Formal Specification of the Plutus Core Language

DRAFT

Plutus Core Team

10th September 2024

Abstract

This is intended to be a reference guide for developers who want to utilise the Plutus Core infrastructure. We lay out the grammar and syntax of untyped Plutus Core terms, and their semantics and evaluation rules. We also describe the built-in types and functions. The Appendices include a list of supported builtins in each era and some aspects of Plutus Core which have been mechanically formalised.

This document only describes untyped Plutus Core: a subsequent version will also include the syntax and semantics of Typed Plutus Core and describe its relation to untyped Plutus Core.

Contents

Chapter 1

Preliminaries

1.1 Introduction

Plutus Core (more correctly, Untyped Plutus Core) is an eagerly-evaluated version of the untyped lambda calculus extended with some "built-in" types and functions; it is intended for the implementation of validation scripts on the Cardano blockchain. This document presents the syntax and semantics of Plutus Core, a specification of an efficient evaluator, a description of the built-in types and functions available in various releases of Cardano, and a specification of the binary serialisation format used by Plutus Core.

Since Plutus Core is intended for use in an environment where computation is potentially expensive and excessively long computations can be problematic we have also developed a costing infrastructure for Plutus Core programs. A description of this will be added in a later version of this document.

We also have a typed version of Plutus Core which provides extra robustness when untyped Plutus Core is used as a compilation target, and we will eventually provide a specification of the type system and semantics of Typed Plutus Core here as well, together with its relationship to Untyped Plutus Core.

1.2 Some basic notation

We begin with some notation which will be used throughout the document.

1.2.1 Sets

- The symbol $\uplus$ denotes a disjoint union of sets; for emphasis we often use this to denote the union of sets which we know to be disjoint.
- Given a set X, X^* denotes the set of finite sequences of elements of X:

$$X^* = \biguplus \{X^n : n \in \mathbb{N}\}.$$

- $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.
- $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$.
- $\mathbb{N}_{[a,b]} = \{n \in \mathbb{N} : a \leq n \leq b\}$.
- $\mathbb{B} = \mathbb{N}_{[0,255]}$, the set of 8-bit bytes.

- $\mathbb{B}^*$ is the set of all bytestrings.
- $\mathbb{b} = \{0, 1\}$, the set of bits.
- $\mathbb{b}^*$ is the set of all bitstrings.
- $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$.
- $\mathbb{F}_q$ denotes a finite field with q elements (q a prime power).
- $\mathbb{F}_q^*$ denotes the multiplicative group of nonzero elements of $\mathbb{F}_q$.
- $\mathbb{U}$ denotes the set of Unicode scalar values, as defined in [44, Definition D76].
- $\mathbb{U}^*$ is the set of all Unicode strings.
- We assume that there is a special symbol $\times$ which does not appear in any other set we mention. The symbol $\times$ is used to indicate that some sort of error condition has occurred, and we will often need to consider situations in which a value is either $\times$ or a member of some set S. For brevity, if S is a set then we define
$$S_\times := S \uplus \{\times\}.$$

1.2.2 Lists

- The symbol [] denotes an empty list.
- The notation $[x_m, \dots, x_n]$ denotes a list containing the elements $x_m, \dots, x_n$. If $m > n$ then the list is empty.
- The length of a list L is denoted by $\ell(L)$.
- Given two lists $L = [x_1, \dots, x_m]$ and $L' = [y_1, \dots, y_n]$, $L{\cdot}L'$ denotes their concatenation $[x_1, \dots, x_m, y_1, \dots, y_n]$.
- Given an object x and a list $L = [x_1, \dots, x_n]$, we denote the list $[x, x_1, \dots, x_n]$ by $x \cdot L$.
- Given a list $L = [x_1, \dots, x_n]$ and an object x, we denote the list $[x_1, \dots, x_n, x]$ by $L{\cdot}x$.
- Given a syntactic category V, the symbol $\overline{V}$ denotes a possibly empty list $[V_1, \dots, V_n]$ of elements $V_i \in V$.

1.2.3 Bytestrings and bitstrings

We make frequent use of bytestrings and bitstrings and for the sake of conciseness we occasionally use special notation. We also define conversion functions between bytestrings and bistrings

- We typically index the bytes in bytestrings starting from the *left* end but the bits in bitstrings from the *right* end.
- The bytestring $[c_0, \dots, c_n]$ may be denoted by $c_0 \cdots c_n$ ($n \geq 1$, $c_i \in \mathbb{B}$); the empty bytestring may be denoted by ϵ.

- The bitstring $[b_n, \ldots, b_0]$ may be denoted by $b_n \cdots b_0$ ($n \geq 1$, $b_i \in \mathbb{b}$); the empty bitstring may be denoted by ϵ: we also use this symbol for the empty bytestring, but this should not cause any confusion.
- In the special case of bitstrings sometimes use notation such as `101110` to denote the list $[1, 0, 1, 1, 1, 0]$; we use a teletype font to avoid confusion with decimal numbers.
- A bytestring can naturally be viewed as a bitstring whose length is a multiple of 8 simply by concatenating the bits of the individual bytes, and vice-versa (by breaking the bitstring into groups of 8 bits). To make this precise we define two conversion functions $\mathsf{bits} : \mathbb{B}^* \to \mathbb{b}^*$ and $\mathsf{bytes} : \{s \in \mathbb{b}^* : 8 \mid \ell(s)\} \to \mathbb{B}^*$. These depend on the fact that any $c \in \mathbb{B}$ can be written uniquely in the form $\Sigma_{i=0}^{7} 2^i b_i$ with $b_0, \ldots, b_7 \in \mathbb{b}$.
  - $\mathsf{bits}([c_0, \ldots, c_{n-1}]) = [b_{8n-1}, \ldots, b_0]$ where $c_j = \Sigma_{i=0}^{7} 2^i b_{8(n-j-1)+i}$
  - $\mathsf{bytes}([b_{8n-1}, \ldots, b_0]) = [c_0, \ldots, c_{n-1}]$ where $c_j = \Sigma_{i=0}^{7} 2^i b_{8(n-j-1)+i}$.

1.2.4 Miscellaneous notation

- Given integers $k \in \mathbb{Z}$ and $n \geq 1$ we write $k \bmod n = \min\{r \in \mathbb{Z} : r \geq 0 \text{ and } n | k - r\}$

Chapter 2

Untyped Plutus Core

2.1 The grammar of Plutus Core

This section presents the grammar of Plutus Core in a Lisp-like form. This is intended as a specification of the abstract syntax of the language; it may also by used by tools as a concrete syntax for working with Plutus Core programs, but this is a secondary use and we do not make any guarantees of its completeness when used in this way. The primary concrete form of Plutus Core programs is the binary format described in Appendix C.

2.1.1 Lexical grammar

Name	n	::=	`[a-zA-Z][a-zA-Z0-9_']*`	name
Var	x	::=	n	term variable
BuiltinName	bn	::=	n	built-in function name
Version	v	::=	`[0-9]`$^+$`.[0-9]`$^+$`.[0-9]`$^+$	version
Natural	k	::=	`[0-9]+`	a natural number
Constant	c	::=	⟨literal constant⟩	

Figure 2.1: Lexical grammar of Plutus Core

2.1.2 Grammar

Term	L, M, N	::=	x	variable
			(con T c)	constant
			(builtin b)	builtin
			(lam x M)	λ abstraction
			[M N]	function application
			(delay M)	delay execution of a term
			(force M)	force execution of a term
			(constr k $M_0 \ldots M_{m-1}$)	constructor with tag k and m arguments
			(case M $N_0 \ldots N_{m-1}$)	case analysis with m alternatives
			(error)	error
Program	P	::=	(program v M)	versioned program

Figure 2.2: Grammar of untyped Plutus Core

2.1.3 Notes

Version numbers. The version number at the start of a program specifies the Plutus Core language version used in the program.

A *Plutus Core language version* describes a version of the basic language with a particular set of features. A language version consists of three non-negative integers separated by decimal points, for example `1.4.2`. Language versions are ordered lexicographically.

The grammar above describes Plutus Core version 1.1.0. Version 1.0.0 is identical, except that `constr` and `case` are not included. Version 1.0.0 is fully forward-compatible with version 1.1.0, so any valid version 1.0.0 program is also a valid version 1.1.0 program. The semantics, evaluator and serialisation formats described later in this document all apply to both versions, except that it is an error to use `constr` or `case` in any program with a version prior to 1.1.0: a parser, deserialiser, or evaluator should fail immediately if `constr` or `case` is encountered when processing such a program.

Scoping. For simplicity, **we assume throughout that the body of a Plutus Core program is a closed term**, ie, that it contains no free variables. Thus `(program 1.0.0 (lam x x))` is a valid program but `(program 1.0.0 (lam x y))` is not, since the variable `y` is free. This condition should be checked before execution of any program commences, and the program should be rejected if its body is not closed. The assumption implies that any variable x occurring in the body of a program must be bound by an occurrence of `lam` in some enclosing term; in this case, we always assume that x refers to the *most recent* (ie, innermost) such binding.

Iterated applications. An application of a term M to a term N is represented by [M N]. We may occasionally write [M $N_1 \ldots N_k$] or [M $\overline{N}$] as an abbreviation for an iterated application [… [[M N_1] N_2] … N_k], and tools may also use this as concrete syntax.

Constructors and case analysis Plutus Core supports creating structured data using `constr` and deconstructing it using `case`. Both of these terms are unusual in that they have (possibly empty) lists of children: `constr` has the (0-based) *tag* and then a list of arguments; `case` has a scrutinee and then a list of case branches. Their behaviour is mostly straightforward: `constr` evaluates its arguments and forms a value; `case` evaluates the scrutinee into a `constr` value, selects the branch corresponding to the tag on the value, and then applies that to the arguments in the value. The only thing to note is that `case` does

not strictly evaluate the case branches, only applying (and hence evaluating) the one that is eventually selected.

Constructor tags Constructor tags can in principle be any natural number. In practice, since they cannot be dynamically constructed, we can limit them to a fixed size without having to worry about overflow. So we limit them to 64 bits, although this is currently only enforced in the binary format (Appendix C).

Built-in types and functions. The language is parameterised by a set $\mathcal{U}$ of *built-in types* (we sometimes refer to $\mathcal{U}$ as the *universe*) and a set $\mathcal{B}$ of *built-in functions* (*builtins* for short), both of which are sets of Names. Briefly, the built-in types represent sets of constants such as integers or strings; constant expressions `(con` T c`)` represent values of the built-in types (the integer 123 or the string `"string"`, for example), and built-in functions are functions operating on these values, and possibly also general Plutus Core terms. Precise details are given in Section 2.2.

See Section 4.3 for a description of the types and functions which have already been deployed on the Cardano blockchain (or will be in the near future).

De Bruijn indices. The grammar defines names to be textual strings, but occasionally (specifically in Appendix C) we want to use de Bruijn indices ([24], [12, C.3]), and for this we redefine names to be natural numbers. In de Bruijn terms, λ-expressions do not need to bind a variable, but in order to re-use our existing syntax we arbitrarily use 0 for the bound variable, so that all λ-expressions are of the form `(lam 0` M`)`; other variables (ie, those not appearing immediately after a `lam` binder) are represented by natural number greater than zero.

Lists in constructor and case terms The grammar defines constructor and case terms to have a variable number of subterms written in sequence with no delimiters. This corresponds to the concrete syntax, e.g. we write `(constr 0` t_1 t_2 t_3`)`. However, in the rest of the specification we will abuse notation and treat these terms as having *lists* of subterms.

2.2 Interpretation of built-in types and functions

As mentioned above, Plutus Core is generic over a universe $\mathcal{U}$ of types and a set $\mathcal{B}$ of built-in functions. As the terminology suggests, built-in functions are interpreted as functions over terms and elements of the built-in types: in this section we make this interpretation precise by giving a specification of built-in types and functions in a set-theoretic denotational style. We require a considerable amount of extra notation in order to do this, and we emphasise that nothing in this section is part of the syntax of Plutus Core: it is meta-notation introduced purely for specification purposes.

2.2.1 Built-in types

We require some extra syntactic notation for built-in types: see Figure 2.3.

at	$::=$	n	Atomic type
op	$::=$	n	Type operator
T	$::=$	$at \mid op(T, T, ..., T)$	Built-in type

Figure 2.3: Type names and operators

We assume that we have a set $\mathcal{U}_0$ of *atomic type names* and a set $\mathcal{O}$ of *type operator names*. Each type operator name $op \in \mathcal{O}$ has an *argument count* $|op| \in \mathbb{N}^+$, and a type name $op(T_1, \ldots, T_n)$ is well-formed if and only if $n = |op|$. We define the *universe* $\mathcal{U}$ to be the closure of $\mathcal{U}_0$ under repeated applications of operators in $\mathcal{O}$:

$$\mathcal{U}_{i+1} = \mathcal{U}_i \cup \{op(T_1, \ldots, T_{|op|}) : op \in \mathcal{O}, T_1, \ldots, T_{|op|} \in \mathcal{U}_i\}$$
$$\mathcal{U} = \bigcup\{\mathcal{U}_i : i \in \mathbb{N}^+\}$$

The universe $\mathcal{U}$ consists entirely of *names*, and the semantics of these names are given by *denotations*. Each built-in type $T \in \mathcal{U}$ is associated with some mathematical set $[\![T]\!]$, the *denotation* of T. For example, we might have $[\![\texttt{bool}]\!] = \{\mathsf{true}, \mathsf{false}\}$ and $[\![\texttt{integer}]\!] = \mathbb{Z}$ and $[\![\texttt{pair}\ (a, b)]\!] = [\![a]\!] \times [\![b]\!]$.

See Section 4.3 for a description of the types and functions which have already been deployed on the Cardano blockchain (or will be in the near future).

For non-atomic type names $T = op(T_1, \ldots, T_r)$ we would generally expect the denotation of T to be obtained in some uniform way (ie, parametrically) from the denotations of $T_1, \ldots, T_r$; we do not insist on this though.

2.2.1.1 Type variables

Built-in functions can be polymorphic, and to deal with this we need *type variables*. An argument of a polymorphic function can be either restricted to built-in types or can be an arbitrary term, and we define two different kinds of type variables to cover these two situations. See Figure 2.4.

TypeVariable	tv	$::=$	n_*	fully polymorphic type variable
			$n_\#$	built-in-polymorphic type variable

Figure 2.4: Type variables

We denote the set of all possible type variables by $\mathcal{V}$, the set of all fully-polymorphic type variables by $\mathcal{V}_*$, and the set of all built-in-polymorphic type variables $v_\#$ by $\mathcal{V}_\#$. Note that $\mathcal{V} \cap \mathcal{U} = \varnothing$ since the symbols $_*$ and $_\#$ do not occur in names in $\mathcal{U}$. The two kinds of type variable are required because we have two different types of polymorphism. Later on we will see that built-in functions can take arguments which can be of a type which is unknown but must be in $\mathcal{U}$, whereas other arguments can range over a larger set of values such as the set of all Plutus Core terms. Type variables in $\mathcal{V}_\#$ are used in the former situation and $\mathcal{V}_*$ in the latter.

Given a variable $v \in \mathcal{V}$ we sometimes write

$$v :: \# \quad \text{if } v \in \mathcal{V}_\#$$

and

$$v :: * \quad \text{if } v \in \mathcal{V}_*.$$

2.2.1.2 Polymorphic types

We also need to talk about polymorphic types, and to do this we define an extended universe of polymorphic types $\mathcal{U}_\#$ by adjoining $\mathcal{V}_\#$ to $\mathcal{U}_0$ and closing under type operators as before:

$$\mathcal{U}_{\#,0} = \mathcal{U}_0 \cup \mathcal{V}_\#$$

$$\mathcal{U}_{\#,i+1} = \mathcal{U}_{\#,i} \cup \{op(T_1, \ldots, T_{|op|}) : op \in \mathcal{O}, T_1, \ldots, T_{|op|} \in \mathcal{U}_{\#,i}\}$$
$$\mathcal{U}_{\#} = \bigcup \{\mathcal{U}_{\#,i} : i \in \mathbb{N}^+\}.$$

We will denote a typical element of $\mathcal{U}_{\#}$ by the symbol P (possibly subscripted).
We define the set of *free #-variables* of an element of $\mathcal{U}_{\#}$ by

$$\mathsf{FV}_{\#}(P) = \varnothing \text{ if } P \in \mathcal{U}_0$$
$$\mathsf{FV}_{\#}(v_{\#}) = \{v_{\#}\}$$
$$\mathsf{FV}_{\#}(op(P_1, \ldots, P_k)) = \mathsf{FV}_{\#}(P_1) \cup \mathsf{FV}_{\#}(P_2) \cup \cdots \cup \mathsf{FV}_{\#}(P_r).$$

Thus $\mathsf{FV}_{\#}(P) \subseteq \mathcal{V}_{\#}$ for all $P \in \mathcal{U}$. We say that a type name $P \in \mathcal{U}_{\#}$ is *monomorphic* if $\mathsf{FV}_{\#}(P) = \varnothing$ (in which case we actually have $P \in \mathcal{U}$); otherwise P is *polymorphic*. The fact that type variables in $\mathcal{U}_{\#}$ are only allowed to come from $\mathcal{V}_{\#}$ will ensure that values of polymorphic types such as lists and pairs can only contain values of built-in types: in particular, we will not be able to construct types representing things such as lists of Plutus Core terms.

2.2.1.3 Type assignments

A *type assignment* is a function $S : D \to \mathcal{U}$ where D is some subset of $\mathcal{V}_{\#}$. As usual we say that D is the *domain* of S and denote it by $\operatorname{dom} S$.

We can extend a type assignment S to a map $\hat{S} : \mathcal{U}_{\#} \uplus \mathcal{V}_* \to \mathcal{U}_{\#} \uplus \mathcal{V}_*$ by defining

$$\begin{aligned}
\hat{S}(v_{\#}) &= S(v_{\#}) \quad \text{if } v_{\#} \in \operatorname{dom} S \\
\hat{S}(v_{\#}) &= v_{\#} \quad \text{if } v_{\#} \in \mathcal{V}_{\#} \setminus \operatorname{dom} S \\
\hat{S}(T) &= T \quad \text{if } T \in \mathcal{U}_0 \\
\hat{S}(op(P_1, \ldots, P_n)) &= op(\hat{S}(P_1), \ldots, \hat{S}(P_n)) \\
\hat{S}(v_*) &= v_* \quad \text{if } v_* \in \mathcal{V}_*.
\end{aligned}$$

If $P \in \mathcal{U}_{\#}$ and S is a type assignment with $\mathsf{FV}_{\#}(P) \subseteq \operatorname{dom} S$ then in fact $\hat{S}(P) \in \mathcal{U}$; in this case we say that $\hat{S}(P)$ is an *instance* or a *monomorphisation* of P (*via* S). If T is an instance of P then there is a unique smallest S (with $\mathsf{FV}_{\#}(P) = \operatorname{dom} S$) such that $T = \hat{S}(P)$: we write $T \preceq_S P$ to indicate that T is an instance of P via S and S is minimal.

Constructing type assignments. We say that a collection $\{S_i : 1 \leq i \leq n\}$ of type assignments is *consistent* if $S_i|_{D_{ij}} = S_j|_{D_{ij}}$ for all i and j, where $|$ denotes function restriction and $D_{ij} = \operatorname{dom} S_i \cap \operatorname{dom} S_j$. If this is the case then (viewing functions as sets of pairs in the usual way) $S_1 \cup \cdots \cup S_n$ is also a well-formed type assignment (each variable in its domain is associated with exactly one type).

Given $T \in \mathcal{U}$ and $P \in \mathcal{U}_{\#}$ it can be shown that $T \preceq_S P$ if and only if one of the following holds:

- $T = P$ and $S = \varnothing$.
- $P \in \mathcal{V}_{\#}$ and $S = \{(v_{\#}, T)\}$.
- – $T = op(T_1, \ldots, T_n)$ with each $T_i \in \mathcal{U}$.

- $P = op(P_1, \ldots, P_n)$ with each $P_i \in \mathcal{U}_\#$.
- $T_i \preceq_{S_i} P_i$ for $1 \leq i \leq n$.
- $\{S_1, \ldots, S_n\}$ is consistent.
- $S = S_1 \cup \cdots \cup S_n$.

This allows us to decide whether $T \in \mathcal{U}$ is an instance of $P \in \mathcal{U}_\#$ and, if so, to construct an S with $T \preceq_S P$.

2.2.2 Built-in functions

2.2.2.1 Inputs to built-in functions

To treat the typed and untyped versions of Plutus Core uniformly it is necessary to make the machinery of built-in functions generic over a set $\mathcal{I}$ of *inputs* which are taken as arguments by built-in functions. In practice $\mathcal{I}$ will be the set of Plutus Core values or something very closely related.

We require $\mathcal{I}$ to have the following two properties:

- $\mathcal{I}$ is disjoint from $[\![T]\!]$ for all $T \in \mathcal{U}$
- There should be disjoint subsets $\mathcal{C}_T \subseteq \mathcal{I}$ (where $T \in \mathcal{U}$) of *constants of type* T and maps $[\![\cdot]\!]_T : \mathcal{C}_T \to [\![T]\!]$ (*denotation*) and $\{\!\{\cdot\}\!\}_T : [\![T]\!] \to \mathcal{C}_T$ (*reification*) such that $\{\!\{[\![c]\!]_T\}\!\}_T = c$ for all $c \in \mathcal{C}_T$. We do not require these maps to be bijective (for example, there may be multiple inputs with the same denotation), but the condition implies that $[\![\cdot]\!]_T$ is surjective and $\{\!\{\cdot\}\!\}_T$ is injective.

It is also convenient to let $[\![\mathcal{I}]\!] = \mathcal{I}$ and define both $[\![\cdot]\!]_\mathcal{I}$ and $\{\!\{\cdot\}\!\}_\mathcal{I}$ to be the identity function.

For example, we could take $\mathcal{I}$ to be the set of all Plutus Core values (see Section 2.3.1), $\mathcal{C}_T$ to be the set of all terms of the form `(con` T c`)`, and $[\![\cdot]\!]_T$ to be the function which maps `(con` T c`)` to c. For simplicity we are assuming that mathematical entities occurring as members of type denotations $[\![T]\!]$ are embedded directly as values c in Plutus Core constant terms. In reality, tools which work with Plutus Core will need some concrete syntactic representation of constants; we do not specify this here, but see Section 4.3 for suggested syntax for the built-in types currently in use on the Cardano blockchain.

2.2.2.2 Signatures and denotations of built-in functions

We will consistently use the symbol τ and subscripted versions of it to denote members of $\mathcal{U}_\# \uplus \mathcal{V}_*$ in the rest of the document; these indicate the types of values consumed and returned by built-in functions.

We also define a class of *quantifications* which are used to introduce type variables: a quantification is a symbol of the form $\forall v$ with $v \in \mathcal{V}$; the set of of all possible quantifications is denoted by $\mathcal{Q}$.

Signatures. Every built-in function $b \in \mathcal{B}$ has a *signature* $\sigma(b)$ which describes the types of its arguments and its return value: a signature is of the form

$$[\iota_1, \ldots, \iota_n] \to \tau$$

with

- $\iota_j \in \mathcal{U}_\# \uplus \mathcal{V}_* \uplus \mathcal{Q}$ for all j
- $\tau \in \mathcal{U}_\# \uplus \mathcal{V}_*$

- $|\{j : \iota_j \notin \mathcal{Q}\}| \geq 1$ (so $n \geq 1$)
- If ι_j involves $v \in \mathcal{V}$ then $\iota_k = \forall v$ for some $k < j$, and similarly for τ; in other words, any type variable v must be introduced by a quantification before it is used. (Here ι *involves* v if either $\iota = T \in \mathcal{U}_\#$ and $v \in \mathsf{FV}_\#(T)$ or $\iota = v$ and $v \in \mathcal{V}_*$.)
- If τ involves $v \in \mathcal{V}$ then some ι_j must involve v; this implies that $\mathsf{FV}_\#(\tau) \subseteq \bigcup\{\mathsf{FV}_\#(\iota_j) : \iota_j \in \mathcal{U}_\#\}$.
- If $j \neq k$ and $\iota_j, \iota_k \in \mathcal{Q}$ then $\iota_j \neq \iota_k$; ie, no quantification appears more than once.
- If $\iota_i = \forall v \in \mathcal{Q}$ then some $i_j \notin \mathcal{Q}$ with $j > i$ must involve v (signatures are not allowed to contain phantom type variables).

For example, in our default set of built-in functions we have the functions `mkCons` with signature $[\forall a_\#, a_\#, \texttt{list}\,(a_\#)] \to \texttt{list}\,(a_\#)$ and `ifThenElse` with signature $[\forall a_*, \texttt{bool}, a_*, a_*] \to a_*$. When we use `mkCons` its arguments must be of built-in types, but the two final arguments of `ifThenElse` can be any Plutus Core values.

If b has signature $[\iota_1, \ldots, \iota_n] \to \tau$ then we define the *arity* of b to be

$$\alpha(b) = [\iota_1, \ldots, \iota_n].$$

We also define

$$\chi(b) = n.$$

We may abuse notation slightly by using the symbol σ to denote a specific signature as well as the function which maps built-in function names to signatures, and similarly with the symbol α.

Given a signature $\sigma = [\iota_1, \ldots, \iota_n] \to \tau$, we define the *reduced signature* $\bar{\sigma}$ to be

$$\bar{\sigma} = [\iota_j : \iota_j \notin \mathcal{Q}] \to \tau$$

Here we have extended the usual set comprehension notation to lists in the obvious way, so $\bar{\sigma}$ just denotes the signature σ with all quantifications omitted. We will often write a reduced signature in the form $[\tau_1, \ldots, \tau_m] \to \tau$ to emphasise that the entries are *types*, and $\forall$ does not appear.

Also, given an arity $= [\iota_1, \ldots, \iota_n]$, the *reduced arity* is

$$\bar{\alpha} = [\iota_j : \iota_j \notin \mathcal{Q}].$$

Commentary. What is the intended meaning of the notation introduced above? In Typed Plutus Core we have to instantiate polymorphic functions (both built-in functions and polymorphic lambda terms) at concrete types before they can be applied, and in Untyped Plutus Core instantiation is replaced by an application of `force`. When we are applying a built-in function we supply its arguments one by one, and we can also apply `force` (or perform type instantiation in the typed case) to a partially-applied builtin "between" arguments (and also after the final argument); no computation occurs until all arguments have been supplied and all `force`s have been applied. The arity (read from left to right) specifies what types of arguments are expected and how they should be interleaved with applications of `force`, and $\chi(b)$ tells

you the total number of arguments and applications of `force` that a built-in function b requires. A fully-polymorphic type variable a_* indicates that an arbitrary value from $\mathcal{I}$ can be provided, whereas a type from $\mathcal{U}_\#$ indicates that a value of the specified built-in type is expected. Occurrences of quantifications indicate that `force` is to be applied to a partially-applied builtin; we allow this purely so that partially-applied builtins can be treated in the same way as delayed lambda-abstractions: `force` has no effect unless it is the very last item in the signature. In Plutus Core, partially-applied builtins are values which can be treated like any others (for example, by being passed as an argument to a `lam`-expression): see Section 2.3.1.

2.2.2.3 Denotations of built-in functions

The basic idea is that a built-in function b should represent some mathematical function on the denotations of the types of its inputs. However, this is complicated by the presence of polymorphism and we have to require that there is such a function for every possible monomorphisation of b.

More precisely, suppose that we have a builtin b with reduced signature $[\tau_1, \dots \tau_n] \to \tau$. For every type assignment S with $\mathrm{dom}\, S = \mathsf{FV}_\#(\tau_1) \cup \cdots \cup \mathsf{FV}_\#(\tau_n)$ (which contains $\mathsf{FV}_\#(\tau)$ by the conditions on signatures in Section 2.2.2.2) we require a *denotation of b at S*, a function

$$[\![b]\!]_S : [\![\hat{S}(\tau_1)]\!] \times \cdots \times [\![\hat{S}(\tau_n)]\!] \to [\![\hat{S}(\tau)]\!]_\times.$$

where

$$[\![v_*]\!] = \mathcal{I} \text{ for } v_* \in \mathcal{V}_*.$$

This makes sense because $\hat{S}(\tau_i) \in \mathcal{U} \uplus \mathcal{I}$ for all i, so $[\![\hat{S}(\tau_i)]\!]$ is always defined, and similarly for τ.

If $\mathsf{FV}_\#(\bar{\sigma}(b)) = \varnothing$ (in which case we say that b is *monomorphic*) then the only relevant type assignment will be the empty one; in this case we have a single denotation

$$[\![b]\!]_\varnothing : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]_\times.$$

Denotations of builtins are mathematical functions which terminate on every possible input; the symbol $\times$ can be returned by a function to indicate that something has gone wrong, for example if an argument is out of range.

In practice we expect most builtins to be *parametrically polymorphic* [46, 39], so that the denotation $[\![b]\!]_S$ will be the "same" for all type assignments S; we do not insist on this though.

2.2.2.4 Results of built-in functions.

If r is the result of the evaluation of some built-in function there are thus three possibilities:

1. $r \in [\![T]\!]$ for some $T \in \mathcal{U}$.
2. $r \in \mathcal{I}$.
3. $r = \times$.

In other words,

$$r \in \mathcal{R} := \biguplus\{[\![T]\!] : T \in \mathcal{U}\} \uplus \mathcal{I} \uplus \{\times\}.$$

Our assumptions on the set $\mathcal{I}$ (Section 2.2.2.1) allow us define a function

$$\{\!|\cdot|\!\} : \mathcal{R} \to \mathcal{I}_\times$$

which converts results of built-in functions back into inputs (or the $\times$ symbol):

1. If $r \in [\![T]\!]$, then $\{\!|r|\!\} = \{\!|r|\!\}_T \in \mathcal{C}_T \subseteq \mathcal{I}$.
2. If $r \in \mathcal{I}$ then $\{\!|r|\!\} = r$.
3. $\{\!|\mathsf{x}|\!\} = \mathsf{x}$.

2.2.2.5 Parametricity for *-polymorphic arguments

A built-in function b can only inspect arguments which are values of built-in types; other arguments (occurring as a_* in $\bar{\sigma}(b)$) are treated opaquely, and can be discarded or returned as (part of) a result, but cannot be altered or examined (in particular, they cannot be compared for equality): b is *parametrically polymorphic* in such arguments. This implies that if a builtin returns a value $v \in \mathcal{I}$, then v must have been an argument of the function.

2.2.3 Evaluation of built-in functions

2.2.3.1 Compatibility of inputs and signature entries

The previous section describes how a built-in function is interpreted as a mathematical function. When a Plutus Core built-in function b is applied to a sequence of arguments, the arguments must have types which are compatible with the signature of b; for example, if b has signature $[\forall a_\#, \forall b_\#, a_\#, b_\#, a_\#, c_*, c_*] \to c_*$ and b is applied to a sequence of inputs V_1, V_2, V_3, V_4, V_5 then V_1, V_2, and V_3 must all be constants of some monomorphic built-in types and the types of V_1 and V_3 must be the same; V_4 and V_5 can be arbitrary inputs. This section describes the conditions for type compatibility.

In detail, given a reduced arity $\bar{\alpha} = [\tau_1, \dots, \tau_n]$, a sequence $\bar{V} = [V_1, \dots, V_m]$, and a type assignment S we say that $\bar{V}$ is *compatible with* $\bar{\alpha}$ (*via* S) if and only if $n = m$ and, letting $I = \{i : 1 \le i \le n, \tau_i \in \mathcal{U}_\#\}$ (so $\tau_j \in \mathcal{V}_*$ if $j \notin I$), there exist type assignments S_i ($1 \le i \le n$) such that all of the following are satisfied

- For all $i \in I$ there exists $T_i \in \mathcal{U}$ such that $V_i \in \mathcal{C}_{T_i}$ and $T_i \preceq_{S_i} \tau_i$.
- $\{S_i : i \in I\}$ is consistent (see Section 2.2.1.3).
- $S = \bigcup\{S_i : i \in I\}$.

If these conditions are all satisfied then we can find suitable S_i using the procedure described in Section 2.2.1.3 and this allows us to construct S explicitly since the S_i are consistent. Note that in this case $\mathrm{dom}\, S = \mathrm{dom}\, S_1 \cup \ldots \cup \mathrm{dom}\, S_n = \mathsf{FV}_\#(\tau_1) \cup \cdots \cup \mathsf{FV}_\#(\tau_n) = \mathsf{FV}_\#(\alpha)$, so S is minimal in the sense that no S' with $\mathrm{dom}\, S'$ strictly smaller than $\mathrm{dom}\, S$ is sufficient to monomorphise all of the τ_i simultaneously. We write

$$[V_1, \dots, V_m] \approx_S [\tau_1, \dots, \tau_n]$$

in this case. If $\bar{V}$ is not compatible with $\bar{\alpha}$ then we write $\bar{V} \not\approx \bar{\alpha}$.

2.2.3.2 Evaluation

For later use we define a function Eval which attempts to evaluate an application of a built-in function b to a sequence of inputs $[V_1, \ldots, V_m]$. This fails if the number of inputs is incorrect or if the inputs are not compatible with $\bar{\alpha}(b)$:

$$\mathsf{Eval}(b, [V_1, \ldots, V_n]) = \mathsf{X} \quad \text{if } [V_1, \ldots, V_n] \not\approx \bar{\alpha}(b).$$

Otherwise, the conditions for the existence of a denotation of b are met and we can apply that denotation to the denotations of the inputs and then reify the result. If $[V_1, \ldots, V_n] \approx_S \bar{\alpha}(b) = [\tau_1, \ldots, \tau_n]$, let $T_i = \hat{S}(\tau_i)$ for $1 \leq i \leq n$; then we define

$$\mathsf{Eval}(b, [V_1, \ldots, V_n]) = \{\!|\, [\![b]\!]_S([\![V_1]\!]_{T_1}, \ldots, [\![V_n]\!]_{T_n}) \,|\!\}.$$

It can be checked that the compatibility condition guarantees that this makes sense according to the definition of $[\![b]\!]_S$ in Section 2.2.2.3.

Notes.

- All of the machinery which we have defined for built-in functions is parametric over the set $\mathcal{I}$ of inputs and the sets $\mathcal{C}_T \subseteq \mathcal{I}$ of constants. This also applies to the Eval function, so its meaning is not fully defined until we have given concrete definitions of the sets of inputs and constants.
- The error value X can occur in two different ways: either because the arguments are not compatible with the signature, or because the builtin itself returns X to signal some error condition.
- The symbol X is not part of Plutus Core; when we define reduction rules and evaluators for Plutus Core later some extra translation will be required to convert the result of Eval into something appropriate to the context.

2.3 Term reduction

This section defines the semantics of (untyped) Plutus Core.

2.3.1 Values in Plutus Core

The semantics of built-in functions in Plutus Core are obtained by instantiating the sets $\mathcal{C}_T$ of constants of type T (see Section 2.2.2.1) to be the expressions of the form `(con T c)` and the set $\mathcal{I}$ to be the set of Plutus Core *values*, terms which cannot immediately undergo any further reduction, such as lambda terms and delayed terms. Values also include partial applications of built-in functions such as `[(builtin modInteger) (con integer 5)]`, which cannot perform any computation until a second integer argument is supplied. However, partial applications must also be *well-formed*, in the sense that applications of `force` must be correctly interleaved with genuine arguments, and the arguments must themselves be values.

We define syntactic classes V of Plutus Core values and A of partial builtin applications simultaneously:

$$
\begin{array}{lrll}
\text{Value} & V & ::= & (\texttt{con}\ T\ c) \\
& & & (\texttt{delay}\ M) \\
& & & (\texttt{lam}\ x\ M) \\
& & & (\texttt{constr}\ i\ \overline{V}) \\
& & & A
\end{array}
$$

Figure 2.5: Values in Plutus Core

Here A is the class of well-formed partial applications, and to define this we first define a class of possibly ill-formed iterated applications B for each built-in function $b \in \mathcal{B}$:

$$
\begin{array}{lrll}
& B & ::= & (\texttt{builtin}\ b) \\
& & & [B\ V] \\
& & & (\texttt{force}\ B)
\end{array}
$$

Figure 2.6: Partial built-in function application

We let B denote the set of terms generated by the grammar in Figure 2.6 and we define a function β which extracts the name of the built-in function occurring in a term in B:

$$
\begin{array}{ll}
\beta((\texttt{builtin}\ b)) & = b \\
\beta([B\ V]) & = \beta(B) \\
\beta((\texttt{force}\ B)) & = \beta(B)
\end{array}
$$

We also define a function $\|\cdot\|$ which measures the size of a term $B \in \mathsf{B}$:

$$
\begin{array}{ll}
\|(\texttt{builtin}\ b)\| & = 0 \\
\|[B\ V]\| & = 1 + \|B\| \\
\|(\texttt{force}\ B)\| & = 1 + \|B\|
\end{array}
$$

Well-formed partial applications. A term $B \in \mathsf{B}$ is an application of $b = \beta(B)$ to a number of values in S, interleaved with applications of `force`. We now define what it means for B to be a *well-formed partial application*. Suppose that $\alpha(b) = [\iota_1, \ldots, \iota_n]$. Firstly we require that $\|B\| < n$, so that b is not fully applied; in this case we put $\iota = \iota_{\|B\|}$, the element of b's signature which describes what kind of "argument" b currently expects. The definition is completed by induction on the structure of B:

1. $B = (\texttt{builtin}\ b)$ is always well-formed.
2. $B = [B'\ V]$ is well-formed if B' is well-formed and $\iota \in \mathcal{U}_\#$ or $\iota \in \mathcal{V}_*$ (equivalently, $\iota \notin \mathcal{Q}$).
3. $B = (\texttt{force}\ B')$ is well-formed if B' is well-formed and $\iota \in \mathcal{Q}$.

The definition of values in Figure 2.5 is now completed by defining A to be the syntactic class of well-formed *partial* built-in function applications:

$$
A = \{B \in \mathsf{B} : B \text{ is a well-formed partial application}\}.
$$

Note that this definition does not impose any requirements of type correctness. For example, with the types and functions defined in Section 4.3.1 the term $X =$ `[(builtin modInteger) (con string "blue")]`

is a valid value which could be treated like any other, for instance by being passed as an argument to a `lam` expression. However, the evaluation rules described in the next section require that when a built-in function b becomes *fully* applied the types of the arguments are checked against the signature of b using the relation $\approx$ and the function Eval defined in Sections 2.2.3.1 and 2.2.3.2, so an error would arise if the term X were ever applied to another argument.

More notation. Suppose that A is a well-formed partial application with $\alpha(\beta(A)) = [\iota_1, \ldots, \iota_n]$. We define a function next which extracts the next argument (or `force`) expected by A:

$$\mathsf{next}(A) = \iota_{\|A\|+1}.$$

This makes sense because in a well-formed partial application A we have $\|A\| < n$.

We also define a function args which extracts the arguments which b has received so far in A:

$$\begin{array}{ll} \mathsf{args}(\texttt{(builtin } b\texttt{)}) & = [] \\ \mathsf{args}(\texttt{[}A\ V\texttt{]}) & = \mathsf{args}(A)\cdot V \\ \mathsf{args}(\texttt{(force } A\texttt{)}) & = \mathsf{args}(A). \end{array}$$

2.3.2 Term reduction

We define the semantics of Plutus Core using contextual semantics (or reduction semantics): see [31] or [29] or [32, 5.3], for example. We use A to denote a partial application of a built-in function as in Section 2.3.1 above. For builtin evaluation, we instantiate the set $\mathcal{I}$ of Section 2.2.2.1 to be the set of Plutus Core values. Thus all builtins take values as arguments and return a value or X. Since values are terms here, we can take $\{\!|V|\!\} = V$.

The notation $[V/x]M$ below denotes substitution of the value V for the variable x in M. This is *capture-avoiding* in that substitution is not performed on occurrences of x inside subterms of M of the form `(lam` x N`)`.

$$
\begin{array}{rrll}
\text{Frame} & f & ::= & \texttt{[\_}\ M\texttt{]} & \text{left application} \\
& & & \texttt{[}V\ \texttt{\_]} & \text{right application} \\
& & & \texttt{(force \_)} & \text{force} \\
& & & \texttt{(constr}\ i\ \overline{V}\ \texttt{\_}\ \overline{M}\texttt{)} & \text{constructor argument} \\
& & & \texttt{(case \_}\ \overline{M}\texttt{)} & \text{case scrutinee}
\end{array}
$$

(a) Grammar of reduction frames for Plutus Core

$\boxed{M \ \to \ M'}$

Term M reduces in one step to term M'.

$$\frac{}{\texttt{[(lam}\ x\ M\texttt{)}\ V\texttt{]} \ \to \ [V/x]M}$$

$$\frac{\ell(A) = \chi(\beta(A)) - 1 \qquad \mathsf{next}(A) \in \mathcal{U}_{\#} \cup \mathcal{V}_{*}}{\texttt{[}A\ V\texttt{]} \ \to \ \mathsf{Eval}'(\beta(A), \mathsf{args}(A) \cdot V)}$$

$$\frac{\ell(A) < \chi(\beta(A)) - 1 \qquad \mathsf{next}(A) \in \mathcal{U}_{\#} \cup \mathcal{V}_{*}}{\texttt{[}A\ V\texttt{]} \ \to \ \texttt{[}A\ V\texttt{]}}$$

$$\frac{}{\texttt{(force (delay}\ M\texttt{))} \ \to \ M}$$

$$\frac{0 \leq i \leq m}{\texttt{(case (constr}\ i\ \overline{V}\texttt{)}\ U_0 \dots U_m\texttt{)} \ \to \ \texttt{[}U_i\ \overline{V}\texttt{]}}$$

$$\frac{\ell(A) = \chi(\beta(A)) - 1 \qquad \mathsf{next}(A) \in \mathcal{Q}}{\texttt{(force}\ A\texttt{)} \ \to \ \mathsf{Eval}'(\beta(A), \mathsf{args}(A))}$$

$$\frac{\ell(A) < \chi(\beta(A)) - 1 \qquad \mathsf{next}(A) \in \mathcal{Q}}{\texttt{(force}\ A\texttt{)} \ \to \ A}$$

$$\frac{}{f\{\texttt{(error)}\} \ \to \ \texttt{(error)}}$$

$$\frac{M \ \to \ M'}{f\{M\} \ \to \ f\{M'\}}$$

(b) Reduction via contextual semantics

$$\mathsf{Eval}'(b, [V_1, \dots, V_n]) = \begin{cases} \texttt{(error)} & \text{if } \mathsf{Eval}(b, [V_1, \dots, V_n]) = \times \\ \mathsf{Eval}(b, [V_1, \dots, V_n]) & \text{otherwise} \end{cases}$$

(c) Built-in function application

Figure 2.7: Term reduction for Plutus Core

It can be shown that any closed Plutus Core term whose evaluation terminates yields either `(error)` or a value. Recall from Section 2.1.3 that we require the body of every Plutus Core program to be closed.

2.4 The CEK machine

This section contains a description of an abstract machine for efficiently executing Plutus Core. This is based on the CEK machine of Felleisen and Friedman [30].
The machine alternates between two main phases: the *compute* phase ($\triangleright$), where it recurses down the AST looking for values, saving surrounding contexts as frames (or *reduction contexts*) on a stack as it goes; and the *return* phase ($\triangleleft$), where it has obtained a value and pops a frame off the stack to tell it how to proceed next. In addition there is an error state $\blacklozenge$ which halts execution with an error, and a halting state $\square$ which halts execution and returns a value to the outside world.

To evaluate a program $(\texttt{program}\ v\ M)$, we first check that the version number v is valid, then start the machine in the state $[];[]\triangleright M$. It can be proved that the transitions in Figure 2.10 always preserve validity of states, so that the machine can never enter a state such as $[]\triangleleft M$ or $s, (\texttt{force}\ \_)\triangleleft(\texttt{lam}\ x\ A\ M)$ which isn't covered by the rules. If such a situation were to occur in an implementation then it would indicate that the machine was incorrectly implemented or that it was attempting to evaluate an ill-formed program (for example, one which attempts to apply a variable to some other term).

State	Σ	$::=$	$s;\rho\triangleright M \mid s\triangleleft V \mid \blacklozenge \mid \square V$
Stack	s	$::=$	f^*
CEK value	V	$::=$	$\langle\texttt{con}\ T\ c\rangle \mid \langle\texttt{delay}\ M\ \rho\rangle \mid \langle\texttt{lam}\ x\ M\ \rho\rangle$
			$\mid \langle\texttt{constr}\ i\ \overline{V}\rangle \mid \langle\texttt{builtin}\ b\ \overline{V}\ \eta\rangle$
Environment	ρ	$::=$	$[] \mid \rho[x\mapsto V]$
Expected builtin arguments	η	$::=$	$[\iota] \mid \iota\cdot\eta$

Figure 2.8: Grammar of CEK machine states for Plutus Core

Frame	f	$::=$	$(\texttt{force}\ \_)$	force
			$[\_\ (M,\rho)]$	left application to term
			$[\_\ V]$	left application to value
			$[V\ \_]$	right application of value
			$(\texttt{constr}\ i\ \overline{V}\ \_\ (\overline{M},\rho))$	constructor argument
			$(\texttt{case}\ \_\ (\overline{M},\rho))$	case scrutinee

Figure 2.9: Grammar of CEK stack frames

Figures 2.8 and 2.9 define some notation for *states* of the CEK machine: these involve a modified type of value adapted to the CEK machine, environments which bind names to values, and a stack which stores partially evaluated terms whose evaluation cannot proceed until some more computation has been performed (for example, since Plutus Core is a strict language function arguments have to be reduced to values before application takes place, and because of this a lambda term may have to be stored on the

stack while its argument is being reduced to a value). Environments are lists of the form $\rho = [x_1 \mapsto V_1, \ldots, x_n \mapsto V_n]$ which grow by having new entries appended on the right; we say that *x is bound in the environment ρ* if ρ contains an entry of the form $x \mapsto V$, and in that case we denote by $\rho[x]$ the value V in the rightmost (ie, most recent) such entry.*

To make the CEK machine fit into the built-in evaluation mechanism defined in Section 2.2 we define $\mathcal{I} = V$ and $\mathcal{C}_T = \{\langle \mathtt{con}\ T\ c \rangle : T \in \mathcal{U}, c \in [\![T]\!]\}$.

The rules in Figure 2.10 show the transitions of the machine; if any situation arises which is not included in these transitions (for example, if a frame $[\langle \mathtt{con}\ T\ c \rangle\ \_]$ is encountered or if an attempt is made to apply `force` to a partial builtin application which is expecting a term argument), then the machine stops immediately in an error state.

*The description of environments we use here is more general than necessary in that it permits a given variable to have multiple bindings; however, in what follows we never actually retrieve bindings other than the most recent one and we never remove bindings to expose earlier ones. The list-based definition has the merit of simplicity and suffices for specification purposes but in an implementation it would be safe to use some data structure where existing bindings of a given variable are discarded when a new binding is added.

$\boxed{\Sigma \mapsto \Sigma'}$

Machine takes one step from state Σ to state Σ'

$$
\begin{array}{rcl}
s;\rho \triangleright x & \mapsto & s \triangleleft \rho[x] \text{ if } x \text{ is bound in } \rho \\
s;\rho \triangleright (\mathtt{con}\ T\ c) & \mapsto & s \triangleleft \langle \mathtt{con}\ T\ c \rangle \\
s;\rho \triangleright (\mathtt{lam}\ x\ M) & \mapsto & s \triangleleft \langle \mathtt{lam}\ x\ M\ \rho \rangle \\
s;\rho \triangleright (\mathtt{delay}\ M) & \mapsto & s \triangleleft \langle \mathtt{delay}\ M\ \rho \rangle \\
s;\rho \triangleright (\mathtt{force}\ M) & \mapsto & (\mathtt{force}\ \_) \cdot s;\rho \triangleright M \\
s;\rho \triangleright [M\ N] & \mapsto & [\_\ (N,\rho)] \cdot s;\rho \triangleright M \\
s;\rho \triangleright (\mathtt{constr}\ i\ M \cdot \overline{M}) & \mapsto & (\mathtt{constr}\ i\ \_\ (\overline{M},\rho)) \cdot s;\rho \triangleright M \\
s;\rho \triangleright (\mathtt{constr}\ i\ []) & \mapsto & s \triangleleft \langle \mathtt{constr}\ i\ [] \rangle \\
s;\rho \triangleright (\mathtt{case}\ N\ \overline{M}) & \mapsto & (\mathtt{case}\ \_\ (\overline{M},\rho)) \cdot s;\rho \triangleright N \\
s;\rho \triangleright (\mathtt{builtin}\ b) & \mapsto & s \triangleleft \langle \mathtt{builtin}\ b\ []\ \alpha(b) \rangle \\
s;\rho \triangleright (\mathtt{error}) & \mapsto & \blacklozenge \\
\\
[] \triangleleft V & \mapsto & \square V \\
[\_\ (M,\rho)] \cdot s \triangleleft V & \mapsto & [V\ \_] \cdot s;\rho \triangleright M \\
[\langle \mathtt{lam}\ x\ M\ \rho \rangle\ \_] \cdot s \triangleleft V & \mapsto & s;\rho[x \mapsto V] \triangleright M \\
[\_\ V] \cdot s \triangleleft \langle \mathtt{lam}\ x\ M\ \rho \rangle & \mapsto & s;\rho[x \mapsto V] \triangleright M \\
[\langle \mathtt{builtin}\ b\ \overline{V}\ (\iota \cdot \eta) \rangle\ \_] \cdot s \triangleleft V & \mapsto & s \triangleleft \langle \mathtt{builtin}\ b\ (\overline{V} \cdot V)\ \eta \rangle \text{ if } \iota \in \mathcal{U}_\# \cup \mathcal{V}_* \\
[\_\ V] \cdot s \triangleleft \langle \mathtt{builtin}\ b\ \overline{V}\ (\iota \cdot \eta) \rangle & \mapsto & s \triangleleft \langle \mathtt{builtin}\ b\ (\overline{V} \cdot V)\ \eta \rangle \text{ if } \iota \in \mathcal{U}_\# \cup \mathcal{V}_* \\
[\langle \mathtt{builtin}\ b\ \overline{V}\ [\iota] \rangle\ \_] \cdot s \triangleleft V & \mapsto & \mathsf{Eval}_{\mathsf{CEK}}(s, b, \overline{V} \cdot V) \text{ if } \iota \in \mathcal{U}_\# \cup \mathcal{V}_* \\
[\_\ V] \cdot s \triangleleft \langle \mathtt{builtin}\ b\ \overline{V}\ [\iota] \rangle & \mapsto & \mathsf{Eval}_{\mathsf{CEK}}(s, b, \overline{V} \cdot V) \text{ if } \iota \in \mathcal{U}_\# \cup \mathcal{V}_* \\
(\mathtt{force}\ \_) \cdot s \triangleleft \langle \mathtt{delay}\ M\ \rho \rangle & \mapsto & s;\rho \triangleright M \\
(\mathtt{force}\ \_) \cdot s \triangleleft \langle \mathtt{builtin}\ b\ \overline{V}\ (\iota \cdot \eta) \rangle & \mapsto & s \triangleleft \langle \mathtt{builtin}\ b\ \overline{V}\ \eta \rangle \text{ if } \iota \in \mathcal{Q} \\
(\mathtt{force}\ \_) \cdot s \triangleleft \langle \mathtt{builtin}\ b\ \overline{V}\ [\iota] \rangle & \mapsto & \mathsf{Eval}_{\mathsf{CEK}}(s, b, \overline{V}) \text{ if } \iota \in \mathcal{Q} \\
(\mathtt{constr}\ i\ \overline{V}\ \_\ (M \cdot \overline{M}, \rho)) \cdot s \triangleleft V & \mapsto & (\mathtt{constr}\ i\ \overline{V} \cdot V\ \_\ (\overline{M}, \rho)) \cdot s;\rho \triangleright M \\
(\mathtt{constr}\ i\ \overline{V}\ \_\ ([], \rho)) \cdot s \triangleleft V & \mapsto & s \triangleleft \langle \mathtt{constr}\ i\ \overline{V} \cdot V \rangle \\
(\mathtt{case}\ \_\ (M_0 \dots M_n, \rho)) \cdot s \triangleleft \langle \mathtt{constr}\ i\ V_0 \dots V_m \rangle & \mapsto & [\_\ V_m] \cdot \dots \cdot [\_\ V_0] \cdot s;\rho \triangleright M_i \text{ if } 0 \le i \le n
\end{array}
$$

(a) CEK machine transitions for Plutus Core

$$
\mathsf{Eval}_{\mathsf{CEK}}(s, b, [V_1, \dots, V_n]) = \begin{cases} \blacklozenge & \text{if } \mathsf{Eval}(b, [V_1, \dots, V_n]) = \times \\ s \triangleleft \mathsf{Eval}(b, [V_1, \dots, V_n]) & \text{otherwise} \end{cases}
$$

(b) Evaluation of built-in functions

Figure 2.10: A CEK machine for Plutus Core

2.4.1 Converting CEK evaluation results into Plutus Core terms

The purpose of the CEK machine is to evaluate Plutus Core terms, but in the definition in Figure 2.10 it does not return a Plutus Core term; instead the machine can halt in two different ways:

- The machine can halt in the state $\square\, V$ for some CEK value V.
- The machine can halt in the state $\blacklozenge$.

To get a complete evaluation strategy for Plutus Core we must convert these states into Plutus Core terms. The term corresponding to $\blacklozenge$ is `(error)`, and to obtain a term from $\square\, V$ we perform a process which we refer to as *discharging* the CEK value V (also known as *unloading*: see [38, pp. 129–130], [28, pp. 71ff]). This process substitutes bindings in environments for variables occurring in the value V to obtain a term $\mathcal{U}(V)$: see Figure 2.11a. Since environments contain bindings $x \mapsto W$ of variables to further CEK values, we have to recursively discharge those bindings first before substituting: see Figure 2.11b, which defines an operation $@\rho$ which does this. As before $[N/x]M$ denotes the usual (capture-avoiding) process of substituting the term N for all unbound occurrences of the variable x in the term M. Note that in Figure 2.11b we substitute the rightmost (ie, the most recent) bindings in the environment first.

$$
\begin{aligned}
\mathcal{U}(\langle \texttt{con}\ T\ c \rangle) &= \texttt{(con}\ T\ c\texttt{)} \\
\mathcal{U}(\langle \texttt{delay}\ M\ \rho \rangle) &= \texttt{(delay}\ M\texttt{)}\,@\rho \\
\mathcal{U}(\langle \texttt{lam}\ x\ M\ \rho \rangle) &= \texttt{(lam}\ x\ M\texttt{)}\,@\rho \\
\mathcal{U}(\langle \texttt{constr}\ i\ \overline{V} \rangle) &= \texttt{(constr}\ i\ \overline{\mathcal{U}(V)}\texttt{)} \\
\mathcal{U}(\langle \texttt{builtin}\ b\ V_1 V_2 \dots V_k\ \eta \rangle) &= \texttt{[}\dots\ \texttt{[[(builtin}\ b\texttt{)}\ (\mathcal{U}(V_1))\texttt{]}\ (\mathcal{U}(V_2))\texttt{]}\dots(\mathcal{U}(V_k))\texttt{]}
\end{aligned}
$$

(a) Discharging CEK values

$$
M@\rho = [(\mathcal{U}(V_1))/x_1] \cdots [(\mathcal{U}(V_n))/x_n]M \quad \text{if } \rho = [x_1 \mapsto V_1, \dots, x_n \mapsto V_n]
$$

(b) Iterated substitution/discharging

Figure 2.11: Discharging CEK values to obtain Plutus Core terms

We can prove that if we evaluate a closed Plutus Core term in the CEK machine and then convert the result back to a term using the above procedure then we get the result that we should get according to the semantics in Figure 2.7.

2.5 Cost accounting for Untyped Plutus Core

To follow.

Chapter 3

Typed Plutus Core

To follow.

Chapter 4

Plutus Core on Cardano

4.1 Protocol versions

The Cardano blockchain controls the introduction of features through the use of *protocol versions*, a field in the protocol parameters. The major protocol version is used to indicate when forwards-incompatible changes (i.e. those that allow blocks that were not previously allowed) are made to the rules of the chain. This is a hard fork of the chain.

The protocol version is part of the history of the chain, as are all protocol parameters. That means that all blocks are associated with the protocol version from when they were created, so that they can be interpreted correctly.

In summary, conditioning on the protocol version is the main way in which we can introduce changes in behaviour.

Table 4.1 lists the protocol versions that are relevant to the use of Plutus Core on Cardano.

Protocol version	Codename	Date
5.0	Alonzo	September 2021
7.0	Vasil	June 2022
8.0	Valentine	February 2023
9.0	Conway	September 2024

Table 4.1: Protocol versions

4.2 Ledger languages

The Cardano ledger uses Plutus Core as the programming language for *scripts*. The ledger in fact supports multiple different interpretations for scripts, and so each script is tagged with a *ledger language* that tells the ledger how to interpret it. Since the ledger must always be able to evaluate old scripts and get the same answer, the ledger language must pin down everything about how the script is evaluated, including:

1. How to interpret the script itself (e.g. as a Plutus Core program, what versions of the Plutus Core language are allowable)

2. Other configuration the script may need in order to run (e.g. the set of builtin types and functions and their interpretations, cost model parameters)

3. How the script is invoked (e.g. after having certain arguments passed to it)

There are currently three "Plutus" ledger languages (i.e. ledger languages whose underlying programming language is Plutus Core) in use on Cardano:*

1. `PlutusV1`
2. `PlutusV2`
3. `PlutusV3`

Table 4.2 shows when each Plutus ledger language was introduced. Ledger languages remain available permanently after they have been introduced.

Protocol version	Ledger language introduced
5.0	`PlutusV1`
7.0	`PlutusV2`
9.0	`PlutusV3`

Table 4.2: Introduction of Plutus ledger languages

Ledger languages can evolve over time. We can make backwards-compatible changes when the major protocol version changes, but backwards-incompatible changes can only be introduced by creating a whole new ledger language.† This means that to fully explain the behaviour of a ledger language we may need to also index by the protocol version.

The following tables show how Plutus ledger languages determine:

- Which Plutus Core language versions are allowable (Table 4.3)
- Which built-in functions and types are available (Table 4.4, given in terms of batches, see Section 4.3)
- How to interpret the built-in functions and types (Table 4.5, given in terms of built-in semantics variants, see Section 4.3)

Currently, once we add a feature for any given protocol version/ledger language, we also make it available for all subsequent protocol versions/ledger languages. For example, Batch 2 of builtins was introduced in `PlutusV2` at protocol version 7.0, so it is also available in `PlutusV2` at protocol versions after 7.0, and `PlutusV3` at protocol versions after 9.0 (when `PlutusV3` itself was first introduced). Hence the tables are simplified to only show when something is *introduced*.

Ledger language	Protocol version	Plutus Core language version introduced
`PlutusV1`	5.0	1.0.0
`PlutusV3`	9.0	1.1.0

Table 4.3: Introduction of Plutus Core language versions

*Note that ledger languages are completely distinct from the point of view of the ledger, the "V1"/"V2" naming is suggestive of the fact that these two ledger languages are related, but in the implementation they are completely independent.

†See [1] for more details on how the process of evolution works.

Ledger language	Protocol version	Built-in functions and types introduced
`PlutusV1`	5.0	Batch 1
`PlutusV2`	7.0	Batch 2
`PlutusV2`	8.0	Batch 3
`PlutusV3`	9.0	Batch 4
`PlutusV3`	10.0	Batch 5

Table 4.4: Introduction of built-in functions and types

Ledger language	Built-in semantics variant used
`PlutusV1`	Built-in semantics 1
`PlutusV3`	Built-in semantics 2

Table 4.5: Selection of built-in semantics variant

4.3 Built-in types and functions

Built-in batches. The built-in types and functions are defined in batches corresponding to how they were added to ledger languages. These batches are given in the following sections.

Built-in semantics variants. In rare cases we can make a mistake or need to change the actual behaviour of a built-in function. To handle this we define a series of built-in semantics variants, which indicate which behaviour should be used. A fix will typically be deployed by defining a new semantics variant, and then using that variant for future ledger languages (but not existing ones, since this is usually a backwards-incompatible change).

Changes are listed alongside the original definition of the built-in function in its original batch, and are indexed in the following table.

Built-in semantics variant	Changes from previous semantics
Built-in semantics 1	None
Built-in semantics 2	`consByteString` (See 2)

Table 4.6: Built-in semantics variants

Concrete syntax for built-in types. Recall that in the abstract notation for built-in types introduced in Section 2.2.1, a built-in type is either an *atomic type* such as `integer` or `string` or an application $op(T_1, \ldots, T_n)$ of a *type operator* to a sequence of built-in types. The concrete syntax of built-in types used in textual Plutus Core programs is slightly different in that we use a curried form of application for type operators: a type is given by

$$\begin{array}{lcll} \mathbf{T} & ::= & \textit{atomic-type} & \text{Atomic type} \\ & & (\textit{op}\ \mathbf{T}_1 \ldots \mathbf{T}_{|op|}) & \text{Type application} \end{array}$$

Note that we again require that all type operators are fully applied. We refer to the syntactic objects $\mathbf{T}$ above as *concrete built-in types*. There is an obvious bijection between these and the abstract built-in types

used elsewhere in this document, and given an abstract built-in type T we will denote the corresponding concrete built-in type by $\bar{T}$.

Concrete syntax for built-in constants. We provide concrete syntax for constants of most (but not all) built-in types. For a built-in type T which has a concrete syntax we specify a set $\mathbf{C}_T$ of strings (using either regular expressions or a BNF-style grammar), and a constant of type T is then represented in the concrete syntax by an expression of the form `(con` $\bar{T}$ c_T`)` with $c_T \in \mathbf{C}_T$. Each string c_T will have an interpretation as a value of type T (ie, an element of $[\![T]\!]$) and since this will generally be the obvious interpretation we will not always spell out the details.

4.3.1 Batch 1

4.3.1.1 Built-in types and type operators

The first batch of built-in types and type operators is defined in Tables 4.7 and 4.8. We also include concrete syntax for these; the concrete syntax is not strictly part of the language, but may be useful for tools working with Plutus Core.

Type	Denotation	Concrete Syntax
`integer`	$\mathbb{Z}$	`-?[0-9]+`
`bytestring`	$\mathbb{B}^*$, the set of sequences of bytes or 8-bit characters.	`#([0-9A-Fa-f][0-9A-Fa-f])*`
`string`	$\mathbb{U}^*$, the set of sequences of Unicode characters.	See note below
`bool`	{true, false}	`True | False`
`unit`	{()}	`()`
`data`	See below	See below

Table 4.7: Atomic types, batch 1

Operator *op*	$\|op\|$	Denotation	Concrete Syntax
`list`	1	$[\![\texttt{list}\,(t)]\!] = [\![t]\!]^*$	See below
`pair`	2	$[\![\texttt{pair}\,(t_1, t_2)]\!] = [\![t_1]\!] \times [\![t_2]\!]$	See below

Table 4.8: Type operators, batch 1

Concrete syntax for strings. Strings are represented as sequences of Unicode characters enclosed in double quotes, and may include standard escape sequences. Surrogate characters in the range `U+D800`–`U+DFFF` are replaced with the Unicode replacement character `U+FFFD`.

Concrete syntax for lists and pairs. A list of type `list`(t) is written as a syntactic list `[`$c_1, \ldots, c_n$`]` where each c_i lies in $\mathbf{C}_t$; a pair of type `pair`(t_1, t_2) is written as a syntactic pair `(`c_1, c_2`)` with $c_1 \in \mathbf{C}_{t_1}$ and $c_2 \in \mathbf{C}_{t_2}$. Some valid constant expressions are thus

```
(con (list integer) [11, 22, 33])
(con (pair bool string) (True, "Plutus")).
(con (list (pair bool (list bytestring)))
   [(True, []), (False, [#,#1F]), (True, [#123456, #AB, #ef2804])])
```

The `data` type. We provide a built-in type `data` which permits the encoding of simple data structures for use as arguments to Plutus Core scripts. This type is defined in Haskell as

```
data Data =
   Constr Integer [Data]
   | Map [(Data, Data)]
   | List [Data]
   | I Integer
   | B ByteString
```

In set-theoretic terms the denotation of `data` is defined to be the least fixed point of the endofunctor F on the category of sets given by $F(X) = (\llbracket\texttt{integer}\rrbracket \times X^*) \uplus (X \times X)^* \uplus X^* \uplus \llbracket\texttt{integer}\rrbracket \uplus \llbracket\texttt{bytestring}\rrbracket$, so that

$$\llbracket\texttt{data}\rrbracket = (\llbracket\texttt{integer}\rrbracket \times \llbracket\texttt{data}\rrbracket^*) \uplus (\llbracket\texttt{data}\rrbracket \times \llbracket\texttt{data}\rrbracket)^* \uplus \llbracket\texttt{data}\rrbracket^* \uplus \llbracket\texttt{integer}\rrbracket \uplus \llbracket\texttt{bytestring}\rrbracket.$$

We have injections

$$\begin{aligned}
\mathrm{inj}_C &: \llbracket\texttt{integer}\rrbracket \times \llbracket\texttt{data}\rrbracket^* \to \llbracket\texttt{data}\rrbracket \\
\mathrm{inj}_M &: \llbracket\texttt{data}\rrbracket \times \llbracket\texttt{data}\rrbracket^* \to \llbracket\texttt{data}\rrbracket \\
\mathrm{inj}_L &: \llbracket\texttt{data}\rrbracket^* \to \llbracket\texttt{data}\rrbracket \\
\mathrm{inj}_I &: \llbracket\texttt{integer}\rrbracket \to \llbracket\texttt{data}\rrbracket \\
inj_B &: \llbracket\texttt{bytestring}\rrbracket \to \llbracket\texttt{data}\rrbracket
\end{aligned}$$

and projections

$$\begin{aligned}
\mathrm{proj}_C &: \llbracket\texttt{data}\rrbracket \to (\llbracket\texttt{integer}\rrbracket \times \llbracket\texttt{data}\rrbracket^*)_\times \\
\mathrm{proj}_M &: \llbracket\texttt{data}\rrbracket \to (\llbracket\texttt{data}\rrbracket \times \llbracket\texttt{data}\rrbracket^*)_\times \\
\mathrm{proj}_L &: \llbracket\texttt{data}\rrbracket \to \llbracket\texttt{data}\rrbracket^*_\times \\
\mathrm{proj}_I &: \llbracket\texttt{data}\rrbracket \to \llbracket\texttt{integer}\rrbracket_\times \\
\mathrm{proj}_B &: \llbracket\texttt{data}\rrbracket \to \llbracket\texttt{bytestring}\rrbracket_\times
\end{aligned}$$

which extract an object of the relevant type from a `data` object D, returning $\times$ if D does not lie in the expected component of the disjoint union; also there are functions

$$\mathrm{is}_C, \mathrm{is}_M, \mathrm{is}_L, \mathrm{is}_I, \mathrm{is}_B : \llbracket\texttt{data}\rrbracket \to \llbracket\texttt{bool}\rrbracket$$

which determine whether a `data` value lies in the relevant component.

Note: `Constr` tag values. The `Constr` constructor of the `data` type is intended to represent values from algebraic data types (also known as sum types and discriminated unions, among other things; `data` itself is an example of such a type), where `Constr` $i\,[d_1, \ldots, d_n]$ represents a tuple of data items together with a tag i indicating which of a number of alternatives the data belongs to. The definition above allows tags to be any integer value, but because of restrictions in the serialisation format for `data` (see Section B.7) we recommend that in practice **only tags i with $0 \leq i \leq 2^{64} - 1$ should be used**: deserialisation will fail for `data` items (and programs which include such items) involving tags outside this range.

Note also that `Constr` is unrelated to the `constr` term in Plutus Core itself. Both provide ways of representing structured data, but the former is part of a built-in type whereas the latter is part of the language itself.

Concrete syntax for `data`. The concrete syntax for `data` is given by

$$\begin{aligned} c_{\texttt{data}} \quad ::= \quad & (\texttt{Constr}\ c_{\texttt{integer}}\ c_{\texttt{list(data)}}) \\ & (\texttt{Map}\ c_{\texttt{list(pair(data,data))}}) \\ & (\texttt{List}\ c_{\texttt{list(data)}}) \\ & (\texttt{I}\ c_{\texttt{integer}}) \\ & (\texttt{B}\ c_{\texttt{bytestring}}). \end{aligned}$$

We interpret these syntactic constants as elements of $[\![\texttt{data}]\!]$ using the various 'inj' functions defined earlier. Some valid `data` constants are

```
(con data (Constr 1 [(I 2), (B #), (Map [])])
(con data (Map [((I 0), (B #00)), ((I 1), (B #0F))]))
(con data (List [(I 0), (I 1), (B #7FFF), (List [])]))
(con data (I -22))
(con data (B #001A)).
```

Note. At the time of writing the syntax accepted by IOG's parser for textual Plutus Core differs slightly from that above in that subobjects of `Constr`, `Map` and `List` objects must *not* be parenthesised: for example one must write `(con data (Constr 1 [I 2, B #,Map []])`. This discrepancy will be resolved in the near future.

4.3.1.2 Built-in functions

The first batch of built-in functions is shown in Table 4.9. The table indicates which functions can fail during execution, and conditions causing failure are specified either in the denotation given in the table or in a relevant note. Recall also that a built-in function will fail if it is given an argument of the wrong type: this is checked in conditions involving the $\sim$ relation and the Eval function in Figures 2.7 and 2.10. Note also that some of the functions are #-polymorphic. According to Section 2.2.2.3 we require a denotation for every possible monomorphisation of these; however all of these functions are parametrically polymorphic so to simplify notation we have given a single denotation for each of them with an implicit assumption that it applies at each possible monomorphisation in an obvious way.

Function	Signature	Denotation	Can fail?	Note
`addInteger`	`[integer, integer]` → `integer`	+	No	
`subtractInteger`	`[integer, integer]` → `integer`	−	No	
`multiplyInteger`	`[integer, integer]` → `integer`	×	No	
`divideInteger`	`[integer, integer]` → `integer`	div	Yes	1
`modInteger`	`[integer, integer]` → `integer`	mod	Yes	1
`quotientInteger`	`[integer, integer]` → `integer`	quot	Yes	1
`remainderInteger`	`[integer, integer]` → `integer`	rem	Yes	1
`equalsInteger`	`[integer, integer]` → `bool`	=	No	
`lessThanInteger`	`[integer, integer]` → `bool`	<	No	

Table 4.9: Built-in functions, batch 1

Function	Type	Denotation	Can fail?	Note
`lessThanEqualsInteger`	`[integer, integer] → bool`	$\leq$	No	
`appendByteString`	`[bytestring, bytestring] → bytestring`	$([c_1, \ldots, c_m], [d_1, \ldots, d_n]) \mapsto [c_1, \ldots, c_m, d_1, \ldots, d_n]$	No	
`consByteString` (Variant 1)	`[integer, bytestring] → bytestring`	$(c, [c_1, \ldots, c_n]) \mapsto [\mathrm{mod}(c, 256), c_1, \ldots, c_n]$	No	2
`consByteString` (Variant 2)	`[integer, bytestring] → bytestring`	$(c, [c_1, \ldots, c_n]) \mapsto \begin{cases} [c, c_1, \ldots, c_n] & \text{if } 0 \leq c \leq 255 \\ \times & \text{otherwise} \end{cases}$	Yes	2
`sliceByteString`	`[integer, integer, bytestring] → bytestring`	$(s, k, [c_0, \ldots, c_n]) \mapsto [c_{\max(s,0)}, \ldots, c_{\min(s+k-1, n-1)}]$	No	3
`lengthOfByteString`	`[bytestring] → integer`	$[] \mapsto 0, [c_1, \ldots, c_n] \mapsto n$	No	
`indexByteString`	`[bytestring, integer] → integer`	$([c_0, \ldots, c_{n-1}], j) \mapsto \begin{cases} c_i & \text{if } 0 \leq j \leq n-1 \\ \times & \text{otherwise} \end{cases}$	Yes	
`equalsByteString`	`[bytestring, bytestring] → bool`	$=$	No	4
`lessThanByteString`	`[bytestring, bytestring] → bool`	$<$	No	4
`lessThanEqualsByteString`	`[bytestring, bytestring] → bool`	$\leq$	No	4
`appendString`	`[string, string] → string`	$([u_1, \ldots, u_m], [v_1, \ldots, v_n]) \mapsto [u_1, \ldots, u_m, v_1, \ldots, v_n]$	No	
`equalsString`	`[string, string] → bool`	$=$	No	
`encodeUtf8`	`[string] → bytestring`	$\mathsf{utf8}$		5
`decodeUtf8`	`[bytestring] → string`	$\mathsf{utf8}^{-1}$	Yes	5
`sha2_256`	`[bytestring] → bytestring`	Hash a `bytestring` using `SHA-256` [23].	No	
`sha3_256`	`[bytestring] → bytestring`	Hash a `bytestring` using `SHA3-256` [26].	No	
`blake2b_256`	`[bytestring] → bytestring`	Hash a `bytestring` using `Blake2b-256` [40].	No	
`verifyEd25519Signature`	`[bytestring, bytestring, bytestring] → bool`	Verify an `Ed25519` digital signature.	Yes	6, 7
`ifThenElse`	$[\forall a_*, \mathtt{bool}, a_*, a_*] \to a_*$	$(\mathsf{true}, t_1, t_2) \mapsto t_1$ $(\mathsf{false}, t_1, t_2) \mapsto t_2$	No	
`chooseUnit`	$[\forall a_*, \mathtt{unit}, a_*] \to a_*$	$((), t) \mapsto t$	No	
`trace`	$[\forall a_*, \mathtt{string}, a_*] \to a_*$	$(s, t) \mapsto t$	No	8
`fstPair`	$[\forall a_\#, \forall b_\#, \mathtt{pair}\,(a_\#, b_\#)] \to a_\#$	$(x, y) \mapsto x$	No	
`sndPair`	$[\forall a_\#, \forall b_\#, \mathtt{pair}\,(a_\#, b_\#)] \to b_\#$	$(x, y) \mapsto y$	No	

Table 4.9: Built-in functions, batch 1

Function	Type	Denotation	Can fail?	Note
`chooseList`	$[\forall a_\#, \forall b_*, \texttt{list}\,(a_\#), b_*, b_*] \to b_*$	$([], t_1, t_2) \mapsto t_1$, $([x_1, \ldots, x_n], t_1, t_2) \mapsto t_2$ $(n \geq 1)$.	No	
`mkCons`	$[\forall a_\#, a_\#, \texttt{list}\,(a_\#)] \to \texttt{list}\,(a_\#)$	$(x, [x_1, \ldots, x_n]) \mapsto [x, x_1, \ldots, x_n]$	No	
`headList`	$[\forall a_\#, \texttt{list}\,(a_\#)] \to a_\#$	$[] \mapsto \times, [x_1, x_2, \ldots, x_n] \mapsto x_1$	Yes	
`tailList`	$[\forall a_\#, \texttt{list}\,(a_\#)] \to \texttt{list}\,(a_\#)$	$[] \mapsto \times$, $[x_1, x_2, \ldots, x_n] \mapsto [x_2, \ldots, x_n]$	Yes	
`nullList`	$[\forall a_\#, \texttt{list}\,(a_\#)] \to \texttt{bool}$	$[] \mapsto \mathsf{true}, [x_1, \ldots, x_n] \mapsto \mathsf{false}$	No	
`chooseData`	$[\forall a_*, \texttt{data}, a_*, a_*, a_*, a_*, a_*] \to a_*$	$(d, t_C, t_M, t_L, t_I, t_B) \mapsto \begin{cases} t_C & \text{if } \mathrm{is}_C(d) \\ t_M & \text{if } \mathrm{is}_M(d) \\ t_L & \text{if } \mathrm{is}_L(d) \\ t_I & \text{if } \mathrm{is}_I(d) \\ t_B & \text{if } \mathrm{is}_B(d) \end{cases}$	No	
`constrData`	$[\texttt{integer}, \texttt{list}\,(\texttt{data})] \to \texttt{data}$	inj_C	No	
`mapData`	$[\texttt{list}\,(\texttt{pair}\,(\texttt{data}, \texttt{data})) \to \texttt{data}$	inj_M	No	
`listData`	$[\texttt{list}\,(\texttt{data})] \to \texttt{data}$	inj_L	No	
`iData`	$[\texttt{integer}] \to \texttt{data}$	inj_I	No	
`bData`	$[\texttt{bytestring}] \to \texttt{data}$	inj_B	No	
`unConstrData`	$[\texttt{data}] \to \texttt{pair}\,(\texttt{integer}, \texttt{list}\,(\texttt{data}))$	proj_C	Yes	
`unMapData`	$[\texttt{data}] \to \texttt{list}\,(\texttt{pair}\,(\texttt{data}, \texttt{data}))$	proj_M	Yes	
`unListData`	$[\texttt{data}] \to \texttt{list}\,(\texttt{data})$	proj_L	Yes	
`unIData`	$[\texttt{data}] \to \texttt{integer}$	proj_I	Yes	
`unBData`	$[\texttt{data}] \to \texttt{bytestring}$	proj_B	Yes	
`equalsData`	$[\texttt{data}, \texttt{data}] \to \texttt{bool}$	$=$		
`mkPairData`	$[\texttt{data}, \texttt{data}] \to \texttt{pair}\,(\texttt{data}, \texttt{data})$	$(x, y) \mapsto (x, y)$	No	
`mkNilData`	$[\texttt{unit}] \to \texttt{list}\,(\texttt{data})$	$() \mapsto []$	No	
`mkNilPairData`	$[\texttt{unit}] \to \texttt{list}\,(\texttt{pair}\,(\texttt{data}, \texttt{data}))$	$() \mapsto []$	No	

Table 4.9: Built-in functions, batch 1 (continued)

Note 1. Integer division functions. We provide four integer division functions: `divideInteger`, `modInteger`, `quotientInteger`, and `remainderInteger`, whose denotations are mathematical functions div, mod, quot, and rem which are modelled on the corresponding Haskell operations. Each of these takes two arguments and will fail (returning $\times$) if the second one is zero. For all $a, b \in \mathbb{Z}$ with $b \neq 0$ we have

$$\mathrm{div}(a, b) \times b + \mathrm{mod}(a, b) = a$$

$$|\operatorname{mod}(a,b)| < |b|$$

and

$$\operatorname{quot}(a,b) \times b + \operatorname{rem}(a,b) = a$$

$$|\operatorname{rem}(a,b)| < |b|.$$

The div and mod functions form a pair, as do quot and rem; div should not be used in combination with mod, not should quot be used with mod.

For positive divisors b, div truncates downwards and mod always returns a non-negative result ($0 \leq \operatorname{mod}(a,b) \leq b-1$). The quot function truncates towards zero. Table 4.10 shows how the signs of the outputs of the division functions depend on the signs of the inputs; + means ≥ 0 and − means ≤ 0, but recall that for $b = 0$ all of these functions return the error value ×.

a	b	div	mod	quot	rem
+	+	+	+	+	+
−	+	−	+	−	−
+	−	−	+	+	+
−	−	+	−	+	−

Table 4.10: Behaviour of integer division functions

Note 2. The `consByteString` function. In built-in semantics 1, the first argument of `consByteString` is an arbitrary integer which will be reduced modulo 256 before being prepended to the second argument. In built-in semantics 2 we require that the first argument lies between 0 and 255 (inclusive): in any other case an error will occur.

Note 3. The `sliceByteString` function. The application `[[(builtin sliceByteString) (con integer s)] (con integer k)] (con bytestring b)]` returns the substring of b of length k starting at position s; indexing is zero-based, so a call with $s = 0$ returns a substring starting with the first element of b, $s = 1$ returns a substring starting with the second, and so on. This function always succeeds, even if the arguments are out of range: if $b = [c_0, \ldots, c_{n-1}]$ then the application above returns the substring $[c_i, \ldots, c_j]$ where $i = \max(s, 0)$ and $j = \min(s+k-1, n-1)$; if $j < i$ then the empty string is returned.

Note 4. Comparisons of bytestrings. Bytestrings are ordered lexicographically in the usual way. If we have $a = [a_1, \ldots, a_m]$ and $b = [b_1, \ldots, b_n]$ then (recalling that if $m = 0$ then $a = []$, and similarly for b),

- $a = b$ if and only if $m = n$ and $a_i = b_i$ for $1 \leq i \leq m$.
- $a \leq b$ if and only if one of the following holds:
  - $a = []$
  - $m, n > 0$ and $a_1 < b_1$
  - $m, n > 0$ and $a_1 = b_1$ and $[a_2, \ldots, a_m] \leq [b_2, \ldots, b_n]$.
- $a < b$ if and only if $a \leq b$ and $a \neq b$.

For example, `#23456789` < `#24` and `#2345` < `#234500`. The empty bytestring is equal only to itself and is strictly less than all other bytestrings.

Note 5. Encoding and decoding bytestrings. The `encodeUtf8` and `decodeUtf8` functions convert between the `string` type and the `bytestring` type. We have defined ⟦`string`⟧ to consist of sequences

of Unicode characters without specifying any particular character representation, whereas $[\![\texttt{bytestring}]\!]$ consists of sequences of 8-bit bytes. We define the denotation of `encodeUtf8` to be the function

$$\mathsf{utf8} : \mathbb{U}^* \to \mathbb{B}^*$$

which converts sequences of Unicode characters to sequences of bytes using the well-known UTF-8 character encoding [44, Definition D92]. The denotation of `decodeUtf8` is the partial inverse function

$$\mathsf{utf8}^{-1} : \mathbb{B}^* \to \mathbb{U}^*_{\mathsf{x}}.$$

UTF-8 encodes Unicode characters encoded using between one and four bytes: thus in general neither function will preserve the length of an object. Moreover, not all sequences of bytes are valid representations of Unicode characters, and `decodeUtf8` will fail if it receives an invalid input (but `encodeUtf8` will always succeed).

Note 6. Digital signature verification functions. We use a uniform interface for digital signature verification algorithms. A digital signature verification function takes three bytestring arguments (in the given order):

- a public key *vk* (in this context *vk* is also known as a *verification key*)
- a message *m*
- a signature *s*.

A signature verification function may require one or more arguments to be well-formed in some sense (in particular an argument may need to be of a specified length), and in this case the function will fail (returning x) if any argument is malformed. If all of the arguments are well-formed then the verification function returns true if the private key corresponding to *vk* was used to sign the message *m* to produce *s*, otherwise it returns false.

Note 7. Ed25519 signature verification. The `verifyEd25519Signature` function performs cryptographic signature verification using the Ed25519 scheme [13, 34], and conforms to the interface described in Note 6. The arguments must have the following sizes:

- *vk*: 32 bytes
- *m*: unrestricted
- *s*: 64 bytes.

Note 8. The `trace` function. An application `[(builtin trace) s v]` (*s* a `string`, *v* any Plutus Core value) returns *v*. We do not specify the semantics any further. An implementation may choose to discard *s* or to perform some side-effect such as writing it to a terminal or log file.

4.3.2 Batch 2

4.3.2.1 Built-in functions

The second batch of built-in functions is defined in Table 4.11. See [2].

Function	Signature	Denotation	Can fail?	Note
`serialiseData`	`[data]` → `bytestring`	$\mathcal{E}_{\texttt{data}}$	No	1

Table 4.11: Built-in functions, batch 2

Note 1. Serialising `data` objects. The `serialiseData` function takes a `data` object and converts it into a bytestring using a CBOR encoding. A full specification of the encoding (including the definition of $\mathcal{E}_{\texttt{data}}$) is provided in Appendix B.

4.3.3 Batch 3

4.3.3.1 Built-in functions

The third batch of built-in functions is defined in Table 4.12. See [3].

Function	Signature	Denotation	Can fail?	Note
`verifyEcdsaSecp256k1Signature`	`[bytestring, bytestring, bytestring]` → `bool`	Verify an SECP-256k1 ECDSA signature	Yes	1
`verifySchnorrSecp256k1Signature`	`[bytestring, bytestring, bytestring]` → `bool`	Verify an SECP-256k1 Schnorr signature	Yes	2

Table 4.12: Built-in functions, batch 3

Note 1. Secp256k1 ECDSA Signature verification. The `verifyEcdsaSecp256k1Signature` function performs elliptic curve digital signature verification [9, 10, 33] over the `secp256k1` curve [20, §2.4.1] and conforms to the interface described in Note 6 of Section 4.3.1.2. The arguments must have the following sizes:

- *vk*: 33 bytes
- *m*: 32 bytes
- *s*: 64 bytes.

The public key *vk* is expected to be in the 33-byte compressed form described in [15]. Moreover, the ECDSA scheme admits two distinct valid signatures for a given message and private key, and we follow the restriction imposed by Bitcoin (see [36], `LOW_S`) and **only accept the smaller signature**; `verifyEcdsaSecp256k1Signature` will return `false` if the larger one is supplied.

Note 2. Secp256k1 Schnorr Signature verification. The `verifySchnorrSecp256k1Signature` function performs verification of Schnorr signatures [42, 35] over the `secp256k1` curve and conforms to the interface described in Note 6 of Section 4.3.1.2. The arguments are expected to be of the forms specified in BIP-340 [35] and thus should have the following sizes:

- *vk*: 32 bytes
- *m*: unrestricted
- *s*: 64 bytes.

4.3.4 Batch 4

The fourth batch of built-in types and functions adds support for

- The `Blake2b-224` and `Keccak-256` hash functions (see [4]).
- Conversion functions from integers to bytestrings and vice-versa (see [6]).
- BLS12-381 elliptic curve pairing operations (see [5], [19], [41, 4.2.1], [37]). For clarity these are described separately in Sections 4.3.4.2 and 4.3.4.3.

4.3.4.1 Miscellaneous built-in functions

Function	Signature	Denotation	Can fail?	Note
`blake2b_224`	`[bytestring] → bytestring`	Hash a `bytestring` using `Blake2b-224` [40]	No	
`keccak_256`	`[bytestring] → bytestring`	Hash a `bytestring` using `Keccak-256` [14]	No	
`integerToByteString`	`[bool, integer, integer] → bytestring`	$(e, w, n) \mapsto \begin{cases} \mathsf{itobs}_{\mathsf{LE}}(w, n) & \text{if } e = \mathsf{false} \\ \mathsf{itobs}_{\mathsf{BE}}(w, n) & \text{if } e = \mathsf{true} \end{cases}$	Yes	1
`byteStringToInteger`	`[bool, bytestring] → bytestring`	$(e, [c_0, \ldots, c_{N-1}]) \mapsto \begin{cases} \sum_{i=0}^{N-1} c_i 256^i & \text{if } e = \mathsf{false} \\ \sum_{i=0}^{N-1} c_i 256^{N-1-i} & \text{if } e = \mathsf{true} \end{cases}$	No	2

Table 4.13: Batch 4: miscellaneous built-in functions

Note 1. Integer to bytestring conversion. The `integerToByteString` function converts non-negative integers to bytestrings. It takes three arguments:

- A boolean endianness flag e.
- An integer width argument w with $0 \leq w < 8192$.
- The integer n to be converted: it is required that $0 \leq n < 256^{8192} = 2^{65536}$.

The conversion is little-endian (LE) if e is `(con bool False)` and big-endian (BE) if e is `(con bool True)`. If the width w is zero then the output is a bytestring which is just large enough to hold the converted integer. If $w > 0$ then the output is exactly w bytes long, and it is an error if n does not fit into a bytestring of that size; if necessary, the output is padded with `0x00` bytes (on the right in the little-endian case and the left in the big-endian case) to make it the correct length. For example, the five-byte little-endian representation of the integer `0x123456` is the bytestring `[0x56, 0x34, 0x12, 0x00, 0x00]` and the five-byte big-endian representation is `[0x00, 0x00, 0x12, 0x34, 0x56]`. In all cases an error occurs error if w or n lies outside the expected range, and in particular if n is negative.

The precise semantics of `integerToByteString` are given by the functions $\mathsf{itobs}_{\mathsf{LE}} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}^*_\times$ and $\mathsf{itobs}_{\mathsf{BE}} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}^*_\times$. Firstly we deal with out-of-range cases and the case $n = 0$:

$$\mathsf{itobs}_{\mathsf{LE}}(w, n) = \mathsf{itobs}_{\mathsf{BE}}(w, n) = \begin{cases} \times & \text{if } n < 0 \text{ or } n \geq 2^{65536} \\ \times & \text{if } w < 0 \text{ or } w > 8192 \\ [] & \text{if } n = 0 \text{ and } 0 \leq w \leq 8192 \end{cases}$$

Now assume that none of the conditions above hold, so $0 < n < 2^{65536}$ and $0 \leq w \leq 8192$. Since $n > 0$ it has a unique base-256 expansion of the form $n = \sum_{i=0}^{N-1} a_i 256^i$ with $N \geq 1$, $a_i \in \mathbb{B}$ for $0 \leq i \leq N-1$ and $a_{N-1} \neq 0$. We then have

$$\mathsf{itobs}_{\mathsf{LE}}(w, n) = \begin{cases} [a_0, \ldots, a_{N-1}] & \text{if } w = 0 \\ [b_0, \ldots, b_{w-1}] & \text{if } w > 0 \text{ and } N \leq w, \text{ where } b_i = \begin{cases} a_i & \text{if } i \leq N-1 \\ 0 & \text{if } i \geq N \end{cases} \\ \times & \text{if } w > 0 \text{ and } N > w \end{cases}$$

and

$$\mathsf{itobs}_{\mathsf{BE}}(w, n) = \begin{cases} [a_{N-1}, \ldots, a_0] & \text{if } w = 0 \\ [b_0, \ldots, b_{w-1}] & \text{if } w > 0 \text{ and } N \leq w, \text{ where } b_i = \begin{cases} 0 & \text{if } i \leq w-1-N \\ a_{w-1-i} & \text{if } i \geq w-N \end{cases} \\ \times & \text{if } w > 0 \text{ and } N > w. \end{cases}$$

Note 2. Bytestring to integer conversion. The `byteStringToInteger` function converts bytestrings to non-negative integers. It takes two arguments:

- A boolean endianness flag e.
- The bytestring s to be converted.

The conversion is little-endian if e is `(con bool False)` and big-endian if e is `(con bool True)`. In both cases the empty bytestring is converted to the integer 0. All bytestrings are legal inputs and there is no limitation on the size of s.

4.3.4.2 BLS12-381 built-in types

Supporting the BLS12-381 operations involves adding three new types and seventeen new built-in functions. The description of the semantics of these types and functions is quite complex and requires a considerable amount of notation, most of which is used only in Sections 4.3.4.2 and 4.3.4.3.

Table 4.14 describes three new built-in types.

Type	Denotation	Concrete Syntax
`bls12_381_G1_element`	G_1	`0x[0-9A-Fa-f]{96}` (see Note 6)
`bls12_381_G2_element`	G_2	`0x[0-9A-Fa-f]{192}` (see Note 6)
`bls12_381_mlresult`	H	None (see Note 6)

Table 4.14: Atomic types, batch 4

Here G_1 and G_2 are both additive cyclic groups of prime order r, where

$$r = \texttt{0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001}.$$

The fields $\mathbb{F}_q$ and $\mathbb{F}_{q^2}$. To define the groups G_1 and G_2 we need the finite field $\mathbb{F}_q$ where

$$q = \texttt{0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf}\\ \texttt{6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab}$$

which is a 381-bit prime. The field $\mathbb{F}_q$ is isomorphic to $\mathbb{Z}_q$, the ring of integers modulo q, and hence there is a natural epimorphism from $\mathbb{Z}$ to $\mathbb{F}_q$ which we denote by $n \mapsto \bar{n}$. Given $x \in \mathbb{F}_q$, we denote by $\tilde{x}$ the smallest non-negative integer n with $\bar{n} = x$. We sometimes use literal integers to represent elements of $\mathbb{F}_q$ in the obvious way.

We also make use of the field $\mathbb{F}_{q^2} = \mathbb{F}_q[X]/(X^2+1)$; we may regard $\mathbb{F}_{q^2}$ as the set $\{a+bu : a, b \in \mathbb{F}_q\}$ where $u^2 = -1$.

It is convenient to say that an element a of $\mathbb{F}_q$ is *larger* than another element b (and b is *smaller* than a) if $\tilde{a} > \tilde{b}$ in $\mathbb{Z}$. We extend this terminology to $\mathbb{F}_{q^2}$ by saying that $a + bu$ is larger than $c + du$ if either b is larger than d or $b = d$ and a is larger than c.

The groups G_1 and G_2. There are elliptic curves E_1 defined over $\mathbb{F}_q$:

$$E_1 : Y^2 = X^3 + 4$$

and E_2 defined over $\mathbb{F}_{q^2}$:

$$E_2 : Y^2 = X^3 + 4(u+1).$$

$E_1(\mathbb{F}_q)$ and $E_2(\mathbb{F}_{q^2})$ are abelian groups under the usual elliptic curve addition operations as described in [43, III.2] or [21, 2.1]. G_1 is a subgroup of $E_1(\mathbb{F}_q)$ and G_2 is a subgroup of $E_2(\mathbb{F}_{q^2})$; explicit generators for G_1 and G_2 are given in [41, 4.2.1]. We denote the identity element (the point at infinity) in G_1 by $\mathcal{O}_{G_1}$ and that in G_2 by $\mathcal{O}_{G_2}$. Given an integer n and a group element a in G_1 or G_2, the scalar multiple na is defined as usual to be $a + \cdots + a$ (n times) if $n > 0$ and $-a + \cdots + -a$ ($-n$ times) if $n < 0$; $0a$ is the identity element of the group.

The `bls12_381_MlResult` type. Values of the `bls12_381_MlResult` type are completely opaque and can only be obtained as a result of `bls12_381_millerLoop` or by multiplying two existing elements of type `bls12_381_MlResult`. We provide neither a serialisation format nor a concrete syntax for values of this type: they exist only ephemerally during computation. We do not specify H, the denotation of `bls12_381_mlresult`, precisely, but it must be a multiplicative abelian group. See Note 7 for more on this.

4.3.4.3 BLS12-381 built-in functions

Function	Signature	Denotation	Can fail?	Note
`bls12_381_G1_add`	[`bls12_381_G1_element`, `bls12_381_G1_element`] → `bls12_381_G1_element`	$(a, b) \mapsto a + b$	No	
`bls12_381_G1_neg`	[`bls12_381_G1_element`] → `bls12_381_G1_element`	$a \mapsto -a$	No	
`bls12_381_G1_scalarMul`	[`integer`, `bls12_381_G1_element`] → `bls12_381_G1_element`	$(n, a) \mapsto na$	No	
`bls12_381_G1_equal`	[`bls12_381_G1_element`, `bls12_381_G1_element`] → `bool`	$=$	No	
`bls12_381_G1_hashToGroup`	[`bytestring`, `bytestring`] → `bls12_381_G1_element`	hash_{G_1}	Yes	3
`bls12_381_G1_compress`	[`bls12_381_G1_element`] → `bytestring`	$\mathsf{compress}_{G_1}$	No	4
`bls12_381_G1_uncompress`	[`bytestring`] → `bls12_381_G1_element`	$\mathsf{uncompress}_{G_1}$	Yes	5
`bls12_381_G2_add`	[`bls12_381_G2_element`, `bls12_381_G2_element`] → `bls12_381_G2_element`	$(a, b) \mapsto a + b$	No	
`bls12_381_G2_neg`	[`bls12_381_G2_element`] → `bls12_381_G2_element`	$a \mapsto -a$	No	
`bls12_381_G2_scalarMul`	[`integer`, `bls12_381_G2_element`] → `bls12_381_G2_element`	$(n, a) \mapsto na$	No	
`bls12_381_G2_equal`	[`bls12_381_G2_element`, `bls12_381_G2_element`] → `bool`	$=$	No	
`bls12_381_G2_hashToGroup`	[`bytestring`, `bytestring`] → `bls12_381_G2_element`	hash_{G_2}	Yes	3
`bls12_381_G2_compress`	[`bls12_381_G2_element`] → `bytestring`	$\mathsf{compress}_{G_2}$	No	4
`bls12_381_G2_uncompress`	[`bytestring`] → `bls12_381_G2_element`	$\mathsf{uncompress}_{G_2}$	Yes	5
`bls12_381_millerLoop`	[`bls12_381_G1_element`, `bls12_381_G2_element`] → `bls12_381_mlresult`	e	No	7
`bls12_381_mulMlResult`	[`bls12_381_mlresult`, `bls12_381_mlresult`] → `bls12_381_mlresult`	$(a, b) \mapsto ab$	No	7
`bls12_381_finalVerify`	[`bls12_381_mlresult`, `bls12_381_mlresult`] → `bool`	ϕ	No	7

Table 4.15: Batch 4: BLS12-381 built-in functions (continued)

Note 3. Hashing into G_1 and G_2. The denotations hash_{G_1} and hash_{G_2} of `bls12_381_G1_hashToGroup` and `bls12_381_G2_hashToGroup` both take an arbitrary bytestring b (the *message*) and a (possibly empty) bytestring of length at most 255 known as a *domain separation tag* (DST) [27, 2.2.5] and hash them to obtain a point in G_1 or G_2 respectively. The details of the hashing process are described in [27] (see specifically Section 8.8), except that **we do not support DSTs of length greater than 255**: an attempt to use a longer DST directly will cause an error. If a longer DST is required then it should be hashed to obtain a short DST as described in [27, 5.3.3], and then this should be supplied as the second argument to the appropriate `hashToGroup` function.

Note 4. Compression for elements of G_1 and G_2. Points in G_1 and G_2 are encoded as bytestrings in a "compressed" format where only the x-coordinate of a point is encoded and some metadata is used to indicate which of two possible y-coordinates the point has. The encoding format is based on the Zcash encoding for BLS12-381 points: see [47] or [37, "Serialization"] or [41, Appendices C and D]. In detail,

- Given an element x of $\mathbb{F}_q$, $\tilde{x}$ can be written as a 381-bit binary number: $\tilde{x} = \sum_{i=0}^{380} b_i 2^i$ with $b_i \in \{0, 1\}$. We define $\mathsf{bits}(x)$ to be the bitstring $b_{380} \cdots b_0$.
- A non-identity element of G_1 can be written in the form (x, y) with $x, y \in \mathbb{F}_q$. Not every element x of $\mathbb{F}_q$ is the x-coordinate of a point in G_1, but those which are in fact occur as the x-coordinate of two distinct points in G_1 whose y-coordinates are the negatives of each other. A similar statement is true for $\mathbb{F}_{q^2}$ and G_2. In both cases we denote the smaller of the possible y-coordinates by $y_{\min}(x)$ and the larger by $y_{\max}(x)$.
- For $(x, y) \in G_1 \backslash \mathcal{O}_{G_1}$ we define

$$\mathsf{compress}_{G_1}(x, y) = \begin{cases} 100 \cdot \mathsf{bits}(x) & \text{if } y = y_{\min}(x) \\ 101 \cdot \mathsf{bits}(x) & \text{if } y = y_{\max}(x) \end{cases}$$

- We encode the identity element of G_1 using

$$\mathsf{compress}_{G_1}(\mathcal{O}_{G_1}) = 110 \cdot 0^{381},$$

  where 0^{381} denotes a string of 381 0 bits.

Thus in all cases the encoding of an element of G_1 requires exactly 384 bits, or 48 bytes.

- Similarly, every non-identity element of G_2 can be written in the form (x, y) with $x, y \in \mathbb{F}_{q^2}$. We define

$$\mathsf{compress}_{G_2}(a + bu, y) = \begin{cases} 100 \cdot \mathsf{bits}(b) \cdot 000 \cdot \mathsf{bits}(a) & \text{if } y = y_{\min}(a + bu) \\ 101 \cdot \mathsf{bits}(b) \cdot 000 \cdot \mathsf{bits}(a) & \text{if } y = y_{\max}(a + bu) \end{cases}$$

- The identity element of G_2 is encoded using

$$\mathsf{compress}_{G_2}(\mathcal{O}_{G_2}) = 110 \cdot 0^{765}.$$

The encoding of an element of G_2 requires exactly 768 bits, or 96 bytes.

Note that in all cases the most significant bit of a compressed point is 1. In the Zcash serialisation scheme this indicates that the point is compressed; Zcash also supports a serialisation format where both

the x- and y-coordinates of a point are encoded, and in that case the leading bit of the encoded point is 0. We do not support this format.

Note 5. Uncompression for elements of G_1 and G_2. There are two (partial) "uncompression" functions $\mathsf{uncompress}_{G_1}$ and $\mathsf{uncompress}_{G_2}$ which convert bytestrings into group elements; these are obtained by inverting the process described in Note 4.

Uncompression for G_1 elements. Given a bytestring b, it is checked that b contains exactly 48 bytes. If not, then $\mathsf{uncompress}_{G_1}(b) = \times$ (ie, uncompression fails). If the length is equal to 48 bytes, write b as a sequence of bits: $b = b_{383} \cdots b_0$.

- If $b_{383} \neq 1$, then $\mathsf{uncompress}_{G_1}(b) = \times$.
- If $b_{383} = b_{382} = 1$ then $\mathsf{uncompress}_{G_1}(b) = \begin{cases} \mathcal{O}_{G_1} & \text{if } b_{381} = b_{380} = \cdots = b_0 = 0 \\ \times & \text{otherwise.} \end{cases}$
- If $b_{383} = 1$ and $b_{382} = 0$, let $c = \sum_{i=0}^{380} b_i 2^i \in \mathbb{N}$.
  - If $c \geq q$, $\mathsf{uncompress}_{G_1}(b) = \times$.
  - Otherwise, let $x = \bar{c} \in \mathbb{F}_q$ and let $z = x^3 + 4$. If z is not a square in $\mathbb{F}_q$, then $\mathsf{uncompress}_{G_1}(b) = \times$.
  - If z is a square then let $y = \begin{cases} y_{\min}(x) & \text{if } b_{381} = 0 \\ y_{\max}(x) & \text{if } b_{381} = 1. \end{cases}$
  - Then $\mathsf{uncompress}_{G_1}(b) = \begin{cases} (x, y) & \text{if } (x, y) \in G_1 \\ \times & \text{otherwise.} \end{cases}$

Uncompression for G_2 elements. Given a bytestring b, it is checked that b contains exactly 96 bytes. If not, then $\mathsf{uncompress}_{G_2}(b) = \times$ (ie, uncompression fails). If the length is equal to 96 bytes, write b as a sequence of bits: $b = b_{767} \cdots b_0$.

- If $b_{767} \neq 1$, then $\mathsf{uncompress}_{G_2}(b) = \times$.
- If $b_{767} = b_{766} = 1$ then $\mathsf{uncompress}_{G_2}(b) = \begin{cases} \mathcal{O}_{G_2} & \text{if } b_{765} = b_{764} = \cdots = b_0 = 0 \\ \times & \text{otherwise.} \end{cases}$
- If $b_{767} = 1$ and $b_{766} = 0$, let $c = \sum_{i=0}^{383} b_i 2^i$ and $d = \sum_{i=384}^{764} b_i 2^{i-384} \in \mathbb{N}$.
  - If $c \geq q$ or $d \geq q$, $\mathsf{uncompress}_{G_2}(b) = \times$.
  - Otherwise, let $x = \bar{c} + \bar{d}u \in \mathbb{F}_{q^2}$ and let $z = x^3 + 4(u + 1)$. If z is not a square in $\mathbb{F}_{q^2}$, then $\mathsf{uncompress}_{G_2}(b) = \times$.
  - If z is a square then let $y = \begin{cases} y_{\min}(x) & \text{if } b_{765} = 0 \\ y_{\max}(x) & \text{if } b_{765} = 1. \end{cases}$
  - Then $\mathsf{uncompress}_{G_2}(b) = \begin{cases} (x, y) & \text{if } (x, y) \in G_2 \\ \times & \text{otherwise.} \end{cases}$

Note 6. Concrete syntax for BLS12-381 types. Concrete syntax for the `bls12_381_G1_element` and `bls12_381_G2_element` types is provided via the compression and decompression functions defined in Notes 4 and 5. Specifically, a value of type `bls12_381_G1_element` is denoted by a term of the form `(con bls12_381_G1_element 0x...)` where `...` consists of 96 hexadecimal digits representing the 48-byte compressed form of the relevant point. Similarly, a value of type `bls12_381_G2_element` is denoted by a term of the form `(con bls12_381_G2_element 0x...)` where `...` consists of 192 hexadecimal digits representing the 96-byte compressed form of the relevant point. **This syntax is provided only for use in textual Plutus Core programs**, for example for experimentation and testing. We do not support constants of any of the BLS12-381 types in serialised programs on the Cardano blockchain: see Section C.3.4. However, for `bls12_381_G1_element` and `bls12_381_G2_element` one can use the appropriate uncompression function on a bytestring constant at runtime: for example, instead of

```
(con bls12_381_G1_element 0xa1e9a0...)
```

write

```
[(builtin bls12_381_G1_uncompress) (con bytestring #a1e9a0...)].
```

No concrete syntax is provided for values of type `bls12_381_mlresult`. It is not possible to parse such values, and they will appear as `(con bls12_381_mlresult <opaque>)` if output by a program.

Note 7. Pairing operations. For efficiency reasons we split the pairing process into two parts: the `bls12_381_millerLoop` and `bls12_381_finalVerify` functions. We assume that we have

- An intermediate multiplicative abelian group H.
- A function (not necessarily itself a pairing) $e : G_1 \times G_2 \to H$.
- A cyclic group μ_r of order r.
- An epimorphism $\psi : H \to \mu_r$ of groups such that $\psi \circ e : G_1 \times G_2 \to \mu_r$ is a (nondegenerate, bilinear) pairing.

Given these ingredients, we define

- $[\![\texttt{bls12\_381\_mlresult}]\!] = H$.
- $[\![\texttt{bls12\_381\_mulMlResult}]\!]$ = the group multiplication operation in H.
- $[\![\texttt{bls12\_381\_millerLoop}]\!] = e$.
- $[\![\texttt{bls12\_381\_finalVerify}]\!] = \phi$, where

$$\phi(a,b) = \begin{cases} \mathsf{true} & \text{if } \psi(ab^{-1}) = 1_{\mu_r} \\ \mathsf{false} & \text{otherwise.} \end{cases}$$

We do not mandate specific choices for H, μ_r, e, and ϕ, but a plausible choice would be

- $H = \mathbb{F}^{\times}_{q^{12}}$.
- e is the Miller loop associated with the optimal Ate pairing for E_1 and E_2 [45].
- $\mu_r = \{x \in \mathbb{F}^{\times}_{q^{12}} : x^r = 1\}$, the group of rth roots of unity in $\mathbb{F}_{q^{12}}$. (There are r distinct rth roots of unity in $\mathbb{F}_{q^{12}}$ because the embedding degree of E_1 and E_2 with respect to r is 12 (see [21, 4.1]).)

- $\psi(x) = x^{\frac{q-1}{r}}$.

The functions `bls12_381_millerLoop` and (especially) `bls12_381_finalVerify` are expected to be expensive, so their use should be kept to a minimum. Fortunately most current use cases do not require many uses of these functions.

4.3.5 Batch 5

The fifth batch of built-in functions adds support for

- Logical and bitwise operations on bytestrings (see [7] and [8]).
- The `RIPEMD-160` hash function.

In the table below most of the functions involve operating on individual bits. We will often view byetstrings as bitstrings $b_{n-1} \cdots b_0$ with $b_i \in \{0, 1\}$ (and n necessarily a multiple of 8). Strictly we should use the functions bits and bytes of Section 1.2.3 to convert back and forth between bytestrings and bitstrings, but we elide this for conciseness and reduce ambiguity by using lower-case names like b,c and d for bits and upper-case names like B for bytes. We denote the complement of a bit $b \in \{0, 1\}$ by $\bar{b}$, so $\bar{0} = 1$ and $\bar{1} = 0$.

Function	Signature	Denotation	Can fail?	Note
`andByteString`	[`bool`, `bytestring`, `bytestring`] → `bytestring`	and	No	1
`orByteString`	[`bool`, `bytestring`, `bytestring`] → `bytestring`	or	No	1
`xorByteString`	[`bool`, `bytestring`, `bytestring`] → `bytestring`	xor	No	1
`complementByteString`	[`bytestring`] → `bytestring`	$b_{n-1} \cdots b_0 \mapsto \bar{b}_{n-1} \cdots \bar{b}_0$	No	
`shiftByteString`	[`bytestring`, `integer`] → `bytestring`	shift	No	2
`rotateByteString`	[`bytestring`, `integer`] → `bytestring`	rotate	No	3
`countSetBits`	[`bytestring`] → `integer`	$b_{n-1} \cdots b_0 \mapsto \lvert\{i : b_i = 1\}\rvert$	No	
`findFirstSetBit`	[`bytestring`] → `integer`	ffs	No	4
`readBit`	[`bytestring`, `integer`] → `bytestring`	$(b_{n-1} \cdots b_0, i) \mapsto \begin{cases} b_i & \text{if } 0 \le i \le n-1 \\ \times & \text{otherwise} \end{cases}$	Yes	
`writeBits`	[`bytestring`, `list (integer)`, `bool`] → `bytestring`	writeBits	Yes	5
`replicateByte`	[`integer`, `integer`] → `bytestring`	replicateByte	Yes	6
`ripemd_160`	[`bytestring`] → `bytestring`	Compute a RIPEMD-160 hash [18, 25]	No	

Table 4.16: Built-in functions, batch 5

Note 1. Bitwise logical operations. The bitwise logical operations and, or, and xor are defined by extending the usual single-bit logical operations $\wedge$, $\vee$, and $\oplus$ (respectively) to bytestrings. However, there is a complication if the bytestrings have different lengths.

- If the first argument of one of the bitwise logical operations is false then the longer bytestring is (conceptually) truncated on the right to have the same length as the shorter one.
- If the first argument is true then the shorter bytestring is (conceptually) extended on the right to have the same length as the longer one. In the case of and the shorter bytestring is extended with 0 bits and in the case of or and xor it is extended with 1 bits.

Formally the truncation operation mentioned above is defined as a function which removes the rightmost k bits from a bitstring:

$$\mathsf{trunc}\,(b_{n-1}\cdots b_0, k) = b_{n-1}\cdots b_k \quad \text{if } 0 \le k \le n$$

and the extension operation is defined as a function which appends k bits to a bitstring:

$$\mathsf{ext}\,(b_{n-1}\cdots b_0, k, x) = b_{n-1}\cdots b_0 \cdot x_{k-1}\cdots x_0 \quad \text{where } x \in \{0,1\},\ k \ge 0, \text{ and } x_0 = x_1 = \cdots = x_{k-1} = x.$$

In both cases, the otuput is the same as the input when $k = 0$.

The denotations of the bitwise logical functions are now defined on bitstring representations of bytestrings as follows, where b is a bitstring of length m and c is a bitstring of length n:

$$\begin{aligned}
\mathsf{and}\,(\mathsf{false}, b, c) &= d_{l-1}\cdots d_0 \quad \text{where } l = \min\{m,n\} \text{ and } d_i = \mathsf{trunc}\,(b, m-l)_i \wedge \mathsf{trunc}\,(c, n-l)_i \\
\mathsf{and}\,(\mathsf{true}, b, c) &= d_{l-1}\cdots d_0 \quad \text{where } l = \max\{m,n\} \text{ and } d_i = \mathsf{ext}\,(b, l-m, 1)_i \wedge \mathsf{ext}\,(c, l-n, 1)_i \\
\\
\mathsf{or}\,(\mathsf{false}, b, c) &= d_{l-1}\cdots d_0 \quad \text{where } l = \min\{m,n\} \text{ and } d_i = \mathsf{trunc}\,(b, m-l)_i \vee \mathsf{trunc}\,(c, n-l)_i \\
\mathsf{or}\,(\mathsf{true}, b, c) &= d_{l-1}\cdots d_0 \quad \text{where } l = \max\{m,n\} \text{ and } d_i = \mathsf{ext}\,(b, l-m, 0)_i \vee \mathsf{ext}\,(c, l-n, 0)_i \\
\\
\mathsf{xor}\,(\mathsf{false}, b, c) &= d_{l-1}\cdots d_0 \quad \text{where } l = \min\{m,n\} \text{ and } d_i = \mathsf{trunc}\,(b, m-l)_i \oplus \mathsf{trunc}\,(c, n-l)_i \\
\mathsf{xor}\,(\mathsf{true}, b, c) &= d_{l-1}\cdots d_0 \quad \text{where } l = \max\{m,n\} \text{ and } d_i = \mathsf{ext}\,(b, l-m, 0)_i \oplus \mathsf{ext}\,(c, l-n, 0)_i.
\end{aligned}$$

Note that although trunc is applied to both arguments in the above definitions, if the arguments have the same lengths then in fact neither will be truncated and if the arguments have different lengths then only the longer one will be truncated. Similarly extension will only occur if the arguments are of different lengths, in which case the shorter one will be extended.

Note 2. Shifting bits. The `shiftByteString` builtin takes a bytestring s and an integer k and shifts the bits of the bytestring k places to the left if $k \ge 0$ and k places to the right if $k < 0$, replacing vacated bits with 0. The length of the output bytestring is the same as that of the input. The denotation (defined on the bitstring representation of s) is

$$\mathsf{shift}\,(b_{n-1}\cdots b_0, k) = d_{n-1}\cdots d_0 \quad \text{where } d_i = \begin{cases} b_{i-k} & \text{if } 0 \le i-k \le n-1 \\ 0 & \text{otherwise.} \end{cases}$$

Note 3. Rotating bits. The `rotateByteString` builtin takes a bytestring s and an integer k and rotates the bits of s k places to the left if $k \ge 0$ and k places to the right if $k < 0$. The length of the

output bytestring is the same as that of the input. The denotation of `rotateByteString` (defined on the bitstring representation of s) is

$$\mathsf{rotate}\,(b_{n-1}\cdots b_0, k) = d_{n-1}\cdots d_0 \quad \text{where } d_i = b_{(i-k) \bmod n}.$$

Note 4. Finding the first set bit in a bytestring. The `findFirstSetBit` builtin returns the index of the first nonzero bit in a bytestring, counting from the *right*. If the bytestring consists entirely of zeros then it returns -1. The denotation $\mathtt{ffs} : \mathbb{b}^* \to \mathbb{Z}$ is

$$\mathtt{ffs}\,(b_{n-1}\cdots b_0) = \begin{cases} -1 & \text{if } b_i = 0 \text{ for } 0 \le i \le n-1 \\ \min\{i \,:\, b_i \neq 0\} & \text{otherwise.} \end{cases}$$

Note 5. Writing bits. The denotation writeBits of the `writeBits` builtin takes a bytestring s, a list $\boldsymbol{J}$ of integer indices, and a boolean value u. An error occurs if any of the indices in $\boldsymbol{J}$ is not a valid bit index for s. If all of the indices are within bounds then for each index j in $\boldsymbol{J}$ the j-th bit of s is updated according to the value of u (`0` for false, `1` for true). The list $\boldsymbol{J}$ is allowed to contain repetitions.

Formally,

$$\mathsf{writeBits}(b_{n-1}\cdots b_0, [j_0, \ldots, j_{l-1}], u) = \begin{cases} \times & \text{if } \exists k \in \{0, \ldots, l-1\} \text{ such that } j_k < 0 \text{ or } j_k \ge n \\ d_{n-1}\cdots d_0 & \text{otherwise} \end{cases}$$

where

$$d_i = \begin{cases} b_i & \text{if } i \notin \{j_0, \ldots, j_{l-1}\} \\ \mathtt{0} & \text{if } i \in \{j_0, \ldots, j_{l-1}\} \text{ and } u = \mathsf{false} \\ \mathtt{1} & \text{if } i \in \{j_0, \ldots, j_{l-1}\} \text{ and } u = \mathsf{true}. \end{cases}$$

Note 6. Replicating bytes. The `replicateByte` builtin takes a length l between 0 and 8192 and an integer B between 0 and 255 and produces a bytestring of length l. It fails if either argument is outside the required bounds. The denotation is

$$\mathsf{replicateByte}\,(l, B) = \begin{cases} B_0\cdots B_{l-1} \quad (B_i = B \text{ for all } i) & \text{if } 0 \le l \le 8192 \text{ and } B \in \mathbb{B} \\ \times & \text{otherwise.} \end{cases}$$

Note that unlike the other denotations in this section we are viewing the output as a bytestring here, not a bitstring.

Appendix A

Formally Verified Behaviours

To follow.

Appendix B

Serialising `data` Objects Using the CBOR Format

B.1 Introduction

In this section we define a CBOR encoding for the `data` type introduced in Section 4.3.1.1. For ease of reference we reproduce the definition of the Haskell `Data` type, which we may regard as the definition of the Plutus `data` type. Other representations are of course possible, but this is useful for the present discussion.

```
data Data =
   Constr Integer [Data]
   | Map [(Data, Data)]
   | List [Data]
   | I Integer
   | B ByteString
```

The CBOR encoding defined here uses basic CBOR encodings as defined in the CBOR standard [17], but with some refinements. Specifically

- We use a restricted encoding for bytestrings which requires that bytestrings are serialised as sequences of blocks, each block being at most 64 bytes long. Any encoding of a bytestring using our scheme is valid according to the CBOR specification, but the CBOR specification permits some encodings which we do not accept. The purpose of the size restriction is to prevent arbitrary data from bring stored on the blockchain.
- Large integers (less than -2^{64} or greater than $2^{64} - 1$) are encoded via the restricted bytestring encoding; other integers are encoded as normal. Again, our restricted encodings are compatible with the CBOR specification.
- The `Constr` case of the `data` type is encoded using a scheme which is an early version of a proposed extension of the CBOR specification to include encodings for discriminated unions. See [22] and [16, Section 9.1].

B.2 Notation

We introduce some extra notation for use here and in Appendix C.

The notation $f : X \rightharpoonup Y$ indicates that f is a partial map from X to Y. We denote the empty bytestring by ϵ and (as in 1.2.2) use $\ell(s)$ to denote the length of a bytestring s and $\cdot$ to denote the concatenation of two bytestrings, and also the operation of prepending or appending a byte to a bytestring. We will also make use of the div and mod functions described in Note 1 in Section 4.3.1.

Encoders and decoders. Recall that $\mathbb{B} = \mathbb{N}_{[0,255]}$, the set of integral values that can be represented in a single byte, and that we identify bytestrings with elements of $\mathbb{B}^*$. We will describe the CBOR encoding of the `data` type by defining families of encoding functions (or *encoders*)

$$\mathcal{E}_X : X \to \mathbb{B}^*$$

and decoding functions (or *decoders*)

$$\mathcal{D}_X : \mathbb{B}^* \rightharpoonup \mathbb{B}^* \times X$$

for various sets X, such as the set $\mathbb{Z}$ of integers and the set of all `data` items. The encoding function $\mathcal{E}_X$ takes an element $x \in X$ and converts it to a bytestring, and the decoding function $\mathcal{D}_X$ takes a bytestring s, decodes some initial prefix of s to a value $x \in X$, and returns the remainder of s together with x. Decoders for complex types will often be built up from decoders for simpler types. Decoders are *partial* functions because they can fail, for instance, if there is insufficient input, or if the input is not well formed, or if a decoded value is outside some specified range.

Many of the decoders which we define below involve a number of cases for different forms of input, and we implicitly assume that the decoder fails if none of the cases applies. We also assume that if a decoder fails then so does any other decoder which invokes it, so any failure when attempting to decode a particular data item in a bytestring will cause the entire decoding process to fail (immediately).

B.3 The CBOR format

A CBOR-encoded item consists of a bytestring beginning with a *head* which occupies 1,2,3,5, or 9 bytes. Depending on the contents of the head, some sequence of bytes following it may also contribute to the encoded item. The first three bits of the head are interpreted as a natural number between 0 and 7 (the *major type*) which gives basic information about the type of the following data. The remainder of the head is called the *argument* of the head and is used to encode further information, such as the value of an encoded integer or the size of a list of encoded items. Encodings of complex objects may occupy the bytes following the head, and these will typically contain further encoded items.

B.4 Encoding and decoding the heads of CBOR items

For $i \in \mathbb{N}$ we define a function $\mathsf{b}_i : \mathbb{N} \to \mathbb{B}$ which returns the i-th byte of an integer, with the 0-th byte being the least significant:

$$\mathsf{b}_i(n) = \mathrm{mod}(\mathrm{div}(n, 256^i), 256).$$

We use this to define for each $k \geq 1$ a partial function $\mathsf{e}_k : \mathbb{N} \rightharpoonup \mathbb{B}^*$ which converts a sufficiently small integer to a bytestring of length k (possibly with leading zeros):

$$\mathsf{e}_k(n) = [\mathsf{b}_{k-1}(n), \ldots, \mathsf{b}_0(n)] \quad \text{if } n \leq 256^k - 1.$$

This function fails if the input is too large to fit into a k-byte bytestring.

We also define inverse functions $\mathsf{d}_k : \mathbb{B}^* \rightharpoonup \mathbb{N}$ which decode a k-byte natural number from the start of a bytestring, failing if there is insufficient input:

$$\mathsf{d}_k(s) = (s', \sum_{i=0}^{k-1} 256^i b_i) \qquad \text{if } s = [b_{k-1}, \dots, b_0] \cdot s'.$$

We now define an encoder $\mathcal{E}_{\mathsf{head}} : \mathbb{N}_{[0,7]} \times \mathbb{N}_{[0,2^{64}-1]} \to \mathbb{B}^*$ which takes a major type and a natural number and encodes them as a CBOR head using the standard encoding:

$$\mathcal{E}_{\mathsf{head}}(m,n) = \begin{cases} [32m+n] & \text{if } n \le 23 \\ (32m+24) \cdot \mathsf{e}_1(n) & \text{if } 24 \le n \le 255 \\ (32m+25) \cdot \mathsf{e}_2(n) & \text{if } 256 \le n \le 256^2 - 1 \\ (32m+26) \cdot \mathsf{e}_4(n) & \text{if } 256^2 \le n \le 256^4 - 1 \\ (32m+27) \cdot \mathsf{e}_8(n) & \text{if } 256^4 \le n \le 256^8 - 1. \end{cases}$$

The corresponding decoder $\mathcal{D}_{\mathsf{head}} : \mathbb{B}^* \rightharpoonup \mathbb{B}^* \times \mathbb{N}_{[0,7]} \times \mathbb{N}_{[0,2^{64}-1]}$ is given by

$$\mathcal{D}_{\mathsf{head}}(n \cdot s) = \begin{cases} (s, \mathrm{div}(n,32), \mathrm{mod}(n,32)) & \text{if } \mathrm{mod}(n,32) \le 23 \\ (s', \mathrm{div}(n,32), k) & \text{if } \mathrm{mod}(n,32) = 24 \text{ and } \mathsf{d}_1(s) = (s',k) \\ (s', \mathrm{div}(n,32), k) & \text{if } \mathrm{mod}(n,32) = 25 \text{ and } \mathsf{d}_2(s) = (s',k) \\ (s', \mathrm{div}(n,32), k) & \text{if } \mathrm{mod}(n,32) = 26 \text{ and } \mathsf{d}_4(s) = (s',k) \\ (s', \mathrm{div}(n,32), k) & \text{if } \mathrm{mod}(n,32) = 27 \text{ and } \mathsf{d}_8(s) = (s',k). \end{cases}$$

This function is undefined if the input is the empty bytestring ϵ, if the input is too short, or if its initial byte is not of the expected form.

Heads for indefinite-length items. The functions $\mathcal{E}_{\mathsf{head}}$ and $\mathcal{D}_{\mathsf{head}}$ defined above are used for a number of purposes. One use is to encode integers less than 64 bits, where the argument of the head is the relevant integer. Another use is for "definite-length" encodings of items such as bytestrings and lists, where the head contains the length n of the object and is followed by some encoding of the object itself (for example a sequence of n bytes for a bytestring or a sequence of n encoded objects for the elements of a list). It is also possible to have "indefinite-length" encodings of objects such as lists and arrays, which do not specify the length of an object in advance: instead a special head with argument 31 is emitted, followed by the encodings of the individual items; the end of the sequence is marked by a "break" byte with value 255. We define an encoder $\mathcal{E}_{\mathsf{indef}} : \mathbb{N}_{[2,5]} \to \mathbb{B}^*$ and a decoder $\mathcal{D}_{\mathsf{indef}} : \mathbb{B}^* \rightharpoonup \mathbb{B}^* \times \mathbb{N}_{[2,5]}$ which deal with indefinite heads for a given major type:

$$\begin{aligned} \mathcal{E}_{\mathsf{indef}}(m) &= [32m+31] \\ \mathcal{D}_{\mathsf{indef}}(n \cdot s) &= (s,m) \qquad \text{if } n = 32m+31. \end{aligned}$$

Note that $\mathcal{E}_{\mathsf{indef}}$ and $\mathcal{D}_{\mathsf{indef}}$ are only defined for $m \in \{2,3,4,5\}$ (and we shall only use them in these cases). The case $m = 31$ corresponds to the break byte and for $m \in \{0,1,6\}$ the value is not well formed: see [17, 3.2.4].

B.5 Encoding and decoding bytestrings

The standard CBOR encoding of bytestrings encodes a bytestring as either a definite-length sequence of bytes (the length being given in the head) or as an indefinite-length sequence of definite-length "chunks" (see [17, §§3.1 and 3.4.2]). We use a similar scheme, but only allow chunks of length up to 64. To this end, suppose that $a = [a_1, \ldots, a_{64k+r}] \in \mathbb{B}^* \backslash \{\epsilon\}$ where $k \geq 0$ and $0 \leq r \leq 63$. We define the *canonical 64-byte decomposition* $\bar{a}$ of a to be

$$\bar{a} = [[a_1, \ldots, a_{64}], [a_{65}, \ldots, a_{128}], \ldots, [a_{64(k-1)+1}, \ldots, a_{64k}]] \in (\mathbb{B}^*)^*$$

if $r = 0$ and

$$\bar{a} = [[a_1, \ldots, a_{64}], [a_{65}, \ldots, a_{128}], \ldots, [a_{64(k-1)+1}, \ldots, a_{64k}], [a_{64k+1}, \ldots, a_{64k+r}]] \in (\mathbb{B}^*)^*$$

if $r > 0$. The canonical decomposition of the empty list is $\bar{\epsilon} = []$.

We define the encoder $\mathcal{E}_{\mathbb{B}^*} : \mathbb{B}^* \to \mathbb{B}^*$ for bytestrings by encoding bytestrings of size up to 64 using the standard CBOR encoding and encoding larger bytestrings by breaking them up into 64-byte chunks (with the final chunk possibly being less than 64 bytes long) and encoding them as an indefinite-length list (major type 2 indicates a bytestring):

$$\mathcal{E}_{\mathbb{B}^*}(s) = \begin{cases} \mathcal{E}_{\mathsf{head}}(2, \ell(s)) \cdot s & \text{if } \ell(s) \leq 64 \\ \mathcal{E}_{\mathsf{indef}}(2) \cdot \mathcal{E}_{\mathsf{head}}(2, \ell(c_1)) \cdot c_1 \cdot \mathcal{E}_{\mathsf{head}}(2, \ell(c_2)) \cdot \cdots & \\ \qquad \cdots \cdot c_{n-1} \cdot \mathcal{E}_{\mathsf{head}}(2, \ell(c_n)) \cdot c_n \cdot 255 & \text{if } \ell(s) > 64 \text{ and } \bar{s} = [c_1, \ldots, c_n]. \end{cases}$$

The decoder is slightly more complicated. Firstly, for every $n \geq 0$ we define a decoder $\mathcal{D}^{(n)}_{\mathsf{bytes}} : \mathbb{B}^* \rightharpoonup \mathbb{B}^* \times \mathbb{B}^*$ which extracts an n-byte prefix from its input (failing in the case of insufficient input):

$$\mathcal{D}^{(n)}_{\mathsf{bytes}}(s) = \begin{cases} (s, \epsilon) & \text{if } n = 0 \\ (s'', b \cdot t) & \text{if } s = b \cdot s' \text{ and } \mathcal{D}^{(n-1)}_{\mathsf{bytes}}(s') = (s'', t). \end{cases}$$

Secondly, we define a decoder $\mathcal{D}_{\mathsf{block}} : \mathbb{B}^* \rightharpoonup \mathbb{B}^* \times \mathbb{B}^*$ which attempts to extract a bytestring of length at most 64 from its input; $\mathcal{D}_{\mathsf{block}}$ (and any other function which calls it) will fail if it encounters a bytestring which is greater than 64 bytes.

$$\mathcal{D}_{\mathsf{block}}(s) = \mathcal{D}^{(n)}_{\mathsf{bytes}}(s') \quad \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 2, n) \text{ and } n \leq 64.$$

Thirdly, we define a decoder $\mathcal{D}_{\mathsf{blocks}} : \mathbb{B}^* \rightharpoonup \mathbb{B}^* \times \mathbb{B}^*$ which decodes a sequence of blocks and returns their concatenation.

$$\mathcal{D}_{\mathsf{blocks}}(s) = \begin{cases} (s', \epsilon) & \text{if } s = 255 \cdot s' \\ (s'', t \cdot t') & \text{if } \mathcal{D}_{\mathsf{block}}(s) = (s', t) \text{ and } \mathcal{D}_{\mathsf{blocks}}(s') = (s'', t'). \end{cases}$$

Finally we define the decoder $\mathcal{D}_{\mathbb{B}^*} : \mathbb{B}^* \rightharpoonup \mathbb{B}^* \times \mathbb{B}^*$ for bytestrings by

$$\mathcal{D}_{\mathbb{B}^*}(s) = \begin{cases} (s', t) & \text{if } \mathcal{D}_{\mathsf{block}}(s) = (s', t) \\ \mathcal{D}_{\mathsf{blocks}}(s') & \text{if } \mathcal{D}_{\mathsf{indef}}(s) = (s', 2). \end{cases}$$

This looks for either a single block or an indefinite-length list of blocks, in the latter case returning their concatenation. It will accept the output of $\mathcal{E}_{\mathbb{B}^*}$ but will reject bytestring encodings containing any blocks greater than 64 bytes long, even if they are valid bytestring encodings according to the CBOR specification.

B.6 Encoding and decoding integers

As with bytestrings we use a specialised encoding scheme for integers which prohibits encodings with overly-long sequences of arbitrary data. We encode integers in $\mathbb{N}_{[-2^{64},2^{64}-1]}$ as normal (see [17, §3.1]: the major type is 0 for positive integers and 1 for negative ones) and larger ones by emitting a CBOR tag (major type 6; argument 2 for positive numbers and 3 for negative numbers) to indicate the sign, then converting the integer to a bytestring and emitting that using the encoder defined above. This encoding scheme is the same as the standard one except for the size limitations.

We firstly define conversion functions $\mathsf{itos} : \mathbb{N} \to \mathbb{B}^*$ and $\mathsf{stoi} : \mathbb{B}^* \to \mathbb{N}$ by

$$\mathsf{itos}(n) = \begin{cases} \epsilon & \text{if } n = 0 \\ \mathsf{itos}(\mathrm{div}(n, 256)) \cdot \mathrm{mod}(n, 256) & \text{if } n > 0. \end{cases}$$

and

$$\mathsf{stoi}(l) = \begin{cases} 0 & \text{if } l = \epsilon \\ 256 \times \mathsf{stoi}(l') + n & \text{if } l = l' \cdot n \text{ with } n \in \mathbb{B}. \end{cases}$$

The encoder $\mathcal{E}_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{B}^*$ for integers is now defined by

$$\mathcal{E}_{\mathbb{Z}}(n) = \begin{cases} \mathcal{E}_{\mathsf{head}}(0, n) & \text{if } 0 \le n \le 2^{64} - 1 \\ \mathcal{E}_{\mathsf{head}}(6, 2) \cdot \mathcal{E}_{\mathbb{B}^*}(\mathsf{itos}(n)) & \text{if } n \ge 2^{64} \\ \mathcal{E}_{\mathsf{head}}(1, -n - 1) & \text{if } -2^{64} \le n \le -1 \\ \mathcal{E}_{\mathsf{head}}(6, 3) \cdot \mathcal{E}_{\mathbb{B}^*}(\mathsf{itos}(-n - 1)) & \text{if } n \le -2^{64} - 1. \end{cases}$$

The decoder $\mathcal{D}_{\mathbb{Z}} : \mathbb{B}^* \rightharpoonup \mathbb{B}^* \times \mathbb{Z}$ inverts this process. The decoder is in fact slightly more permissive than the encoder because it also accepts small integers encoded using the scheme for larger ones. However, the CBOR standard permits integer encodings which contain bytestrings longer than 64 bytes and it will not accept those.

$$\mathcal{D}_{\mathbb{Z}}(s) = \begin{cases} (s', n) & \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 0, n) \\ (s', -n - 1) & \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 1, n) \\ (s'', \mathsf{stoi}(b)) & \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 6, 2) \text{ and } \mathcal{D}_{\mathbb{B}^*}(s') = (s'', b) \\ (s'', -\mathsf{stoi}(b) - 1) & \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 6, 3) \text{ and } \mathcal{D}_{\mathbb{B}^*}(s') = (s'', b). \end{cases}$$

B.7 Encoding and decoding `data`

It is now quite straightforward to encode most `data` values. The main complication is in the encoding of constructor tags (the number i in `Constr` i l).

The encoder. The encoder is given by

$$\begin{aligned} \mathcal{E}_{\mathtt{data}}(\mathtt{Map}\ l) &= \mathcal{E}_{\mathsf{head}}(5, \ell(l)) \cdot \mathcal{E}_{(\mathtt{data}^2)^*}(l) \\ \mathcal{E}_{\mathtt{data}}(\mathtt{List}\ l) &= \mathcal{E}_{\mathsf{indef}}(4) \cdot \mathcal{E}_{\mathtt{data}^*}(l) \cdot 255 \\ \mathcal{E}_{\mathtt{data}}(\mathtt{Constr}\ i\ l) &= \mathcal{E}_{\mathsf{ctag}}(i) \cdot \mathcal{E}_{\mathsf{indef}}(4) \cdot \mathcal{E}_{\mathtt{data}^*}(l) \cdot 255 \\ \mathcal{E}_{\mathtt{data}}(\mathtt{I}\ n) &= \mathcal{E}_{\mathbb{Z}}(n) \\ \mathcal{E}_{\mathtt{data}}(\mathtt{B}\ s) &= \mathcal{E}_{\mathbb{B}^*}(s). \end{aligned}$$

This definition uses encoders for lists of data items, lists of pairs of data items, and constructor tags as follows:

$$\mathcal{E}_{\mathtt{data}^*}([d_1, \ldots, d_n]) = \mathcal{E}_{\mathtt{data}}(d_1) \cdot \cdots \cdot \mathcal{E}_{\mathtt{data}}(d_n)$$
$$\mathcal{E}_{(\mathtt{data}^2)^*}([(k_1, d_1), \ldots, (k_n, d_n)]) = \mathcal{E}_{\mathtt{data}}(k_1) \cdot \mathcal{E}_{\mathtt{data}}(d_1) \cdot \cdots \cdot \mathcal{E}_{\mathtt{data}}(k_n) \cdot \mathcal{E}_{\mathtt{data}}(d_n)$$
$$\mathcal{E}_{\mathtt{ctag}}(i) = \begin{cases} \mathcal{E}_{\mathsf{head}}(6, 121 + i) & \text{if } 0 \le i \le 6 \\ \mathcal{E}_{\mathsf{head}}(6, 1280 + (i - 7)) & \text{if } 7 \le i \le 127 \\ \mathcal{E}_{\mathsf{head}}(6, 102) \cdot \mathcal{E}_{\mathsf{head}}(4, 2) \cdot \mathcal{E}_{\mathbb{Z}}(i) & \text{otherwise.} \end{cases}$$

In the final case of $\mathcal{E}_{\mathtt{ctag}}$ we emit a head with major type 4 and argument 2. This indicates that an encoding of a list of length 2 will follow: the first element of the list is the constructor number and the second is the argument list of the constructor, which is actually encoded in $\mathcal{E}_{\mathtt{data}}$. It might be conceptually more accurate to have a single encoder which would encode both the constructor tag and the argument list, but this would increase the complexity of the notation even further. Similar remarks apply to $\mathcal{D}_{\mathtt{ctag}}$ below.

The decoder. The decoder is given by

$$\mathcal{D}_{\mathtt{data}}(s) = \begin{cases} (s'', \mathtt{Map}\ l) & \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 5, n) \text{ and } \mathcal{D}^{(n)}_{(\mathtt{data}^2)^*}(s') = (s'', l) \\ (s', \mathtt{List}\ l) & \text{if } \mathcal{D}_{\mathtt{data}^*}(s) = (s', l) \\ (s'', \mathtt{Constr}\ i\ l) & \text{if } \mathcal{D}_{\mathtt{ctag}}(s) = (s', i) \text{ and } \mathcal{D}_{\mathtt{data}^*}(s') = (s'', l) \\ (s', \mathtt{I}\ n) & \text{if } \mathcal{D}_{\mathbb{Z}}(s) = (s', n) \\ (s', \mathtt{B}\ b) & \text{if } \mathcal{D}_{\mathbb{B}^*}(s) = (s', b) \end{cases}$$

where

$$\mathcal{D}_{\mathtt{data}^*}(s) = \begin{cases} \mathcal{D}^{(n)}_{\mathtt{data}^*}(s') & \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 4, n) \\ \mathcal{D}^{\mathsf{indef}}_{\mathtt{data}^*}(s') & \text{if } \mathcal{D}_{\mathsf{indef}}(s) = (s', 4) \end{cases}$$

$$\mathcal{D}^{(n)}_{\mathtt{data}^*}(s) = \begin{cases} (s, \epsilon) & \text{if } n = 0 \\ (s'', d \cdot l) & \text{if } \mathcal{D}_{\mathtt{data}}(s) = (s', d) \text{ and } \mathcal{D}^{(n-1)}_{\mathtt{data}^*}(s') = (s'', l) \end{cases}$$

$$\mathcal{D}^{\mathsf{indef}}_{\mathtt{data}^*}(s) = \begin{cases} (s', \epsilon) & \text{if } s = 255 \cdot s' \\ (s'', d \cdot l) & \text{if } \mathcal{D}_{\mathtt{data}}(s) = (s', d) \text{ and } \mathcal{D}^{\mathsf{indef}}_{\mathtt{data}^*}(s') = (s'', l) \end{cases}$$

$$\mathcal{D}^{(n)}_{(\mathtt{data}^2)^*}(s) = \begin{cases} (s, \epsilon) & \text{if } n = 0 \\ (s''', (k, d) \cdot l) & \begin{cases} \text{if } n > 0 \\ \text{and } \mathcal{D}_{\mathtt{data}}(s) = (s', k) \\ \text{and } \mathcal{D}_{\mathtt{data}}(s') = (s'', d) \\ \text{and } \mathcal{D}^{(n-1)}_{(\mathtt{data}^2)^*}(s'') = (s''', l) \end{cases} \end{cases}$$

$$\mathcal{D}_{\mathtt{ctag}}(s) = \begin{cases} (s', i - 121) & \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 6, i) \text{ and } 121 \le i \le 127 \\ (s', (i - 1280) + 7) & \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 6, i) \text{ and } 1280 \le i \le 1400 \\ (s''', i) & \begin{cases} \text{if } \mathcal{D}_{\mathsf{head}}(s) = (s', 6, 102) \\ \text{and } \mathcal{D}_{\mathsf{head}}(s') = (s'', 4, 2) \\ \text{and } \mathcal{D}_{\mathbb{Z}}(s'') = (s''', i) \\ \text{and } 0 \le i \le 2^{64} - 1. \end{cases} \end{cases}$$

Note that the decoders for `List` and `Constr` accept both definite-length and indefinite-length lists of encoded `data` values, but the decoder for `Map` only accepts definite-length lists (and the length is the number of *pairs* in the map). This is consistent with CBOR's standard encoding of arrays and lists (major type 4) and maps (major type 5).

Note also that the encoder $\mathcal{E}_{\texttt{ctag}}$ accepts arbitrary integer values for `Constr` tags, but (for compatibility with [22]) the decoder $\mathcal{D}_{\texttt{ctag}}$ only accepts tags in $\mathbb{N}_{[0,2^{64}-1]}$. This means that some valid Plutus Core programs can be serialised but not deserialised, and is the reason for the recommendation in Section 4.3.1.1 that only constructor tags between 0 and $2^{64} - 1$ should be used.

Appendix C

Serialising Plutus Core Terms and Programs Using the `flat` Format

We use the `flat` format [11] to serialise Plutus Core terms, and we regard this format as being the definitive concrete representation of Plutus Core programs. For compactness we generally (and *always* for scripts on the blockchain) replace names with de Bruijn indices (see Section 2.1.3) in serialised programs.

We use bytestrings for serialisation, but it is convenient to define the serialisation and deserialisation process in terms of strings of bits. Some extra bits of padding are added at the end of the encoding of a program to ensure that the number of bits in the output is a multiple of 8, and this allows us to regard serialised programs as bytestrings in the obvious way.

See Section C.4 for some restrictions on serialisation specific to the Cardano blockchain.

Note: `flat` versus CBOR. Much of the Cardano codebase uses the CBOR format for serialisation; however, it is important that serialised scripts not be too large. CBOR pays a price for being a self-describing format. The size of the serialised terms is consistently larger than a format that is not self-describing: benchmarks show that `flat` encodings of Plutus Core scripts are smaller than CBOR encodings by about 35% (without using compression).

C.1 Encoding and decoding

Firstly recall some notation from Section 1.2. The set of all finite sequences of bits is denoted by $\mathsf{b}^* = \{0,1\}^*$. For brevity we write a sequence of bits in the form $b_{n-1}\cdots b_0$ instead of $[b_{n-1},\ldots,b_0]$: thus `011001` instead of $[0,1,1,0,0,1]$). We denote the empty sequence by ϵ, and use $\ell(s)$ to denote the length of a sequence of bits, and $\cdot$ to denote concatenation (or prepending or appending a single bit to a sequence of bits).

Similarly to the CBOR encoding for `data` described in Appendix B, we will describe the flat encoding by defining families of encoding functions (or *encoders*)

$$\mathsf{E}_X : \mathsf{b}^* \times X \to \mathsf{b}^*$$

and (partial) decoding functions (or *decoders*)

$$\mathsf{D}_X : \mathsf{b}^* \rightharpoonup \mathsf{b}^* \times X$$

for various sets X, such as the set $\mathbb{Z}$ of integers and the set of all Plutus Core terms. The encoding function E_X takes a sequence $s \in \mathsf{b}^*$ and an element $x \in X$ and produces a new sequence of bits by appending the encoding of x to s, and the decoding function D_X takes a sequence of bits, decodes some initial prefix of s to a value $x \in X$, and returns the remainder of s together with x.

Encoding functions basically operate by decomposing an object into subobjects and concatenating the encodings of the subobject; however it is sometimes necessary to add some padding between subobjects in order to make sure that parts of the output are aligned on byte boundaries, and for this reason (unlike the CBOR encoding for `data`) all of our encoding functions have a first argument containing all of the previous output, so that it can be examined to determine how much alignment is required.

As in the case of CBOR, decoding functions are partial: they can fail if, for instance, there is insufficient input, or if a decoded value is outside some specified range. To simplify notation we will mention any preconditions separately, with the assumption that the decoder will fail if the preconditions are not met; we also make a blanket assumption that all decoders fail if there is not enough input for them to proceed. Many of the definitions of decoders construct objects by calling other decoders to obtain subobjects which are then composed, and these are often introduced by a condition of the form "if $\mathsf{D}_X(s) = x$". Conditions like this should be read as implicitly saying that if the decoder D_X fails then the whole decoding process fails.

C.1.1 Padding

The encoding functions mentioned above produce sequences of *bits*, but we sometimes need sequences of *bytes*. To this end we introduce a functions $\mathsf{pad} : \mathsf{b}^* \to \mathsf{b}^*$ which adds a sequence of 0s followed by a 1 to a sequence s to get a sequence whose length is a multiple of 8; if s is a sequence such that $\ell(s)$ is already a multiple of 8 then pad still adds an extra byte of padding; pad is used both for internal alignment (for example, to make sure that the contents of a bytestring are aligned on byte boundaries) and at the end of a complete encoding of a Plutus Core program to to make the length a multiple of 8 bits. Symbolically,

$$\mathsf{pad}(s) = s \cdot \mathsf{p}_k \quad \text{if } \ell(s) = 8n + k \text{ with } n, k \in \mathbb{N} \text{ and } 0 \leq k \leq 7$$

where

$$\begin{aligned}
\mathsf{p}_0 &= \mathtt{00000001} \\
\mathsf{p}_1 &= \mathtt{0000001} \\
\mathsf{p}_2 &= \mathtt{000001} \\
\mathsf{p}_3 &= \mathtt{00001} \\
\mathsf{p}_4 &= \mathtt{0001} \\
\mathsf{p}_5 &= \mathtt{001} \\
\mathsf{p}_6 &= \mathtt{01} \\
\mathsf{p}_7 &= \mathtt{1}.
\end{aligned}$$

We also define a (partial) inverse function $\mathsf{unpad} : \mathsf{b}^* \rightharpoonup \mathsf{b}^*$ which discards padding:

$$\mathsf{unpad}(q \cdot s) = s \quad \text{if } q = \mathsf{p}_i \text{ for some } i \in \{0, 1, 2, 3, 4, 5, 6, 7\}.$$

This can fail if the padding is not of the expected form or if the input is the empty sequence ϵ.

C.2 Basic `flat` encodings

C.2.1 Fixed-width natural numbers

We often wish to encode and decode natural numbers which fit into some fixed number of bits, and we do this simply by encoding them as their binary expansion (most significant bit first), adding leading zeros if necessary. More precisely for $n \geq 1$ we define an encoder

$$\mathsf{E}_n : \mathsf{b}^* \times \mathbb{N}_{[0,2^{n-1}-1]} \to \mathsf{b}^*$$

by

$$\mathsf{E}_n(s, \sum_{i=0}^{n-1} b_i 2^i) = s \cdot b_{n-1} \cdots b_0 \quad (b_i \in \{0, 1\})$$

and a decoder

$$\mathsf{D}_n : \mathsf{b}^* \rightharpoonup \mathsf{b}^* \times \mathbb{N}_{[0,2^{n-1}-1]}$$

by

$$\mathsf{D}_n(b_{n-1} \cdots b_0 \cdot s) = (s, \sum_{i=0}^{n-1} b_i 2^i).$$

As in Appendix B, $\mathbb{N}_{[a,b]}$ denotes the closed interval of integers $\{n \in \mathbb{Z} : a \leq n \leq b\}$. Note that n here is a variable (not a fixed label) so we are defining whole families of encoders $\mathsf{E}_1, \mathsf{E}_2, \mathsf{E}_3, \ldots$ and and decoders $\mathsf{D}_1, \mathsf{D}_2, \mathsf{D}_3 \ldots$.

C.2.2 Lists

Suppose that we have a set X for which we have defined an encoder E_X and a decoder D_X; we define an encoder $\vec{\mathsf{E}}_X$ which encodes lists of elements of X by emitting the encodings of the elements of the list, each preceded by a `1` bit, then emitting a `0` bit to mark the end of the list.

$$\vec{\mathsf{E}}_X(s, []) = s \cdot 0$$
$$\vec{\mathsf{E}}_X(s, [x_1, \ldots, x_n]) = \vec{\mathsf{E}}_X(s \cdot 1 \cdot \mathsf{E}_X(x_1), [x_2, \ldots, x_n]).$$

The corresponding decoder is given by

$$\vec{\mathsf{D}}_X(0 \cdot s) = (s, [])$$
$$\vec{\mathsf{D}}_X(1 \cdot s) = (s'', x \cdot l) \quad \text{if } D_X(s) = (s', x) \text{ and } \vec{\mathsf{D}}_X(s') = (s'', l).$$

C.2.3 Natural numbers

We encode natural numbers by splitting their binary representations into sequences of 7-bit blocks, then emitting these as a list with the **least significant block first**:

$$\mathsf{E}_{\mathbb{N}}(s, \sum_{i=0}^{n-1} k_i 2^{7i}) = \vec{\mathsf{E}}_7(s, [k_0, \ldots, k_{n-1}])$$

(where $k_i \in \mathbb{Z}$ and $0 \leq k_i \leq 127$). The decoder is

$$\mathsf{D}_{\mathbb{N}}(s) = (s', \sum_{i=0}^{n-1} k_i 2^{7i}) \quad \text{if } \vec{\mathsf{D}}_7(s) = (s', [k_0, \ldots, k_{n-1}]).$$

C.2.4 Integers

Signed integers are encoded by converting them to natural numbers using the zigzag encoding ($0 \mapsto 0, -1 \mapsto 1, 1 \mapsto 2, -2 \mapsto 3, 2 \mapsto 4, \ldots$) and then encoding the result using $\mathsf{E}_{\mathbb{N}}$:

$$\mathsf{E}_{\mathbb{Z}}(s, n) = \begin{cases} \mathsf{E}_{\mathbb{N}}(s, 2n) & \text{if } n \geq 0 \\ \mathsf{E}_{\mathbb{N}}(s, -2n - 1) & \text{if } n < 0. \end{cases}$$

The decoder is

$$\mathsf{D}_{\mathbb{Z}}(s) = \begin{cases} (s', \frac{n}{2}) & \text{if } n \equiv 0 \pmod 2 \\ (s', -\frac{n+1}{2}) & \text{if } n \equiv 1 \pmod 2 \end{cases} \quad \text{if } \mathsf{D}_{\mathbb{N}}(s) = (s', n).$$

C.2.5 Bytestrings

Bytestrings are encoded by dividing them into nonempty blocks of up to 255 bytes and emitting each block in sequence. Each block is preceded by a single unsigned byte containing its length, and the end of the encoding is marked by a zero-length block (so the empty bytestring is encoded just as a zero-length block). Before emitting a bytestring, the preceding output is padded so that its length (in bits) is a multiple of 8; if this is already the case a single padding byte is still added; this ensures that contents of the bytestring are aligned to byte boundaries in the output.

Recall that $\mathbb{B}$ denotes the set of 8-bit bytes, $\{0, 1, \ldots, 255\}$. For specification purposes we may identify the set of bytestrings with the set $\mathbb{B}^*$ of (possibly empty) lists of elements of $\mathbb{B}$. We denote by C the set of *bytestring chunks* of **nonempty** bytestrings of length at most 255: $C = \{[b_1, \ldots, b_n] : b_i \in \mathbb{B}, 1 \leq n \leq 255\}$, and define a function $E_C : C \to \mathsf{b}^*$ by

$$E_C([b_1, \ldots, b_n]) = \mathsf{E}_8(n) \cdot \mathsf{E}_8(b_1) \cdot \cdots \cdot \mathsf{E}_8(b_n).$$

We define an encoder E_{C^*} for lists of chunks by

$$\mathsf{E}_{C^*}(s, [c_1, \ldots, c_n]) = s \cdot E_C(c_1) \cdot \cdots \cdot E_C(c_n) \cdot \mathtt{00000000}.$$

Note that each c_i is required to be nonempty but that we allow the case $n = 0$, so that an empty list of chunks encodes as `00000000`.

To encode a bytestring we decompose it into a list L of chunks and then apply E_{C^*} to L. However, there will usually be many ways to decompose a given bytestring a into chunks. For definiteness we recommend (but do not demand) that a is decomposed into a sequence of chunks of length 255 possibly followed by a smaller chunk. Formally, suppose that $a = [a_1, \ldots, a_{255k+r}] \in \mathbb{B}^* \backslash \{\epsilon\}$ where $k \geq 0$ and $0 \leq r \leq 254$. We define the *canonical 256-byte decomposition* $\tilde{a}$ of a to be

$$\tilde{a} = [[a_1, \ldots, a_{255}], [a_{256}, \ldots, a_{510}], \ldots [a_{255(k-1)+1}, \ldots, a_{255k}]] \in C^*$$

if $r = 0$ and

$$\tilde{a} = [[a_1, \ldots, a_{255}], [a_{256}, \ldots, a_{510}], \ldots [a_{255(k-1)+1}, \ldots, a_{255k}], [a_{255k+1}, \ldots, a_{255k+r}]] \in C^*$$

if $r > 0$.

For the empty bytestring we define

$$\tilde{\epsilon} = [].$$

Given all of the above, we define the canonical encoding function $\mathsf{E}_{\mathbb{B}^*}$ for bytestrings to be

$$\mathsf{E}_{\mathbb{B}^*}(s, a) = E_{C^*}(\mathsf{pad}(s), \tilde{a}).$$

Non-canonical encodings can be obtained by replacing $\tilde{a}$ with any other decomposition of a into nonempty chunks, and the decoder below will accept these as well.

To define a decoder for bytestrings we first define a decoder D_C for bytestring chunks:

$$\mathsf{D}_C(s) = \mathsf{D}_C^{(n)}(s', []) \quad \text{if } \mathsf{D}_8(s) = (s', n)$$

where

$$\mathsf{D}_C^{(n)}(s, l) = \begin{cases} (s, l) & \text{if } n = 0 \\ \mathsf{D}_C^{(n-1)}(s', l \cdot x) & \text{if } n > 0 \text{ and } \mathsf{D}_8(s) = (s', x). \end{cases}$$

Now we define

$$\mathsf{D}_{C^*}(s) = \begin{cases} (s', []) & \text{if } D_C(s) = (s', []) \\ (s'', x \cdot l) & \text{if } \mathsf{D}_C(s) = (s', x) \text{ with } x \neq [] \text{ and } \mathsf{D}_{C^*}(s') = (s'', l). \end{cases}$$

The notation is slightly misleading here: D_{C^*} does not decode to a list of bytestring chunks, but to a single bytestring. We could alternatively decode to a list of bytestrings and then concatenate them later, but this would have the same overall effect.

Finally, we define the decoder for bytestrings by

$$\mathsf{D}_{\mathbb{B}^*}(s) = \mathsf{D}_{C^*}(\mathsf{unpad}(s)).$$

C.2.6 Strings

We have defined values of the `string` type to be sequences of Unicode characters. As mentioned earlier we do not specify any particular internal representation of Unicode characters, but for serialisation we use the UTF-8 representation to convert between strings and bytestrings and then use the bytestring encoder and decoder:

$$\mathsf{E}_{\mathbb{U}^*}(s, u) = \mathsf{E}_{\mathbb{B}^*}(s, \mathsf{utf8}(u))$$

$$\mathsf{D}_{\mathbb{U}^*}(s) = (s', \mathsf{utf8}^{-1}(a)) \quad \text{if } \mathsf{D}_{\mathbb{B}^*}(s) = (s', a)$$

where $\mathsf{utf8}$ and $\mathsf{utf8}^{-1}$ are the UTF8 encoding and decoding functions mentioned in Section 4.3.1. Recall that $\mathsf{utf8}^{-1}$ is partial (not all bytestrings represent valid Unicode sequences), so $\mathsf{D}_{\mathbb{U}^*}$ may fail if the input is invalid.

C.3 Encoding and decoding Plutus Core

C.3.1 Programs

A program is encoded by encoding the three components of the version number in sequence then encoding the body, and possibly adding some padding to ensure that the total number of bits in the output is a multiple of 8 (and hence the output can be viewed as a bytestring).

$$\mathsf{E}_{\mathsf{program}}((\texttt{program}\ a.b.c\ t)) = \mathsf{pad}(\mathsf{E}_{\mathsf{term}}(\mathsf{E}_{\mathbb{N}}(\mathsf{E}_{\mathbb{N}}(\mathsf{E}_{\mathbb{N}}(\epsilon, a), b), c), t)).$$

The decoding process is the inverse of the encoding process: three natural numbers are read to obtain the version number and then the body is decoded. After this we discard any padding in the remaining input and check that all of the input has been consumed.

$$\mathsf{D}_{\mathsf{program}}(s) = (\texttt{program}\ a.b.c\ t) \quad \begin{cases} \text{if} & \mathsf{D}_{\mathbb{N}}(s) = (s', a) \\ \text{and} & \mathsf{D}_{\mathbb{N}}(s') = (s'', b) \\ \text{and} & \mathsf{D}_{\mathbb{N}}(s'') = (s''', c) \\ \text{and} & \mathsf{D}_{\mathsf{term}}(s''') = (r, t) \\ \text{and} & \mathsf{unpad}(r) = \epsilon. \end{cases}$$

C.3.2 Terms

Plutus Core terms are encoded by emitting a 4-bit tag identifying the type of the term (see Table C.1; recall that `[]` denotes application) then emitting the encodings for any subterms. We currently only use ten of the sixteen available tags: the remainder are reserved for potential future expansion.

Term type	Binary	Decimal
Variable	0000	0
`delay`	0001	1
`lam`	0010	2
`[]`	0011	3
`const`	0100	4
`force`	0101	5
`error`	0110	6
`builtin`	0111	7
`constr`	1000	8
`case`	1001	9

Table C.1: Term tags

The encoder for terms is given below: it refers to other encoders (for names, types, and constants) which will be defined later.

$$\begin{aligned}
\mathsf{E}_{\mathsf{term}}(s, x) &= \mathsf{E}_{\mathsf{name}}(s \cdot 0000, x) \\
\mathsf{E}_{\mathsf{term}}(s, (\texttt{delay}\ t)) &= \mathsf{E}_{\mathsf{term}}(s \cdot 0001, t) \\
\mathsf{E}_{\mathsf{term}}(s, (\texttt{lam}\ x\ t)) &= \mathsf{E}_{\mathsf{term}}(\mathsf{E}_{\lambda\mathsf{var}}(s \cdot 0010, x), t) \\
\mathsf{E}_{\mathsf{term}}(s, [t_1\ t_2]) &= \mathsf{E}_{\mathsf{term}}(\mathsf{E}_{\mathsf{term}}(s \cdot 0011, t_1), t_2) \\
\mathsf{E}_{\mathsf{term}}(s, (\texttt{const}\ tn\ c)) &= \mathsf{E}^{tn}_{\mathsf{constant}}(\mathsf{E}_{\mathsf{type}}(s \cdot 0100, T), c) \\
\mathsf{E}_{\mathsf{term}}(s, (\texttt{force}\ t)) &= \mathsf{E}_{\mathsf{term}}(s \cdot 0101, t) \\
\mathsf{E}_{\mathsf{term}}(s, (\texttt{error})) &= s \cdot 0110 \\
\mathsf{E}_{\mathsf{term}}(s, (\texttt{builtin}\ b)) &= \mathsf{E}_{\mathsf{builtin}}(s \cdot 0111, b) \\
\mathsf{E}_{\mathsf{term}}(s, (\texttt{constr}\ i\ l)) &= \vec{\mathsf{E}}_{\mathsf{term}}(\mathsf{E}_{64}(s \cdot 1000, i), l) \\
\mathsf{E}_{\mathsf{term}}(s, (\texttt{case}\ u\ l)) &= \vec{\mathsf{E}}_{\mathsf{term}}(\mathsf{E}_{\mathsf{term}}(s \cdot 1001, u), l)
\end{aligned}$$

The decoder for terms is given below. To simplify the definition we use some pattern-matching syntax for inputs to decoders: for example the argument $\texttt{0101} \cdot s$ indicates that when the input is a string beginning with `0101` the definition after the = sign should be used (and the remainder of the input is available in s there). If the input is not long enough to permit the indicated decomposition then the decoder fails. The decoder also fails if the input begins with a prefix which is not listed; that does not happen here, but does in some later decoders.

$$
\begin{array}{lll}
\mathsf{D}_{\mathsf{term}}(\texttt{0000} \cdot s) = (s', x) & \text{if } \mathsf{D}_{\mathsf{name}}(s) = (s', x) & \\
\mathsf{D}_{\mathsf{term}}(\texttt{0001} \cdot s) = (s', (\texttt{delay}\ t)) & \text{if } \mathsf{D}_{\mathsf{term}}(s) = (s', t) & \\
\mathsf{D}_{\mathsf{term}}(\texttt{0010} \cdot s) = (s'', (\texttt{lam}\ x\ t)) & \text{if } \mathsf{D}_{\lambda\mathsf{var}}(s) = (s', x) & \text{and } \mathsf{D}_{\mathsf{term}}(s') = (s'', t) \\
\mathsf{D}_{\mathsf{term}}(\texttt{0011} \cdot s) = (s'', [t_1\ t_2]) & \text{if } \mathsf{D}_{\mathsf{term}}(s) = (s', t_1) & \text{and } \mathsf{D}_{\mathsf{term}}(s') = (s'', t_2) \\
\mathsf{D}_{\mathsf{term}}(\texttt{0100} \cdot s) = (s'', (\texttt{const}\ tn\ c)) & \text{if } \mathsf{D}_{\mathsf{type}}(s) = (s', T) & \text{and } \mathsf{D}^{T}_{\mathsf{constant}}(s') = (s'', c) \\
\mathsf{D}_{\mathsf{term}}(\texttt{0101} \cdot s) = (s', (\texttt{force}\ t)) & \text{if } \mathsf{D}_{\mathsf{term}}(s) = (s', t) & \\
\mathsf{D}_{\mathsf{term}}(\texttt{0110} \cdot s) = (s, (\texttt{error})) & & \\
\mathsf{D}_{\mathsf{term}}(\texttt{0111} \cdot s) = (s', b) & \text{if } \mathsf{D}_{\mathsf{builtin}}(s) = (s', b) & \\
\mathsf{D}_{\mathsf{term}}(\texttt{1000} \cdot s) = (s', (\texttt{constr}\ i\ l)) & \text{if } \mathsf{D}_{64}(s) = (s', i) & \text{and } \vec{\mathsf{D}}_{\mathsf{term}}(s') = (s'', l) \\
\mathsf{D}_{\mathsf{term}}(\texttt{1001} \cdot s) = (s', (\texttt{case}\ u\ l)) & \text{if } \mathsf{D}_{\mathsf{term}}(s) = (s', u) & \text{and } \vec{\mathsf{D}}_{\mathsf{term}}(s') = (s'', l)
\end{array}
$$

NOTE. The decoder $\mathsf{D}_{\mathsf{term}}$ should fail if we are decoding a program with a version less than 1.1.0 and an input of the form $\texttt{1000} \cdot s$ or $\texttt{1001} \cdot s$ is encountered.

C.3.3 Built-in types

Constants from built-in types are essentially encoded by emitting a sequence of 4-bit tags representing the constant's type and then emitting the encoding of the constant itself. However the encoding of types is somewhat complex because it has to be able to deal with type operators such as `list` and `pair`. The tags are given in Table C.2: they include tags for the basic types together with a tag for a type application operator.

Type	Binary	Decimal
`integer`	0000	0
`bytestring`	0001	1
`string`	0010	2
`unit`	0011	3
`bool`	0100	4
`list`	0101	5
`pair`	0110	6
(type application)	0111	7
`data`	1000	8
`bls12_381_G1_element`	1001	9
`bls12_381_G2_element`	1010	10
`bls12_381_MlResult`	1011	11

Table C.2: Type tags

We define auxiliary functions $\mathsf{e}_{\mathsf{type}} : \mathcal{U} \to \mathbb{N}^*$ and $\mathsf{d}_{\mathsf{type}} : \mathbb{N}^* \rightharpoonup \mathbb{N}^* \times \mathcal{U}$ ($\mathsf{d}_{\mathsf{type}}$ is partial and $\mathcal{U}$ denotes the universe of types defined in Sections 4.3.1, 4.3.2, and 4.3.3).

$$
\begin{aligned}
\mathsf{e}_{\mathsf{type}}(\texttt{integer}) &= [0] \\
\mathsf{e}_{\mathsf{type}}(\texttt{bytestring}) &= [1] \\
\mathsf{e}_{\mathsf{type}}(\texttt{string}) &= [2] \\
\mathsf{e}_{\mathsf{type}}(\texttt{unit}) &= [3] \\
\mathsf{e}_{\mathsf{type}}(\texttt{bool}) &= [4] \\
\mathsf{e}_{\mathsf{type}}(\texttt{list}\,(t)) &= [7,5] \cdot \mathsf{e}_{\mathsf{type}}(t) \\
\mathsf{e}_{\mathsf{type}}(\texttt{pair}\,(t_1,t_2)) &= [7,7,6] \cdot \mathsf{e}_{\mathsf{type}}(t_1) \cdot \mathsf{e}_{\mathsf{type}}(t_2) \\
\mathsf{e}_{\mathsf{type}}(\texttt{data}) &= [8].
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{d}_{\mathsf{type}}(0 \cdot l) &= (l, \texttt{integer}) \\
\mathsf{d}_{\mathsf{type}}(1 \cdot l) &= (l, \texttt{bytestring}) \\
\mathsf{d}_{\mathsf{type}}(2 \cdot l) &= (l, \texttt{string})) \\
\mathsf{d}_{\mathsf{type}}(3 \cdot l) &= (l, \texttt{unit}) \\
\mathsf{d}_{\mathsf{type}}(4 \cdot l) &= (l, \texttt{bool}) \\
\mathsf{d}_{\mathsf{type}}([7,5] \cdot l) &= (l', \texttt{list}\,(t)) && \text{if } \mathsf{d}_{\mathsf{type}}(l) = (l', t) \\
\mathsf{d}_{\mathsf{type}}([7,7,6] \cdot l) &= (l'', \texttt{pair}\,(t_1,t_2)) && \begin{cases} \text{if} & \mathsf{d}_{\mathsf{type}}(l) = (l', t_1) \\ \text{and} & \mathsf{d}_{\mathsf{type}}(l') = (l'', t_2) \end{cases} \\
\mathsf{d}_{\mathsf{type}}(8 \cdot l) &= (l, \texttt{data}).
\end{aligned}
$$

The encoder and decoder for types is obtained by combining $\mathsf{e}_{\mathsf{type}}$ and $\mathsf{d}_{\mathsf{type}}$ with $\vec{\mathsf{E}}_4$ and $\vec{\mathsf{D}}_4$, the encoder and decoder for lists of four-bit integers (see Section C.2).

$$\mathsf{E}_{\mathsf{type}}(s,t) = \vec{\mathsf{E}}_4(s, \mathsf{e}_{\mathsf{type}}(t))$$

$$\mathsf{D}_{\mathsf{type}}(s) = (s', t) \quad \text{if } \vec{\mathsf{D}}_4(s) = (s', l) \text{ and } \mathsf{d}_{\mathsf{type}}(l) = ([], t).$$

C.3.4 Constants

Values of built-in types can mostly be encoded quite simply by using encoders already defined. Note that the unit value `(con unit ())` does not have an explicit encoding: the type has only one possible value, so there is no need to use any space to serialise it.

The `data` type is encoded by converting to a bytestring using the CBOR encoder $\mathcal{E}_{\texttt{data}}$ described in Appendix B and then using $\mathsf{E}_{\mathbb{B}^*}$. The decoding process is the opposite of this: a bytestring is obtained using $\mathsf{D}_{\mathbb{B}^*}$ and this is then decoded from CBOR using $\mathcal{D}_{\texttt{data}}$ to obtain a `data` object.

We do not provide serialisation and deserialisation for constants of type `bls12_381_G1_element`, `bls12_381_G2_element`, or `bls12_381_mlresult`. We have specified tags for these types, but if one of these tags is encountered during deserialisation then deserialisation fails and any subsequent input is ignored. Note however that constants of the first two types can be serialised by using the compression

functions defined in Section 4.3.4.3 and serialising the resulting bytestrings. Decoding can similarly be performed indirectly by using `bls12_381_G1_uncompress` and `bls12_381_G2_uncompress` on bytestring constants during program execution.

$$
\begin{aligned}
\mathsf{E}^{\mathtt{integer}}_{\mathsf{constant}}(s,n) &= \mathsf{E}_{\mathbb{Z}}(s,n)\\
\mathsf{E}^{\mathtt{bytestring}}_{\mathsf{constant}}(s,a) &= \mathsf{E}_{\mathbb{B}^*}(s,a)\\
\mathsf{E}^{\mathtt{string}}_{\mathsf{constant}}(s,t) &= \mathsf{E}_{\mathbb{U}^*}(s,t)\\
\mathsf{E}^{\mathtt{unit}}_{\mathsf{constant}}(s,c) &= s\\
\mathsf{E}^{\mathtt{bool}}_{\mathsf{constant}}(s,\mathtt{False}) &= s\cdot 0\\
\mathsf{E}^{\mathtt{bool}}_{\mathsf{constant}}(s,\mathtt{True}) &= s\cdot 1\\
\mathsf{E}^{\mathtt{list}(T)}_{\mathsf{constant}}(s,l) &= \vec{\mathsf{E}}^{T}_{\mathsf{constant}}(s,l)\\
\mathsf{E}^{\mathtt{pair}(T_1,T_2)}_{\mathsf{constant}}(s,(c_1,c_2)) &= \mathsf{E}^{T_2}_{\mathsf{constant}}(\mathsf{E}^{T_1}_{\mathsf{constant}}(s,c_1),c_2)\\
\mathsf{E}^{\mathtt{data}}_{\mathsf{constant}}(s,d) &= \mathsf{E}_{\mathbb{B}^*}(s,\mathcal{E}_{\mathtt{data}}(d))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{D}^{\mathtt{integer}}_{\mathsf{constant}}(s) &= \mathsf{D}_{\mathbb{Z}}(s)\\
\mathsf{D}^{\mathtt{bytestring}}_{\mathsf{constant}}(s) &= \mathsf{D}_{\mathbb{B}^*}(s)\\
\mathsf{D}^{\mathtt{string}}_{\mathsf{constant}}(s) &= \mathsf{D}_{\mathbb{U}^*}(s)\\
\mathsf{D}^{\mathtt{unit}}_{\mathsf{constant}}(s) &= s\\
\mathsf{D}^{\mathtt{bool}}_{\mathsf{constant}}(0\cdot s) &= (s,\mathtt{False})\\
\mathsf{D}^{\mathtt{bool}}_{\mathsf{constant}}(1\cdot s) &= (s,\mathtt{True})\\
\mathsf{D}^{\mathtt{list}(T)}_{\mathsf{constant}}(s) &= \vec{\mathsf{D}}^{T}_{\mathsf{constant}}(s,l)\\
\mathsf{D}^{\mathtt{pair}(T_1,T_2)}_{\mathsf{constant}}(s) &= (s'',(c_1,c_2)) \begin{cases} \text{if} & \mathsf{D}^{T_1}_{\mathsf{constant}}(s) = (s',c_1)\\ \text{and} & \mathsf{D}^{T_2}_{\mathsf{constant}}(s') = (s'',c_2)\end{cases}\\
\mathsf{D}^{\mathtt{data}}_{\mathsf{constant}}(s) &= (s',d) \qquad \text{if } \mathsf{D}_{\mathbb{B}^*}(s) = (s',t) \text{ and } \mathcal{D}_{\mathtt{data}}(t) = (t',d) \text{ for some } t'
\end{aligned}
$$

C.3.5 Built-in functions

Built-in functions are represented by seven-bit integer tags and encoded and decoded using E_7 and D_7. The tags are specified in Tables C.3–C.7. We assume that there are (partial) functions tag and tag^{-1} which convert back and forth between builtin names and their tags.

$$
\begin{aligned}
\mathsf{E}_{\mathsf{builtin}}(s,b) &= \mathsf{E}_7(s,\mathsf{tag}(b))\\
\mathsf{D}_{\mathsf{builtin}}(s) &= (s',\mathsf{tag}^{-1}(n)) \quad \text{if } \mathsf{D}_7(s) = (s',n).
\end{aligned}
$$

Builtin	Binary	Decimal	Builtin	Binary	Decimal
addInteger	0000000	0	ifThenElse	0011010	26
subtractInteger	0000001	1	chooseUnit	0011011	27
multiplyInteger	0000010	2	trace	0011100	28
divideInteger	0000011	3	fstPair	0011101	29
quotientInteger	0000100	4	sndPair	0011110	30
remainderInteger	0000101	5	chooseList	0011111	31
modInteger	0000110	6	mkCons	0100000	32
equalsInteger	0000111	7	headList	0100001	33
lessThanInteger	0001000	8	tailList	0100010	34
lessThanEqualsInteger	0001001	9	nullList	0100011	35
appendByteString	0001010	10	chooseData	0100100	36
consByteString	0001011	11	constrData	0100101	37
sliceByteString	0001100	12	mapData	0100110	38
lengthOfByteString	0001101	13	listData	0100111	39
indexByteString	0001110	14	iData	0101000	40
equalsByteString	0001111	15	bData	0101001	41
lessThanByteString	0010000	16	unConstrData	0101010	42
lessThanEqualsByteString	0010001	17	unMapData	0101011	43
sha2_256	0010010	18	unListData	0101100	44
sha3_256	0010011	19	unIData	0101101	45
blake2b_256	0010100	20	unBData	0101110	46
verifyEd25519Signature	0010101	21	equalsData	0101111	47
appendString	0010110	22	mkPairData	0110000	48
equalsString	0010111	23	mkNilData	0110001	49
encodeUtf8	0011000	24	mkNilPairData	0110010	50
decodeUtf8	0011001	25			

Table C.3: Tags for built-in functions (batch 1)

Builtin	Binary	Decimal
serialiseData	0110011	51

Table C.4: Tags for built-in functions (batch 2)

Builtin	Binary	Decimal
verifyEcdsaSecp256k1Signature	0110100	52
verifySchnorrSecp256k1Signature	0110101	53

Table C.5: Tags for built-in functions (batch 3)

Builtin	Binary	Decimal
`bls12_381_G1_add`	0110110	54
`bls12_381_G1_neg`	0110111	55
`bls12_381_G1_scalarMul`	0111000	56
`bls12_381_G1_equal`	0111001	57
`bls12_381_G1_hashToGroup`	0111010	58
`bls12_381_G1_compress`	0111011	59
`bls12_381_G1_uncompress`	0111100	60
`bls12_381_G2_add`	0111101	61
`bls12_381_G2_neg`	0111110	62
`bls12_381_G2_scalarMul`	0111111	63
`bls12_381_G2_equal`	1000000	64
`bls12_381_G2_hashToGroup`	1000001	65
`bls12_381_G2_compress`	1000010	66
`bls12_381_G2_uncompress`	1000011	67
`bls12_381_millerLoop`	1000100	68
`bls12_381_mulMlResult`	1000101	69
`bls12_381_finalVerify`	1000110	70
`keccak_256`	1000111	71
`blake2b_224`	1001000	72
`integerToByteString`	1001000	73
`byteStringToInteger`	1001000	74

Table C.6: Tags for built-in functions (batch 4)

Builtin	Binary	Decimal
`andByteString`	1001011	75
`orByteString`	1001100	76
`xorByteString`	1001101	77
`complementByteString`	1001110	78
`readBit`	1001111	79
`writeBits`	1010000	80
`replicateByte`	1010001	81
`shiftByteString`	1010010	82
`rotateByteString`	1010011	83
`countSetBits`	1010100	84
`findFirstSetBit`	1010101	85
`ripemd_160`	1010110	86

Table C.7: Tags for built-in functions (batch 5)

C.3.6 Variable names

Variable names are encoded and decoded using the $\mathsf{E}_{\mathsf{name}}$ and $\mathsf{D}_{\mathsf{name}}$ functions, and variables bound in `lam` expressions are encoded and decoded by the $\mathsf{E}_{\lambda\mathsf{var}}$ and $\mathsf{D}_{\lambda\mathsf{var}}$ functions.

De Bruijn indices. We use serialised de Bruijn-indexed terms for script transmission because this makes serialised scripts significantly smaller. Recall from Section 2.1.3 that when we want to use our syntax with de Bruijn indices we replace names with natural numbers and the bound variable in a `lam` expression with 0. During serialisation the zero is ignored, and during deserialisation no input is consumed and the index 0 is always returned:

$$\mathsf{E}_{\lambda\mathsf{var}}(s, n) = s$$
$$\mathsf{D}_{\lambda\mathsf{var}}(s) = 0.$$

For variables we always use indices which are greater than zero, and our encoder and decoder for names are given by

$$\mathsf{E}_{\mathsf{name}} = \mathsf{E}_{\mathbb{N}}$$

and

$$\mathsf{D}_{\mathsf{name}}(s) = (s', n) \quad \text{if } \mathsf{D}_{\mathbb{N}} = (s', n) \text{ and } n > 0.$$

Other types of name. One can serialise code involving other types of name by providing suitable encoders and decoders for name. For example, for textual names one could use $\mathsf{E}_{\lambda\mathsf{var}} = \mathsf{E}_{\mathsf{name}} = \mathsf{E}_{\mathbb{U}^*}$ and $\mathsf{D}_{\lambda\mathsf{var}} = \mathsf{D}_{\mathsf{name}} = \mathsf{D}_{\mathbb{U}^*}$. Depending on the method used to represent variable names it may also be necessary to check during deserialisation the more general requirement that variables are well-scoped, but this problem will not arise if de Bruijn indices are used.

C.4 Cardano-specific serialisation issues

C.4.1 Scope checking

To execute a Plutus Core program on the blockchain it will be necessary to deserialise it to some in-memory representation, and during or immediately after deserialisation it should be checked that the body of the program is a closed term (see the requirement in Section 2.1.3); if this is not the case then evaluation should fail immediately.

C.4.2 CBOR wrapping

Plutus Core programs are not stored on the Cardano chain directly as `flat` bytestrings; for consistency with other objects used on the chain, the `flat` bytestrings are in fact wrapped in a CBOR encoding. This should not concern most users, but we mention it here to avoid possible confusion.

C.5 Example

Consider the program

```
(program 5.0.2
 [
  [(builtin indexByteString)(con bytestring #1a5f783625ee8c)]
  (con integer 54321)
 ])
```

Suppose this is stored in `index.uplc`. We can convert it to `flat` by running

```
$ cabal run exec uplc convert -- -i index.uplc --of flat -o index.flat
```

The serialised program looks like this:

```
$ xxd -b index.flat
00000000: 00000101 00000000 00000010 00110011 01110001 11001001  ...3q.
00000006: 00010001 00000111 00011010 01011111 01111000 00110110  ..._x6
0000000c: 00100101 11101110 10001100 00000000 01001000 00111000  %...H8
00000012: 10110100 00000001 10000001
```

Figure C.1 shows how this encodes the original program. Sequences of bits are followed by explanatory comments and lines beginning with # provide further commentary on preceding bit sequences.

```
00000101 : Final integer chunk: 0000101 → 5
00000000 : Final integer chunk: 0000000 → 0
00000010 : Final integer chunk: 0000000 → 2
         # Version: 5.0.2
0011     : Term tag 3: apply
0011     : Term tag 3: apply
0111     : Term tag 7: builtin
0001110  : Builtin tag 14
         # builtin indexByteString
0100     : Term tag 4: constant
1        : Start of type tag list
0001     : Type tag 1
0        : End of list
         # Type tags: [1] → bytestring
001      : Padding before bytestring
00000111 : Bytestring chunk size: 7
00011010 : 0x1a
01011111 : 0x5f
01111000 : 0x78
00110110 : 0x36
00100101 : 0x25
11101110 : 0xee
10001100 : 0x8c
00000000 : Bytestring chunk size: 0 (end of list of chunks)
         # con bytestring #1a5f783625ee8c
0100     : Term tag 4: constant
1        : Start of type tag list
0000     : Type tag 0
0        : End of list
         # Type tags: [0] → integer
11100010 : Integer chunk 1100010 (least significant)
11010000 : Integer chunk 1010000
00000110 : Final integer chunk 0000110 (most significant)
         # 0000110 · 1010000 · 1100010 → 108642 decimal
         # Zigzag encoding: 108642/2 → +54321
         # con integer 54321
000001   : Padding
```

Figure C.1: `flat` encoding of `index.uplc`

Bibliography

[1] Cardano Improvement Proposal 0035 (CIP 0035) – Changes to Plutus Core. https://cips.cardano.org/cip/CIP-0035/.

[2] Cardano Improvement Proposal 0042 (CIP 0042) – New Plutus Builtin serialiseData. https://cips.cardano.org/cip/CIP-0042/.

[3] Cardano Improvement Proposal 0049 (CIP 0049) – ECDSA and Schnorr signatures in Plutus Core. https://cips.cardano.org/cip/CIP-0049/.

[4] Cardano Improvement Proposal 0101 (CIP 0101) – Integration of `keccak256` into Plutus. https://cips.cardano.org/cip/CIP-0101/.

[5] Cardano Improvement Proposal 0381 (CIP 0381) – Plutus support for Pairings over BLS12_381. https://cips.cardano.org/cips/cip0381/.

[6] Draft Cardano Improvement Proposal 0121 (CIP 0121) – Integer-Bytestring conversions. https://github.com/cardano-foundation/CIPs/tree/master/CIP-0121.

[7] Draft Cardano Improvement Proposal 0122 (CIP 0122) – Logical operations over BuiltinByteString. https://github.com/cardano-foundation/CIPs/tree/master/CIP-0122.

[8] Draft Cardano Improvement Proposal 0123 (CIP 0123) – Bitwise operations over BuiltinByteString. https://github.com/cardano-foundation/CIPs/tree/master/CIP-0123.

[9] ANSI. X9.62: Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA), 2005.

[10] ANSI. X9.142: Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA), 2020.

[11] Pasqualino 'Titto' Assini. Flat format specification. http://quid2.org/docs/Flat.pdf.

[12] Hendrik Pieter Barendregt. *The Lambda Calculus - its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1985.

[13] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In *CHES*, volume 6917 of *Lecture Notes in Computer Science*, pages 124–142. Springer, 2011.

[14] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The Keccak Reference. Submission to NIST (Round 3), January 2011.

[15] Bitcoin Wiki. Elliptic Curve Digital Signature Algorithm, 2022.

[16] Carsten Bormann. Notable CBOR Tags. https://www.ietf.org/archive/id/draft-bormann-cbor-notable-tags-06.html.

[17] Carsten Bormann and Paul E. Hoffman. RFC 8949: Concise Binary Object Representation (CBOR). https://datatracker.ietf.org/doc/html/rfc8949, December 2020.

[18] Antoon Bosselaers. The hash function RIPEMD-160. https://homes.esat.kuleuven.be/~bosselae/ripemd160.html, 2012.

[19] Sean Bowe. BLS12-381: New zk-SNARK Elliptic Curve Construction. https://electriccoin.co/blog/new-snark-curve/, March 2017.

[20] Certicom Research. Standards for Efficient Cryptography 2 (SEC 2). https://www.secg.org/SEC2-Ver-2.0.pdf, 2010.

[21] Craig Costello. Pairings for beginners. https://static1.squarespace.com/static/5fdbb09f31d71c1227082339/t/5ff394720493bd28278889c6/1609798774687/PairingsForBeginners.pdf.

[22] Duncan Coutts, Michael Peyton Jones, and Carsten Bormann. CBOR Tags for Discriminated Unions. https://github.com/cabo/cbor-discriminated-unions/.

[23] Quynh Dang. Secure Hash Standard (SHS). FIPS PUB 180-4, https://www.nist.gov/publications/secure-hash-standard, March 2012.

[24] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[25] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1996.

[26] Morris Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 202, https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions, August 2015.

[27] A. Faz-Hernandez, S. Scott, N. Sullivan, and R.S. Wahby. Hashing to Elliptic Curves (Version 16). https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve, June 2022.

[28] Matthias Felleisen. Programming languages and lambda calculi, 2007.

[29] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009.

[30] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, August 1986.

[31] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, September 1992.

[32] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[33] Don Johnson, Alfred Menezes, and Scott A. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *Int. J. Inf. Sec.*, 1(1):36–63, 2001.

[34] Simon Josefsson and Ilari Liusvaara. RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA). https://datatracker.ietf.org/doc/html/rfc8032, January 2017.

[35] Johnson Lau, Jonas Nick, and Tim Ruffing. Bitcoin Improvement Proposal 340: Schnorr Signatures for secp256k1. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki, 2020.

[36] Johnson Lau and Pieter Wuilie. Bitcoin Improvement Proposal 146: Dealing with signature encoding malleability. https://github.com/bitcoin/bips/blob/master/bip-0146.mediawiki, 2016.

[37] Supranational LLC. `blst library`. https://github.com/supranational/blst.

[38] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[39] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[40] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693, https://datatracker.ietf.org/doc/html/rfc7693, November 2015.

[41] Y. Sakemi, T. Kobayashi, T. Saito, and R. Wahby. Pairing-Friendly Curves (Version 11). https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves-11, November 2022.

[42] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 1989.

[43] Joseph H Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, 2009.

[44] The Unicode Consortium. The Unicode Standard. https://www.unicode.org/versions/latest/.

[45] F. Vercauteren. Optimal pairings. Cryptology ePrint Archive, Paper 2008/096, 2008. https://eprint.iacr.org/2008/096.

[46] Philip Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM.

[47] Zcash. Zcash BLS12-381 serialization format. https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md#serialization.