ErgoTree Specification for Ergo Protocol 1.0

Alexander Slesarenko

March 1, 2020

Abstract

In this document we describe a typed abstract syntax of the language called ErgoTree which is used to define logical propositions protecting boxes (generalization of coins) in the Ergo blockchain. Serialized ErgoTree expressions are written into UTXO boxes and then evaluated by the transaction verifier. Most of Ergo users don't use ErgoTree directly since they are developing contracts in higher-level language, such as ErgoScript, which is then compiled to ErgoTree. The reference implementation of ErgoTree uses Scala, however alternative implementations can use other languages. This document provides a language neutral specification of ErgoTree for developers of alternative ErgoTree implementations.

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. To serve as a platform for contractual money, the language for writing contracts on blockchain must have a number of properties.

First, the language and the contract execution environment should be *deterministic*. Once created and stored in the blockchain, a 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 using any *compliant* language implementation. No general purpose programming language is deterministic at least because all of them provide non-deterministic operations. ErgoTree doesn't have non-deterministic operations.

Second, the language should facilitate *spam-resistantance*, i.e. defending against attacks when malicious contracts can overload the network nodes and bring the blockchain down [Ler17]. To fullfill this goal transaction model of ErgoTree support *predictable cost estimation*, the fast calculation of contract execution costs to ensure the evaluation cost of the given transaction is always within acceptable bounds. In a general (turing-complete) case, such cost prediction is not possible and require special mechanisms such as Gas [Woo14]. Gas limits on transactions indeed protect the network from spam attacks, but at the expence of the users who need to be careful to specifiy the gas limit large enough for the transaction to complete, otherwise the gas used for the failed transaction will be kept by the miners for their work and the user *will not* get it back.

Third, the contracts language should be simple yet *expressive enough*. It should be possible to efficiently implement wide range of practical applications. For example ErgoTree is *not* turing-complete but it is co-designed with the capabilities of the Ergo blockchain platform itself, making the whole system *turing-complete* as demonstrated in [CKM18]. Simplicity of the language enables spam-resistantance.

Forth, simplicity and expressivity are often the characteristics of domain-specific languages [Fow10, Hud96]. From this perspective ErgoTree is an intermediate representation of a DSL for writing smart contracts. The language directly captures the *ubiquitous language* [Ubi] of the Ergo blockchain directly manipulating with Boxes, Tokens, Zero-Knowledge Sigma-Propostions etc. These are the novel first-class features of Ergo platform which it provides for user applicatons. It has complementary frontend language called ErgoScript with syntax of Scala/Kotlin. ErgoScript aims to address the large audience of programmers with minimum surprise and WTF ratio [WTF]. The syntax of ErgoScript is inspired by Scala/Kotlin and also shares a common subset with Java and C#, thus if you are proficient in any of these languages you will be right at home using ErgoScript as well.

And last, but not the least, ErgoTree as a core language of Ergo platform, should be optimized for compact storage and fast execution.

We implemented a reference implementation of ErgoTree according to the specification described in this document and provide a guidance in section 6 for development of an alternative and compliant protocol implementation. We don't describe ErgoScript is this document and focus exclusively on ErgoTree.

2 ErgoTree As A Language

In this section we define an abstract syntax for the ErgoTree language. It is a typed call-by-value, higher-order functional language without recursion. It supports single-assignment blocks, tuples, optional values, indexed collections with higher-order operations, short-cutting logicals, ternary 'if' with lazy branches. All operations are deterministic, without side effects and all values are immutable.

The semantics of ErgoTree is specified by first translating it to a core calculus (`Core-`λ) and then by giving its denotational evaluation semantics. Typing rules are given in section 3 and the evaluation semantics is given in section 4. In section 5 we describe serialization format of ErgoTree. Guidance on compliant interpreter implementation is provided in section 6.

ErgoTree is defined below using abstract syntax notation as shown in Figure 1. This corresponds to `Value` class of the reference implementation, which can be serialized to an array of bytes using `ValueSerializer`. The mnemonic names shown in the figure correspond to classes of the reference implementation.

Set Name		Syntax	Mnemonic	Description
$\mathcal{T} \ni T$	::=	`P`	`SPredefType`	predefined types (see Appendix A)
	|	$(T_1, \ldots, T_n)$	`STuple`	tuple of n elements (see `Tuple` type)
	|	$(T_1, \ldots, T_n) \to T$	`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
	|	$\lambda(\overline{x_i : T_i}).e$	`FuncValue`	lambda expression
	|	$e_f\langle\overline{e_i}\rangle$	`Apply`	application of functional expression
	|	$e.m\langle\overline{e_i}\rangle$	`MethodCall`	method invocation
	|	$(e_1, \ldots, e_n)$	`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
	|	$\{\overline{\texttt{val } x_i = e_i;}\ e\}$	`BlockValue`	block expression
cd	::=	`trait` T $\{\overline{ms_i}\}$	`STypeCompanion`	type 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 or a `val` bound variable.

Lambda expressions can take a list of lambda-bound variables which can be used in the body expression, which can be a *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 types*. If expression e has type T and and method m is declared in the type T then method invocation $e.m(args)$ is defined for the appropriate $args$. See section A for the specification of types and their methods.

Conditional expressions of ErgoTree are strict in the 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 refer to 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 and which doesn't have lazy operations. This *lowering* translation is shown in Figure 2.

$Term_{ErgoTree}$		$Term_{Core}$
$\mathcal{L}[\![\lambda(\overline{x_i : T_i}).e]\!]$	$\mapsto$	$\lambda(\overline{x_i : T_i}).\mathcal{L}[\![e]\!]$
$\mathcal{L}[\![e_f\langle\overline{e_i}\rangle]\!]$	$\mapsto$	$\mathcal{L}[\![e_f]\!]\langle\mathcal{L}[\![(\overline{e_i})]\!]\rangle$
$\mathcal{L}[\![e.m\langle\overline{e_i}\rangle]\!]$	$\mapsto$	$\mathcal{L}[\![e]\!].m\langle\overline{\mathcal{L}[\![e_i]\!]}\rangle$
$\mathcal{L}[\![(e_1, \ldots, e_n)]\!]$	$\mapsto$	$(\mathcal{L}[\![e_1]\!], \ldots, \mathcal{L}[\![e_n]\!])$
$\mathcal{L}[\![e_1$ `||` $e_2]\!]$	$\mapsto$	$\mathcal{L}[\![$`if` (e_1) `true else` $e_2]\!]$
$\mathcal{L}[\![e_1$ `&&` $e_2]\!]$	$\mapsto$	$\mathcal{L}[\![$`if` (e_1) e_2 `else false`$]\!]$
$\mathcal{L}[\![$`if` (e_{cond}) e_1 `else` $e_2]\!]$	$\mapsto$	$(if(\mathcal{L}[\![e_{cond}]\!],\ \lambda(\_ : Unit).\mathcal{L}[\![e_1]\!],\ \lambda(\_ : Unit).\mathcal{L}[\![e_2]\!]))\langle\rangle$
$\mathcal{L}[\![\{\overline{$`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

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` $\Rightarrow$ `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 semantics of `Core-`λ is specified in Section 4.

Note, that we use lowering transformation only to specify semantics. Implementations can optimize by evaluating ErgoTree directly as long as the semantics is preserved.

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.

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

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

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

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

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

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

Figure 3: Typing rules of ErgoTree

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

Primitive operations can be parameterized with type variables, for example addition (B.0.19) has the signature `def +[T](left: T, right: T): T` where `T` is one of the numeric types (Table 8). Function *ptype* returns the type of a primitive operation specialized for the concrete types of its arguments, for example $\mathit{ptype}(+, \texttt{Int}, \texttt{Int}) = (\texttt{Int}, \texttt{Int}) \to \texttt{Int}$.

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

`BlockValue` 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 it cannot be typed and evaluated.

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

4 Evaluation

In this section we describe evaluation semantics of the ErgoTree language and the corresponding reference implementation of the interpreter.

4.1 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 `Core-`λ terms are called values:*

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

- *All `Core-`λ 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 $\{\texttt{true}, \texttt{false}\}$, 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 (hence the semantics is call-by-value).

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 the 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.

Figure 4: Denotational semantics of `Core-`λ

The denotations of `Core-`λ types

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

The denotations of `Core-`λ terms which together specify the function $[\![\_]\!] : [\![\Gamma]\!] \to [\![T]\!]$

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

5 Serialization

This section defines a binary format, which is used to store ErgoTree contracts in persistent stores, to transfer them over the 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 (represented by the `ErgoTree` class), which can be evaluated as it is specified in Section 4.

Here we specify the serialization procedure in general. The serialization format of ErgoTree types (`SType` class) and nodes (`Value` class) is specified in section 5.1 and Appendix C correspondingly.

Table 1 shows size limits which are checked during contract deserialization, which is important to resist malicious script attacks.

Table 1: Serialization limits

Constant	Value	Description
VLQ_{max}	10	Maximum size of VLQ encoded byte sequence (See VLQ formats E.1)
T_{max}	100	Maximum size of serialized type term (see Type format 5.1)
D_{max}	$4Kb$	Maximum size of serialized data instance (see Data format 5.2)
C_{max}	$= T_{max} + D_{max}$	Maximum size of serialized data instance (see Const format 5.3)
$Expr_{max}$	$4Kb$	Maximum size of serialized ErgoTree term (see Expr format 5.4)
$ErgoTree_{max}$	$4Kb$	Maximum size of serialized ErgoTree contract (see ErgoTree format 5.5)

All the serialization formats which are uses and defined thoughout this section are listed in Table 2 which introduces a name for each format and also shows the number of bytes each format may occupy in the byte stream.

Table 2: Serialization formats

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 of bytes, which size is stored elsewhere or wellknown.
`Type`	$[1..T_{max}]$	Serialized type terms of ErgoTree. See 5.1
`Data`	$[1..D_{max}]$	Serialized data values of ErgoTree. 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
`AvlTree`	44	Serialized dynamic dictionary digest. See 5.2.3
`Constant`	$[1..C_{max}]$	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

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 and very fast deserialization. In many cases serialization procedure is data dependent and thus have branching logic. To express this complex serialization logic in the specification we use a *pseudo-language* with operators like `for, match, if, optional`. The language allows to specify a *structure* out of *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 resulting stream of bytes. The notation is summarized in Table 3.

Table 3: Serialization Notation

Notation	Description
$[\![T]\!]$ where T - type	Denotes a set of values of type T
$v \in [\![T]\!]$	The value v belongs to the set $[\![T]\!]$
$v : T$	Same as $v \in [\![T]\!]$
`match` (t, v)	Pattern match on pair (t, v) where t, v - values
`with` $(Unit, v \in [\![Unit]\!])$	Pattern case

`for` $i = 1$ `to` len
`serialize(`v_i`)`
`end for`
	Call the given `serialize` function repeatedly. The outputs bytes of all invocations are concatenated and become the output of the `for` statement.

`if` *condition* `then`
`serialize1(`v_1`)`
`else`
`serialize2(`v_2`)`
`end if`
	Serialize one of the branches depending of the *condition*. The output bytes of the executed branch becomes the output of the `if` statement.

In the next section we describe how 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 will proceed to serialization of arbitrary ErgoTree trees.

5.1 Type Serialization

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

5.1.1 Distribution of type codes

The whole space of 256 one byte codes is divided as shown in Figure 4.

Table 4: Distribution of type codes between Data and Function types

Value/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 [1]

5.1.2 Encoding of Data Types

There are eight different values for *embeddable* types and 3 more are reserved for the future extensions. Each embeddable type has a type code in the range 1,...,11 as shown in Figure 5.

Table 5: Embeddable Types

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

Table 6: Code Ranges of Data Types

Interval	Constructor	Description
0x01 - 0x0B(11)		embeddable types (including 3 reserved)
0x0C(12)	`Coll[_]`	Collection of non-embeddable types (`Coll[(Int,Boolean)]`)
0x0D(13) - 0x17(23)	`Coll[_]`	Collection of embeddable types (`Coll[Byte]`, `Coll[Int]`, etc.)
0x18(24)	`Coll[Coll[_]]`	Nested collection of non-embeddable types (`Coll[Coll[(Int,Boolean)]]`)
0x19(25) - 0x23(35)	`Coll[Coll[_]]`	Nested collection of embeddable types (`Coll[Coll[Byte]]`, `Coll[Coll[Int]]`)
0x24(36)	`Option[_]`	Option of non-embeddable type (`Option[(Int, Byte)]`)
0x25(37) - 0x2F(47)	`Option[_]`	Option of embeddable type (`Option[Int]`)
0x30(48)	`Option[Coll[_]]`	Option of Coll of non-embeddable type (`Option[Coll[(Int, Boolean)]]`)
0x31(49) - 0x3B(59)	`Option[Coll[_]]`	Option of Coll of embeddable type (`Option[Coll[Int]]`)
0x3C(60)	`(_,_)`	Pair of non-embeddable types (`((Int, Byte), (Boolean,Box))`, etc.)
0x3D(61) - 0x47(71)	`(_, Int)`	Pair of types where first is embeddable (`(_, Int)`)
0x48(72)	`(_,_,_)`	Triple of types
0x49(73) - 0x53(83)	`(Int, _)`	Pair of types where second is embeddable (`(Int, _)`)
0x54(84)	`(_,_,_,_)`	Quadruple of types
0x55(85) - 0x5F(95)	`(_, _)`	Symmetric pair of embeddable 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
0x66(102)		reserved for String
0x67(103)		reserved for TypeVar
0x68(104)	`Header`	Header type
0x69(105)	`PreHeader`	PreHeader type
0x6A(106)	`Global`	Global type
0x6B(107)-0x6E(110)		reserved for future use
0x6F(111)		Reserved for future `Class` type (e.g. user-defined types)

For each type constructor like `Coll` or `Option` we use the encoding schema defined below. Type constructor has an associated *base code* which is multiple of 12 (e.g. 12 for `Coll[_]`, 24 for `Coll[Coll[_]]` etc.). The base code can be added to the embeddable type code to produce the code of the constructed type, for example $12 + 1 = 13$ is a code of `Coll[Byte]`. The code of type constructor (e.g. 12 in this example) is used when type parameter is non-embeddable 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 fast decoding by using `div` and `mod` operations.

Following the above encoding schema the interval of codes for data types is divided as shown

in Table 6.

5.1.3 Encoding of 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 the function types (i.e $F \in \{112, \ldots, 255\}$) can be represented as $F = D*12+R+112$, where $D, R \in \{0, \ldots, 11\}$ - indices of domain and range types correspondingly, 112 - is the first code in an interval of function types.

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

If $R = 0$ then the range type is not embeddable and the recursive descent is necessary to write/read the 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 which case we emit the separate 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 the elements type of the collection is not embeddable
2. Recursively serialize `(Int, Boolean)`
3. Emit `0x41(=0x3D+4)` because the first type of the pair is embeddable and its code is 4
4. Recursivley serialize `Boolean`
5. Emit `0x02` - the code for embeddable type `Boolean`

More examples of type serialization are shown in Table 7

Table 7: Examples of type serialization

Type	**D**	**R**	**Serialized Bytes**	**#Bytes**	**Comments**
`Byte`			2	1	simple embeddable type
`Coll[Byte]`			12 + 2 = 14	1	embeddable type in Coll
`Coll[Coll[Byte]]`			24 + 2 = 26	1	embeddable type in nested Coll
`Option[Byte]`			36 + 2 = 38	1	embeddable type in Option
`Option[Coll[Byte]]`			48 + 2 = 50	1	embeddable type in Coll nested in Option
`(Int,Int)`			84 + 4 = 88	1	symmetric pair of embeddable type
`Int=>Boolean`	4	1	161 = 4*12+1+112	1	embeddable domain and range
`(Int,Int)=>Int`	0	4	115=0*12+4+112, 88	2	embeddable range, then symmetric pair
`(Int,Boolean)`			60 + 4, 1	2	Int embedded in pair, then Boolean
`(Int,Box)=>Boolean`	0	1	0*12+1+112, 60+4, 99	3	func with embedded range, then Int embedded, then Box

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.

It is done is such a way that 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, by examining the structure of the type, that `d` is a tuple with 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). The data serialization procedure is recursive over a type tree and the corresponding sub-components of the data object. For primitive types (the leaves of the type tree) the format is fixed. The data values of ErgoTree types are serialized according to the predefined recursive function shown in Figure 5 which uses the notation from Table 3.

Figure 5: Data serialization format

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 if $v = false$ or 1 otherwise
`with` $(Byte, v \in [\![Byte]\!])$			
v	`Byte`	1	in a single byte
`with` $(N, v \in [\![N]\!]), N \in Short, Int, Long$			
v	`VLQ(ZigZag(`N`))`	[1..3]	16,32,64-bit signed integer encoded using ZigZag and then 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 **??**
`with` $(AvlTree, v \in [\![AvlTree]\!])$			
v	`AvlTree`		serialization of AvlTree data. See 5.2.3
`with` $(Coll[T], v \in [\![Coll[T]]\!])$			
len	`VLQ(UShort)`	[1..3]	length of the collection
`match` (T, v)			
`with` $(Boolean, v \in [\![Coll[Boolean]]\!])$			
v	`Bits`	[1..1024]	boolean values packed in bits
`with` $(Byte, v \in [\![Coll[Byte]]\!])$			
v	`Bytes`	[1..len]	items of the collection
`otherwise`			
`for` $i = 1$ `to` len `do serializeData(`T, v_i`) end for`			
`end match`			
`end match`			
`end serializeData`			

5.2.1 GroupElement serialization

A value of the `GroupElement` type is represented in reference implementation using `SecP256K1Point` class of the `org.bouncycastle.math.ec.custom.sec` package and serialized into ASN.1 encoding. During deserialization the different encodings are taken into account including point compression for F_p (see X9.62 sec. 4.2.1 pg. 17).

Figure 6: GroupElement serialization format

Slot	Format	#bytes	Description
`def serialize(`ge`)`			
`if` $ge.isInfinity$ `then`			
$bytes$	`rep`$(0, 33)$	33	all bytes = 0
`else`			
$bytes$	Bytes	33	where $bytes(0) \neq 0$, see `sigmastate.serialization.GroupElementSerializer`
`end if`			
`end def`			

5.2.2 SigmaProp serialization

In reference implementation values of `SigmaProp` type are serialized using `SigmaBoolean.serializer`

Figure 7: SigmaProp serialization format

Slot	Format	#bytes	Description
`def serializeSigma(`$sp : SigmaTree$`)`			
$sp.opCode$	`Byte`	1	opcode of SigmaTree node
`match` sp			
`with` $dl : ProveDlog$			
$dl.value$	`GroupElement`	33	see 5.2.1
`with` $dht : ProveDHTuple$			
$dht.gv$	`GroupElement`	33	see 5.2.1
$dht.hv$	`GroupElement`	33	
$dht.uv$	`GroupElement`	33	
$dht.vv$	`GroupElement`	33	
`with` $and : CAND$			
$nChildren$	`VLQ(UShort)`	1..3	number of children
`for` $i = 1$ `to` $nChildren$ `do serializeSigma(`and`.children(`i`)) end for`			
`with` $or : COR$			
$nChildren$	`VLQ(UShort)`	1..3	number of children
`for` $i = 1$ `to` $nChildren$ `do serializeSigma(`or`.children(`i`)) end for`			
`with` $th : CTHRESHOLD$			
$th.k$	`VLQ(UShort)`	1..3	k out of n
$nChildren$	`VLQ(UShort)`	1..3	number of children
`for` $i = 1$ `to` $nChildren$ `do serializeSigma(`th`.children(`i`)) end for`			
`with` $\_ : TrivialProp$ `// besides opCode no additional bytes`			
`end match`			
`end def`			

5.2.3 AvlTree serialization

In reference implementation values of `AvlTree` type are serialized using `AvlTreeData.serializer`.

Figure 8: AvlTree serialization format

Slot	Format	#bytes	Description
digest	`Bytes`	*DigestSize*	authenticated tree digest: root hash along with tree height
treeFlags	`UByte`	1	allowed modifications of the tree. The operation is allowed when bit is set to 1. bit0 - insert, bit1 - update, bit2 - remove
keyLength	`VLQ(UInt)`	[1..5]	the length of each key in the tree
`optional` *valueLength*			
tag	`Byte`	1	0 - no value; 1 - has value
`when` *tag* == 1			
valueLength	`VLQ(UInt)`	[1, *]	the length of all the values in the tree
`end optional`			

5.3 Constant Serialization

`Constant` format is simple and self sufficient to represent any data value. Serialized bytes of the `Constant` format contain both the type bytes and the data bytes, thus it can be stored or wire transfered and then later unambiguously interpreted. The format is shown in Figure 9

Figure 9: Constant serialization format

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

In order to parse the `Constant` format first use type serializer form section 5.1 and read the type. Then use the parsed type as an argument of data serializer given in section 5.2.

5.4 Expression Serialization

Expressions of ErgoTree are serialized as tree data structure using recursive procedure described in Figure 10. Expression nodes are represented in the reference implementation using `Value` class.

Figure 10: `Expr` 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	`Constant`	$[1..C_{max}]$	Constant serializaton according to 5.3
`else`			
body	`Op`	$[1..Expr_{max}]$	serialization of the operation depending on *e.opCode* as defined in Appendix C
`end if`			
`end serializeExpr`			

5.5 ErgoTree serialization

The ErgoTree propostions stored in UTXO boxes are represented in the reference implementation using `ErgoTree` class. The class is serialized using the `ErgoTree` serialization format shown in Figure 11.

Figure 11: ErgoTree 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
`if` $header[3] = 1$ `then`			
size	`VLQ(UInt)`	[1, *]	size of the constants and root, i.e. the number of bytes after *header* and *size*
`end for`			
numConstants	`VLQ(UInt)`	[1, *]	size of *constants* array
`for` $i = 0$ `to` $numConstants - 1$			
$const_i$	`Const`	[1, *]	constant in i-th position
`end for`			
root	`Expr`	[1, *]	If $header[4] = 1$, the *root* tree may contain ConstantPlaceholder nodes instead of Constant nodes (and may by only some of them, not all). Otherwise (i.e. if $header[4] = 0$) the root cannot contain placeholders (exception should be thrown). It is possible to have both constants and placeholders in the tree, but for every placeholder there should be a constant in *constants* array of ErgoTree instance.

Serialized instances of `ErgoTree` class 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. The meaning of the bits is shown in Figure 12.

Figure 12: ErgoTree *header* bits

Bits	Default	Description
Bits 0-2	0	language version (current version == 0)
Bit 3	0	= 1 if size of the whole tree is serialized after the header byte
Bit 4	0	= 1 if constant segregation is used for this ErgoTree
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)

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 Bit4 is set to 1 then *constants* collection contains the constants for which there may be `ConstantPlaceholder` nodes in the tree. However, if the constant segregation bit is 0, then *constants* collection should be empty and any placeholder in the tree will lead to exception.

6 Compliant Implementation

References

[CKM18] Alexander Chepurnoy, Vasily Kharin, and Dmitry Meshkov. Self-reproducing coins as universal turing machine, 2018. https://arxiv.org/abs/1806.10116.

[Fow10] Martin Fowler. *Domain-Specific Languages*. 01 2010.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196–es, December 1996.

[Ler17] Sergio Lerner. A bitcoin transaction that takes 5 hours to verify, 2017. https://bitslog.wordpress.com/2017/01/08/a-bitcoin-transaction-that-takes-5-hours-to-verify/.

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

[Ubi] Ubiquitous language. https://www.martinfowler.com/bliki/UbiquitousLanguage.html.

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

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

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Available at http://gavwood.com/Paper.pdf, 2014.

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

Table 8: Predefined types of ErgoTree

There is a section for each type with sub-sections for all available methods. Each method is characterized by the description, signature (i.e. name, parameters and return type), description of all parameters and reference to the serialization format.

There is universal primitive which can represent any method invocation (`MethodCall`). However, many method are also mapped to the special primitive operations to save storage space.

The following sub-sections are auto-generated from type descriptors of ErgoTree reference implementation.

A.1 Byte type

A.1.1 `Byte.toByte` method (Code 106.1)

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Signature	`def toByte: Byte`
Serialized as	`Downcast`

A.1.2 `Byte.toShort` method (Code 106.2)

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Signature	`def toShort: Short`
Serialized as	`Upcast`

A.1.3 `Byte.toInt` method (Code 106.3)

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Signature	`def toInt: Int`
Serialized as	`Upcast`

A.1.4 `Byte.toLong` method (Code 106.4)

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Signature	`def toLong: Long`
Serialized as	`Upcast`

A.1.5 `Byte.toBigInt` method (Code 106.5)

Description	Converts this numeric value to `BigInt`
Signature	`def toBigInt: BigInt`
Serialized as	`Upcast`

A.2 Short type

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

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Signature	`def toByte: Byte`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Signature	`def toShort: Short`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Signature	`def toInt: Int`
Serialized as	`Upcast`

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

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Signature	`def toLong: Long`
Serialized as	`Upcast`

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

Description	Converts this numeric value to `BigInt`
Signature	`def toBigInt: BigInt`
Serialized as	`Upcast`

A.3 Int type

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

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Signature	`def toByte: Byte`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Signature	`def toShort: Short`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Signature	`def toInt: Int`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Signature	`def toLong: Long`
Serialized as	`Upcast`

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

Description	Converts this numeric value to `BigInt`
Signature	`def toBigInt: BigInt`
Serialized as	`Upcast`

A.4 Long type

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

Description	Converts this numeric value to `Byte`, throwing exception if overflow.
Signature	`def toByte: Byte`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `Short`, throwing exception if overflow.
Signature	`def toShort: Short`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `Int`, throwing exception if overflow.
Signature	`def toInt: Int`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `Long`, throwing exception if overflow.
Signature	`def toLong: Long`
Serialized as	`Downcast`

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

Description	Converts this numeric value to `BigInt`
Signature	`def toBigInt: BigInt`
Serialized as	`Upcast`

A.5 BigInt type

A.5.1 `BigInt.toBigInt` method (Code 106.5)

Description	Converts this numeric value to `BigInt`
Signature	`def toBigInt: BigInt`
Serialized as	`Downcast`

A.6 GroupElement type

A.6.1 `GroupElement.getEncoded` method (Code 7.2)

Description	Get an encoding of the point value.
Signature	`def getEncoded: Coll[Byte]`
Serialized as	`PropertyCall`

A.6.2 `GroupElement.exp` method (Code 7.3)

Description	Exponentiate this `GroupElement` to the given number. Returns this to the power of k
Signature	`def exp(k: BigInt): GroupElement`
Parameters	`k` The power
Serialized as	`Exponentiate`

A.6.3 `GroupElement.multiply` method (Code 7.4)

Description	Group operation.
Signature	`def multiply(other: GroupElement): GroupElement`
Parameters	`other` other element of the group
Serialized as	`MultiplyGroup`

A.6.4 `GroupElement.negate` method (Code 7.5)

Description	Inverse element of the group.
Signature	`def negate: GroupElement`
Serialized as	`PropertyCall`

A.7 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 13.

Figure 13: Abstract syntax of sigma propositions

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`$(k, t_1, \ldots, t_n)$	`CTHRESHOLD`	knowledge of k out of n secrets
	|	`OR`$(t_1, \ldots, t_n)$	`COR`	knowledge of any one of n secrets
	|	`AND`$(t_1, \ldots, t_n)$	`CAND`	knowledge of all n secrets

Every well-formed tree of sigma proposition is a value of type `SigmaProp`, thus following the notation of Section 4 we can define a denotation of the `SigmaProp` type (i.e. a set of possible values)

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

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

A.7.1 `SigmaProp.propBytes` method (Code 8.1)

Description	Serialized bytes of this sigma proposition taken as ErgoTree.
Signature	`def propBytes: Coll[Byte]`
Serialized as	`SigmaPropBytes`

Additionally, for a list of primitive operations on `SigmaProp` type see Appendix B.

A.8 Box type

A.8.1 `Box.value` method (Code 99.1)

Description	Monetary value in NanoERGs stored in this box.
Signature	`def value: Long`
Serialized as	`ExtractAmount`

A.8.2 `Box.propositionBytes` method (Code 99.2)

Description	Serialized bytes of the guarding script which should be evaluated to true in order to open this box (spend it in a transaction).
Signature	`def propositionBytes: Coll[Byte]`
Serialized as	`ExtractScriptBytes`

A.8.3 `Box.bytes` method (Code 99.3)

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

A.8.4 `Box.bytesWithoutRef` method (Code 99.4)

Description	Serialized bytes of this box's content, excluding transactionId and index of output.
Signature	`def bytesWithoutRef: Coll[Byte]`
Serialized as	`ExtractBytesWithNoRef`

A.8.5 `Box.id` method (Code 99.5)

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

A.8.6 `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 bytes of the transaction identifier followed by the serialized bytes of the box index in the transaction outputs.
Signature	`def creationInfo: (Int,Coll[Byte])`
Serialized as	`ExtractCreationInfo`

A.8.7 `Box.tokens` method (Code 99.8)

Description	Secondary tokens
Signature	`def tokens: Coll[(Coll[Byte],Long)]`
Serialized as	`PropertyCall`

A.8.8 `Box.R4` method (Code 99.13)

Description	Non-mandatory register
Signature	`def R4[T]: Option[T]`
Serialized as	`ExtractRegisterAs`

A.8.9 `Box.R5` method (Code 99.14)

Description	Non-mandatory register
Signature	`def R5[T]: Option[T]`
Serialized as	`ExtractRegisterAs`

A.8.10 `Box.R6` method (Code 99.15)

Description	Non-mandatory register
Signature	`def R6[T]: Option[T]`
Serialized as	`ExtractRegisterAs`

A.8.11 `Box.R7` method (Code 99.16)

Description	Non-mandatory register
Signature	`def R7[T]: Option[T]`
Serialized as	`ExtractRegisterAs`

A.8.12 `Box.R8` method (Code 99.17)

Description	Non-mandatory register
Signature	`def R8[T]: Option[T]`
Serialized as	`ExtractRegisterAs`

A.8.13 `Box.R9` method (Code 99.18)

Description	Non-mandatory register
Signature	`def R9[T]: Option[T]`
Serialized as	`ExtractRegisterAs`

A.9 `AvlTree` type

A.9.1 `AvlTree.digest` method (Code 100.1)

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

A.9.2 `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`

Signature	`def enabledOperations: Byte`
Serialized as	`PropertyCall`

A.9.3 `AvlTree.keyLength` method (Code 100.3)

Description	All the elements under the tree have the same given length of the keys.
Signature	`def keyLength: Int`
Serialized as	`PropertyCall`

A.9.4 `AvlTree.valueLengthOpt` method (Code 100.4)

Description	If non-empty, all the values under the tree are of the same given length.
Signature	`def valueLengthOpt: Option[Int]`
Serialized as	`PropertyCall`

A.9.5 `AvlTree.isInsertAllowed` method (Code 100.5)

Description	Checks if Insert operation is allowed for this tree instance.
Signature	`def isInsertAllowed: Boolean`
Serialized as	`PropertyCall`

A.9.6 `AvlTree.isUpdateAllowed` method (Code 100.6)

Description	Checks if Update operation is allowed for this tree instance.
Signature	`def isUpdateAllowed: Boolean`
Serialized as	`PropertyCall`

A.9.7 `AvlTree.isRemoveAllowed` method (Code 100.7)

Description	Checks if Remove operation is allowed for this tree instance.
Signature	`def isRemoveAllowed: Boolean`
Serialized as	`PropertyCall`

A.9.8 `AvlTree.updateOperations` method (Code 100.8)

Description	Enable/disable operations of this tree producing a new tree. Since `AvlTree` is immutable, `this` tree instance remains unchanged. Returns a copy of this AvlTree instance where `this.enabledOperations` replaced by `newOperations`.
Signature	`def updateOperations(newOperations: Byte): AvlTree`
Parameters	`newOperations` a new flags which specify available operations on a new tree
Serialized as	`MethodCall`

A.9.9 `AvlTree.contains` method (Code 100.9)

Description	Checks if an entry with key `key` exists in this tree using proof `proof`. NOTE, does not support multiple keys check, use `getMany` instead. Returns `true` if a leaf with the key `key` exists. Returns `false` if leaf with provided key does not exist.
Signature	`def contains(key: Coll[Byte], proof: Coll[Byte]): Boolean`
Parameters	`key` a key of an element of this authenticated dictionary `proof` proof that they tree with `this.digest` contains the given key
Serialized as	`MethodCall`

A.9.10 `AvlTree.get` method (Code 100.10)

Description	Perform a lookup of key `key` in this tree using `proof`. Throws exception if proof is incorrect. NOTE, does not support multiple keys check, use `getMany` instead. Return `Some(bytes)` of leaf with key `key` if it exists Return `None` if leaf with provided key does not exist.
Signature	`def get(key: Coll[Byte], proof: Coll[Byte]): Option[Coll[Byte]]`
Parameters	`key` a key of an element of this authenticated dictionary `proof` proof that they tree with `this.digest` contains the given key
Serialized as	`MethodCall`

A.9.11 `AvlTree.getMany` method (Code 100.11)

Description	Perform a lookup of many keys `keys` in this tree using proof `proof`. NOTE, keys must be ordered the same way they were in lookup before proof was generated. For each key return `Some(bytes)` of leaf if it exists and `None` if is doesn't.
Signature	`def getMany(keys: Coll[Coll[Byte]], proof: Coll[Byte]): Coll[Option[Coll[Byte]]]`
Parameters	`keys` keys of elements of this authenticated dictionary `proof` proof that they tree with `this.digest` contains the given key
Serialized as	`MethodCall`

A.9.12 `AvlTree.insert` method (Code 100.12)

Description	Perform insertions of key-value entries into this authenticated dictionary using proof `proof`. Throws exception if proof is incorrect. NOTE, pairs must be ordered the same way they were in insert ops before proof was generated. Returns `Some(newTree)` if successful. Returns `None` if operations were not performed.
Signature	`def insert(operations: Coll[(Coll[Byte],Coll[Byte])], proof: Coll[Byte]): Option[AvlTree]`
Parameters	`operations` a collection of key-value pairs inserted in this dictionary `proof` a proof that the key-value pairs were inserted
Serialized as	`MethodCall`

A.9.13 `AvlTree.update` method (Code 100.13)

Description	Perform updates of key-value entries into this authenticated dictionary using proof `proof`. Throws exception if proof is incorrect. Note, pairs must be ordered the same way they were in update ops before proof was generated. Returns `Some(newTree)` if successful. Returns `None` if operations were not performed.
Signature	`def update(operations: Coll[(Coll[Byte],Coll[Byte])], proof: Coll[Byte]): Option[AvlTree]`
Parameters	`operations` a collection of key-value pairs updated in this dictionary `proof` a proof that the key-value pairs were updated
Serialized as	`MethodCall`

A.9.14 `AvlTree.remove` method (Code 100.14)

Description	Perform removal of entries into this authenticated dictionary using `proof`. Throws exception if the proof is incorrect. Returns `Some(newTree)` if successful. Returns `None` if operations were not performed. NOTE, keys must be ordered the same way they were in remove ops before proof was generated.
Signature	`def remove(operations: Coll[Coll[Byte]], proof: Coll[Byte]): Option[AvlTree]`
Parameters	`operations` a collection of key-value pairs removed from this dictionary `proof` a proof that the key-value pairs were removed
Serialized as	`MethodCall`

A.9.15 `AvlTree.updateDigest` method (Code 100.15)

Description	Replace digest of this tree producing a new tree. Since AvlTree is immutable, this tree instance remains unchanged. Returns a copy of this AvlTree instance where `this.digest` replaced by `newDigest`.
Signature	`def updateDigest(newDigest: Coll[Byte]): AvlTree`
Parameters	`newDigest` a new digest
Serialized as	`MethodCall`

A.10 Header type

A.10.1 `Header.id` method (Code 104.1)

Description	Bytes representation of ModifierId of this Header
Signature	`def id: Coll[Byte]`
Serialized as	`PropertyCall`

A.10.2 `Header.version` method (Code 104.2)

Description	Block version, to be increased on every soft and hard-fork.
Signature	`def version: Byte`
Serialized as	`PropertyCall`

A.10.3 `Header.parentId` method (Code 104.3)

Description	Bytes representation of ModifierId of the parent block
Signature	`def parentId: Coll[Byte]`
Serialized as	`PropertyCall`

A.10.4 `Header.ADProofsRoot` method (Code 104.4)

Description	Hash of ADProofs for transactions in a block
Signature	`def ADProofsRoot: Coll[Byte]`
Serialized as	`PropertyCall`

A.10.5 `Header.stateRoot` method (Code 104.5)

Description	AvlTree of a state after block application
Signature	`def stateRoot: AvlTree`
Serialized as	`PropertyCall`

A.10.6 `Header.transactionsRoot` method (Code 104.6)

Description	Root hash (for a Merkle tree) of transactions in a block.
Signature	`def transactionsRoot: Coll[Byte]`
Serialized as	`PropertyCall`

A.10.7 `Header.timestamp` method (Code 104.7)

Description	Block timestamp (in milliseconds since beginning of Unix Epoch)
Signature	`def timestamp: Long`
Serialized as	`PropertyCall`

A.10.8 `Header.nBits` method (Code 104.8)

Description	Current difficulty in a compressed view. NOTE: actually it is unsigned Int.
Signature	`def nBits: Long`
Serialized as	`PropertyCall`

A.10.9 `Header.height` method (Code 104.9)

Description	Block height
Signature	`def height: Int`
Serialized as	`PropertyCall`

A.10.10 `Header.extensionRoot` method (Code 104.10)

Description	Root hash of extension section
Signature	`def extensionRoot: Coll[Byte]`
Serialized as	`PropertyCall`

A.10.11 `Header.minerPk` method (Code 104.11)

Description	Miner public key. Should be used to collect block rewards. Part of Autolykos solution.
Signature	`def minerPk: GroupElement`
Serialized as	`PropertyCall`

A.10.12 `Header.powOnetimePk` method (Code 104.12)

Description	One-time public key. Prevents revealing of miners secret.
Signature	`def powOnetimePk: GroupElement`
Serialized as	`PropertyCall`

A.10.13 `Header.powNonce` method (Code 104.13)

Description	The nonce value generated during mining.
Signature	`def powNonce: Coll[Byte]`
Serialized as	`PropertyCall`

A.10.14 `Header.powDistance` method (Code 104.14)

Description	Distance between pseudo-random number, corresponding to nonce `powNonce` and a secret, corresponding to `minerPk`. The lower `powDistance` is, the harder it was to find this solution.
Signature	`def powDistance: BigInt`
Serialized as	`PropertyCall`

A.10.15 `Header.votes` method (Code 104.15)

Description	A collection of votes set up by the block miner.
Signature	`def votes: Coll[Byte]`
Serialized as	`PropertyCall`

A.11 PreHeader type

A.11.1 `PreHeader.version` method (Code 105.1)

Description	Block version, to be increased on every soft and hard-fork.
Signature	`def version: Byte`
Serialized as	`PropertyCall`

A.11.2 `PreHeader.parentId` method (Code 105.2)

Description	Id of parent block
Signature	`def parentId: Coll[Byte]`
Serialized as	`PropertyCall`

A.11.3 `PreHeader.timestamp` method (Code 105.3)

Description	Block timestamp (in milliseconds since beginning of Unix Epoch)
Signature	`def timestamp: Long`
Serialized as	`PropertyCall`

A.11.4 `PreHeader.nBits` method (Code 105.4)

Description	Current difficulty in a compressed view. NOTE: actually it is unsigned Int.
Signature	`def nBits: Long`
Serialized as	`PropertyCall`

A.11.5 `PreHeader.height` method (Code 105.5)

Description	Block height
Signature	`def height: Int`
Serialized as	`PropertyCall`

A.11.6 `PreHeader.minerPk` method (Code 105.6)

Description	Miner public key. Should be used to collect block rewards.
Signature	`def minerPk: GroupElement`
Serialized as	`PropertyCall`

A.11.7 `PreHeader.votes` method (Code 105.7)

Description	A collection of votes set up by the block miner.
Signature	`def votes: Coll[Byte]`
Serialized as	`PropertyCall`

A.12 Context type

A.12.1 `Context.dataInputs` method (Code 101.1)

Description	A collection of inputs of the current transaction that will not be spent.
Signature	`def dataInputs: Coll[Box]`
Serialized as	`PropertyCall`

A.12.2 `Context.headers` method (Code 101.2)

Description	A fixed number of last block headers in descending order (first header is the newest one)
Signature	`def headers: Coll[Header]`
Serialized as	`PropertyCall`

A.12.3 `Context.preHeader` method (Code 101.3)

Description	Only header fields that can be predicted by a miner when the spending transaction is added to a new block candidate.
Signature	`def preHeader: PreHeader`
Serialized as	`PropertyCall`

A.12.4 `Context.INPUTS` method (Code 101.4)

Description	A collection of inputs of the current transaction, where the `SELF` box is one of the inputs.
Signature	`def INPUTS: Coll[Box]`
Serialized as	`Inputs`

A.12.5 `Context.OUTPUTS` method (Code 101.5)

Description	A collection of outputs of the current transaction.
Signature	`def OUTPUTS: Coll[Box]`
Serialized as	`Outputs`

A.12.6 `Context.HEIGHT` method (Code 101.6)

Description	Height (block number) of the block which is currently being validated.
Signature	`def HEIGHT: Int`
Serialized as	`Height`

A.12.7 `Context.SELF` method (Code 101.7)

Description	Box whose proposition is being currently executing
Signature	`def SELF: Box`
Serialized as	`Self`

A.12.8 `Context.LastBlockUtxoRootHash` method (Code 101.9)

Description	Authenticated dynamic dictionary digest representing Utxo state before current state.
Signature	`def LastBlockUtxoRootHash: AvlTree`
Serialized as	`LastBlockUtxoRootHash`

A.12.9 `Context.minerPubKey` method (Code 101.10)

Description	Encoded bytes of public key of the miner who created the block. Equals to `preHeader.minerPk.getEncoded`
Signature	`def minerPubKey: Coll[Byte]`
Serialized as	`MinerPubkey`

A.12.10 `Context.getVar` method (Code 101.11)

Description	Get context variable with given `varId` and type. Example: `getVar[Coll[Byte]](10).get` extract a collection of bytes from the variable with varId = 10.
Signature	`def getVar[T](varId: Byte): Option[T]`
Parameters	`varId` `Byte` identifier of context variable
Serialized as	`GetVar`

A.13 Global type

A.13.1 `SigmaDslBuilder.groupGenerator` method (Code 106.1)

Description	The generator g of the group is an element of the group such that, when written multiplicatively, every element of the group is a power of g. Returns the generator of the SecP256K1 group.
Signature	`def groupGenerator: GroupElement`
Serialized as	`GroupGenerator`

A.13.2 `SigmaDslBuilder.xor` method (Code 106.2)

Description	Byte-wise XOR of two collections of bytes
Signature	`def xor(left: Coll[Byte], right: Coll[Byte]): Coll[Byte]`
Parameters	`left` left operand `right` right operand
Serialized as	`Xor`

A.14 Coll type

A.14.1 `SCollection.size` method (Code 12.1)

Description	The size of the collection in elements.
Signature	`def size: Int`
Serialized as	`SizeOf`

A.14.2 `SCollection.getOrElse` method (Code 12.2)

Description	Return the element of collection if `index` is in range `0 .. size-1`
Signature	`def getOrElse(index: Int, default: IV): IV`
Parameters	`index` index of the element of this collection `default` value to return when `index` is out of range
Serialized as	`ByIndex`

A.14.3 `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.
Signature	`def map[OV](f: (IV) => OV): Coll[OV]`
Parameters	`f` the function to apply to each element
Serialized as	`MapCollection`

A.14.4 `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`
Signature	`def exists(p: (IV) => Boolean): Boolean`
Parameters	`p` the predicate used to test elements
Serialized as	`Exists`

A.14.5 `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.
Signature	`def fold[OV](zero: OV, op: (OV,IV) => OV): OV`
Parameters	`zero` a starting value `op` the binary operator
Serialized as	`Fold`

A.14.6 `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`.
Signature	`def forall(p: (IV) => Boolean): Boolean`
Parameters	`p` the predicate used to test elements
Serialized as	`ForAll`

A.14.7 `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`
Signature	`def slice(from: Int, until: Int): Coll[IV]`
Parameters	`from` the lowest index to include from this collection `until` the lowest index to EXCLUDE from this collection
Serialized as	`Slice`

A.14.8 `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.
Signature	`def filter(p: (IV) => Boolean): Coll[IV]`
Parameters	`p` the predicate used to test elements.
Serialized as	`Filter`

A.14.9 `SCollection.append` method (Code 12.9)

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

A.14.10 `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`
Signature	`def apply(i: Int): IV`
Parameters	`i` the index
Serialized as	`ByIndex`

A.14.11 `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.
Signature	`def indices: Coll[Int]`
Serialized as	`PropertyCall`

A.14.12 `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.
Signature	`def flatMap[OV](f: (IV) => Coll[OV]): Coll[OV]`
Parameters	`f` the function to apply to each element.
Serialized as	`MethodCall`

A.14.13 `SCollection.patch` method (Code 12.19)

Description	Produces a new collection where a slice of elements in this collection is replaced by another collection. Returns a new collection consisting of all elements of this collection except that `replaced` elements starting from `from` are replaced by `patch`.
Signature	`def patch(from: Int, patch: Coll[IV], replaced: Int): Coll[IV]`
Parameters	`from` the index of the first replaced element `patch` the replacement sequence `replaced` the number of elements to drop in the original collection
Serialized as	`MethodCall`

A.14.14 `SCollection.updated` method (Code 12.20)

Description	A copy of this collection with one single replaced element. Returns a new collection which is a copy of this collection with the element at position `index` replaced by `elem`. Throws IndexOutOfBoundsException if `index` does not satisfy `0 <= index < length`.
Signature	`def updated(index: Int, elem: IV): Coll[IV]`
Parameters	`index` the position of the replacement `elem` the replacing element
Serialized as	`MethodCall`

A.14.15 `SCollection.updateMany` method (Code 12.21)

Description	Returns a copy of this collection where elements at `indexes` are replaced with `values`.
Signature	`def updateMany(indexes: Coll[Int], values: Coll[IV]): Coll[IV]`
Parameters	`indexes` the positions of the replacement `values` the values to be put in the corresponding position
Serialized as	`MethodCall`

A.14.16 `SCollection.indexOf` method (Code 12.26)

Description	Finds index of first occurrence of some value in this collection after or at some start index. Returns an index `>= from` of the first element of this collection that is equal (as determined by `==`) to `elem`, or `-1`, if none exists.
Signature	`def indexOf(elem: IV, from: Int): Int`
Parameters	`elem` the element value to search for `from` the start index
Serialized as	`MethodCall`

A.14.17 `SCollection.zip` method (Code 12.29)

Description	For this collection $(x_0, \ldots, x_N)$ and other collection $(y_0, \ldots, y_M)$ produces a collection $((x_0, y_0), \ldots, (x_K, y_K))$ where $K = min(N, M)$.
Signature	`def zip[OV](ys: Coll[OV]): Coll[(IV,OV)]`
Parameters	`ys` other collection
Serialized as	`MethodCall`

A.15 Option type

A.15.1 `SOption.isDefined` method (Code 36.2)

Description	Returns `true` if the option is an instance of `Some`, `false` otherwise.
Signature	`def isDefined: Boolean`
Serialized as	`OptionIsDefined`

A.15.2 `SOption.get` method (Code 36.3)

Description	Returns the option's value. The option must be nonempty. Throws exception if the option is empty.
Signature	`def get: T`
Serialized as	`OptionGet`

A.15.3 `SOption.getOrElse` method (Code 36.4)

Description	Returns the option's value if the option is nonempty, otherwise returns `default`.
Signature	`def getOrElse(default: T): T`
Parameters	`default` the default value
Serialized as	`OptionGetOrElse`

A.15.4 `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`.
Signature	`def map[R](f: (T) => R): Option[R]`
Parameters	`f` the function to apply
Serialized as	`MethodCall`

A.15.5 `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`.
Signature	`def filter(p: (T) => Boolean): Option[T]`
Parameters	`p` the predicate used for testing
Serialized as	`MethodCall`

B Predefined global functions

Note, the following table and sub-sections are autogenerated from sigma operation descriptors. See `SigmaPredef.scala`

Code	Mnemonic	Description
116	`SubstConstants`	See B.0.1
122	`LongToByteArray`	Converts `Long` value to big-endian bytes representation. See B.0.2
123	`ByteArrayToBigInt`	Convert big-endian bytes representation (Coll[Byte]) to BigInt value. See B.0.3
124	`ByteArrayToLong`	Convert big-endian bytes representation (Coll[Byte]) to Long value. See B.0.4
125	`Downcast`	Cast this numeric value to a smaller type (e.g. Long to Int). Throws exception if overflow. See B.0.5
126	`Upcast`	Cast this numeric value to a bigger type (e.g. Int to Long) See B.0.6
140	`SelectField`	Select tuple field by its 1-based index. E.g. `input._1` is transformed to `SelectField(input, 1)` See B.0.7
143	`LT`	Returns `true` is the left operand is less then the right operand, `false` otherwise. See B.0.8
144	`LE`	Returns `true` is the left operand is less then or equal to the right operand, `false` otherwise. See B.0.9
145	`GT`	Returns `true` is the left operand is greater then the right operand, `false` otherwise. See B.0.10
146	`GE`	Returns `true` is the left operand is greater then or equal to the right operand, `false` otherwise. See B.0.11
147	`EQ`	Compare equality of `left` and `right` arguments See B.0.12
148	`NEQ`	Compare inequality of `left` and `right` arguments See B.0.13
149	`If`	Compute condition, if true then compute trueBranch else compute falseBranch See B.0.14
150	`AND`	Returns true if *all* the elements in collection are `true`. See B.0.15
151	`OR`	Returns true if *any* the elements in collection are `true`. See B.0.16
152	`AtLeast`	See B.0.17
153	`Minus`	Returns a result of subtracting second numeric operand from the first. See B.0.18
154	`Plus`	Returns a sum of two numeric operands See B.0.19
155	`Xor`	Byte-wise XOR of two collections of bytes. Example: `xs | ys`. See B.0.20
156	`Multiply`	Returns a multiplication of two numeric operands See B.0.21
157	`Division`	Integer division of the first operand by the second operand. See B.0.22
158	`Modulo`	Reminder from division of the first operand by the second operand. See B.0.23
161	`Min`	Minimum value of two operands. See B.0.24
162	`Max`	Maximum value of two operands. See B.0.25
203	`CalcBlake2b256`	Calculate Blake2b hash from `input` bytes. See B.0.26
204	`CalcSha256`	Calculate Sha256 hash from `input` bytes. See B.0.27
205	`CreateProveDlog`	ErgoTree operation to create a new `SigmaProp` value representing public key of discrete logarithm signature protocol. See B.0.28
206	`CreateProveDHTuple`	ErgoTree operation to create a new SigmaProp value representing public key of Diffie Hellman signature protocol. Common input: (g,h,u,v) See B.0.29
209	`BoolToSigmaProp`	See B.0.30
212	`DeserializeContext`	See B.0.31
213	`DeserializeRegister`	See B.0.32
218	`Apply`	Apply the function to the arguments. See B.0.33
227	`GetVar`	Get context variable with given `varId` and type. See B.0.34
234	`SigmaAnd`	Returns sigma proposition which is proven when *all* the elements in collection are proven. See B.0.35
235	`SigmaOr`	Returns sigma proposition which is proven when *any* of the elements in collection is proven. See B.0.36
236	`BinOr`	Logical OR of two operands See B.0.37
237	`BinAnd`	Logical AND of two operands See B.0.38
238	`DecodePoint`	Convert `Coll[Byte]` to `GroupElement` using `GroupElementSerializer` See B.0.39
239	`LogicalNot`	Logical NOT operation. Returns `true` if input is `false` and `false` if input is `true`. See B.0.40
240	`Negation`	Negates numeric value `x` by returning `-x`. See B.0.41
244	`BinXor`	Logical XOR of two operands See B.0.42
255	`XorOf`	Similar to `allOf`, but performing logical XOR operation between all conditions instead of `&&` See B.0.43

B.0.1 `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.
Signature	`def substConstants[T](scriptBytes: Coll[Byte], positions: Coll[Int], newValues: Coll[T]): Coll[Byte]`
Parameters	`scriptBytes` serialized ErgoTree with ConstantSegregationFlag set to 1. `positions` 0-based indexes in ErgoTree.constants `newValues` values to be put into the corresponding positions
Serialized as	`SubstConstants`

B.0.2 `longToByteArray` method (Code 122)

Description	Converts `Long` value to big-endian bytes representation.
Signature	`def longToByteArray(input: Long): Coll[Byte]`
Parameters	`input` value to convert
Serialized as	`LongToByteArray`

B.0.3 `byteArrayToBigInt` method (Code 123)

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

B.0.4 `byteArrayToLong` method (Code 124)

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

B.0.5 `downcast` method (Code 125)

Description	Cast this numeric value to a smaller type (e.g. Long to Int). Throws exception if overflow.
Signature	`def downcast[T, R](input: T): R`
Parameters	`input` value to cast
Serialized as	`Downcast`

B.0.6 `upcast` method (Code 126)

Description	Cast this numeric value to a bigger type (e.g. Int to Long)
Signature	`def upcast[T, R](input: T): R`
Parameters	`input` value to cast
Serialized as	`Upcast`

B.0.7 `selectField` method (Code 140)

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

B.0.8 < method (Code 143)

Description	Returns `true` is the left operand is less then the right operand, `false` otherwise.
Signature	`def <[T](left: T, right: T): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`LT`

B.0.9 <= method (Code 144)

Description	Returns `true` is the left operand is less then or equal to the right operand, `false` otherwise.
Signature	`def <=[T](left: T, right: T): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`LE`

B.0.10 > method (Code 145)

Description	Returns `true` is the left operand is greater then the right operand, `false` otherwise.
Signature	`def >[T](left: T, right: T): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`GT`

B.0.11 >= method (Code 146)

Description	Returns `true` is the left operand is greater then or equal to the right operand, `false` otherwise.
Signature	`def >=[T](left: T, right: T): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`GE`

B.0.12 == method (Code 147)

Description	Compare equality of `left` and `right` arguments
Signature	`def ==[T](left: T, right: T): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`EQ`

B.0.13 != method (Code 148)

Description	Compare inequality of `left` and `right` arguments
Signature	`def !=[T](left: T, right: T): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`NEQ`

B.0.14 if method (Code 149)

Description	Compute condition, if true then compute trueBranch else compute falseBranch
Signature	`def if[T](condition: Boolean, trueBranch: T, falseBranch: T): T`

Parameters	`condition` condition expression
	`trueBranch` expression to execute when `condition == true`
	`falseBranch` expression to execute when `condition == false`

Serialized as	`If`

B.0.15 allOf method (Code 150)

Description	Returns true if *all* the elements in collection are `true`.
Signature	`def allOf(conditions: Coll[Boolean]): Boolean`
Parameters	`conditions` a collection of conditions
Serialized as	`AND`

B.0.16 anyOf method (Code 151)

Description	Returns true if *any* the elements in collection are `true`.
Signature	`def anyOf(conditions: Coll[Boolean]): Boolean`
Parameters	`conditions` a collection of conditions
Serialized as	`OR`

B.0.17 `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.
Signature	`def atLeast(bound: Int, children: Coll[SigmaProp]): SigmaProp`
Parameters	`bound` required minimum of proven children `children` proposition to be proven/validated
Serialized as	`AtLeast`

B.0.18 `-` method (Code 153)

Description	Returns a result of subtracting second numeric operand from the first.
Signature	`def -[T](left: T, right: T): T`
Parameters	`left` left operand `right` right operand
Serialized as	`Minus`

B.0.19 `+` method (Code 154)

Description	Returns a sum of two numeric operands
Signature	`def +[T](left: T, right: T): T`
Parameters	`left` left operand `right` right operand
Serialized as	`Plus`

B.0.20 `binary_|` method (Code 155)

Description	Byte-wise XOR of two collections of bytes. Example: `xs | ys`.
Signature	`def binary_|(left: Coll[Byte], right: Coll[Byte]): Coll[Byte]`
Parameters	`left` left operand `right` right operand
Serialized as	`Xor`

B.0.21 `*` method (Code 156)

Description	Returns a multiplication of two numeric operands
Signature	`def *[T](left: T, right: T): T`
Parameters	`left` left operand `right` right operand
Serialized as	`Multiply`

B.0.22 / method (Code 157)

Description	Integer division of the first operand by the second operand.
Signature	`def /[T](left: T, right: T): T`
Parameters	`left` left operand `right` right operand
Serialized as	`Division`

B.0.23 % method (Code 158)

Description	Reminder from division of the first operand by the second operand.
Signature	`def %[T](left: T, right: T): T`
Parameters	`left` left operand `right` right operand
Serialized as	`Modulo`

B.0.24 min method (Code 161)

Description	Minimum value of two operands.
Signature	`def min[T](left: T, right: T): T`
Parameters	`left` left operand `right` right operand
Serialized as	`Min`

B.0.25 max method (Code 162)

Description	Maximum value of two operands.
Signature	`def max[T](left: T, right: T): T`
Parameters	`left` left operand `right` right operand
Serialized as	`Max`

B.0.26 blake2b256 method (Code 203)

Description	Calculate Blake2b hash from `input` bytes.
Signature	`def blake2b256(input: Coll[Byte]): Coll[Byte]`
Parameters	`input` collection of bytes
Serialized as	`CalcBlake2b256`

B.0.27 sha256 method (Code 204)

Description	Calculate Sha256 hash from `input` bytes.
Signature	`def sha256(input: Coll[Byte]): Coll[Byte]`
Parameters	`input` collection of bytes
Serialized as	`CalcSha256`

B.0.28 `proveDlog` method (Code 205)

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

B.0.29 `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)
Signature	`def proveDHTuple(g: GroupElement, h: GroupElement, u: GroupElement, v: GroupElement): SigmaProp`

Parameters	`g`
	`h`
	`u`
	`v`

Serialized as	`CreateProveDHTuple`

B.0.30 `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.
Signature	`def sigmaProp(condition: Boolean): SigmaProp`
Parameters	`condition` boolean value to embed in SigmaProp value
Serialized as	`BoolToSigmaProp`

B.0.31 `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
Signature	`def executeFromVar[T](id: Byte): T`
Parameters	`id` identifier of the context variable
Serialized as	`DeserializeContext`

B.0.32 `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
Signature	`def executeFromSelfReg[T](id: Byte, default: Option[T]): T`
Parameters	`id` identifier of the register `default` optional default value, if register is not available
Serialized as	`DeserializeRegister`

B.0.33 `apply` method (Code 218)

Description	Apply the function to the arguments.
Signature	`def apply[T, R](func: (T) => R, args: T): R`
Parameters	`func` function which is applied `args` list of arguments
Serialized as	`Apply`

B.0.34 `getVar` method (Code 227)

Description	Get context variable with given `varId` and type.
Signature	`def getVar[T](varId: Byte): Option[T]`
Parameters	`varId` `Byte` identifier of context variable
Serialized as	`GetVar`

B.0.35 `allZK` method (Code 234)

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

B.0.36 `anyZK` method (Code 235)

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

B.0.37 `||` method (Code 236)

Description	Logical OR of two operands
Signature	`def ||(left: Boolean, right: Boolean): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`BinOr`

B.0.38 `&&` method (Code 237)

Description	Logical AND of two operands
Signature	`def &&(left: Boolean, right: Boolean): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`BinAnd`

B.0.39 `decodePoint` method (Code 238)

Description	Convert `Coll[Byte]` to `GroupElement` using `GroupElementSerializer`
Signature	`def decodePoint(input: Coll[Byte]): GroupElement`
Parameters	`input` serialized bytes of some `GroupElement` value
Serialized as	`DecodePoint`

B.0.40 `unary_!` method (Code 239)

Description	Logical NOT operation. Returns `true` if input is `false` and `false` if input is `true`.
Signature	`def unary_!(input: Boolean): Boolean`
Parameters	`input` input `Boolean` value
Serialized as	`LogicalNot`

B.0.41 `unary_-` method (Code 240)

Description	Negates numeric value `x` by returning `-x`.
Signature	`def unary_-[T](input: T): T`
Parameters	`input` value of numeric type
Serialized as	`Negation`

B.0.42 `^` method (Code 244)

Description	Logical XOR of two operands
Signature	`def ^(left: Boolean, right: Boolean): Boolean`
Parameters	`left` left operand `right` right operand
Serialized as	`BinXor`

B.0.43 `xorOf` method (Code 255)

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

C Serialization format of ErgoTree nodes

All operations have the same serialization format, in which OpCode byte is serialized first and then the content of the operation is serialized as described in the following subsections.

Note, these subsections are autogenerated from instrumented ValueSerializers of the reference implementation.

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` $(left, right)$			
`with` $(Constant(l, Boolean), Constant(r, Boolean))$			
opCode	`Byte`	1	always contains OpCode 133
(l, r)	`Bits`	1	two higher bits in a byte
`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` $(left, right)$			
`with` $(Constant(l, Boolean), Constant(r, Boolean))$			
opCode	`Byte`	1	always contains OpCode 133
(l, r)	`Bits`	1	two higher bits in a byte
`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. Example: `xs | ys`. See `SigmaDslBuilder.xor`

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)

Monetary value in NanoERGs stored in this box. See `Box.value`

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

C.0.34 `ExtractScriptBytes` operation (OpCode 194)

Serialized bytes of the guarding script which should be evaluated to true in order to open this box (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 bytes of the transaction identifier followed by the serialized bytes of the 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*			
$item_i$	`ValDef`	[1, *]	block's definition 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 8)
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 8)
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 `Context.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 returns `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` $(left, right)$			
`with` $(Constant(l, Boolean), Constant(r, Boolean))$			
opCode	`Byte`	1	always contains OpCode 133
(l, r)	`Bits`	1	two higher bits in a byte
`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, *]	0-based indexes in ErgoTree.constants
newValues	`Expr`	[1, *]	values to be put into the corresponding positions

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

ErgoTree types terms are serialized using special encoding designed for compact storage yet fast deserialization. In this section we describe the motivation.

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 type parameters should also be serialized as part of the operation. The following encoding is designed to minimize a number of bytes required to represent type in the serialization format of ErgoTree. Since most of the scripts will use simple types so we want them the take a single byte of the storage.

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 whould 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 (`Boolean`, `Byte`, `Short`, `Int`, `Long`, `BigInt`, `GroupElement`, `SigmaProp`, `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 preferably by a single byte (see examples in Figure 7). For other types, we do recursive descent down the type tree as it is defined in section 5.1.4.

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 at minimum. Additionally, the block validation time should be as small as possible so that a miner can start solving the PoW puzzle as soon as possible to increase the probability of the successful mining.

For every script (of an input `box`) the following is done in order to validate it (and should be executed as fast as possible):

1. A Context object is created with `SELF` = box
2. ErgoTree is traversed to build a cost graph - the graph for the cost estimation
3. Cost estimation is computed by evaluating the cost graph with the current context
4. If the cost within the limit, the ErgoTree is evaluated using the context to obtain sigma proposition (see `SigmaProp`)
5. Sigma protocol verification procedure is executed

D.2.2 The Potential Script Processing Optimization

Before an ErgoScript contract can be stored in a blockchain it should be first compiled from its source code into ErgoTree and then serialized into byte array. Because the ErgoTree is purely functional graph-based IR, the compiler may perform various optimizations for reducing a size of the tree. This will have an effect of normalization/unification, in which different original scripts may be compiled into the identical ErgoTrees and as a result the identical serialized bytes.

In many cases two boxes will have the same ErgoTree up to a substitution of constants. For example all pay-to-public-key scripts have the same ErgoTree template in which only public key (constant of GroupElement type) is replaced.

Because of normalization, and also because of script template reusability, the number different scripts templates is much less than the number of actual ErgoTrees in the UTXO boxes. For example we may have 1000s of different script templates in a blockchain with millions of UTXO boxes.

The average reusability ratio is 1000 in this case. And even those 1000 different scripts may have different usage frequency. Having big reusability ratio we can optimize script evaluation by performing the step 2 from section D.2.1 only once per unique script.

The compiled cost graph can be cached in `Map[Array[Byte], Context => Int]`. Every ErgoTree template extracted from an input box can be used as the key in this map to obtain the graph which is ready to execute.

However, there is an obstacle to the optimization by caching, i.e. the constants embedded in contracts. In many cases it is natural to embed constants in the ErgoTree body with the most notable scenario is when public keys are embedded. As the result two functionally identical scripts are serialized to the different byte arrays because they have the different embedded constants.

D.2.3 Templatized ErgoTree

A 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 an indexed placeholder node (see

`ConstantPlaceholder`). Each placeholder have an index of the constant in the `constants` collection of `ErgoTree`.

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. The index of the placeholder can be assigned by breadth-first topological order of the graph traversal during compilation of ErgoScript into ErgoTree. Whatever method is used, a placeholder should always refer to an existing constant.

Thus the format of serialized ErgoTree with is shown in Figure 11 which contains:

1. The bytes of collection with *segregated constants*
2. The bytes of script expression with placeholders

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

Using such script format we can use the script expression bytes as a key in the cache. The observation is that after the constants are segregated, what remains is the template. Thus, instead of applying steps 1-2 from section D.2.1 to *constant-full* scripts we can apply them to *constant-less* templates. Before applying the steps 3 - 5 we need to bind placeholders with actual values taken from the constants collection and then evaluate both cost graph and ErgoTree.

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);
}
```