value to cast
type	`Type`	[1, *]	resulting type of the cast operation

C.0.73 `Upcast` operation (OpCode 126)

Cast this numeric value to a bigger type (e.g. Int to Long) See `upcast`

Slot	Format	#bytes	Description
input	`Expr`	[1, *]	value to cast
type	`Type`	[1, *]	resulting type of the cast operation

D Motivations

D.1 Type Serialization format rationale

Some operations of ErgoTree have type parameters, for which concrete types should be specified (since ErgoTree is monomorphic IR). When the operation (such as `ExtractRegisterAs`) is serialized those types should also be serialized as part of operation. The following encoding is designed to minimize a number of bytes required to represent type in the serialization format of ErgoTree.

In most cases type term serialises into a single byte. In the intermediate representation of ErgoTree each type is represented by a tree of nodes where leaves are primitive types and other nodes are type constructors. Simple (but sub-optimal) way to serialize a type would be to give each primitive type and each type constructor a unique type code. Then, to serialize a node, we need to emit its code and then perform recursive descent to serialize all children. However, to save storage space, we use special encoding schema to save bytes for the types that are used more often.

We assume the most frequently used types are:

- primitive types (`Int, Byte, Boolean, BigInt, GroupElement, Box, AvlTree`)
- Collections of primitive types (`Coll[Byte]` etc)
- Options of primitive types (`Option[Int]` etc.)
- Nested arrays of primitive types (`Coll[Coll[Int]]` etc.)
- Functions of primitive types (`Box => Boolean` etc.)
- First biased pair of types (`(_, Int)` when we know the first component is a primitive type).
- Second biased pair of types (`(Int, _)` when we know the second component is a primitive type)
- Symmetric pair of types (`(Int, Int)` when we know both types are the same)

All the types above should be represented in an optimized way (preferable by a single byte). For other types, we do recursive descent down the type tree as it is defined in section 5.1

D.2 Constant Segregation rationale

D.2.1 Massive script validation

Consider a transaction `tx` which have `INPUTS` collection of boxes to spend. Every input box can have a script protecting it (`propostionBytes` property). This script should be executed in a context of the current transaction. The simplest transaction have 1 input box. Thus if we want to have a sustained block validation of 1000 transactions per second we need to be able to validate 1000 scripts per second.

For every script (of input `box`) the following is done in order to validate it:

1. Context is created with `SELF` = box
2. The script is deserialized into ErgoTree

3. ErgoTree is traversed to build costGraph and calcGraph, two graphs for cost estimation function and script calculation function.
4. Cost estimation is computed by evaluating costGraph with current context data
5. If cost and data size limits are not exceeded, calcGraph is evaluated using context data to obtain sigma proposition (see `SigmaProp`)
6. Verification procedure is executed

D.2.2 Potential for Script processing optimization

Before an ErgoTree contract can be stored in a blockchain it should be first compiled from its source text into ErgoTree and then serialized into byte array.

Because the language is purely functional and IR is graph-based, the compilation process has an effect of normalization/unification. This means that different original scripts may have identical ErgoTrees and as the result identical serialized bytes.

Because of normalization, and also because of script reusability, the number of conceptually (or logically) different scripts is much less than the number of individual scripts in a blockchain. For example we may have 1000s of different scripts in a blockchain with millions of boxes.

The average reusability ratio is 1000 in this case. And even those different scripts may have different usage frequency. Having big reusability ratio we can optimize script evaluation by performing steps 1 - 4 only once per unique script.

The compiled calcGraph can be cached in `Map[Array[Byte], Context => SigmaBoolean]`. Every script extracted from an input box can be used as a key in this map to obtain ready to execute graph.

However, we have a problem with constants embedded in contracts. There is one obstacle to the optimization by caching. In many cases it is very natural to embed constants in the script body, most notable scenario is when public keys are embedded. As result two functionally identical scripts may serialize to different byte arrays because they have different embedded constants.

D.2.3 Constant-less ErgoTree

The solution to the problem with embedded constants is simple, we don't need to embed constants. Each constant in the body of ErgoTree can be replaced with indexed placeholder (see `ConstantPlaceholder`). Each placeholder have an index field. The index of the placeholder is assigned by breadth-first topological order of the graph traversal.

The transformation is part of compilation and is performed ahead of time. Each ErgoTree have an array of all the constants extracted from its body. Each placeholder refers to the constant by the constant's index in the array.

Thus the format of serialized script is shown in Figure 13 which contains:

1. number of constants
2. constants collection
3. script expression with placeholders

The constants collection contains serialized constant data (using ConstantSerializer) one after another. The script expression is a serialized ErgoTree with placeholders.

Using this new script format we can use script expression part as a key in the cache. An observation is that after the constants are extracted, what remains is a template. Thus instead of applying steps 1-4 to *constant-full* scripts we can apply them to *constant-less* templates. Before applying steps 4 and 5 we need to bind placeholders with actual values taken from the cconstants collection.

E Compressed encoding of integer values

E.1 VLQ encoding

```
public final void putULong(long value) {
    while (true) {
        if ((value & ~0x7FL) == 0) {
            buffer[position++] = (byte) value;
            return;
        } else {
            buffer[position++] = (byte) (((int) value & 0x7F) | 0x80);
            value >>>= 7;
        }
    }
}
```

E.2 ZigZag encoding

Encode a ZigZag-encoded 64-bit value. ZigZag encodes signed integers into values that can be efficiently encoded with varint. (Otherwise, negative values must be sign-extended to 64 bits to be varint encoded, thus always taking 10 bytes in the buffer.

Parameter `n` is a signed 64-bit integer. This Java method returns an unsigned 64-bit integer, stored in a signed int because Java has no explicit unsigned support.

```
public static long encodeZigZag64(final long n) {
  // Note:  the right-shift must be arithmetic
  return (n << 1) ^ (n >> 63);
}
```