Formal Specification of the Plutus Core Language (version 3.0)

Plutus Team

January 26, 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 decribes 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 an object x and a list $L = [x_1, \ldots, x_n]$, we denote the list $[x, x_1, \ldots, x_n]$ by $x :: L$.
- 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 C.

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
			$(\texttt{con}\ tn\ c)$	constant
			$(\texttt{builtin}\ b)$	builtin
			$(\texttt{lam}\ x\ M)$	λ abstraction
			$[M\ N]$	function application
			$(\texttt{delay}\ M)$	delay execution of a term
			$(\texttt{force}\ M)$	force execution of a term
			$(\texttt{error})$	error
Program	P	$::=$	$(\texttt{program}\ v\ M)$	versioned program

Figure 2: Grammar of untyped Plutus Core

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 $(\texttt{con}\ tn\ c)$ 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.

4 Interpretation of built-in types and functions.

As mentioned earlier, 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.
- 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 $\times$ which does not appear in any other set we mention. The symbol $\times$ is used to indicate that some sort of error condition has occurred, and we will often need to consider situations in which a value is either $\times$ or a member of some set S. For brevity, if S is a set then we define
$$S_\times := S \uplus \{\times\}.$$

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 possbile 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:

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

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

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

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 the *arity* of b is

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

and the *argument count* of b is $|b| = n$
We may abuse notation slightly by using the symbol σ to denote a specific signature as well as the function which maps built-in function names to signatures, and similarly with the symbol α.

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

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

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 $[\tau_1, \ldots, \tau_m] \to \tau$ 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 signature (read from left to right) specifies what types of arguments are expected and how they should be interleaved with applications of `force`. 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.

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

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

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

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

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

and

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

Denotations of builtins are mathematical functions which terminate on every possible input; the symbol $\times$ can be returned by a function to indicate that something has gone wrong, 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. $r = \times$

In other words,

$$r \in \mathcal{R} := \bigcup\!\!\!\!+ \{[\![tn]\!] : tn \in \mathcal{U}\} \uplus \mathcal{I} \uplus \{\times\}.$$

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

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

which converts results of built-in functions back into inputs (or the $\times$ 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 $\{\!|\times|\!\} = \times$

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:

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

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 $b \in \mathcal{B}$:

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

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:

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

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

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

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*. Firstly we let $n = |b|$ and we require that $l \leq n$, so that b is not over-applied. In this case we put $\iota = \iota_l$, 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 $l = n$ 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 $[(\texttt{builtin}\ b)\ U\ V\ W]$ 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

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

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

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

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

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 Felleisen and Hieb [1992] or [Harper, 2012, 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 $\mathcal{I}$ 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 $\{\!|V|\!\} = V$.

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

(a) Grammar of reduction frames for Plutus Core

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

Term M reduces in one step to term M'.

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

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

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

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

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

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

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

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

(b) Reduction via Contextual Semantics

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

(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.

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 [Felleisen and Friedman, 1986].
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	$::=$	$[\iota] \mid \iota :: \varepsilon$

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, enivronments 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 $\rho = [x_1 \mapsto V_1, \dots, x_n \mapsto V_n]$ 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'

$$
\begin{array}{rcl}
s;\rho \rhd x & \mapsto & s \lhd \rho[x] \text{ if } x \text{ is bound in } \rho \\
s;\rho \rhd (\texttt{con}\ tn\ c) & \mapsto & s \lhd \langle \texttt{con}\ tn\ c \rangle \\
s;\rho \rhd (\texttt{lam}\ x\ M) & \mapsto & s \lhd \langle \texttt{lam}\ x\ M\ \rho \rangle \\
s;\rho \rhd (\texttt{delay}\ M) & \mapsto & s \lhd \langle \texttt{delay}\ M\ \rho \rangle \\
s;\rho \rhd (\texttt{force}\ M) & \mapsto & (\texttt{force}\ \_) :: s;\rho \rhd M \\
s;\rho \rhd [M\ N] & \mapsto & [\_\ (N,\rho)] :: s;\rho \rhd M \\
s;\rho \rhd (\texttt{builtin}\ b) & \mapsto & s \lhd \langle \texttt{builtin}\ b\ [\ ]\ \alpha(b) \rangle \\
s;\rho \rhd (\texttt{error}) & \mapsto & \blacklozenge \\
\\
[\,] \lhd V & \mapsto & \square V \\
[\_\ (M,\rho)] :: s \lhd V & \mapsto & [V\ \_] :: s;\rho \rhd M \\
[\langle \texttt{lam}\ x\ M\ \rho \rangle\ \_] :: s \lhd V & \mapsto & s;\rho[x \mapsto V] \rhd M \\
(\texttt{force}\ \_) :: s \lhd \langle \texttt{delay}\ M\ \rho \rangle & \mapsto & s;\rho \rhd M \\
(\texttt{force}\ \_) :: s \lhd \langle \texttt{builtin}\ b\ \overline{V}\ (\iota :: \varepsilon) \rangle & \mapsto & s \lhd \langle \texttt{builtin}\ b\ \overline{V}\ \varepsilon \rangle \text{ if } \iota \in \mathcal{Q} \\
(\texttt{force}\ \_) :: s \lhd \langle \texttt{builtin}\ b\ \overline{V}\ [\iota] \rangle & \mapsto & \mathsf{Eval}(s, b, \overline{V}) \text{ if } \iota \in \mathcal{Q} \\
[\langle \texttt{builtin}\ b\ \overline{V}\ (\iota :: \varepsilon) \rangle\ \_] :: s \lhd V & \mapsto & s \lhd \langle \texttt{builtin}\ b\ (\overline{V} \cdot V)\ \varepsilon \rangle \text{ if } V \sim \iota \\
[\langle \texttt{builtin}\ b\ \overline{V}\ [\iota] \rangle\ \_] :: s \lhd V & \mapsto & \mathsf{Eval}(s, b, \overline{V} \cdot V) \text{ if } V \sim \iota
\end{array}
$$

(a) CEK machine transitions for Plutus Core

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

(b) Evaluation of built-in functions

Figure 11: A CEK machine for Plutus Core

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 supports a default set of built-in types and type operators defined in Figures 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`	The set of sequences of Unicode scalar values, as defined in §3.9, definition D76 of the Unicode 5.2 standard.	See note below
`bool`	{`true, false`}	`True | False`
`unit`	{()}	`()`
`data`	See below.	Not yet supported

Table 1: Atomic Types

Operator *op*	$\lvert op \rvert$	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.

A.2 Built-in functions

The default set of built-in functions for the Alonzo release is shown in Figure 3.

Function	Signature	Denotation	Can Fail?	Note
`addInteger`	`[integer, integer]` → `integer`	+		

Table 3: Built-in Functions

Function	Type	Denotation	Can Fail?	Note
`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	Convert a `string` to a `bytestring`.		4
`decodeUtf8`	[bytestring] → string	Convert a `bytestring` to a `string`.	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`.		
`verifySignature`	[bytestring, bytestring, bytestring] → bool	Verify the signature using `Ed25519`.	Yes	5
`ifThenElse`	$[\forall a_*, \texttt{bool}, a_*, a_*] \to a_*$	$(\texttt{true}, t_1, t_2) \mapsto t_1$, $(\texttt{false}, t_1, t_2) \mapsto t_2$		
`chooseUnit`	$[\forall a_*, \texttt{unit}, a_*] \to a_*$	$((), t) \mapsto t$		
`trace`	$[\forall a_*, \texttt{string}, a_*] \to a_*$	$(s, t) \mapsto t$		6
`fstPair`	$[\forall a_\#, \forall b_\#, \texttt{pair}\,(a_\#, b_\#)] \to a_\#$	$(x, y) \mapsto x$		
`sndPair`	$[\forall a_\#, \forall b_\#, \texttt{pair}\,(a_\#, b_\#)] \to b_\#$	$(x, y) \mapsto y$		

Table 3: Built-in Functions

Function	Type	Denotation	Can Fail?	Note
`chooseList`	$[\forall a_\#, \forall b_*, \texttt{list}\,(a_\#), b_*, b_*] \to b_*$	$([], t_1, t_2) \mapsto t_1$, $([x_1, \ldots, x_n], t_1, t_2) \mapsto t_2$ $(n \geq 1)$.		
`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`	$[\forall a_*, \texttt{data}, a_*, a_*, a_*, a_*] \to a_*$	$(d, t_C, t_M, t_L, t_I, t_B) \mapsto \begin{cases} t_C & \text{if is}_C(d) \\ t_M & \text{if is}_M(d) \\ t_L & \text{if is}_L(d) \\ t_I & \text{if is}_I(d) \\ t_B & \text{if is}_B(d) \end{cases}$		
`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`	$[\texttt{data}] \to \texttt{pair}\,(\texttt{integer}, \texttt{data})$	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 negative. For all $a, b \in \mathbb{Z}$ with $a \neq 0$ we have

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

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

and

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

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

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 ($0 \leq \mathrm{mod}(a, b) \leq b-1$). The quot function truncates towards zero. Figure 4 shows how the signs of the outputs of the division functions depend on the signs of the inputs (+ means non-negative, so includes 0).

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, but $[\![\texttt{bytestring}]\!]$ consists of sequences of 8-bit bytes. As the names suggest, both functions use the well-known UTF-8 character encoding, where each Unicode character is 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. Signature verification. The `verifySignature` performs cryptographic signature verification using the Ed25519 scheme (Bernstein et al. [2011]). The function takes three bytestring arguments: a public key k, a message m, and a signature s (in that order). The key k must be exactly 32 bytes long and the signature s must be exactly 64 bytes, and the function will fail if either is the wrong length; there is no restriction on the length of the message. If k and s are the correct lengths then `verifySignature` 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. 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.

Appendix B Formally Verified Behaviours

To follow.

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

[Possibly incomplete and/or inaccurate.]

We use the `flat` [Assini] format to serialise Plutus Core terms. The `flat` format encodes sum types as tagged unions and products by concatenating their contents. We proceed by defining the structure and the data types of untyped Plutus Core and how they get serialised.

C.1 Variable length data

`Non-empty lists` are encoded by prefixing the element stored with '0' if this is the `last` element or '1' if there is `more` data following.
We encode `Integers` as a non-empty list of chunks, 7 bits each, with the least significant chunk first and the most significant bit first in the chunk.

Let's calculate the encoding of the `32768` index (unsigned, arbitrary length integer):

1. Converting `32768` to binary:
   `32768` → `0b1000000000000000`
2. Split into 7 bit chunks:
   `0b1000000000000000` → `0000010 0000000 0000000`
3. Reorder chunks (least significant chunk first):
   `0000010 0000000 0000000` → `0000000 0000000 0000010`
4. Add list constructor tags:
   `0000000 0000000 0000010` → `10000000 10000000 00000010`

For `ByteStrings` and `Strings` we use a byte aligned array of bytes (in the case of `String` the bytes correspond to the UTF-8 encoding of the text). The structure is pre-aligned to the byte boundary by using the '0' bit as a filler and the '1' bit as the final bit. Following the filler we have the number of bytes that the data uses, a number from 0 to 255 (1 byte), followed by the bytes themselves, and a final 0 length block (the byte '0').

C.2 Constants

Constants are encoded as a combination of a sequence of 4-bit tags indicating the type of value that is serialised and the value itself: see Figure 12. Constants use 4 bits to encode the type tags, so they allow for a maximum of 16 constructors, of which 9 are used by the default set of builtin types.

Name	Tag	Encoding
`integer`	0	ZigZag + Variable length
`bytestring`	1	Variable length
`string`	2	UTF-8
`unit`	3	Empty
`bool`	4	'1' is True, '0' is False
`list`	5	See below
`pair`	6	See below
`Type application`	7	See below
`data`	8	See below

Figure 12: Serialising constants

Encoding types. Basic types (those in $\mathcal{U}_0$) and type operators (in $\mathcal{O}$) are encoded using a single tag. Complex types such as `list (integer)` or `pair (bool, list (string))` are regarded as iterated applications $(\ldots((op(t_1))(t_2)\ldots)(t_n)$ and are encoded by emitting a special tag for each application $t(t')$ followed by the encodings of t and t' (this approach permits some extra flexibility which we do not currently use). Thus in the default set of built-in types, `list (integer)` would be encoded as the concatenation of the sequence of 4-bit numbers $(7, 5, 0)$ and `pair (bool, list (string))` would be encoded as the concatenation of the sequence $(7, 7, 6, 4, 7, 5, 2)$.

Encoding data. Values of type `data` are encoded by emitting the tag for `data` followed by the CBOR encoding (see Bormann and Hoffman [2020]) of the Haskell type

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

See the Haskell code in `plutus-core/plutus-core/src/PlutusCore/Data.hs` in the Plutus GitHub repository IOHK [2019] for full details of this encoding.

Encoding primitive values.

- Unsigned values of type `integer` and `bytestring` are encoded using the previously introduced encoding for variable length data types.
- Signed `integer` values are first converted to an unsigned value using the `ZigZag`* encoding, then they are encoded as variable length data types.
- `strings` are encoded as lists of characters and use the `UTF-8` encoding.

*The `ZigZag` encoding interleaves positive and negative numbers such that small negative numbers are stored using a small number of bytes.

- The single `()` value of the `unit` type is removed from the serialised data, as are the constructors of any data structure which has only one constructor.
- Variable names are encoded using DeBruijn indices, which are unsigned, arbitrary length integers.

Possibly empty lists are encoded in the standard way, by the tag for the constructor, '0' for 'Nil' and '1' for 'Cons'. The 'Cons' constructor is followed by the serialised element, and then, recursively another list.

Encoded values are aligned to byte/word boundary using a meaningless sequence of '0' bits terminated with a '1' bit.

C.3 Untyped terms

Terms are encoded using 4 bit tags, which allows a total of 16 kinds of term, of which 8 are currently used.

Name	Tag	Arguments
Variable	0	name
Delay	1	term
Lambda abstraction	2	name, term
Application	3	term, term
Constant	4	constant
Force	5	term
Error	6	term
Builtin	7	builtin

Figure 13: Untyped terms

C.4 Built-in functions

Built-in functions use 8 bits for their tags, allowing for a maximum of 128 builtin functions of which 51 are currently used.

Name	Tag	Name	Tag	Name	Tag
addInteger	0	blake2b_256	20	iData	40
subtractInteger	1	verifySignature	21	bData	41
multiplyInteger	2	appendString	22	unConstrData	42
divideInteger	3	equalsString	23	unMapData	43
quotientInteger	4	encodeUtf8	24	unListData	44
remainderInteger	5	decodeUtf8	25	unIData	45
modInteger	6	ifThenElse	26	unBData	46
equalsInteger	7	chooseUnit	27	equalsData	47
lessThanInteger	8	trace	28	mkPairData	48
lessThanEqualsInteger	9	fstPair	29	mkNilData	49
appendByteString	10	sndPair	30	MkNilPairData	50
consByteString	11	chooseList	31		
sliceByteString	12	mkCons	32		
lengthOfByteString	13	headList	33		
indexByteString	14	tailList	34		
equalsByteString	15	nullList	35		
lessThanByteString	16	chooseData	36		
lessThanEqualsByteString	17	constrData	37		
sha2_256	18	mapData	38		
sha3_256	19	listData	39		

Figure 14: Builtin tags

C.5 Example

We will serialise the program `(program 11.22.33 (con integer 11))` compiled to untyped Plutus Core, using DeBruijn indices.

First, lets convert the program to the desired representation:

```
> stack exec plc -- convert --untyped --if plc --of flat -o program.flat <<EOF
> (program 11.22.33 (con integer 11))
> EOF
```

Now, let's take a look at the output.

```
> xxd -b program.flat
> 00000000: 00001011 00010110 00100001 01001000 00000101 10000001  ..!H..
```

C.5.1 The program preamble.

We define ‘Program‘ in the ‘PlutusCore.Core.Type‘ haskell module like this:

```
-- | A 'Program' is simply a 'Term' coupled with a 'Version'
--   of the core language.
data Program tyname name uni fun ann =
       Program ann (Version ann) (Term tyname name uni fun ann)
         deriving (Show, Functor, Generic, NFData, Hashable)
```

Because the `Program` data type has only one constructor we know that flat will not waste any space serialising it. ‘ann’ will always be (for serialised ASTs) ‘()’, which similarly to the `Program` data type has only one constructor and flat will not serialise it.

Next, the ‘Version’ is a tuple of 3 ‘Natural’ numbers, which are encoded as variable length unsigned integers. Because all the version numbers can fit in a 7 bit word, we only need one byte to store each of them. Also, the first bit, which represents the non-empty list constructor will always be ‘0’ (standing for ‘Last’), resulting in:

```
0 (*Last*) 000 (*Unused*) 1011   (*11 in binary*)
0 (*Last*) 00  (*Unused*) 10110  (*22 in binary*)
0 (*Last*) 0   (*Unused*) 100001 (*33 in binary*)
```

C.5.2 The integer constant ‘1‘.

Let’s take a quick look at how we defined untyped Plutus Core terms, in Figure 13.

We need to encode the ‘Constant’, signed integer value ‘11’. Terms are encoded using 4 bits, and the ‘Constant’ term has tag 4. This results in:

```
0 (*Unused*) 100 (*4 in binary*)
```

For the ‘Default’ universe we have the constant tags defined from Figure 12, wrapped in a list, followed by the encoding for the constant’s value.

So we see how, for the integer type we care about the type is encoded as a list containing the id ‘0’. We know that we are using 3 bits to store the type of constant, so the encoding will be:

```
1 (*Cons*) 0000 (The ‘0‘ tag using 4 bits for storage) 0 (*Nil*)
```

The annotation will not be serialised, and we are left with the constant itself. Because it is an variable length signed integer, we first need to find out it’s value after conversion to the ‘ZigZag’ format.

```
> stack repl plutus-core:exe:plc
> ghci> import Data.ZigZag
> ghci> zigZag (11 :: Integer)
> 22
```

Next, we need to encode the variable length unsigned integer ‘22’. We only need one byte (as it fits in the available 7 bits), so we end up with the following:

```
0 (*Last*) 00 (*Unused*) 10110 (*22 in binary*) 000001 (*Padding to byte size*)
```

C.5.3 Note

You may notice how in the rest of the codebase we use the 'CBOR' format to serialise everything.

So why did we choose to switch to 'Flat' for on-chain serialisation?

'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. Running the 'flat' benchmarks will show flat consistently out-performing 'CBOR' by about 35% without using compression.

```
> stack bench plutus-benchmark:flat
> cat plutus-benchmark/flat-sizes.md

** Contract: crowdfunding-indices **
Codec            Size    Of minimum   Of maximum
flat             8148    2.240308     0.62652826
cbor             13005   3.5757492    1.0

** Contract: escrow-indices **
Codec            Size    Of minimum   Of maximum
flat             8529    2.2004645    0.6302838
cbor             13532   3.491228     1.0

** Contract: future-indices **
Codec            Size    Of minimum   Of maximum
flat             17654   2.19141      0.6628619
cbor             26633   3.305983     1.0

** Contract: game-indices **
Codec            Size    Of minimum   Of maximum
flat             5158    2.2290406    0.6254395
cbor             8247    3.5639584    1.0

** Contract: vesting-indices **
Codec            Size    Of minimum   Of maximum
flat             8367    2.2288227    0.6273525
cbor             13337   3.5527437    1.0
```

References

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

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. doi: 10.1007/978-3-642-23951-9_9. URL https://www.iacr.org/archive/ches2011/69170125/69170125.pdf.

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

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.

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. ISSN 0304-3975.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576.

IOHK. Plutus GitHub repository. https://github.com/input-output-hk/plutus, 2019.