Formal Specification of the Plutus Core Language

Plutus Team

14th July 2022

DRAFT

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. Appendix A includes a list of supported builtins in each era and the formally verified behaviour.

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

1 Introduction

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 the Alonzo release 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.

2 Some Basic Notation

- The symbol [] denotes an empty list.
- The notation $[x_1, \ldots, x_n]$ denotes a list containing the elements $x_1, \ldots, x_n$. If $n < 1$ then the list is empty.
- Given two lists $L = [x_1, \ldots, x_m]$ and $L' = [y_1, \ldots, y_n]$, $L \cdot L'$ denotes their concatenation $[x_1, \ldots, x_m, y_1, \ldots, y_n]$.
- Given an object x and a list $L = [x_1, \ldots, x_n]$, we denote the list $[x, x_1, \ldots, x_n]$ by $x \cdot L$.
- Given a list $L = [x_1, \ldots, x_n]$ and an object x, we denote the list $[x_1, \ldots, x_n, x]$ by $L \cdot x$.
- Given a syntactic category V, the symbol $\overline{V}$ denotes a possibly empty list $[V_1, \ldots, V_n]$ of elements $V_i \in V$.

3 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 E.

3.1 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
Constant	c	::=	⟨literal constant⟩	

Figure 1: Lexical grammar of Plutus Core

3.2 Grammar

Term	L, M, N	$::=$	x	variable
			`(con` tn 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
			`(error)`	error
Program	P	$::=$	`(program` v M`)`	versioned program

Figure 2: Grammar of untyped Plutus Core

3.3 Notes

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 … N_k`]` as an abbreviation for an iterated application [… [[M N_1] N_2] … N_k], and tools may also use this as concrete syntax.

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` tn 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 4. Plutus Core comes with a default universe and a default set of builtins, which are described in Appendix A.

De Bruijn indices. The grammar defines names to be textual strings, but occasionally (specifically in Appendix E) we want to use de Bruijn indices ([10], [4, 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-expresssions 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.

4 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.

Set-theoretic notation. We begin with some extra set-theoretic notation:

- $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.
- $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$.
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- $\mathbb{B} = \{n \in \mathbb{Z} : 0 \leq n \leq 255\}$, the set of 8-bit bytes.
- $\mathbb{U}$ denotes the set of Unicode scalar values, as defined in [21, Definition D76].
- 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:

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

- We assume that there is a special symbol X which does not appear in any other set we mention. The symbol X 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 X or a member of some set S. For brevity, if S is a set then we define

$$S_{\mathsf{X}} := S \uplus \{\mathsf{X}\}.$$

4.1 Built-in types

We require some extra syntactic notation for built-in types: see Figure 3.

atn	$::=$	n	Atomic type
op	$::=$	n	Type operator
tn	$::=$	$atn \mid op(tn, tn, ..., tn)$	Type

Figure 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 $op(tn_1, \ldots, tn_n)$ 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 $\mathcal{O}$:

$$\mathcal{U}_{i+1} = \mathcal{U}_i \cup \{op(tn_1, \ldots, tn_{|op|}) : op \in \mathcal{O}, tn_1, \ldots, tn_{|op|} \in \mathcal{U}_i\}$$

$$\mathcal{U} = \bigcup \{\mathcal{U}_i : i \in \mathbb{N}^+\}$$

The universe $\mathcal{U}$ consists entirely of *names*, and the semantics of these names are given by *denotations*. Each type name $tn \in \mathcal{U}$ is associated with some mathematical set $[\![tn]\!]$, the *denotation* of tn. For example,

we might have $[\![\texttt{boolean}]\!] = \{\mathsf{true}, \mathsf{false}\}$ and $[\![\texttt{integer}]\!] = \mathbb{Z}$ and $[\![\texttt{pair}\,(a, b)]\!] = [\![a]\!] \times [\![b]\!]$. See Appendix A for a description of the built-in types and type operators available in the Alonzo release of Plutus Core.

For non-atomic type names $tn = op(tn_1, \ldots, tn_r)$ we require the denotation of tn to be obtained in some uniform way from the denotations of $tn_1, \ldots, tn_r$.

4.1.1 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 4, where we also define a class of *quantifications* which are used to introduce type variables.

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

Figure 4: Type variables

We denote the set of all possible quantifications by $\mathcal{Q}$, 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 $\mathcal{V} \cup \mathcal{U} = \varnothing$ since the symbols $_*$ and $_\#$ do not occur 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

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

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

$$\hat{\mathcal{U}}_0 = \mathcal{U}_0 \cup \mathcal{V}_\#$$
$$\hat{\mathcal{U}}_{i+1} = \hat{\mathcal{U}}_i \cup \{op(tn_1, \ldots, tn_{|op|}) : op \in \mathcal{O}, tn_1, \ldots, tn_{|op|} \in \hat{\mathcal{U}}_i\}$$
$$\hat{\mathcal{U}} = \bigcup\{\hat{\mathcal{U}}_i : i \in \mathbb{N}^+\}$$

We define the set of *free variables* of an element of $\hat{\mathcal{U}}$ by

$$\mathsf{FV}(tn) = \varnothing \text{ if } tn \in \mathcal{U}_0$$
$$\mathsf{FV}(v_\#) = \{v_\#\}$$
$$\mathsf{FV}(op(tn_1, \ldots, tn_k)) = \mathsf{FV}(tn_1) \cup \mathsf{FV}(tn_2) \cup \cdots \cup \mathsf{FV}(tn_r)$$

Thus $\mathsf{FV}(tn) \subseteq \mathcal{V}_\#$ for all $tn \in \mathcal{U}$. We say that a type name $tn \in \hat{\mathcal{U}}$ is *monomorphic* if $\mathsf{FV}(tn) = \varnothing$; otherwise tn is *polymorphic*. The fact that type variables in $\hat{\mathcal{U}}$ 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.

4.2 Arguments of 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 properties:

- $\mathcal{I}$ is disjoint from $[\![tn]\!]$ for all $tn \in \mathcal{U}$
- We require disjoint subsets $\mathcal{C}_{tn} \subseteq \mathcal{I}$ ($tn \in \mathcal{U}$) of *constants of type tn* and maps $[\![\cdot]\!]_{tn} : \mathcal{C}_{tn} \to [\![tn]\!]$ (*denotation*) and $\{\!|\cdot|\!\}_{tn} : [\![tn]\!] \to \mathcal{C}_{tn}$ (*reification*) such that $\{\!|[\![c]\!]_{tn}|\!\}_{tn} = c$ for all $c \in \mathcal{C}_{tn}$. 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 $[\![\cdot]\!]_{tn}$ is surjective and $\{\!|\cdot|\!\}_{tn}$ is injective.

For example, we could take $\mathcal{I}$ to be the set of all Plutus Core values (see Section 5.1), $\mathcal{C}_{tn}$ to be the set of all terms `(con` *tn c*`)`, and $[\![\cdot]\!]_{tn}$ to be the function which maps `(con` *tn c*`)` to c. For simplicity we are assuming that mathematical entities occurring as members of type denotations $[\![tn]\!]$ 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 A.1 for suggested syntax for the built-in types included in the Alonzo release.

We will consistently use the symbol τ (and subscripted versions of it) to denote a member of $\hat{\mathcal{U}} \uplus \mathcal{V}_*$ in the rest of the document.

4.3 Built-in functions

Signatures. Every built-in function $b \in \mathcal{B}$ has a *signature* $\sigma(b)$ of the form

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

with

- $\iota_j \in \hat{\mathcal{U}} \uplus \mathcal{V}_* \uplus \mathcal{Q}$ for all j
- $\tau \in \hat{\mathcal{U}} \uplus \mathcal{V}_*$
- $|\{j : \iota_j \notin \mathcal{Q}\}| \geq 1$ (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 $\iota = tn \in \hat{\mathcal{U}}$ and $v \in \mathsf{FV}(tn)$ or $\iota = v$ and $v :: *$.)
- 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.

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 $[\forall a_*, \texttt{boolean}, a_*, a_*] \to a_*$. 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 $\sigma = [\iota_1, \dots, \iota_n] \to \tau$, we define the *reduced signature* $\bar{\sigma}$ to be

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

We extend the usual set comprehension notation to lists in the obvious way, so this just denotes the signature σ with all quantifications omitted. We will often write a reduced signature in the form $[\tau_1, \dots, \tau_m] \to \tau$ to emphasise that the entries are *types*, and $\forall$ does not appear.

What is the intended meaning of this notation? 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 $\hat{\mathcal{U}}$ 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 5.1.

To make some of the above remarks more precise and simplify some of the later exposition we introduce a relation $\sim \subseteq \mathcal{I} \times (\hat{\mathcal{U}} \uplus \mathcal{V}_* \uplus \mathcal{Q})$ of *compatibility* between inputs and signature entries: this is defined in Figure 5.

$$\begin{array}{lll} V \sim \iota & \text{if} & \iota \in \mathcal{U} \text{ and } V \in \mathcal{C}_\iota \\ & \text{or} & \iota \in \hat{\mathcal{U}} \setminus \mathcal{U} \\ & \text{or} & \iota \in \mathcal{V}_* \end{array}$$

Figure 5: Compatibility of inputs with signature entries

Note that we can never have $V \sim \forall v$.

Denotations of built-in functions. If we have a built-in function b with reduced signature

$$\bar{\sigma}(b) = [\tau_1, \dots, \tau_m] \to \tau,$$

then we require b to have a *denotation* (or *meaning*), a function

$$[\![b]\!] \,:\, [\![\tau_1]\!] \times \cdots \times [\![\tau_m]\!] \to [\![\tau]\!]_\times$$

where for a name a

$$[\![a_\#]\!] = \bigsqcup\nolimits_{+} \{[\![tn]\!] \,:\, tn \in \mathcal{U}\}$$

and

$$[\![a_*]\!] = \mathcal{I}.$$

Denotations of builtins are mathematical functions which terminate on every possible input; the symbol X can be returned by a function to indicate that something has gone wrong, such as an attempted division by zero.

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

1. $r \in [\![tn]\!]$ for some $tn \in \mathcal{U}$
2. $r \in \mathcal{I}$
3. $r = \mathsf{X}$

In other words,

$$r \in \mathcal{R} := \biguplus \{[\![tn]\!] : tn \in \mathcal{U}\} \uplus \mathcal{I} \uplus \{\mathsf{X}\}.$$

Our assumptions on the set $\mathcal{I}$ (Section 4.2) allow us define a function

$$\{\!|-|\!\} : \mathcal{R} \to \mathcal{I}_{\mathsf{X}}$$

which converts results of built-in functions back into inputs (or the X symbol)

1. If $r \in [\![tn]\!]$, then we let $\{\!|r|\!\} = \{\!|r|\!\}_{tn} \in \mathcal{C}_{tn} \subseteq \mathcal{I}$.
2. If $r \in \mathcal{I}$ then we let $\{\!|r|\!\} = r$
3. We let $\{\!|\mathsf{X}|\!\} = \mathsf{X}$

Behaviour of built-in functions. 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.

We also require built-in functions to be parametrically polymorphic in arguments which are of polymorphic built-in types, such as lists, and when a function signature contains type variables in $\mathcal{V}_\#$ we will expect the actual arguments supplied during application to have consistent types (for a given type variable $a_\#$, all arguments to which it refers should have the same built-in type at run time). However we do not enforce this in the notation above: instead consistency conditions of this sort will be included in the specifications of the semantics of the full Plutus Core language.

When (the meaning of) a built-in function b is applied (perhaps partially) to arguments, the types of constant arguments must correspond to the types in $\bar{\sigma}(b)$, and the function will return X if this is not the case; builtins may also return X in other circumstances, for example if an argument is out of range.

5 Term Reduction

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

5.1 Values in Plutus Core

The semantics of built-in functions in Plutus Core are obtained by instantiating the sets $\mathcal{C}_{tn}$ of constants of type tn (see Section 4.2) to be the expressions of the form `(con` tn 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*: for example, applications of `force` must be correctly interleaved with genuine arguments, and the arguments must (a) themselves be values, and (2) must be of the types expected by the function, so if `modInteger` has signature `[integer, integer]` → `integer` then `[(builtin modInteger) (con string "green")]` is illegal. The occurrence of partially-applied builtins complicates the definition of general values considerably.

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

$$
\begin{array}{llll}
\text{Value} & V & ::= & \texttt{(con}\ tn\ c\texttt{)} \\
 & & & \texttt{(delay}\ M\texttt{)} \\
 & & & \texttt{(lam}\ x\ M\texttt{)} \\
 & & & A
\end{array}
$$

Figure 6: 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 for each built-in function $b \in \mathcal{B}$:

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

Figure 7: Partial built-in function application

We let P denote the set of terms generated by the grammar in Figure 7 and we define a function β which extracts the name of the built-in function occurring in a term in P:

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

We also define a function ℓ which measures the size of a term $P \in \mathsf{P}$:

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

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

1. $P = (\texttt{builtin}\ b)$ is always well-formed.
2. $P = (\texttt{force}\ P')$ is well-formed if P' is well-formed and $\iota \in \mathbb{Q}$.
3. $P = [P'\ V]$ is well-formed if P' is well-formed and $V \sim \iota$ (see Figure 5 for the definition of $\sim$).
4. Furthermore, if $\ell(P) = n$ then we require that built-in polymorphic types are used consistently in P.

Conditions (2) and (3) say the arguments of b are properly interleaved with occurrences of `force`, and that the arguments are of the expected types. For type consistency, the compatibility condition says that (a) if the signature specifies a monomorphic built-in type then the type of V must match it exactly; (b) if the signature specifies a polymorphic built-in type then V must be a constant of *some* built-in type; and (c) if the signature specifies a full-polymorphic type then any input is acceptable.

In case (b) further checks will be required if and when b becomes fully applied, to make sure that polymorphic type variables are instantiated consistently.

Consistency of arguments and signatures. The meaning of condition (4) should be fairly obvious; for example if we have a builtin b with signature

$$[\forall a_{\#}, \forall b_{\#}, a_{\#}, \texttt{list}\,(a_{\#}), \texttt{pair}\,(a_{\#}, b_{\#})] \to \texttt{pair}\,(\texttt{list}\,(a_{\#}), \texttt{list}\,(b_{\#}))$$

then in a well-formed saturated application `[(builtin b) U V W]` there must be (monomorphic) types $t, u \in \mathcal{U}$ such that U is a constant of type t, V is a constant of type $\texttt{list}\,(t)$, and W is a constant of type $\texttt{pair}\,(t, u)$. A full definition of consistency will be added in a subsequent version of this document. We will define consistency to be a binary relation $\approx$ between lists of values and reduced arities and we will use this notation later in the document even though the full definition is not available yet.

We can now complete the definition of values in Figure 6 by defining A to be the set of well-formed *partial* built-in function applications

$$A = \{P \in \mathsf{P} : P \text{ is well-formed and } \ell(P) < \chi(\beta(P))\}.$$

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:

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

This makes sense because in a well-formed partial application we have $\ell(P) < n$.

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

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

5.2 Term reduction

We define the semantics of Plutus Core using contextual semantics (or reduction semantics): see [13] or [14, 5.3], for example. We use A to denote a partial application of a built-in function as in Section 5.1 above. For builtin evaluation, we instantiate the set $\mathbb{I}$ of Section 4.2 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`)`.

$$\begin{array}{rrcll} \text{Frame} & f & ::= & \texttt{[\_}\ M\texttt{]} & \text{left application} \\ & & & \texttt{[}V\ \texttt{\_]} & \text{right application} \\ & & & \texttt{(force \_)} & \text{force} \end{array}$$

(a) Grammar of reduction frames for Plutus Core

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

Term M reduces in one step to term M'.

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

$$\frac{\ell(A) = \chi(\beta(A)) - 1 \qquad V \sim \mathsf{next}(A)}{\texttt{[}A\ V\texttt{]} \ \to \ \mathsf{Eval}\,(\beta(A), (\mathsf{args}(A)) \cdot V)}$$

$$\frac{\ell(A) < \chi(\beta(A)) - 1 \qquad V \sim \mathsf{next}(A)}{\texttt{[}A\ V\texttt{]} \ \to \ \texttt{[}A\ V\texttt{]}}$$

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

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

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

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

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

(b) Reduction via Contextual Semantics

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

(c) Built-in function application

Figure 8: 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 3.3 that we require the body of every Plutus Core program to be closed.

Frame	f	$::=$	$(\texttt{force}\ \_)$	force
			$[\_\ (M, \rho)]$	left application
			$[V\ \_]$	right application

Figure 10: Grammar of reduction frames for Plutus Core

6 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 [12].
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 11 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).

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

Figure 9: Grammar of CEK machine states for Plutus Core

Figures 9 and 10 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 4 we define $\mathcal{I} = V$ and $\mathcal{C}_{tn} = \{\langle \texttt{con}\ tn\ c \rangle : tn \in \mathcal{U}, c \in [\![tn]\!]\}$.

The rules in Figure 11 show the transitions of the machine; if any situation arises which is not included in these transitions (for example, if a frame $[\langle \texttt{con}\ tn\ c \rangle\ \_]$ 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.

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

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}\ tn\ c) & \mapsto & s \triangleleft \langle \mathtt{con}\ tn\ 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{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 \\
(\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 \varepsilon) \rangle & \mapsto & s \triangleleft \langle \mathtt{builtin}\ b\ \overline{V}\ \varepsilon \rangle \text{ if } \iota \in \mathcal{Q} \\
(\mathtt{force}\ \_) \cdot s \triangleleft \langle \mathtt{builtin}\ b\ \overline{V}\ [\iota] \rangle & \mapsto & \mathsf{Eval}(s, b, \overline{V}) \text{ if } \iota \in \mathcal{Q} \\
[\langle \mathtt{builtin}\ b\ \overline{V}\ (\iota \cdot \varepsilon) \rangle\ \_] \cdot s \triangleleft V & \mapsto & s \triangleleft \langle \mathtt{builtin}\ b\ (\overline{V} \cdot V)\ \varepsilon \rangle \text{ if } V \sim \iota \\
[\langle \mathtt{builtin}\ b\ \overline{V}\ [\iota] \rangle\ \_] \cdot s \triangleleft V & \mapsto & \mathsf{Eval}(s, b, \overline{V} \cdot V) \text{ if } V \sim \iota
\end{array}
$$

(a) CEK machine transitions for Plutus Core

$$
\mathsf{Eval}(s, b, [V_1, \ldots, V_n]) \equiv \begin{cases} \blacklozenge & \text{if } [V_1, \ldots, V_n] \not\approx \bar{\alpha}(b) \\ \blacklozenge & \text{if } [\![b]\!]([\![V_1]\!], \ldots, [\![V_n]\!]) = \mathsf{X} \\ s \triangleleft \{\!| [\![b]\!]([\![V_1]\!], \ldots, [\![V_n]\!]) |\!\} & \text{otherwise} \end{cases}
$$

(b) Evaluation of built-in functions

Figure 11: A CEK machine for Plutus Core

6.1 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 11 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 [19, pp. 129–130], [11, pp. 71ff]). This process substitutes bindings in environments for variables occurring in the value V to obtain

a term $\mathcal{U}(V)$: see Figure 12a. 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 12b, where 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 12b we substitute the rightmost (ie, the most recent) bindings in ρ first.

$$\begin{aligned}
\mathcal{U}(\langle \texttt{con}\ tn\ c \rangle) &= (\texttt{con}\ tn\ c) \\
\mathcal{U}(\langle \texttt{delay}\ M\ [x_1 \mapsto V_1, \dots, x_n \mapsto V_n] \rangle) &= (\texttt{delay}\ (M\,@\,\rho)) \\
\mathcal{U}(\langle \texttt{lam}\ x\ M\ \rho \rangle) &= (\texttt{lam}\ x\ (M\,@\,\rho)) \\
\mathcal{U}(\langle \texttt{builtin}\ b\ V_1 V_2 \dots V_k\ \varepsilon \rangle) &= [\dots\ [[(\texttt{builtin}\ b)\ \mathcal{U}(V_1)]\ \mathcal{U}(V_2)] \dots \mathcal{U}(V_k)]
\end{aligned}$$

(a) Discharging CEK values

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

(b) Iterated substitution/discharging

Figure 12: 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 8.

7 Typed Plutus Core

To follow.

Appendix A Built-in Types and Functions Supported in the Alonzo Release

A.1 Built-in types and type operators

The Alonzo release of the Cardano blockchain (September 2021) supports a default set of built-in types and type operators defined in Tables 1 and 2. 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.	Not yet supported.

Table 1: Atomic Types

Operator *op*	$\|op\|$	Denotation	Concrete Syntax
`list`	1	$[\![\texttt{list}\,(t)]\!] = [\![t]\!]^*$	Not yet supported
`pair`	2	$[\![\texttt{pair}\,(t_1, t_2)]\!] = [\![t_1]\!] \times [\![t_2]\!]$	Not yet supported

Table 2: Type Operators

Concrete syntax for strings. Strings are represented as sequences of Unicode characters enclosed in double quotes, and may include standard escape sequences.

Concrete syntax for higher-order types. Types such as `list` (`integer`) and `pair` (`bool`, `string`)) are represented by application at the type level, thus: `[(con list) (con integer)]` and `[(con pair) (con bool) (con string)]`. Each higher-order type will need further syntax for representing constants of those types. For example, we might use `[]` for list values and `(,)` for pairs, so the list [11, 22, 33] might be written as

```
(con [(con list) (con integer)]
     [(con integer 11), (con integer 22), (con integer 33)]
)
```

and the pair (True, "Plutus") as

```
(con [(con pair) (con bool) (con string)]
     ((con bool True), (con string "Plutus"))
).
```

Note however that this syntax is not currently supported by most Plutus Core tools at the time of writing.

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 D.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.

A.2 Alonzo built-in functions

The default set of built-in functions for the Alonzo release is shown in Table 3. 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 $\sim$ and $\approx$ relations in Figures 8 and 11.

Function	Signature	Denotation	Can Fail?	Note
`addInteger`	[`integer`, `integer`] → `integer`	$+$		
`subtractInteger`	[`integer`, `integer`] → `integer`	$-$		
`multiplyInteger`	[`integer`, `integer`] → `integer`	$\times$		
`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`	$=$		
`lessThanInteger`	[`integer`, `integer`] → `bool`	$<$		
`lessThanEqualsInteger`	[`integer`, `integer`] → `bool`	$\leq$		
`appendByteString`	[`bytestring`, `bytestring`] → `bytestring`	$([c_1, \ldots, c_m], [d_1, \ldots, d_n]) \mapsto [c_1, \ldots, c_m, d_1, \ldots, d_n]$		
`consByteString`	[`integer`, `bytestring`] → `bytestring`	$(c, [c_1, \ldots, c_n]) \mapsto [\text{mod}(c, 256), c_1, \ldots, c_n]$		
`sliceByteString`	[`integer`, `integer`, `bytestring`] → `bytestring`	$(s, k, [c_0, \ldots, c_n]) \mapsto [c_{\max(s,0)}, \ldots, c_{\min(s+k-1, n-1)}]$		2
`lengthOfByteString`	[`bytestring`] → `integer`	$[] \mapsto 0, [c_1, \ldots, c_n] \mapsto n$		
`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`	$=$		3
`lessThanByteString`	[`bytestring`, `bytestring`] → `bool`	$<$		3
`lessThanEqualsByteString`	[`bytestring`, `bytestring`] → `bool`	$\leq$		3
`appendString`	[`string`, `string`] → `string`	$([u_1, \ldots, u_m], [v_1, \ldots, v_n]) \mapsto [u_1, \ldots, u_m, v_1, \ldots, v_n]$		
`equalsString`	[`string`, `string`] → `bool`	$=$		
`encodeUtf8`	[`string`] → `bytestring`	`utf8`		4
`decodeUtf8`	[`bytestring`] → `string`	$\texttt{utf8}^{-1}$	Yes	4
`sha2_256`	[`bytestring`] → `bytestring`	Hash a `bytestring` using `SHA256`.		
`sha3_256`	[`bytestring`] → `bytestring`	Hash a `bytestring` using `SHA3-256`.		
`blake2b_256`	[`bytestring`] → `bytestring`	Hash a `bytestring` using `Blake2B256`.		
`verifyEd25519Signature`	[`bytestring`, `bytestring`, `bytestring`] → `bool`	Verify an `Ed25519` digital signature.	Yes	5, 6
`ifThenElse`	$[\forall a_*, \texttt{bool}, a_*, a_*] \to a_*$	$(\texttt{true}, t_1, t_2) \mapsto t_1$ $(\texttt{false}, t_1, t_2) \mapsto t_2$		

Table 3: Built-in Functions

Function	Type	Denotation	Can Fail?	Note
`chooseUnit`	$[\forall a_*, \texttt{unit}, a_*] \to a_*$	$((), t) \mapsto t$		
`trace`	$[\forall a_*, \texttt{string}, a_*] \to a_*$	$(s, t) \mapsto t$		7
`fstPair`	$[\forall a_\#, \forall b_\#, \texttt{pair}\,(a_\#, b_\#)] \to a_\#$	$(x, y) \mapsto x$		
`sndPair`	$[\forall a_\#, \forall b_\#, \texttt{pair}\,(a_\#, b_\#)] \to b_\#$	$(x, y) \mapsto y$		
`chooseList`	$[\forall a_\#, \forall b_*, \texttt{list}\,(a_\#), b_*, b_*] \to b_*$	$([], t_1, t_2) \mapsto t_1$, $([x_1, \ldots, x_n], t_1, t_2) \mapsto t_2\ (n \geq 1)$.		
`mkCons`	$[\forall a_\#, a_\#, \texttt{list}\,(a_\#)] \to \texttt{list}\,(a_\#)$	$(x, [x_1, \ldots, x_n]) \mapsto [x, x_1, \ldots, x_n]$	Yes	
`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`	$[\forall a_\#, \texttt{list}\,(a_\#)] \to \texttt{bool}$	$[] \mapsto \texttt{true}, [x_1, \ldots, x_n] \mapsto \texttt{false}$		
`chooseData`	$[\forall a_*, \texttt{data}, a_*, a_*, a_*, a_*, a_*] \to a_*$	$(d, t_C, t_M, t_L, t_I, t_B) \mapsto \begin{cases} t_C & \text{if } \text{is}_C(d) \\ t_M & \text{if } \text{is}_M(d) \\ t_L & \text{if } \text{is}_L(d) \\ t_I & \text{if } \text{is}_I(d) \\ t_B & \text{if } \text{is}_B(d) \end{cases}$		
`constrData`	$[\texttt{integer}, \texttt{list}\,(\texttt{data})] \to \texttt{data}$	inj_C		
`mapData`	$[\texttt{list}\,(\texttt{pair}\,(\texttt{data}, \texttt{data})) \to \texttt{data}$	inj_M		
`listData`	$[\texttt{list}\,(\texttt{data})] \to \texttt{data}$	inj_L		
`iData`	$[\texttt{integer}] \to \texttt{data}$	inj_I		
`bData`	$[\texttt{bytestring}] \to \texttt{data}$	inj_B		
`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`	$[\texttt{data}, \texttt{data}] \to \texttt{pair}\,(\texttt{data}, \texttt{data})$	$(x, y) \mapsto (x, y)$		
`mkNilData`	$[\texttt{unit}] \to \texttt{list}\,(\texttt{data})$	$() \mapsto []$		
`mkNilPairData`	$[\texttt{unit}] \to \texttt{list}\,(\texttt{pair}\,(\texttt{data}, \texttt{data}))$	$() \mapsto []$		

Table 3: Built-in Functions (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

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

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

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 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: Behaviour of integer division functions

Note 2. 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 3. Comparisons of bytestrings. Bytestrings are ordered lexicographically. If we have $a = [c_1, \ldots, c_m]$ and $b = [d_1, \ldots, d_n]$ then

- $a = b$ if and only if $m = n$ and $c_i = d_i$ for $1 \leq i \leq m$
- $a \leq b$ if $c_i = d_i$ for $1 \leq i \leq \min(m, n)$
- $a < b$ if $a \leq b$ and $a \neq b$.

The empty bytestring is equal only to itself and is strictly less than all other bytestrings.

Note 4. Encoding and decoding bytestrings. The `encodeUtf8` and `decodeUtf8` functions convert between the `string` type and the `bytestring` type. We have defined $[\![\texttt{string}]\!]$ 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 [21, 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 5. Digital signature verification functions. We use a uniform interface for digital signature verification algorithms. A digital signature verification function takes three bytestring arguments:

- a public key k
- a message m
- a signature s

(in that order). A particular verification function may require one or more arguments to be of specified lengths, and will fail (returning ×) if any argument is the wrong length. If the arguments are the correct lengths then the verification function returns `true` if the private key corresponding to k was used to sign the message m to produce s, otherwise it returns `false`.

Note 6. Ed25519 signature verification. The `verifyEd25519Signature` function* performs cryptographic signature verification using the Ed25519 scheme [5, 16], and conforms to the interface described in Note 5. The arguments must have the following sizes:

- k: 32 bytes
- m: unrestricted
- s: 64 bytes.

Note 7. 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.

A.3 Cost accounting for built-in functions

To follow.

*`verifyEd25519Signature` was formerly called `verifySignature` but was renamed to avoid ambiguity when further signature verification functions were introduced in the Vasil release (see Section B.2).

Appendix B Built-in Types and Functions Supported in the Vasil Release

The Vasil release of Cardano (June 2022) extends the set of built-in functions slightly.

B.1 Built-in types and type operators

The built-in types and type operators remain unchanged from the Alonzo (Appendix A.1).

B.2 Built-in functions

The Vasil release continues to support the Alonzo built-in functions (Table 3) and adds three new ones: these are described in Table 5.

Function	Signature	Denotation	Can Fail?	Note
`serialiseData`	`[data]` → `bytestring`	$\mathcal{E}_{\texttt{data}}$		1
`verifyEcdsaSecp256k1Signature`	`[bytestring, bytestring, bytestring]` → `bool`	Verify an SECP-256k1 ECDSA signature	Yes	2
`verifySchnorrSecp256k1Signature`	`[bytestring, bytestring, bytestring]` → `bool`	Verify an SECP-256k1 Schnorr signature	Yes	3

Table 5: Built-in Functions

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 D.

Note 2. Secp256k1 ECDSA Signature verification. The `verifyEcdsaSecp256k1Signature` function performs elliptic curve digital signature verification [1, 2, 15] over the `secp256k1` curve [8, §2.4.1] and conforms to the interface described in Note 5 of Section A.2. The arguments must have the following sizes:

- k: 64 bytes
- m: 32 bytes
- s: 64 bytes.

The ECDSA scheme admits two distinct valid signatures for a given message and private key. We follow the restriction imposed by Bitcoin (see [18], `LOW_S`) and **only accept the smaller signature**: `verifyEcdsaSecp256k1Signature` will return `false` if the larger one is supplied.

Note 3. Secp256k1 Schnorr Signature verification. The `verifySchnorrSecp256k1Signature` function performs verification of Schnorr signatures [20, 17] over the secp256k1 curve and conforms to the interface described in Note 5 of Section A.2. The arguments must have the following sizes:

- k: 64 bytes
- m: unrestricted
- s: 64 bytes.

Appendix C Formally Verified Behaviours

To follow.

Appendix D The CBOR encoding for `data`

D.1 Introduction

In this section we define a CBOR encoding for the `data` type introduced in Section A.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 [7], 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 [9] and [6, Section 9.1].

D.2 Notation

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

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 use $l(s)$ 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 Appendix A.
To avoid confusion with our notation for lists we use some slightly non-standard notation for intervals in the set of natural numbers:

$$\mathbb{N}_{[a,b]} = \{n \in \mathbb{N} : a \leq n \leq b\}.$$

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).

D.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.

D.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 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 an inverse function $\mathsf{d}_k : \mathbb{B}^* \rightharpoonup \mathbb{N}$ which decodes a k-byte natural number from the start of a bytestring, failing if there is insufficient input:

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

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 [7, 3.2.4].

D.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 [7, §§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 $k \ge 0$ and $0 \le r \le 63$. 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

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

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):

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

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.

D.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 [7, §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.

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

D.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

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

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 $\mathcal{E}_{\texttt{ctag}}$ 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 $\mathcal{E}_{\texttt{data}}$. 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 $\mathcal{D}_{\texttt{ctag}}$ below.

The decoder. The decoder is given by

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

where

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

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

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

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

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

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 [9]) 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 A.1 that only constructor tags between 0 and $2^{64} - 1$ should be used.

Appendix E A Binary Serialisation Format for Plutus Core Terms and Programs

We use the `flat` format [3] 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 3.3) 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 E.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).

E.1 Encoding and decoding

Let $\mathbb{S} = \{0,1\}^*$, the set of all finite sequences of bits. 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 D, we will describe the flat encoding by defining families of encoding functions (or *encoders*)

$$\mathsf{E}_X : \mathbb{S} \times X \to \mathbb{S}$$

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

$$\mathsf{D}_X : \mathbb{S} \rightharpoonup \mathbb{S} \times X$$

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 $s \in \mathbb{S}$ 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.

E.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 $\mathsf{pad} : \mathbb{S} \to \mathbb{S}$ 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 $\mathsf{unpad} : \mathbb{S} \rightharpoonup \mathbb{S}$ 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.

E.2 Basic `flat` encodings

E.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

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

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

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

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 D, $\mathbb{N}_{[a,b]}$ denotes the closed interval of integers $\{n \in \mathbb{Z} : a \leq n \leq b\}$.

E.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

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

E.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

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

E.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 $\mathsf{E}_{\mathbb{N}}$:

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

The decoder is

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

E.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 $E_C : C \to \mathbb{S}$ 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

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

E.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 Appendix A. 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.

E.3 Encoding and decoding Plutus Core

E.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.

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

E.3.2 Terms

Plutus Core terms are encoded by emitting a 4-bit tag identifying the type of the term (see Table 6; recall that `[]` denotes application) then emitting the encodings for any subterms. We currently only use eight 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

Table 6: Term tags

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

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

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 $0101 \cdot s$ 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.

$$
\begin{aligned}
&\mathsf{D}_{\mathsf{term}}(0000 \cdot s) = (s', x) && \text{if } \mathsf{D}_{\mathsf{name}}(s) = (s', x) \\
&\mathsf{D}_{\mathsf{term}}(0001 \cdot s) = (s', (\texttt{delay}\ t)) && \text{if } \mathsf{D}_{\mathsf{term}}(s) = (s', t) \\
&\mathsf{D}_{\mathsf{term}}(0010 \cdot s) = (s'', (\texttt{lam}\ x\ t)) && \text{if } \mathsf{D}_{\lambda\mathsf{var}}(s) = (s', x) \quad \text{and } \mathsf{D}_{\mathsf{term}}(s') = (s'', t) \\
&\mathsf{D}_{\mathsf{term}}(0011 \cdot s) = (s'', [t_1\ t_2]) && \text{if } \mathsf{D}_{\mathsf{term}}(s) = (s', t_1) \quad \text{and } \mathsf{D}_{\mathsf{term}}(s') = (s'', t_2) \\
&\mathsf{D}_{\mathsf{term}}(0100 \cdot s) = (s'', (\texttt{const}\ tn\ c)) && \text{if } \mathsf{D}_{\mathsf{type}}(s) = (s', tn) \quad \text{and } \mathsf{D}^{tn}_{\mathsf{constant}}(s') = (s'', c) \\
&\mathsf{D}_{\mathsf{term}}(0101 \cdot s) = (s', (\texttt{force}\ t)) && \text{if } \mathsf{D}_{\mathsf{term}}(s) = (s', t) \\
&\mathsf{D}_{\mathsf{term}}(0110 \cdot s) = (s, (\texttt{error})) \\
&\mathsf{D}_{\mathsf{term}}(0111 \cdot s) = (s', b) && \text{if } \mathsf{D}_{\mathsf{builtin}}(s) = (s', b).
\end{aligned}
$$

E.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 7: 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

Table 7: 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 used in Alonzo and Vasil) by

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

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

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 E.2).

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

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

E.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 $\mathcal{E}_{\mathtt{data}}$ described in Appendix D 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 $\mathcal{D}_{\mathtt{data}}$ to obtain a `data` object.

$$\begin{aligned}
\mathsf{E}^{\mathtt{integer}}_{\mathsf{constant}}(s,n) &= \mathsf{E}_{\mathbb{Z}}(s,n)\\
\mathsf{E}^{\mathtt{bytestring}}_{\mathsf{constant}}(s,a) &= \mathsf{E}_{\mathbb{B}^*}(s,a)\\
\mathsf{E}^{\mathtt{string}}_{\mathsf{constant}}(s,t) &= \mathsf{E}_{\mathbb{U}^*}(s,t)\\
\mathsf{E}^{\mathtt{unit}}_{\mathsf{constant}}(s,c) &= s\\
\mathsf{E}^{\mathtt{bool}}_{\mathsf{constant}}(s,\mathtt{False}) &= s \cdot 0\\
\mathsf{E}^{\mathtt{bool}}_{\mathsf{constant}}(s,\mathtt{True}) &= s \cdot 1\\
\mathsf{E}^{\mathtt{list}\,(tn)}_{\mathsf{constant}}(s,l) &= \vec{\mathsf{E}}^{tn}_{\mathsf{constant}}(s,l)\\
\mathsf{E}^{\mathtt{pair}\,(tn_1,tn_2)}_{\mathsf{constant}}(s,(c_1,c_2)) &= \mathsf{E}^{tn_2}_{\mathsf{constant}}(\mathsf{E}^{tn_1}_{\mathsf{constant}}(s,c_1),c_2)\\
\mathsf{E}^{\mathtt{data}}_{\mathsf{constant}}(s,d) &= \mathsf{E}_{\mathbb{B}^*}(s,\mathcal{E}_{\mathtt{data}}(d)).
\end{aligned}$$

$$\begin{aligned}
\mathsf{D}^{\mathtt{integer}}_{\mathsf{constant}}(s) &= \mathsf{D}_{\mathbb{Z}}(s)\\
\mathsf{D}^{\mathtt{bytestring}}_{\mathsf{constant}}(s) &= \mathsf{D}_{\mathbb{B}^*}(s)\\
\mathsf{D}^{\mathtt{string}}_{\mathsf{constant}}(s) &= \mathsf{D}_{\mathbb{U}^*}(s)\\
\mathsf{D}^{\mathtt{unit}}_{\mathsf{constant}}(s) &= s\\
\mathsf{D}^{\mathtt{bool}}_{\mathsf{constant}}(0 \cdot s) &= (s,\mathtt{False})\\
\mathsf{D}^{\mathtt{bool}}_{\mathsf{constant}}(1 \cdot s) &= (s,\mathtt{True})\\
\mathsf{D}^{\mathtt{list}\,(tn)}_{\mathsf{constant}}(s) &= \vec{\mathsf{D}}^{tn}_{\mathsf{constant}}(s,l)\\
\mathsf{D}^{\mathtt{pair}\,(tn_1,tn_2)}_{\mathsf{constant}}(s) = (s'',(c_1,c_2)) &\begin{cases} \text{if} & \mathsf{D}^{tn_1}_{\mathsf{constant}}(s) = (s',c_1)\\ \text{and} & \mathsf{D}^{tn_2}_{\mathsf{constant}}(s') = (s'',c_2)\end{cases}\\
\mathsf{D}^{\mathtt{data}}_{\mathsf{constant}}(s) &= (s',d) \qquad \text{if } \mathsf{D}_{\mathbb{B}*}(s) = (s',t) \text{ and } \mathcal{D}_{\mathtt{data}}(t) = (t',d) \text{ for some } t'.
\end{aligned}$$

E.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 8 and 9. We assume that there are (partial) functions tag and tag^{-1} which convert back and forth between builtin names and their tags.

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

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 8: Tags for Alonzo builtins

Builtin	Binary	Decimal
serialiseData	0110011	51
verifyEcdsaSecp256k1Signature	0110100	52
verifySchnorrSecp256k1Signature	0110101	53

Table 9: Extra tags for Vasil builtins

E.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 3.3 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.

E.4 Cardano-specific serialisation issues

E.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 3.3); if this is not the case then evaluation should fail immediately.

E.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 13 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 13: `flat` encoding of `index.uplc`

References

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

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

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

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

[5] 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.

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

[7] Carsten Bormann and Paul E. Hoffman. RFC 8949: Concise Binary Object Representation (CBOR). https://www.rfc-editor.org/info/rfc8949, December 2020.

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] Simon Josefsson and Ilari Liusvaara. RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA). https://www.rfc-editor.org/info/rfc8032, January 2017.

[17] 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.

[18] 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.

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

[20] 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.

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