Formal Specification of the Plutus Core Language v1.0 (RC5.4)

I. Plutus Core

Plutus Core is the target for compilation of Plutus TX, the smart contract language to validate transactions on Cardano. Plutus Core is designed to be simple and easy to reason about using proof assistants and automated theorem provers.

Plutus Core has a strong formal basis. For those familiar with programming language theory, it can be described in one line: higher kinded System F with isorecursive types, and suitable primitive types and values. There are no constructs for data types, which are represented using Scott encodings—we do not use Church encodings since they may require linear rather than constant time to access the components of a data structure. Primitive types, such as Integer and Bytestring, are indexed by their size, which allows the cost (in gas) of operations such as addition to be determined at compile time. In contrast, IELE uses unbounded integers, which never overflow, but require that the gas used is calculated at run time. We use F^{ω} (as opposed to F) because it supports types indexed by size and parameterised types (such as "List of A").

Plutus Core also notable lacks obvious built-in support for recursion, either general or otherwise. The reason for this is simply that in the presence of type left fixed points, it's possible to give a valid type to recursion combinators. This is analogous to the fact that in Haskell, you can define the Y combinator but you need to use a newtype declaration to do. Precisely which combinators one defines is up to the user, but as Plutus itself is eagerly reduced, some of them will lead to looping behavior, including Y. Libraries can of course be defined with various combinators provided as a convenient way to gain access to recursion.

II. Syntax

The grammar of Plutus Core is given in 2 and 1. This grammar describes the abstract syntax trees of Plutus Core, in a convenient notation, and also describes the string syntax to be used when referring to those ASTs. The string syntax is not fundamental to Plutus Core, and only exists because we must refer to programs in documents such as this. Plutus Core programs are intended exist only as ASTs produced by compilers from higher languages, and as serialized representations on blockchains, and therefore we do not expect anyone to write programs in Plutus Core, nor need to use a parser for the language.

Lexemes are described in standard regular expression notation. The only other lexemes are round `()`, square `[]`, and curly `{}` brackets. Spaces and tabs are allowed anywhere, and have no effect save to separate lexemes.

Application in both terms and types is indicated by square brackets, and instantiation in terms is indicated by curly brackets. We permit the use of multi-argument application and instantiation as syntactic sugar for iterated application. For instance,

$$[M_0\ M_1\ M_2\ M_3]$$

is sugar for

$$[[[M_0\ M_1]\ M_2]\ M_3]$$

All subsequent definitions assume iterated application and instantiation has been expanded out, and use only the binary form. To the extent that a standard utility parser for Plutus Core might be made, for debugging purposes and other such things, iterated application and instantiation ought to be included as sugar.

Name	n	::=	`[a-zA-Z][a-zA-Z0-9_']`*	name
Var	x	::=	n	term variable
TyVar	α	::=	n	type variable
BuiltinName	bn	::=	n	builtin term name
TyBuiltinName	bt	::=	n	builtin type name
Integer	i	::=	`[+-]`$^?$`[0-9]`$^+$	integer
ByteString	b	::=	`#([a-fA-F0-9][a-fA-F0-9])`$^+$	hex string
Size	s	::=	`[0-9]`$^+$	size
Version	v	::=	`[0-9]`$^+$`(.[0-9]`$^+$`)`*	version
Constant	cn	::=	s `!` i	integer constant
			s `!` b	bytestring constant
			s	size constant
			bn	builtin name
TyConstant	tcn	::=	s	size
			bt	builtin type name

Fig. 1. Lexical Grammar of Plutus Core

III. Type Correctness

We define for Plutus Core a number of typing judgments which explain ways that a program can be well-formed. First, in Figure 3, we define the grammar of variable contexts that these judgments hold under. Variable contexts contain information about the nature of variables — type variables with their kind, and term variables with their type.

Then, in Figure 4, we define what it means for a type synthesize a kind. Plutus Core is a higher-kinded version of System F, so we have a number of standard System F rules together with some obvious extensions. In Figure 5, we define the type synthesis judgment, which explains how a term synthesizes a type.

Finally, type synthesis for constants (`(con` bn`)` and `(con` bt`)`) is given in tabular form rather than in inference rule form, in Figure 13, which also gives the reduction semantics. This table also specifies what conditions trigger an error.

IV. Reduction and Execution

In figure 9, we define a standard eager, small-step contextual semantics for Plutus Core in terms of the reduction relation for types ($A \rightarrow_{ty} A'$) and terms ($M \rightarrow M'$), which incorporates both β reduction and contextual congruence. We make use of the transitive closure of these stepping relations via the usual Kleene star notation.

In the context of a blockchain system, it can be useful to also have a step indexed version of stepping, indicated by a superscript count of steps ($M \rightarrow^n M'$). In order to prevent transaction validation from looping indefinitely, or from simply taking an inordinate amount of time, which would be a serious security flaw in the blockchain system, we can use step indexing to put an upper bound on the number of computational steps that a program can have. In this setting, we would pick some upper bound max and then perform steps of terms M by computing which M' is such that $M \rightarrow^{max} M'$.

Term	L, M, N	::=	x	variable
			`(abs` α K V`)`	type abstraction
			`{`M A`}`	type instantiation
			`(wrap` α A M`)`	fix type's wrap
			`(unwrap` M`)`	fix type's unwrap
			`(lam` x A M`)`	λ abstraction
			`[`M N`]`	function application
			`(con` cn`)`	builtin
			`(error` A`)`	error
Value	U, V	::=	`(abs` α K V`)`	type abstraction
			`(wrap` α A V`)`	fix type's wrap
			`(lam` x A M`)`	λ abstraction
			BV	builtin value
Builtin Value	BV	::=	`(con` cn`)`	builtin
			`[`BV V`]`	function application
			`{`BV A`}`	type instantiation
Type	A, B, C	::=	α	type variable
			`(fun` A B`)`	function type
			`(all` α K A`)`	polymorphic type
			`(fix` α A`)`	fixed point type
			`(lam` α K A`)`	λ abstraction
			`[`A B`]`	function application
			`(con` tcn`)`	builtin type
Type Value	R, S, T	::=	`(fun` S T`)`	function type
			`(all` α K S`)`	polymorphic type
			`(fix` α S`)`	fixed point type
			`(lam` α K S`)`	λ abstraction
			`(con` tcn`)`	builtin type
			W	neutral type
Neutral Type	W		α	type variable
			`[`W S`]`	function application
Kind	J, K	::=	`(type)`	type kind
			`(fun` J K`)`	arrow kind
			`(size)`	size kind
Program	P	::=	`(version` v M`)`	versioned program

Fig. 2. Grammar of Plutus Core

$$\begin{array}{rrcll} \text{Ctx} & \Gamma & ::= & \epsilon & \text{empty context} \\ & & & \Gamma, \alpha :: K & \text{type variable} \\ & & & \Gamma, x : A & \text{term variable} \end{array}$$

$\boxed{\Gamma \ni J}$

Context Γ contains judgment J

$$\frac{}{\Gamma, \alpha :: K \ni \alpha :: K}$$

$$\frac{}{\Gamma, x : A \ni x : A}$$

$$\frac{\Gamma \ni \alpha :: K \qquad \alpha \neq \beta}{\Gamma, \beta :: J \ni \alpha :: K}$$

$$\frac{\Gamma \ni \alpha :: K}{\Gamma, y : B \ni \alpha :: B}$$

$$\frac{\Gamma \ni x : A}{\Gamma, \beta :: J \ni x : A}$$

$$\frac{\Gamma \ni x : A \qquad x \neq y}{\Gamma, y : B \ni x : A}$$

$\boxed{\Gamma \text{ valid}}$

Context Γ is valid

$$\frac{}{\epsilon \text{ valid}}$$

$$\frac{\Gamma \text{ valid} \qquad \alpha \text{ is free in } \Gamma}{\Gamma, \alpha :: K}$$

$$\frac{\Gamma \text{ valid} \qquad x \text{ is free in } \Gamma \qquad \Gamma \vdash A :: \texttt{(type)}}{\Gamma, x : A}$$

Fig. 3. Contexts

$\boxed{\Gamma \vdash A :: K}$

In context Γ, type A has kind K

$$\frac{\Gamma \ni \alpha :: K}{\Gamma \vdash \alpha :: K}\ \text{tyvar}$$

$$\frac{\Gamma, \alpha :: K \vdash A :: \texttt{(type)}}{\Gamma \vdash \texttt{(all } \alpha\ K\ A\texttt{)} :: \texttt{(type)}}\ \text{tyall}$$

$$\frac{\Gamma, \alpha :: \texttt{(type)} \vdash A :: K \qquad K = \texttt{(type)}}{\Gamma \vdash \texttt{(fix } \alpha\ A\texttt{)} :: \texttt{(type)}}\ \text{tyfix}$$

$$\frac{\Gamma \vdash B :: \texttt{(type)} \qquad \Gamma \vdash A :: \texttt{(type)}}{\Gamma \vdash \texttt{(fun } B\ A\texttt{)} :: \texttt{(type)}}\ \text{tyfun}$$

$$\frac{\Gamma, \alpha :: J \vdash A :: K}{\Gamma \vdash \texttt{(lam } \alpha\ J\ A\texttt{)} :: \texttt{(fun } J\ K\texttt{)}}\ \text{tylam}$$

$$\frac{\Gamma \vdash A :: \texttt{(fun } J\ K\texttt{)} \qquad \Gamma \vdash B :: J}{\Gamma \vdash \texttt{[}A\ B\texttt{]} :: K}\ \text{tyapp}$$

$$\frac{bt \text{ has kind } K \text{ in Fig. 13}}{\Gamma \vdash \texttt{(con } tcn\texttt{)} :: K}\ \text{tybuiltin}$$

Fig. 4. Kind Synthesis

$\boxed{\Gamma \vdash M : A}$

In context Γ, term M has type A

$$\frac{\Gamma \ni x : A}{\Gamma \vdash x : A}\ \text{var}$$

$$\frac{\Gamma, \alpha :: K \vdash M : B}{\Gamma \vdash \texttt{(abs}\ \alpha\ K\ M\texttt{)} : \texttt{(all}\ \alpha\ K\ B\texttt{)}}\ \text{abs}$$

$$\frac{\Gamma \vdash L : \texttt{(all}\ \alpha\ K\ B\texttt{)} \qquad \Gamma \vdash A :: K}{\Gamma \vdash \texttt{\{}L\ A\texttt{\}} : [A/\alpha]B}\ \text{inst}$$

$$\frac{\begin{array}{c}\Gamma, \alpha :: \texttt{(type)} \vdash A :: \texttt{(type)} \\ \Gamma \vdash M : [\texttt{(fix}\ \alpha\ A\texttt{)}/\alpha]A\end{array}}{\Gamma \vdash \texttt{(wrap}\ \alpha\ A\ M\texttt{)} : \texttt{(fix}\ \alpha\ A\texttt{)}}\ \text{wrap}$$

$$\frac{\Gamma \vdash M : \texttt{(fix}\ \alpha\ A\texttt{)}}{\Gamma \vdash \texttt{(unwrap}\ M\texttt{)} : [\texttt{(fix}\ \alpha\ A\texttt{)}/\alpha]A}\ \text{unwrap}$$

$$\frac{\Gamma, y : A \vdash M : B}{\Gamma \vdash \texttt{(lam}\ y\ A\ M\texttt{)} : \texttt{(fun}\ A\ B\texttt{)}}\ \text{lam}$$

$$\frac{\Gamma \vdash L : \texttt{(fun}\ A\ B\texttt{)} \qquad \Gamma \vdash M : A}{\Gamma \vdash \texttt{[}L\ M\texttt{]} : B}\ \text{app}$$

$$\frac{bi \text{ has type } S \text{ in Fig. 13}}{\Gamma \vdash \texttt{(con}\ cn\texttt{)} : A}\ \text{builtin}$$

$$\frac{\Gamma \vdash A :: \texttt{(type)}}{\Gamma \vdash \texttt{(error}\ A\texttt{)} : A}\ \text{error}$$

Fig. 5. Type Synthesis

$$\begin{array}{lrll} \text{Type Frame} & f & ::= & \texttt{(fun}\ \_\ A\texttt{)} & \text{left arrow} \\ & & & \texttt{(fun}\ S\ \_\texttt{)} & \text{right arrow} \\ & & & \texttt{(all}\ \alpha\ K\ \_\texttt{)} & \text{all} \\ & & & \texttt{(lam}\ \alpha\ K\ \_\texttt{)} & \lambda \\ & & & \texttt{[}\_\ A\texttt{]} & \text{left app} \\ & & & \texttt{[}S\ \_\texttt{]} & \text{right app} \end{array}$$

Fig. 6. Grammar of Type Reduction Frames

$\boxed{A \ \rightarrow_{ty} \ A'}$

Type A reduces in one step to type A'

$$\frac{}{\texttt{[(lam}\ \alpha\ K\ B\texttt{)}\ S\texttt{]} \ \rightarrow_{ty} \ [S/\alpha]B}$$

$$\frac{A \ \rightarrow_{ty} \ A'}{f\{A\} \ \rightarrow_{ty} \ f\{A'\}}$$

Fig. 7. Type Reduction via Contextual Dynamics

Frame	$\texttt{\{\_}\ A\texttt{\}}$	left instantiation
	$\texttt{(wrap}\ \alpha\ A\ \texttt{\_)}$	right wrap
	$\texttt{(unwrap \_)}$	unwrap
	$\texttt{[\_}\ M\texttt{]}$	left app
	$\texttt{[}V\ \texttt{\_]}$	right app

Fig. 8. Grammar of Reduction Frames

$\boxed{M \ \rightarrow \ M'}$

Term M reduces in one step to term M'

$$\frac{}{\texttt{\{(abs}\ \alpha\ K\ M\texttt{)}\ A\texttt{\}} \ \rightarrow \ [A/\alpha]M}$$

$$\frac{}{\texttt{(unwrap (wrap}\ \alpha\ A\ V\texttt{))} \ \rightarrow \ V}$$

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

$$\frac{M \text{ is a fully saturated constant} \quad M \text{ computes to } V \text{ according to Fig 13}}{M \ \rightarrow \ V}$$

$$\frac{M \ \rightarrow \ M' \qquad M' = \texttt{(error}\ B\texttt{)}}{f\{M\} \ \rightarrow \ \texttt{(error}\ A\texttt{)}} \textit{ (A is the type of the frame, B is the type of its hole)}$$

$$\frac{M \ \rightarrow \ M' \qquad M' \neq \texttt{(error}\ B\texttt{)}}{f\{M\} \ \rightarrow \ f\{M'\}}$$

Fig. 9. Reduction via Contextual Dynamics

Stack	s	$::=$	f^*	stacks
State	σ	$::=$	$s \triangleright M$	computing a term
			$s \triangleleft V$	returning a term value
			$\blacklozenge$	throwing an error

Fig. 10. Grammar of CK Machine States

$\boxed{\sigma \;\mapsto\; \sigma'}$

Machine state σ transitions in one step to machine state σ'

$$
\begin{array}{l}
s \triangleright (\texttt{abs}\ \alpha\ K\ M) \;\mapsto\; s \triangleleft (\texttt{abs}\ \alpha\ K\ M) \\
s \triangleright \{M\ A\} \;\mapsto\; s, \{\_\ A\} \triangleright M \\
s \triangleright (\texttt{wrap}\ \alpha\ A\ M) \;\mapsto\; s, (\texttt{wrap}\ \alpha\ A\ \_) \triangleright M \\
s \triangleright (\texttt{unwrap}\ M) \;\mapsto\; s, (\texttt{unwrap}\ \_) \triangleright M \\
s \triangleright (\texttt{lam}\ x\ A\ M) \;\mapsto\; s \triangleleft (\texttt{lam}\ x\ A\ M) \\
s \triangleright [M\ N] \;\mapsto\; s, [\_\ N] \triangleright M \\
s \triangleright (\texttt{con}\ cn) \;\mapsto\; s \triangleleft (\texttt{con}\ cn) \\
s \triangleright (\texttt{error}\ A) \;\mapsto\; \blacklozenge \\
s, \{\_\ A\} \triangleleft (\texttt{abs}\ \alpha\ K\ M) \;\mapsto\; s \triangleright M \\
s, \{\_\ A\} \triangleleft M \;\mapsto\; s \triangleleft V \quad \textit{fully saturated constant} \\
s, \{\_\ A\} \triangleleft M \;\mapsto\; s \triangleleft \{M\ A\} \quad \textit{partially saturated constant} \\
s, (\texttt{wrap}\ \alpha\ A\ \_) \triangleleft V \;\mapsto\; s \triangleleft (\texttt{wrap}\ \alpha\ A\ V) \\
s, (\texttt{unwrap}\ \_) \triangleleft (\texttt{wrap}\ \alpha\ A\ V) \;\mapsto\; s \triangleleft V \\
s, [\_\ N] \triangleleft V \;\mapsto\; s, [V\ \_] \triangleright N \\
s, [\,(\texttt{lam}\ x\ A\ M)\ \_] \triangleleft V \;\mapsto\; s \triangleright [V/x]M \\
s, [M\ \_] \triangleleft N \;\mapsto\; s \triangleleft V \quad \textit{fully saturated constant} \\
s, [M\ \_] \triangleleft N \;\mapsto\; s \triangleleft [M\ N] \quad \textit{partially saturated constant}
\end{array}
$$

Fig. 11. CK Machine

Abbreviation	Expanded
$\forall \alpha :: K.\ B$	`(all` α K B`)`
$\forall \alpha, \beta, \ldots :: K, \ldots .\ C$	$\forall \alpha :: K.\ \forall \beta :: K.\ \ldots .\ C$
$A \to B$	`(fun` A B`)`
$err(A)$	`(error` A`)`
$integer_s$	`[(con integer)` s`]`
$bytestring_s$	`[(con bytestring)` s`]`
$size_s$	`[(con size)` s`]`
$\star$	`(type)`
$size$	`(size)`
$unit$	$\forall \alpha :: \star.\ \alpha \to \alpha$
$unitval$	`(abs` α `(type) (lam` x α x`))`
$boolean$	$\forall \alpha :: \star.\ (unit \to \alpha) \to (unit \to \alpha) \to \alpha$
$true$	`(abs` α `(type) (lam` t $(unit \to \alpha)$ `(lam` f $(unit \to \alpha)$ `[`t $unitval$`])))`
$false$	`(abs` α `(type) (lam` t $(unit \to \alpha)$ `(lam` f $(unit \to \alpha)$ `[`f $unitval$`])))`

Fig. 12. Abbreviations

Builtin Type Name	Kind	Arguments	Semantics
$integer$	`(fun (size) (type))`	s	$[-2^{8s-1}, 2^{8s-1})$
$bytestring$	`(fun (size) (type))`	s	$\cup_{0<s'\leq s}\{0,1\}^{8s'}$
$size$	`(fun (size) (type))`	s	$\{s\}$
s	`(size)`		s

Let txh be the transaction hash, $bnum$ be the block number, and $btime$ be the block time, all as global parameters to normalization.

Builtin Name	Type	Arguments	Semantics	Success Conditions
$s!i$	$integer_s$	—	$s!i$	
$s!b$	$bytestring_s$	—	$s!b$	
z	$size_s$	—	z	
`addInteger`	$\forall s :: size.\ integer_s \to integer_s \to integer_s$	$s!i_0, s!i_1$	$s!(i_0 + i_1)$	$-2^{8s-1} \leq i_0 + i_1 < 2^{8s-1}$
`subtractInteger`	$\forall s :: size.\ integer_s \to integer_s \to integer_s$	$s!i_0, s!i_1$	$s!(i_0 - i_1)$	$-2^{8s-1} \leq i_0 - i_1 < 2^{8s-1}$
`multiplyInteger`	$\forall s :: size.\ integer_s \to integer_s \to integer_s$	$s!i_0, s!i_1$	$s!(i_0 * i_1)$	$-2^{8s-1} \leq i_0 * i_1 < 2^{8s-1}$
`divideInteger`	$\forall s :: size.\ integer_s \to integer_s \to integer_s$	$s!i_0, s!i_1$	$s!(\text{div}\ i_0\ i_1)$	$i_1 \neq 0$
`remainderInteger`	$\forall s :: size.\ integer_s \to integer_s \to integer_s$	$s!i_0, s!i_1$	$s!(\text{mod}\ i_0\ i_1)$	$i_1 \neq 0$
`lessThanInteger`	$\forall s :: size.\ integer_s \to integer_s \to boolean$	$s!i_0, s!i_1$	$i_0 < i_1$	
`lessThanEqualsInteger`	$\forall s :: size.\ integer_s \to integer_s \to boolean$	$s!i_0, s!i_1$	$i_0 <= i_1$	
`greaterThanInteger`	$\forall s :: size.\ integer_s \to integer_s \to boolean$	$s!i_0, s!i_1$	$i_0 > i_1$	
`greaterThanEqualsInteger`	$\forall s :: size.\ integer_s \to integer_s \to boolean$	$s!i_0, s!i_1$	$i_0 >= i_1$	
`equalsInteger`	$\forall s :: size.\ integer_s \to integer_s \to boolean$	$s!i_0, s!i_1$	$i_0 == i_1$	
`resizeInteger`	$\forall s_0, s_1 :: size.\ size_{s_1} \to integer_{s_0} \to integer_{s_1}$	$z, s_0!i$	$s_1!i$	$-2^{8s_1-1} \leq i < 2^{8s_1-1}$
`intToByteString`	$\forall s_0, s_1 :: size.\ size_{s_1} \to integer_{s_0} \to bytestring_{s_1}$	$z, s_0!i$	the binary representation of i 0 padded to a most-significant-bit-first s_1-byte bytestring	$-2^{8s_1-1} \leq i < 2^{8s_1-1}$
`concatenate`	$\forall s :: size.\ bytestring_s \to bytestring_s \to bytestring_s$	$s!b_0, s!b_1$	$s!(b_0 \cdot b_1)$	$\lvert b_0 \cdot b_1 \rvert \leq s$
`takeByteString`	$\forall s_0, s_1 :: size.\ integer_{s_0} \to bytestring_{s_1} \to bytestring_{s_1}$	$s_0!i, s_1!b$	$s_1!(take\ i\ b)$	
`dropByteString`	$\forall s_0, s_1 :: size.\ integer_{s_0} \to bytestring_{s_1} \to bytestring_{s_1}$	$s_0 1i, s_1!b$	$s_1!(drop\ i\ b)$	
`sha2_256`	$\forall s :: size.\ bytestring_s \to bytestring_{\texttt{(con 256)}}$	$s!b$	$256!(sha2\_256\ b)$	
`sha3_256`	$\forall s :: size.\ bytestring_s \to bytestring_{\texttt{(con 256)}}$	$s!b$	$256!(sha3\_256\ b)$	
`verifySignature`	$\forall s_0, s_1, s_2 :: size.\ bytestring_{s_0} \to bytestring_{s_1} \to bytestring_{s_2} \to boolean$	k, d, s	$true$ if the private key corresponding to public key k was used to sign d to produce s, otherwise $false$	
`resizeByteString`	$\forall s_0, s_1 :: size.\ size_{s_1} \to bytestring_{s_0} \to bytestring_{s_1}$	$z, s_0!b$	$s_1!b$	$\lvert b \rvert \leq s_1$
`equalsByteString`	$\forall s :: size.\ bytestring_s \to bytestring_s \to boolean$	b_0, b_1	$b_0 == b_1$	
`txhash`†	$bytestring_{\texttt{(con 256)}}$	—	$256!txh$	
`blocknum`	$\forall s :: size.\ size_s \to integer_s$	z	$s!bnum$	$0 \leq bnum < 2^{8s-1}$

† `txhash` and its meaning txh are the hash of the transaction, containing information as specified by the host blockchain.

Fig. 13. Builtin Types and Reductions

V. Basic Validation Program Structure

The basic way that validation is done in Plutus Core is somewhat similar to what's in Bitcoin Script. Whereas in Bitcoin Script, a validation is successful if the validating script successfully executes and leaves *true* on the top of the stack, in Plutus Core, validation is successful when the script reduces to any value other than `(error A)` in the allotted number of steps.

VI. Erasure

TO WRITE

VII. Example

We illustrate the use of Plutus Core by constructing a simple validator program. We present components of this program in a high-level style. i.e., we write

```
one : unit
one = (abs a (type) (lam x a x))
```

for the element of the type $unit$ (defined in 12).

We stress that declarations in this style are **not part of the Plutus Core language**. We merely use the familiar syntax to present out example. If the high-level definitions in our example were compiled to a Plutus Core expression, it would result in something like figure 22.

We proceed by defining the booleans. Like $unit$, the type $boolean$ below is an abbreviation in the specification. Some built-in constants return values of type $boolean$. When needed, user programs should contain the declarations below. We have

```
true : boolean
true = (abs a (type)
        (lam x (fun unit a)
        (lam y (fun unit a)
          [x one])))
```

and similarly

```
false : boolean
false = (abs a (type)
         (lam x (fun unit a)
         (lam y (fun unit a)
           [y one])))
```

Next, we define the "case" function for the type $boolean$ as follows:

```
case : (all a (type)
       (fun boolean
       (fun a (fun a a))))
case = (abs a (type)
       (lam b boolean
       (lam t a
       (lam f a
         [ {b a}
           (lam x unit t)
           (lam x unit f)
         ]
       ))))
```

The reader is encouraged to verify that

```
[{case a} true x y] →* x
```

and

```
[{case a} false x y] →* y
```

We can use `case` to define the following function:

```
verifyIdentity :
  (fun [(con bytestring) 2048]
  (fun [(con bytestring) 256] unit))
verifyIdentity =
  (lam pubkey [(con bytestring) 2048]
  (lam signed [(con bytestring) 256]
    [ case [ {verifySignature 256
                              256
                              2048}
               signed
               txhash
               pubkey
           ]
      one
      (error unit)
    ]))
```

the idea being that the first arguemnt is a public key, and the second argument is the result of signing the hash of the current transaction (accessible via $txhash$: `[(con bytestring) 256]`) with that public key. The function terminates if and only if the signature is valid, raising an `error` otherwise. Now, given Alice's public key we can apply our function to obtain one that verifies

Term	L, M, N		x	variable
			$(\texttt{fix}\ x\ M)$	fixed point term
			$(\texttt{lam}\ x\ M)$	λ abstraction
			$[M\ N^{+}]$	function application
			$(\texttt{con}\ bi)$	builtin
			$(\texttt{error})$	error
Value	V, W	$::=$	$(\texttt{lam}\ x\ M)$	λ abstraction
			$(\texttt{con}\ bi)$	builtin
			$(\texttt{error})$	error

Fig. 14. Grammar of Plutus Core Erasure

Frame	f	$::=$	$[\_\ M]$	left app
			$[V\ \_]$	right app

Fig. 15. Grammar of Erasure Reduction Frames

$\boxed{M \to M'}$

Term M reduces in one step to term M'

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

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

$$\frac{M \to M' \qquad M' = (\texttt{error})}{f\{M\} \to (\texttt{error})}$$

$$\frac{M \to M' \qquad M' \neq (\texttt{error})}{f\{M\} \to f\{M'\}}$$

Fig. 16. Erasure Reduction via Contextual Dynamics

Stack	s	$::=$	f^{*}	stacks
State	σ	$::=$	$s \triangleright M$	computing a term
			$s \triangleleft V$	returning a term value

Fig. 17. Grammar of Erasure CK Machine States

whether or not its input is the result of Alice signing the current block number. Again, we stress that the Plutus Core expression corresponding to `verifyIdentity` is going to look something like figure 22.

With minimal modification we might turn our function into one that verifies a signature of the current block number. Specifically, by replacing $txhash$ with

```
[ {intToByteString 128 256}
  256
  [{blocknum 128} 128]
]
```

Notice that we must supply $blocknum$ with the size we wish to use to store the result twice, once at the type level and again at the term level. This is necessary because we want to have the size

$\boxed{\sigma \mapsto \sigma'}$

Machine state σ transitions in one step to machine state σ'

$$
\begin{aligned}
& s \triangleright \texttt{(lam}\ x\ M\texttt{)} \mapsto s \triangleleft \texttt{(lam}\ x\ M\texttt{)} \\
& s \triangleright \texttt{[}M\ N\texttt{]} \mapsto s, \texttt{[\_}\ N\texttt{]} \triangleright M \\
& s \triangleright \texttt{(error)} \mapsto s \triangleleft \texttt{(error)} \\
& s, \texttt{[\_}\ N\texttt{]} \triangleleft V \mapsto s, \texttt{[}V\ \texttt{\_]} \triangleright N \\
& s, \texttt{[(lam}\ x\ M\texttt{)}\ \texttt{\_]} \triangleleft V \mapsto s \triangleright [V/x]M \\
& s, f \triangleleft \texttt{(error)} \mapsto s \triangleleft \texttt{(error)}
\end{aligned}
$$

Fig. 18. Erasure CK Machine

$$
\begin{aligned}
\lfloor x \rfloor &= x \\
\lfloor \texttt{(fix}\ x\ A\ M\texttt{)} \rfloor &= \texttt{(fix}\ x\ \lfloor M \rfloor\texttt{)} \\
\lfloor \texttt{(abs}\ \alpha\ K\ M\texttt{)} \rfloor &= \lfloor M \rfloor \\
\lfloor \texttt{\{}M\ A\texttt{\}} \rfloor &= \lfloor M \rfloor \\
\lfloor \texttt{(wrap}\ \alpha\ A\ M\texttt{)} \rfloor &= \lfloor M \rfloor \\
\lfloor \texttt{(unwrap}\ M\texttt{)} \rfloor &= \lfloor M \rfloor \\
\lfloor \texttt{(lam}\ x\ A\ M\texttt{)} \rfloor &= \texttt{(lam}\ x\ \lfloor M \rfloor\texttt{)} \\
\lfloor \texttt{[}M\ N\texttt{]} \rfloor &= \texttt{[}\lfloor M \rfloor\ \lfloor N \rfloor\texttt{]} \\
\lfloor \texttt{(con}\ bi\texttt{)} \rfloor &= \texttt{(con}\ bi\texttt{)} \\
\lfloor \texttt{(error}\ A\texttt{)} \rfloor &= \texttt{(error)}
\end{aligned}
$$

Fig. 19. Plutus Core Erase Definition

information available both at the type level, to facilitate gas calculations, and at the term level, so that once types are erased the runtime will know how much memory to allocate. This quirk is present in a number of the built in functions.

Theorem: if $M \;\to\; M'$ then $\lfloor M \rfloor \;\to^*\; \lfloor M' \rfloor$

Proof: Induction on the proof $\mathcal{D}$ of $M \;\to\; M'$

Case $\mathcal{D} =$

$$\frac{}{\texttt{\{(abs}\;\alpha\;K\;M\texttt{)}\;S\texttt{\}} \;\to\; [S/\alpha]M}$$

Proof:

$$\begin{array}{ll} & \lfloor \texttt{\{(abs}\;\alpha\;K\;M\texttt{)}\;S\texttt{\}} \rfloor \;\to^*\; \lfloor [S/\alpha]M \rfloor \\ \Leftrightarrow & \{\text{definition of } \lfloor\_\rfloor \text{ twice}\} \\ & \lfloor M \rfloor \;\to^*\; [S/\alpha]M \\ \square & \{\text{type substitution lemma}\} \end{array}$$

Case $\mathcal{D} =$

$$\frac{}{\texttt{(unwrap (wrap}\;\alpha\;A\;M\texttt{))} \;\to\; M}$$

Proof:

$$\begin{array}{ll} & \lfloor \texttt{(unwrap (wrap}\;\alpha\;A\;M\texttt{))} \rfloor \;\to^*\; \lfloor M \rfloor \\ \Leftrightarrow & \{\text{definition of } \lfloor\_\rfloor \text{ twice}\} \\ & \lfloor M \rfloor \;\to^*\; \lfloor M \rfloor \\ \square & \end{array}$$

Case $\mathcal{D} =$

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

Proof:

$$\begin{array}{ll} & \lfloor \texttt{[(lam}\;x\;A\;M\texttt{)}\;V\texttt{]} \rfloor \;\to^*\; \lfloor [V/x]M \rfloor \\ \Leftrightarrow & \{\text{definition of } \lfloor\_\rfloor \text{ twice}\} \\ & \texttt{[(lam}\;x\;\lfloor M \rfloor\texttt{)}\;\lfloor V \rfloor\texttt{]} \;\to^*\; \lfloor [V/\rfloor]xM \\ \Leftrightarrow & \{\beta \text{ reduction}\} \\ & [\lfloor V \rfloor/x]\lfloor M \rfloor \;\to^*\; \lfloor [V/x]M \rfloor \\ \Leftrightarrow & \{\text{term substitution lemma}\} \\ & [\lfloor V \rfloor/x]\lfloor M \rfloor \;\to^*\; [\lfloor V \rfloor/x]\lfloor M \rfloor \\ \square & \end{array}$$

Fig. 20. Plutus Core Erasure Theorem

Case $\mathcal{D} =$

$$\frac{}{\texttt{(fix}\ x\ A\ M\texttt{)}\ \rightarrow\ [\texttt{(fix}\ x\ A\ M\texttt{)}/x]M}$$

Proof:

$$\begin{array}{ll} & \lfloor \texttt{(fix}\ x\ A\ M\texttt{)} \rfloor\ \rightarrow^*\ \lfloor [\texttt{(fix}\ x\ A\ M\texttt{)}/x]M \rfloor \\ \Leftrightarrow & \{\text{term substitution lemma}\} \\ & \lfloor \texttt{(fix}\ x\ A\ M\texttt{)} \rfloor\ \rightarrow^*\ [\lfloor \texttt{(fix}\ x\ A\ M\texttt{)} \rfloor/x]\lfloor M \rfloor \\ \Leftrightarrow & \{\text{definition of } \lfloor \_ \rfloor \text{ twice}\} \\ & \texttt{(fix}\ x\ \lfloor M \rfloor\texttt{)}\ \rightarrow^*\ [\texttt{(fix}\ x\ \lfloor M \rfloor\texttt{)}/x]\lfloor M \rfloor \\ \Leftrightarrow & \{\beta \text{ reduction}\} \\ & [\texttt{(fix}\ x\ \lfloor M \rfloor\texttt{)}/x]\lfloor M \rfloor\ \rightarrow^*\ [\texttt{(fix}\ x\ \lfloor M \rfloor\texttt{)}/x]\lfloor M \rfloor \\ \square & \end{array}$$

Case $\mathcal{D} =$

$$\frac{A\ \rightarrow_{ty}\ A'}{f\{A\}\ \rightarrow\ f\{A'\}}$$

Show: $\lfloor f\{A\} \rfloor\ \rightarrow^*\ \lfloor f\{A'\} \rfloor$
Proof (omitted): Cases on f, then definition of $\lfloor \_ \rfloor$.
Case $\mathcal{D} =$

$$\frac{M\ \rightarrow\ M' \qquad M' = \texttt{(error}\ A\texttt{)}}{f\{M\}\ \rightarrow\ \texttt{(error}\ A\texttt{)}}$$

Show: $\lfloor f\{M\} \rfloor\ \rightarrow^*\ \lfloor \texttt{(error}\ A\texttt{)} \rfloor$
Proof (omitted): Cases on f, then definition of $\lfloor \_ \rfloor$, $\_ \rightarrow \_$, and inductive hypotheses.
Case $\mathcal{D} =$

$$\frac{M\ \rightarrow\ M' \qquad M' \neq \texttt{(error}\ A\texttt{)}}{f\{M\}\ \rightarrow\ f\{M'\}}$$

Show: $\lfloor f\{M\} \rfloor\ \rightarrow^*\ \lfloor \texttt{(error}\ A\texttt{)} \rfloor$
Proof (omitted): Cases on f, then definition of $\lfloor \_ \rfloor$, $\_ \rightarrow \_$, and inductive hypotheses.

Type Substitution Lemma: $\lfloor [A/\alpha]M \rfloor = \lfloor M \rfloor$
Proof (omitted): Cases on M, then definition of $\lfloor \_ \rfloor$

Term Substitution Lemma: $\lfloor [M/x]N \rfloor = [\lfloor M \rfloor/x]\lfloor N \rfloor$
Proof (omitted): Cases on N, then definition of $\lfloor \_ \rfloor$

Fig. 21. Plutus Core Erasure Theorem (cont.)

```
[
  (lam prog
    (fun unit
    (fun boolean
    (fun boolean
    (fun (all a (type) (fun boolean (fun a (fun a a))))
    (fun [(con bytestring) 256]
    (fun [(con bytestring) 2048] (fun [(con bytestring) 256] unit)))))))))
    [{prog (all a (type) (fun a a))
         (all a (type) (fun (fun unit a) (fun (fun unit a) a)))}
         (abs a (type) (lam x a x))
         (abs a (type) (lam x (fun unit a) (lam y (fun unit a) [x one])))
         (abs a (type) (lam x (fun unit a) (lam y (fun unit a) [y one])))
         (abs a (type) (lam b boolean (lam t a (lam f a
                                  [{b a} (lam x unit t) (lam x unit f)])))))])
    (lam one unit
    (lam true boolean
    (lam false boolean
    (lam case (all a (type) (fun boolean (fun a (fun a a))))
      (lam pubkey [(con bytestring) 2048]
      (lam signed [(con bytestring) 256]
        [case [{verifySignature 256 256 2048} signed txhash pubkey]
              one
              (error unit)])))))))))
]
```

Fig. 22. Example of Section VII written out in full